UNIVERSITY OF TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

International Doctorate School
in Information and Communication Technologies

PhD Dissertation

ENABLING ACCESS TO AND EXPLORATION OF INFORMATION GRAPHS

Matteo Lissandrini

Advisor:
Prof. Yannis Velegrakis
University of Trento

Prof. Themis Palpanas
Paris Descartes University

ACADEMIC YEAR 2017-2018

"fatti non foste a viver come bruti, ma per seguir virtute e canoscenza"

Dante Alighieri – The Divine Comedy, Inferno, Canto XXVI (vv. 119-120)

---

* *"You were not formed to live the life of brutes, but to pursue virtue and knowledge."*

Abstract

Exploratory search is the new frontier of information consumption as it goes well beyond simple *lookups*. Information repositories are ubiquitous and grow larger every day, and automated search systems help users find information in such collections. To extract knowledge from these repositories, the common "query lookup" retrieval paradigm accepts a set of specifications (the query) that describes the objects of interest and then collects such objects. Yet, the query lookup retrieval paradigms commonly in use are no more sufficient to support complex information needs, as they can only provide candidate starting points, but do not help the user in expanding their knowledge. To ease access and consumption of rich information repositories, we address the crucial problem of data exploration. Exploratory tasks match the natural need for finding answers to open-ended information needs within an unfamiliar environment.

In particular, in this dissertation, we focus on enabling access to and exploration of rich information graphs. Within businesses, organizations, and among researchers, data is produced in many forms, large volumes, and different contexts. As a consequence of this heterogeneity, many applications find more useful modelling their datasets with the graph model, where information is represented with entities (nodes) and relationships (edges). Those are the data graphs, the graph databases, the knowledge graphs, or more generally information graphs. The richness of their schema and of their content makes it challenging for users to express appropriate queries and retrieve the desired results. Hence, to allow an effective exploration of a graph, we require: (i) an expressive *query paradigm*, (ii) an intuitive *query mechanism*, and (iii) an appropriate *storage and query processing system*. In this work, we address these three requirements.

An exploratory query should be simple enough to avoid complicate declarative languages (such as SQL or SPARQL), and at the same time, it should retain the flexibility and expressiveness of such languages. For this reason, with respect to the query paradigm, we introduce the notion of *exemplar queries* and propose extensions to handle multiple incomplete examples. An exemplar query is a query method in which the user, or the analyst, circumvents query languages by using examples as input. In particular, the solution we design allows flexible matching in the case of incomplete or partially specified examples.

Moreover, to enable this query paradigm, there is the need for interactive systems that implement an incremental query-constructions mechanism and interactive explorations. To address this need, we study algorithms and implementations based on pseudo-relevance feedback for *exemplar query suggestion*, along with an in-depth study of their effectiveness.

Finally, as there exist many graph databases, high heterogeneity can be observed in the functionalities and performances of these systems. We provide an exhaustive evaluation methodology and a comprehensive study of the existing systems that allow to understand their capabilities and limitations. In particular, we design a novel micro-benchmarking framework for the assessment of the functionalities of some graph databases among the most prominent in the area and provide detailed insights on their performance.

Keywords

[Database usability, query answering, information graphs]

Acknowledgements

The gratitude at the end of a years-long journey cannot be summarized on a pageful of words. The list of names of those to which I'm thankful is longer than I can fathom. Some names you'll find on the front-page of this document, next to my name in the proceedings of some conferences, on the door-bell of my house, and in the recent contacts on my smart-phone, but that's an incomplete list.

To Whom It May Concern.

Contents

Chapter 1

Introduction

Information search usually relies on "query lookup" paradigms. These paradigms are modeled to query-matching problems where the query contains the conditions that describe a well defined information need, and answers are all those items that match the conditions. Yet, this information retrieval model falls short in supporting complex search tasks that arise when the information need is open-ended, poorly understood, and when the user is exploring unfamiliar information landscapes [WR09]. These are all the cases in which the query specification is unknown, vague, or hard-to-define.

Recently, a lot of research has been devoted to exploratory search as a complementary search task to overcome the limits of traditional "query lookup" search paradigms. Exploratory search refers to an open-ended information need, for which the searcher is only aware of the starting point from which to expand their knowledge. As a matter of fact, exploratory search is an important portion of search activities on the web [MBL15, WKW+10]. Exploratory systems have been quite effective in allowing users finding their way through complex, and poorly understood datasets [WR09]. In this work we focus on exploratory search for a particular type of data repositories that has reached widespread use in many different contexts, namely *information graphs*.

Recent advancements in structured knowledge extraction from web documents, alongside the increasing traction of the Linked Data effort, have resulted in the emergence of

a number of large-scale knowledge bases, ontologies, and knowledge graphs, like DBPedia[1], WikiData[2], YAGO [SKW07], and Google Knowledge Vault [DGH+14]. Knowledge graphs represent facts about the world and human knowledge in a structured way [Zha02, RSH+16, DGH+14]. In this model, entities like concepts, people, objects, or places assume the role of vertices, which in turn are connected by facts represented as edges. Hence, they are information networks, where substructures can represent rich and complex situations. As those resources are getting larger and richer, it has become of paramount importance to ease access to the information they contain and to adopt efficient systems for their storage, search, and management.

Motivation. The web has been undoubtedly one of the most important revolutions in the democratization of access to information. Such a revolution has been made possible by the rising of the search engines as an intuitive and powerful tool to allow users to find the informations they needed. In general we can easily see that any collection of data and knowledge is per-se useless if it is not paired with an effective way to actually retrieve the pieces of information for which we are looking. The same is true for any knowledge graph.

Historically, knowledge graphs have been employed to support application in artificial intelligence [Zha02], and for this reason a great body of research has focused on how to allow machines and software easy access to these repositories of information. Yet, they are invaluable assets for humans too. As a matter of fact, a great deal of works have focused recently on different ways to provide query capabilities on top of these repositories and graph-shaped data in general [SCSS15, ZH10, EB11, CZY13].

The two main advantages of popular knowledge graphs are their flexible structure, and their sheer size. These advantages constitute a challenge when it comes to querying these repositories. Their flexible (or better their undefined) schema makes it hard to produce an appropriate query defining the conditions that describe the information of interest, while their size requires an appropriate query processing system able to produce results in reasonable time. Unfortunately, existing approaches assume that the searcher is perfectly aware of the structure and characteristics of interest, and do not provide valuable support for all those cases in which the information need is actually not well defined or simply open ended. Moreover, when it comes to the choice of the graph query

[1] dbpedia.org
[2] wikidata.org

system to adopt, studies on performance capabilities of existing graph databases are generally incomplete, limited in the type and scope of the operations they include, and hard to generalize.

Contributions and outline. In this work we focus on the task of exploratory search in information graphs, and we address the issue of enabling users to perform complex exploratory search on information graphs and graph data in general. Moreover, we present an evaluation methodology to assess the capabilities of the graph data management system at hand. As a result, we provide three important contributions, each of them is introduced below in this chapter and more extensively presented with theorems, algorithms, and experiments in the rest of this dissertation.

In particular, **(1)** we study a new query paradigm to efficiently search complex information within a large and heterogeneous information graph by providing simple examples of aspects of interest, the paradigm is called *exemplar queries* and in this work we study its generalization for multiple examples, i.e., *multi-exemplar queries* (Section 1.1, and Chapter 3). To enable users, and especially novice users, to fully exploit such query paradigm **(2)** we provide solutions for effective query suggestion systems that assist query-construction by providing relevant query reformulations and expansions (Section 1.2, and Chapter 4). Finally, **(3)** we investigate a core component of any knowledge graph storage and search system: graph databases, providing both an evaluation methodology and important insights in the realization of a data management solution for graph data (Section 1.3, and Chapter 5).

The next chapter (Chapter 2), will provide an overview of the state of the art and the related work that exist in the areas of graph search and of graph databases.

Scientific Outcome. Exemplar queries have been introduced in a prior work describing algorithm for both exact and approximate search for isomorphic structures [MLVP14a], later it has been extended to support a more flexible congruence relation, namely strong simulation [MLVP16a]. Their application to support more advanced document search has also been considered [LMP+15] and demonstrated in practice in a prototype system [MLVP14b]. In this line of work, this dissertation contributes with the extension of the exemplar query paradigm to support multiple partial examples [LMVP18], while the

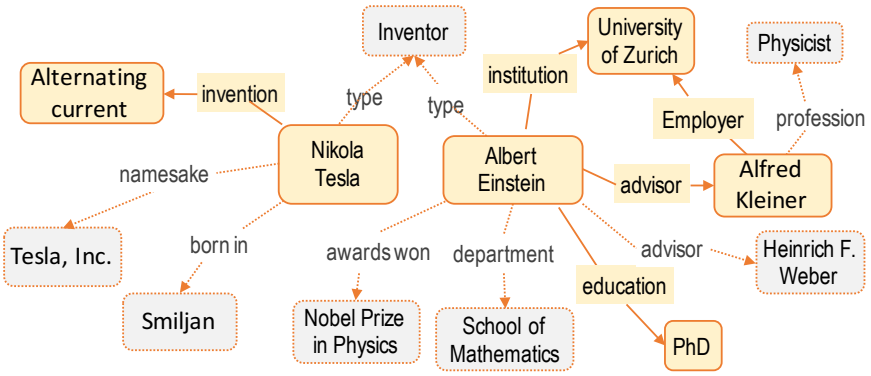

Figure 1.1: Simple Knowledge Graph: describing Einstein and Tesla among others.

study of the methods to enable interactive query suggestion has been recently submitted to a conference [LMPV18a]. A system prototype showcasing an exploratory system integrating all these solutions has been developed [LMPV18b] (under submission). Last but not least, this work presents an evaluation methodology and the results of its application to a set of state-of-the-art graph databases systems, which has been submitted to a conference as well [LBV18].

1.1 Exemplar Queries

We are witnessing a great deal of work towards novel query paradigms that better support both expert and non-expert users in coping with the increasing complexity of data structures and schemas [JCE+07]. User friendliness, language independence, and lack of full schema awareness, have become fundamental factors in these efforts. There are many real world situations where such paradigms found applications, including the exploration of complex big data collections like open data [DAB16a]. This trend is particularly prominent in the field of knowledge-graph search [MLVP17, MLVP14b].

A knowledge-graph models facts in the form of subject-predicate-object triples that can be conceptualized as a directed labeled multi-graph.

Example 1.1 (Figure 1.1)**.** *A knowledge graph can represent entities like "Albert Einstein" or "Nikola Tesla", and concepts like "Inventor". Facts are represented as connections, like "A. Einstein has education PhD".*

Knowledge-graphs have been extensively exploited, especially by popular search engines, to allow effective query expansion [CNGR08], and to enrich search results. By

mapping concepts in the user query to entities in a knowledge graph, additional related informations could be retrieved and presented to the user. Following this intuition, in information retrieval, knowledge-graphs have been integrated with search engines to support and enhance the task for document search, suggest queries refinements, and improve the understanding of ambiguous or underspecified keyword-queries [SWW+11, ZCCW09, GXX13, PIW10]. On the other hand, modern knowledge-graphs are so rich and widespread [KRSW09, DGH+14, CDSE05] that they are no more relegate to the role of a mere facilitator for document search, but are actually the center of specialized search tasks like pattern-matching [CZY13, MLVP16a], entity search [SCSS15, ZCC+17], graph queries [ZH10], and also knowledge explorations [SVLV16, SVJ14, NDC15, WKW+10].

1.1.1 By-Example methods for Knowledge Graph Search

The first contribution of this work is to study *by-example methods* in the context of knowledge graph search. As a matter of fact, *by-example* methods have become a popular paradigm in many contexts. In general, they aim at simplifying information access by facilitating the specification process of the user's need [DPD14, DG16, MSS13, ZW14, MLVP16a, JKL+15, LMP+15]. In particular *exemplar queries* let the user provide an example of the elements of interest instead of a query and require the system to infer the conditions that the elements in the result-set should satisfy. Existing works in relational databases expect the user to present some partially specified tuples that should be contained in the desired result-set [PDCC], provide examples that are marked as relevant or irrelevant [DPD14], specify tuples alongside explanations [DG16], or desired entities [MSS13]. For graph-data, the user is expected to provide as an example a subgraph that constitutes a part of the desired result set [MLVP16a, JKL+15]. There, the example that the user provides does not contain only information about components of interest, but also information on how these components are connected. In a previous work ([MLVP16a, MLVP14a, MLVP14b]) we defined the exemplar query paradigm where the user would submit the description of a situation of interest as a small graph, and the answers retrieved are all the subgraphs in the knowledge graph with the same or similar structure.

Example 1.2 (Exemplar Query)**.** *To search for scientists and their advisors, the query could describe "Albert Einstein member of the University of Zurich with advisor Alfred Kleiner" (Figure 1.1). Answers would be all other structures describing similar entities connected by the same edges, e.g, "Alan Turing member of Princeton University with advisor Alonzo Church".*

A limitation of the aforementioned approaches is that they assume there exists one example (structure or template) that describes the user needs and they expect that such example is known by the user [PDCC, YGCC12, SCC$^+$]. This is not always the case, especially when the information need is complex or not well understood, which is typical in exploration tasks [WR09]. Some relational approaches have allowed for incomplete examples [PDCC, SCC$^+$], but partial examples in the case of graphs have not been studied so far [MLVP16a, JKL$^+$15, MLVP17]. This becomes a limitation since users often are not aware of a single example that fully characterize what they are looking for. Previous work for relational databases [PDCC, SCC$^+$] have showed that in several domains it is often easier to provide multiple partial examples, and expect the system to infer the complete specification by combining the information from the many examples.

The importance of combining and connecting distinct examples in the context of a single query has also been argued in a recent study on partial topology-based network search [XBCW17], which focuses on finding the connections between node-label isomorphic structures in an undirected graph. The provided exact solution, unfortunately, only considers simple networks with labeled nodes, and does not scale to large graphs [XBCW17].

Searching by providing multiple examples finds application in many different real world scenarios. When, for example, lawyers are searching for similar court cases that involve the combination of more than one complex infringement, each example can refer to details of prior judgments and results are other judgments where those infringements appeared together. For biologists, the ability to provide multiple examples facilitates the search for complex molecular structures that contain certain substructures of interest. As a third example, advertisers could use friends, posts, and products from a network of existing customers to the identify target audience for their campaign. In all the above scenarios, it is not possible for one to come up with a single example describing all the desired specification, but even if this was possible, the different ways that

these specifications can be combined are never considered from the existing approaches. The conjunction of the specifications is the unique and default option that has been considered.

1.1.2 Multi-Exemplar Queries

To provide a much needed flexibility in the field of by-example search, in this work, we propose a novel method for query answering that is based on the ability to identify elements that were not known to the user, but share properties with a user-provided set of distinct examples. This kind of queries are then called *multi-exemplar* queries, to emphasize its nature as an extension of previous works on example-driven query paradigms [MLVP16a, JKL+15], that required unique, complete, and complex examples as inputs. We focus specifically on the case of complex large graphs. We assume that the examples that the user provides are in the form of a graph, and we look to find similar cases within a large graph knowledge base. Once these cases have been identified, parts of them will have to be combined to form elements of the answer set.

Example 1.3 (Multi-Exemplar Query)**.** *Consider the previous query presenting as example "Albert Einstein member of the University of Zurich with advisor Alfred Kleiner" (Figure 1.1), and consider also as additional example "Nikola Tesla invented Alternating Current". These two examples can be used in conjunction in a multi-exemplar query. In this case, answers would be all instances in the graph describing combinations of those structures, e.g, "Alan Turing invented the Turing Machine and is member of Princeton University with Advisor Alonzo Church".*

There are different challenges in performing the above tasks successfully to implement a multi-exemplar query mechanism. First, after the provided examples have been identified in the knowledge base, combining parts of them to form the final answers is a combinatorial problem that requires similar structure identification. In the case of graphs, similar structures are often identified through isomorphic structure discovery [MLVP16a, JKL+15, CZY13]. This makes the task exponential, since all the possible combinations of all the structures similar to those provided by the user will have to be considered. Finally, the very nature of the graph data and their size, poses some additional performance challenges. In certain cases, intermediate results may reach the tens of millions of graphs when the technique is applied to some real worlds knowledge bases.

To cope with the above issues, we have developed efficient algorithms that selectively construct the solution by limiting the number of isomorphic searches to be performed. Since the complete result-set may be too large to be consumed by the user and still too costly to compute, we developed a top-k solution based on a general family of ranking functions that takes into account weights on the nodes of the graph. We also study approximate techniques able to reduce query-time with a limited loss in the completeness of the set of retrieved answers. The efficiency and effectiveness of our solution at scale is demonstrated with a set of extensive experiments on large real world information graphs

1.2 Exploratory Query Suggestion

After defining the exemplar query paradigm and its extension to multiple examples, we need to allow users to easily formulate such type of queries. Existing approaches assume the user to be able to fully specify such a query graph (or query graphs) [ZH10, CZY13, MLVP16a, JGK+14, LMVP18]. This is not always true. In information retrieval, it is not always the case that the user is able to completely define a specific item of interest [WR09, WKW+10]. Users are often looking for expanding their knowledge within a specific knowledge domain by expanding the search around some initial items of interest that are known to them [WR09, WKW+10, SVJ14]. These needs have usually a known starting point, but not a well-defined end-goal. These are the exploratory search tasks. [MBL15, WKW+10].

Nonetheless, existing graph-query techniques either lack any interactive (explore-and-refine) set-up that would more easily accommodate exploratory needs [MLVP16a, JGK+14], or limit the task to subsets of similar entities while completely disregarding connections and related objects [SCSS15, ZCC+17]. One way to assist users in this task is interactive query expansion (IQE), where the system suggests possible related concepts and the user indicates which should be added to their query. Unfortunately, there is no study that provides a principled approach for interactive graph-query expansion.

1.2.1 Interactive Graph Query Expansion

The second contribution of this work is the study of various methods to provide IQE functionality to knowledge graph search engines. The end goal is to help the user build

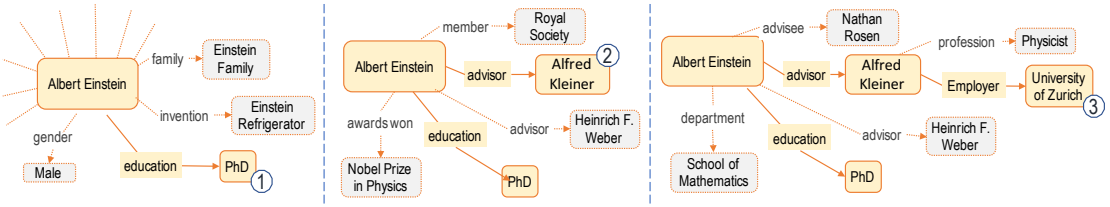

Figure 1.2: Composing a graph query: "*Einstein academic Education*".

a graph-query, through an expansion/refinement process.

The task of query-expansion requires a way to assess the likelihood a candidate expansion represents the actual user need. Graph query suggestions have not been studied formally so far and traditional IR approaches have not been applied in this domain. While, for the case of document search, word co-occurrence has been successfully exploited as an indicator that two keywords are related, this notion cannot be directly adopted for knowledge bases, since they are not collections of graphs within which to search, but are typically consisting of a single large and complex structure (graph). Thus, how to model the problem of graph-query expansion, and how effectively the adopted approach for keyword-query suggestion could be translated into this domain, are two main issues that are tackled in this study.

We developed a novel approach on how to propose query expansions, given an initial graph-query. Graph-query expansions have the form of additional edges and entities that the user can decide to add to their current query. Such suggestions, especially for the case of exploratory navigation, can represent the information the user was looking for, helping the user enrich their knowledge and understand of the domain. Alternatively, they can be used as actual graph-queries to retrieve other similar situations from the knowledge graph, applying traditional graph-search and the *exemplar query* methods described in this dissertation [ZH10, CZY13, MLVP16a, LMVP18, JGK+14].

1.2.2 Pseudo-Relevance Feedback for Graph Search

This work studies different methods of implementing such a graph-query suggestion system and report on their effectiveness. In particular, we propose methods that are grounded in the theory of pseudo-relevance feedback and language-models [CNGR08, PC98]. Such model for a graph query, and its results, represents the user need by means

of the structural information described by those graphs, i.e., their edge labels. Graph-query expansions are then ranked by the model that estimates the likelihood of their appearance in the user query.

This work considers an exploratory use case that is typical of a novice user tapping into the rich information of a knowledge graph, and trying to find insights for specific topics of which with they have limited familiarity.

For instance, to search for information about the education of famous scientists, the user could search for entities similar to Einstein. Yet, this search would have produced Nobel Prize winners, inventors, or physicists. To obtain information about their education, it is necessary to specify in the query that Alfred Kleiner was Einstein's advisor and that Einstein himself had a PhD. As mentioned earlier, an exemplar query search [MLVP16a, JGK+14, CZY13] will then retrieve all other similar situations that include those aspects. A student not familiar with the topic will be oblivious to these details; they would only know Einstein's name and the fact that Einstein is a good subject for their search.

In our example, the user starts with the query "Albert Einstein". The system will then suggest possible additional information, e.g., the facts that he invented the "Einstein Refrigerator", or that he had a PhD. The user will then select the most suitable suggestion among the options, e.g., "education-PhD", as depicted in Figure 1.2(left).

The system will respond with a new set of suggestions to expand the query even further, with the suggestions of each subsequent step of the interaction being more focused towards the direction that the user wants to move. For example, after the user selects "education-PhD", the system proposes more options related to education and academics (i.e., advisors, awards, etc.), as shown in Figure 1.2(middle); finally, after the user selects "advisor-Alfred Kleiner", the system focuses more on advisors and their affiliations, allowing the user to complete the query by selecting "employer-University of Zurich", depicted in Figure 1.2(right). Therefore, the system helps the user to quickly arrive at the intended query. Without such a system, the user would be forced to a laborious and cumbersome search within the entire list of available edges, which may be hundreds or thousands.

1.3 Evaluating Graph Data Management Systems

As we have seen, graph data has become increasingly important nowadays, since it can model knowledge graphs [LMP+15, SKW07], but also transportation networks, social networks, biological graphs, and many others [GSSZ15]. As the graph datasets are becoming larger and larger, so does the need for their efficient and effective management, analysis, and exploitation. This has led to the development of many graph data management and processing systems. Hence, to assure effective access to information graphs, and graph data in general, it is of paramount importance to understand the capabilities of those systems that are currently used for their storage.

There are two kinds of systems that handle graph data (Figure 1.3). One is the *graph processing systems* [LBG+12, MAB+10, HDA+14, LCYW14, CHI+15, FRP15]. They are systems that analyze graphs with the goal of discovering characteristic properties in their structures, e.g., average degree of connectivity, density, or modularity. They also perform batch analytics at large-scale that implement a number of computationally expensive graph algorithms like PageRank [PBMW], SVD[DFK+04], strongly connected component identification [Tar72], core identification [CKCO11], and others. Examples in this category include systems like Giraph, GraphLab, Graph Engine, and GraphX [YBT+16]. The second kind of graph management systems comprises the *graph databases*, or GDB for short [AG08]. Graph Databases focus on storage and querying tasks where the priority is the high-throughput interrogations of the data, and the execution of transactional operations. Originally, they were implemented by exploiting specialized schemas on relational systems. As the sizes of the graphs was becoming larger and more complex, it became apparent that more dedicated systems were needed. This gave rise to a whole new wave of graph databases, that include Neo4j [neo], OrientDB [ori], Sparksee [spa] (formerly known as DEX), Titan[tit] (now superseded by Janus[jan]), ArangoDB[ara] and BlazeGraph [bla].

The two aforementioned categories cover two very different requirements. To make this distinction clear, graph processing systems, can, in some sense, be seen as the parallel of the OLAP systems in the graph world, while graph databases as the parallel of the OLTP systems. Graph processing systems have received considerable attention both in terms of research and also in terms of the evaluation methodologies and frameworks

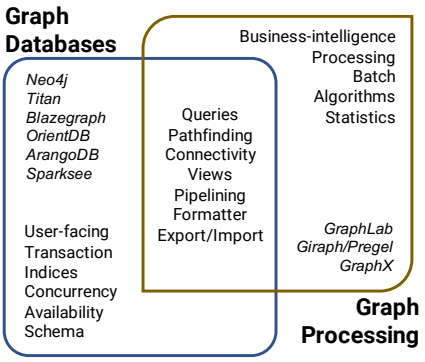

Figure 1.3: Overview of the distinction between Graph Databases and Graph Processing systems

that have been designed [LBG+12, MAB+10, FRP15, HDA+14]. Instead, the work on graph databases lags far behind. The NoSQL movement has triggered an acceleration on the development of graph databases, but their requirements and functionalities are yet not fully understood or commonly agreed. To cope with this problem there is a need for effective evaluation methodologies. Graph databases are of great importance for the implementation of efficient graph query systems and graph exploration systems. For this reason, graph databases (not graph processing systems) and their evaluation are at the centre of the third contribution of this dissertation.

1.3.1 Comparing Graph Databases

The third contribution of this work is both a novel evaluation methodology that complement existing approaches, and a report of useful insights on the performance of the existing graph databases and the effectiveness of their respective design choices.

Given the increased popularity that graph databases are enjoying, there is a need for comparative evaluations of their available options. Such evaluations are critically important for practitioners in order to better understand both the capabilities and limitations of each system, as well as the conditions under which perform well, so that they can choose the system that better fits the task at hand. A comparative study is also important for researchers, since they can find where they should invest their future studies. Last but not least, it is of great value for the developers, since it gives them an idea of how graph data management systems compare to the competitors and what parts of their systems need improvement. There is already a number of experimental comparisons on graph databases [DSUBGVn+10, JV13, KSM13], but they do not provide the

kind of complete and exhaustive study needed. They test a limited number of features (i.e., queries), which provide only a partial understanding of each system. Furthermore, existing studies do not perform experiments at large scale, but make prediction on how the systems will perform based on tests on smaller sizes. Apart from the fact that they tend to provide contradictory conclusions, when we performed the experiments at larger scale, results were highly different from those they had predicted. Finally, many of the tests performed are either constrained to a specific use-case or comprise only some arbitrary settings, to a point that it is not easy to interpret the results and identify the exact limitations of each system.

Given the above motivations, in this work we provide a complete and systematic evaluation of the state-of-the-art graph database systems. Our approach is based not only on the previous works of this kind, but also on the principles that are followed when designing benchmarks [DSUBGVn$^+$10, JV13, KSM13, ATV08, IRV13].

1.3.2 A Micro-Benchmark Methodology

In the design of the tests, we follow a microbenchmark model [BD84]. Instead of considering elaborate situations, we have identified primitive operators and we designed tests that provide a clear understanding of each such elementary operator.

We show how the results produced by complex queries are ambiguous and limited in providing a clear picture on the advantages of each system. So, instead of considering the complex queries, we consider and evaluate a set of primitive operators. The primitive operators are created by decomposing the queries in benchmarks like the LDBC [EALP$^+$15], queries in the literature, or queries in real application scenarios. The use of primitive operators makes clear the specific parts of a system that are limited. Complex tasks can be typically decomposed into combinations of basic steps, thus, their performance can be explained by the performance of the components implementing the primitive operators. Moreover, primitive operators are often implemented by opaque components in the system, therefore, by identifying the underperformed operators we can pinpoint the exact system components that underperform. Query optimizers may change the order of the basic operators, or select among different implementations, but bottom line, there will always be the primitive operators. This evaluation model is similar to the principles that have been successfully followed in the design of many

benchmarks in other areas [DSUBGVn$^+$10, JV13, KSM13, ATV08, IRV13]. Note that we do not intend to replace complex query-based benchmarks. They are important to evaluate optimizers, caches, and other higher-level components. What we intend is to complement these benchmarks for more fine-grained evaluations.

Based on an extensive study of the literature, we have made an effort to cover all the scenarios that have so far been identified. As result, we scrupulously test all the types of insert-select-update-delete queries considered so far, with special attention to the various use-cases, and extend such tests to cover the whole spectrum of tasks, data-types and scale.

As an indication of the extent of our work, we test 35 classes of operations with single queries and batch workloads as well (for a total of about 70 different tests) as opposed to 4-13 that existing studies have done, and we scale our experiments up to 28M nodes/ 31M edges, as opposed to the 250K nodes/2.2M edges of existing works.

Chapter 2

State of the art

In this chapter we provide an analysis of the different aspects that relate our work with exploratory search and query by example techniques. In particular we first present an overview of by example methods that have been successfully employed in different tasks for exploratory graph search (Section 2.1). Then we focus on the main differences between our multi-exemplar approach and the other existing methods for exemplar query answering (Section 2.2). On the topic of exploratory search and graph query suggestion, we present an analysis of methods related to interactive query expansion and interactive exploratory tasks in general (Section 2.3). Finally, we survey current studies on evaluations of graph data management systems (Section 2.4).

Through this survey we aim to (1) motivate the importance of the study of advanced methods for graph exploration and (2) to identify the gaps in the literature that prompted us with the need for both a study of more flexible and expressive graph query paradigms, and need for a more exhaustive experimental methodology for comparing graph databases.

2.1 By Example Methods for Graph Search

In this section we survey existing techniques for identifying elements and structures of interest within a large graph. First we review methods to identify important nodes based on how they are connected with an initial set of seed nodes. Such methods work for generic unlabeled undirected graphs, or for graphs where nodes are annotated with a set of features or attributes. Then, moving to a richer model, we discuss approaches

designed for identifying entities of interest in a knowledge graph. In this second case, entity properties and types determine when two elements are similar.

Finally, we look into methods designed to retrieve more complex structures. The first set of techniques try to reverse-engineer some form of graph-queries based on the user input. All structures matching the query are then returned as answers. The second set of techniques, instead, allows the user to provide a specification of structures of interest in the graph, i.e., subgraphs, and we present solutions to identify all other subgraphs with the same structure. When this is not possible, or too time-consuming, only the top-k results will be produced.

2.1.1 Search by Example Nodes

The first family of search tasks on graphs has nodes as the object of the search. We start with the simpler model of a network and consider unlabeled graphs. In these cases the user input is a set of nodes that are known to be relevant, and the output is a subset of the network that comprises both the nodes of interest, as well as additional nodes that are similarly related, or explain the connections among the former. We can think of a subset of people in a social network, where edges describe mutual-friendship relationships, and the output are members of the same community. Hence, the only aspect that we can study to discriminate among them, is how closely they are connected. For the case of attributed graphs, similarity among node attributes can be exploited in order to prioritize among connections. While for richer models, that is the case of RDF graphs (or in general information graphs), node attributes and edge types help discriminate among entities that are involved in similar relationships.

Seed set expansion. In one work [KK14], the query nodes are also called *seed nodes*, and the answer nodes are member of the same cluster or community C. The problem is hence identified with the name of *seed set expansion*.

In their study, they survey a number of methods for this task and evaluate their effectiveness and trade-offs. Hence, equipped with an indicator function that recognizes whether the node v belongs to the community C, basically a ground-truth, given a subset of nodes Q from C, $Q \subseteq C$, they compute the precision and recall for the retrieved set of nodes in C'.

Within a network, the intuitive definition of a community is a portion of the graph where nodes have much more connections within it than outside. Following this intuition, it comes as no surprise that the most efficient algorithm to retrieve the members of a community given some seed members is based on the computation of a Personalized Page-Rank (for instance as in [JW03]) [KK14]. As a matter of fact, the result of the page rank computation estimates with which probability a random walker starting from any of the seed nodes will end on any other node in the graph. Hence, the computation captures the following idea: if the walker is visiting a member of the community at time t, when transitioning at time $t+1$ it will more likely end on another member of the same community.

Center-Piece Subgraph search. Yet, communities may be extremely large, so retrieving the complete list of all its members may be superfluous or not really practical. A human could never browse or consider all the members. On the other hand, communities usually have a leader, or a small set of central and influential figures. Following this intuition, other works [TF06, GMU15, RBGS$^+$15] address a different problem: given a set of seed nodes, retrieve a connected portion of the graph that explains in some way how the query nodes are connected. In this way, instead of retrieving all the members of a community, they retrieve a smaller subset of characteristic nodes.

The first problem in this area is described as the problem of *Center-Piece Subgraph search* [TF06], where, given a set of query nodes Q and a graph G, the task is to determine a set of nodes $N_{\mathcal{H}}$ that identifies a connected subgraph $\mathcal{H}$ which contains all (or almost all) the nodes in Q and for which the nodes in such subgraph maximizes a *goodness* function $g(\mathcal{H})$. It follows that, depending on the definition of *goodness* employed, different set of nodes will be retrieved. The concept of center-piece subgraph search has been applied, for instance, to identify hubs or influencer in a subnetwork, and to explain why a group of nodes are related one to the other.

Here, a goodness function $g(N_{\mathcal{H}})$ is defined over each node $v_i \in N_{\mathcal{H}}$ based on the nodes in N_Q such that $g(N_{\mathcal{H}}) = \sum_{v_i \in N_{\mathcal{H}}} g(v_i) = \sum_{v_i \in N_{\mathcal{H}}} r(N_Q, v_i)$, for some function r that provides a score for each single node. This means that the goodness of each node in the answer is evaluated against the entire set of query nodes. Moreover, they require for the resulting graph to be connected, although they do not require for it to be fully connected.

In particular, they introduce the *K_softAND* property, where nodes in the output graph are only required to have connections to at least k of the query nodes (with $k \leq |N_Q|$).

The solution proposed to find the Center-Piece Subgraph is composed by three steps ([TF06]):

1. **Individual score calculation:** for each node $q_i \in N_Q$ and $v_j \in N_G$, compute $r(q_i, v_j)$;
2. **Individual score aggregation:**, combine all scores $r(q_i, v_j)$, for a given v_j, in order to obtain $r(N_Q, v_j)$;
3. **CEPS extraction:** extract the connected subgraph $\mathcal{H}$ which maximizes $g(N_\mathcal{H})$, computed through each $r(N_Q, v_j)$ score.

At this point, we note that there is no implied limit enforced by the goodness function g. Yet, extracting the CEPS is hardly useful if the resulting graph $\mathcal{H}$ is extremely large. For this reason, an extra condition is added [TF06]. Namely, they introduce a budget b that limits the maximum size of nodes in $\mathcal{H}$, i.e., the extra condition requires that $|N_\mathcal{H}| \leq b$.

The other important element to define is the scoring function $r(q_i, v_j)$ and consequently $r(N_Q, v_j)$. The score $r(N_Q, v_j)$ is actually computed based on the result of a personalized page rank computation [TF06], which, as a choice, is consistent with the result seen above [KK14]. In particular, in the case of the *K_softAND* formulation, $r(N_Q, v_j)$ is actually computed as the meeting probability of k particles $r(N_Q, k, v_j)$, i.e., the steady-state probability that at least k of N_Q particles of a random walk from the query nodes N_Q, all find themselves at node v_j. When all scores have been computed, a dynamic programming algorithm starts from the query nodes and try to connect those with other destination nodes based on their score. This algorithm effectively grows the target graph $\mathcal{H}$ until the desired budget b has been allocated.

The Wiener Connector. Among the approaches that try to explain the connection between a set of nodes, one very effective solution is to find the *minimum Weiner connector* of a set of nodes. This approach takes the name from the *Wiener index* (introduced by [Wie47]) which measures the *compactness* of a graph as the sum of all pairwise shortest-path distances between its vertices. The problem is proved to be **NP**-hard [RBGS+15].

This approach tries to optimize the inter-node distance between each pair of members, and as byproduct this formulation tends to limit the size of the subgraph identified as solution. In contrast, other methods (like those seen above) either return very large subgraphs, since they reconstruct the entire community, do not guarantee the solution to be fully connected, or tend to be slower because of the cost of the various random walks that have to be computed. Moreover, they usually require a user-defined budget for the size of the solution retrieved, while the Minimum Wiener Connector allows a parameter-free solution.

Although the problem of finding a small connected subgraph that brings together all the query nodes may resemble the intuition behind the Steiner Tree [HRW92], the latter does not provide, in general, a good solution for the former [RBGS$^+$15]. This is because, being the Steiner Tree a tree, it may cause the solution to discard edges that are important to describe the centrality of some of the query nodes. Yet, it exists a constant-factor approximation algorithm for the solution of the problem [RBGS$^+$15], and such solution is actually obtained by the summarization of the graph by means of multiple steiner trees.

In particular, in their solution, each query node is, in turn, considered as a candidate root node for a minimum-weight Steiner Tree. Weights on the edges of the tree are computed w.r.t. the distance between the chosen root and each other node. With this approximation strategy, the algorithm finds solutions that are quite near to the optimal in few minutes for graphs with millions of nodes [RBGS$^+$15]. This means that it is not suitable for on-line applications, but could be efficiently employed for both business analytics use-cases, or precomputed solutions for recommending systems.

Local discrepancy maximization. Another work [GMU15] studies the idea of finding regions of the graph that explain the connections of some restricted set of query nodes N_Q. In particular, the problem tackled by this work is called *local discrepancy maximization*, or "bump hunting", and describes the task of retrieving, given as a set of query nodes that exhibit a certain property of interest, a connected subgraph where such nodes (or a subset) appear more often compared to non-query nodes.

Nevertheless, here the set of input nodes is assumed to be possibly very large, and the output instead should focus only on a smaller portion, where those are more dense

(the "bump"). In this sense, this work does not completely adhere to the example-based query paradigm, since not all the input examples are actually part of the desired output. Yet, one feature that makes this work particularly relevant for graph-search, and in particular node-based search and exploration, is its ability to include in the output nodes that where not part of the input and that are relevant because of their connection with the input. In this sense, the local discrepancy identifies the other nodes that are relevant to the information need subsumed by the user input, despite not being part of it. Moreover, the possibility to actually discard some of the input nodes can prove particularly useful for approximate-search.

The linear discrepancy maximization problem is proved to be **NP**-hard in general. Yet, it is demonstrated [GMU15] that when the input graph is actually a tree, the solution can be found in linear time, i.e., $O(|G|)$. Hence, for the general case, their approach is heuristic, and they solve the problem by reducing the graph to a tree (they study various alternatives for this step), producing in this way a reduced input graph, and then applying the linear-time algorithm on such restricted search space. Yet, those solutions all require multiple scans of the entire graph, which initially is also assumed to completely reside into main memory. This solution may be too costly for very large graphs, and for this reason a different settings is also taken in consideration in their work.

To take into account the case where the entire graph G is not provided, they study a *local-access model*. In such model, the graph is not completely accessible as input, instead only the query nodes N_Q are given, while the rest of the graph can be accessed through a *neighbor function*. This function takes as input a single node and, by querying the graph, obtains the list of all other adjacency nodes.

In these settings, the key-element for an efficient solution is to invoke the neighbor function only a limited number of times. To this end, three different strategies are studied [GMU15], all with a common element: to iteratively build a smaller subgraph G' of G by expanding the query nodes (or only some of those) in N_Q and deciding then when to stop expanding. The stopping criterion is based on the proportion between the number of query nodes, the size of the currently loaded graph G', and the distance between the candidate node to expand and the nearest query node. Of the three expansion techniques studied, two have a fixed stopping criterion, while the third (and most

effective in general) implements an adaptive expansion strategy, which at every iteration computes the upper-bound of the optimal solution scores, and based on the result of that computation decides whether to proceed with the expansion or to stop and proceed looking for a solution. An important outcome of the experimental evaluation is that, for sparse graphs, even a fixed stopping criterion performs exceptionally well both in terms of running time and quality of the output.

Focused clustering. Up to this point, we have considered problems that retrieve nodes whose relevance is based on their connectivity and respective closeness. In some cases, the above algorithms may take into consideration also edge-weights as a predefined and static (hence, not query-dependent) information about the strength or cost of the edges. Yet, all these solutions retrieve nodes considering only the topology of the graph.

On the other hand, many real-world graphs are provided with richer information in the form of node attributes. Consider, for instance, a social network where nodes are people connected by friendship relationships, and for each person the network store a profiles as a set of node-attributes (e.g., name, demographics, or preferences). In this model, node-attributes are represented as *feature-vectors*, and while the user input is still a set of query nodes N_Q as in the previous problems, in this case, the query nodes provide an additional information carried by the values of the feature-vector of the query nodes themselves.

In this case, the task is to infer attribute weights, which represent an implicit measure of the user preference towards some of the node attributes [PAISM14]. Equipped with this information, they consider that, when considering the connectivity among nodes, not all connections have the same importance or strength. In these settings, the problem proposed is called *Focused clustering*.

Additionally, as a result of comparing members in each cluster, they solve a second complementary problem: *focused outlier detection*, i.e., they identify in each cluster those nodes that deviate from the other nodes in the same cluster for some of their attribute values.

The approach to find both clusters and outliers, is composed by three steps [PAISM14]:

1. **Attribute weights inference:** this translates to an instantiation of the distance learning problem [XJRN03], where the goal is to find an appropriate distance

function for computing the distance between two feature vectors f_i, f_j, for nodes $v_i, v_j \in N$, such that given two nodes their distance is smaller if they both are in the same cluster C that if one of the two laid outside it. The outcome of this step is the weighting function ω_F;

2. **Focused clusters extraction:** this steps extracts clusters of nodes with high connectivity and also high feature-value coherence, i.e., those that form community of nodes with almost the same values for those attributes with high weight in ω_F. In this step, the edges in the graph are re-weighted according to the scores of ω_F, and only connected components with high-score weights are kept. These are the cores of the final cluster, with an iterative process those cores are then progressively expanded. During expansion the weighted conductance [ACL06] of each cluster is computed, and the expansion proceeds until no further improvement can be gained.

3. **Outlier detection:** in this process outliers are retrieved while computing clusters. In particular outliers are those nodes that have a high un-weighted conductance, but a low weighted conductance.

Up to this point, we have seen how we can retrieve nodes that are relevant to the user, either because they are member of a community of interest, or because they are important connectors that characterize the proximity of the exemplar nodes. All these approaches can retrieve answers based on simple topological informations, and in some cases they can exploit slightly richer representations with node-attributes.

We can see here a common pattern employed in order to retrieve good solution in reasonable time, i.e., to iteratively expand around the query nodes, with the goal of avoiding entire scans of the whole graph.

In the following we are going to consider instead more expressive data-models, where the edges carry important information as edge-labels, and queries then are characterized as richer edge-labeled structures.

2.1.2 Similar Entity Search in Information Graphs

In the previous section we described, among others, the idea of *focused clustering*, where the discovery of answer nodes was guided not only by their connectivity, but also, and with equal importance, by their attributes.

In this section, this same intuition is considered in a different form for the task of entity search for knowledge graphs [MSS13, SCSS15]. As mentioned earlier, knowledge graphs are special directed edge-labeled graphs, where information about entities and concepts is represented as subject-predicate-object triples (as in the RDF data model). In this data model, an entity can represent real-world entities, like a person, a place or an abstract concept, e.g., *Emma Watson*, or *Paris*. Additional information is represented as edges between nodes. In this way, for each triple, the nodes represent subjects and objects, while predicates are translated to edge labels. These structures are part of the so-called *Fact Graph*. Moreover, within the same graph, ontological information is also stored as an *Ontology Tree*, e.g., *Emma Watson* **is an** *Actor*, and an *Actor* **is an** *Artist*. Within a knowledge graph, given an entity e, e.g., *Emma Watson*, all the facts that are connected to it, i.e., both edges in the fact graph or in the ontology tree, are called its aspects $A(e)$.

In this context, two works [MSS13, SCSS15] describe the problem of *query by entity examples*, also called *aspect based entity retrieval*.

Consequently, in this formulation [MSS13], the problem is to identify a set of aspects that are common to all entities in the query, and that can retrieve new entities that are not present in it. In the graph terminology, the aspects of an entity are all the edges incident to it. They also distinguish the set of *basic aspects*, which are the type of edges that are incident to the entity [MSS13]. For instance, **was born in** *Paris* is an aspect, while **being born** (with no reference to the place) is a basic aspect. Moreover, through each aspect a they identify the set of entities that share that same aspect, called the entity set of a, e.g., the set of all the entities that are incident to an edge for **was born in** *Paris*.

Intuitively, some of the entities in the query have some aspects that are not shared by some other user-provided entities, and as such we assume they are not relevant for the user. Consequently, we need to consider only aspects that are common to all the entities.

At the same time, if we keep *all* the aspects that are common to *all* the entities in the query, we could end up considering a set of aspects which is too restrictive and does not identify any other entity that is not already known to the user. Hence, the solution is to retrieve a set of aspects which contains as many aspects as possible, and, a the same time, which includes at least one entity that is not in N_Q, this is called a *maximal aspect set* [MSS13].

To retrieve such maximal aspects, first they retrieve the set of *basic aspects*, then the basic aspects values shared by all the other entities is considered. A special role is given to aspects related with the ontology tree, i.e., the types. Each set of maximal aspects should contain at least one type, and among all the types, they first consider those that are more specific, i.e., those that are lower in the in the ontology tree (e.g., actor instead of artist).

For a group of entities, there are usually multiple disjoint sets of aspects that satisfy the maximal condition. Hence, it is important to discriminate among the possible alternatives. To this end, they also perform a task called *aspect ranking*, where each aspect is assigned a score based on its selectivity (i.e., how frequent or infrequent it is in the knowledge graph), and based on some popularity score of the entities involved [MSS13].

Their approach is implemented in the QBEES framework [MSS13], and later extended in the iQbees framework [SCSS15]. In their extension, they enable an interactive query mechanism. In practice the user is able to provide a initial query set, and later refine it by adding additional entities, so that the set of maximal aspects is refined at every interaction.

2.1.3 Reverse Engineering Queries on Graphs

Up to this point we presented solutions to find nodes when examples where other nodes that were strictly connected/related or similar in terms of attributes and aspects.

In the following we move our attention to specific sub-structures of the graph. In particular these methods can reverse-engineer specific types of graph-queries, namely: path queries, and SPARQL queries.

Learning Path Queries on Graphs. In the first work we study [BCL15], the user need can be represented by a *path query*. Those are queries that can be described by a regular expression on edge labels. Consider for example the query `(child_of|married_with)`* `+acted_in`[1], here we consider actors, or people related to actors (in our example children or spouse) and their movies.

These queries are quite flexible and enjoy vast applicability, see for instance [BB13] and [Woo12]. Yet, they are not so easy to be expressed by a novice user, or by somebody that is not aware of the structure of the knowledge graph. To overcome this limitation, they allow the user to express their need by presenting some nodes in the form of positive and negative examples [BCL15]. In practice, continuing with our example, the user may provide a set of actors or people related to actors alongside some movies as positive examples, while it might present some singers and songs as examples of entities that are not of interest.

The task is then to derive a path query that is able to (i) generate paths covered by all the positive examples, and (ii) none of the negative ones. The example above present a case where the query specifies both starting and ending nodes. Nonetheless, the problem studied is even more generic, and allows the user to specify just one end of the path of interest, e.g., the starting node [BCL15]. In this formulation, this approach can be also used for similarity entity search, as described in the previous section, but for cases in which simple aspects are not sufficient to describe why the entities are relevant.

As mentioned earlier, path queries are based on the concept of regular expressions. For example, the query $(a|b)^* \cdot c$ generate a language containing words like c, ac, $aabc$ and so on.

A path is obtained from a sequence of edges by considering the edge-labels encountered. For instance, given the subgraph represented by *Emma Watson* **born in** *Paris* **located in** *France*, we obtain the path **born in** $\cdot$ **located in**. The path matches the path query q, if it is member of the language $L(q)$ generated by it. Then, given a node $v \in N_G$ we identify all the paths that originate from it with $paths(v)$. In their work, paths are generated with a directed traversal of the graph starting from the designated node.

[1] the symbol | is the disjunction, $*$ is the equivalent of the Kleene star, and + is the concatenation symbol.

Moreover, if a directed loop is encountered, we generate an infinite series of paths, hence the result is that the set $paths(v)$ is actually an infinite set.

Given the definition above, the answer to a path query q on the graph G is then the set of nodes which are source of at least one path that is member of the language generated by the query, i.e., $q(G) = \{v \in G | L(q) \cap paths(v) \neq \emptyset\}$.

As we mentioned, the user is providing two sets of examples, positive and negative examples [BCL15]. In general, they indicate the pair made with the set of positive and negative examples as the *sample* $S : \langle S^+, S^- \rangle$ for the query.

In their study [BCL15], they first define when a sample (a set of positive and negative examples) is consistent, i.e., doesn't contradict itself, and as such allow for a solution to exist. Then, they describe the problem of actually determining whether a generic input is consistent, and they demonstrate that, in general, the problem is **PSPACE**-complete. Finally, they provide both a characterization of what class of queries are actually learnable, and the proof that the problem of learning generic graph pattern queries is **NP**-complete.

To overcome the complexity of the problem, they characterize a class of queries which is *learnable with abstain*. Namely, they describe a learning algorithm that, always in polynomial time, returns either a query consistent with the sample or a special *null* value that indicates that not enough examples are provided in order to return an answer or that such query does not exists. In particular, the algorithm provides both the following guarantees.

1. **Soundess with Abstain:** For every graph G, and sample $S : \{S^+, S^-\}$, the algorithm returns either a query in $q \in Q$ that is consistent with the examples, or null if no such query exists or it cannot be constructed efficiently.

2. **Completeness:** For every query q there exists a polynomially-sized characteristic sample $CS : \{S^+_{CS}, S^-_{CS}\}$ on G, s.t. for every other sample $\bar{S}$ extending CS consistently with q (i.e., $S^+_{CS} \subseteq \bar{S}^+, S^-_{CS} \subseteq \bar{S}^-$) the algorithm on input G and $\bar{S}$ returns q.

These guarantees allow the algorithm to return very quickly, either the correct answer, or the possibility to the user to extend the initial sample when the provided input is not

sufficient to identify efficiently the correct answer, enabling in this way an interactive query-discovery process.

To obtain answers in polynomial time, they fix a maximum path length k, which can be decided based on common practical cases and according with the structure of the graph. Given the maximum length k, the algorithm enumerates all paths originated from the positive examples S^+, and subtracts from such set all those that are generated by the negative samples S^-. The paths obtained in this step are called *smallest consistent paths* (SCP). Yet, those use only disjunction and concatenation. To exploit the full expressibility of the language, a *generalization step* is performed. This steps compacts queries and is able to introduce the use of the kleene star. This process builds first a Prefix-Tree Acceptor (PTA) [DlH10] for the language generated by the SCP, then it compacts it into a Deterministic Finite-state Automaton (DFA).

The process is demonstrated to generate queries that satisfy the user need. Moreover, in their work [BCL15], as a result of the completeness property of the algorithm, they provide a framework for an interactive query-learning process. In particular, they describe an active process, where the system poses to the users a question regarding specific nodes to be labeled as positive or negative examples. The challenge is then in minimizing the number of nodes to present to the user for labeling. The solution they devise identifies a set of nodes as *informative*, i.e., nodes that generate at least one path that is not generated by any node among the negative examples.

Up to now, in this chapter, this last work [BCL15] and the previous iQbees system [SCSS15], are the only two approaches that specifically target an interactive use-case.

Reverse Engineering SPARQL Queries. The last work we cover in this section regards the SPARQL query language [GFMPdlF11, PAG09]. The SPARQL query language is specifically designed as web standard by the W3C (World Wide Web Consortium)[2] for RDF datasets. Informally, given a RDF dataset as a set of RDF subject-predicate-object (s-p-o) triples, a SPARQL query has the form of template matching query, which is composed by s-p-o triples itself, but where some subject, objects or predicates are replaced by variables. Triples in a SPARQL query may be joined by **AND** conjunctions, or enriched by **FILTER** statements where conditions are posed on top of the variables defined in some of the triples. For instance, the query that selects

[2] `https://www.w3.org/TR/2004/REC-rdf-primer-20040210/`

actors *born in Paris*, which acted in some movie is `(?X, born_in, ‘‘Paris’’)` **AND** `(?X, acted_in ,?Y)`, with `?X` and `?Y` as symbols for variables. A filter condition could be of the form `(?X, age, ?A) FILTER ?A > 18`

Answers to such queries are all the variable assignments that are satisfied in the dataset by some set of triples. The SPARQL query language adopts also an additional **OPT** operator, which retrieves a triple if it exists, but is not mandatory for an answer to be able to satisfy such condition. For instance, we could try to retrieve also the child of an actor by rewriting the query above as `(?X born_in ‘‘Paris’’)` **AND** `(?X acted_in ?Y)` **OPT** `(?X has_child ?W)`. The result set of this last query is a superset of the previous query, since we retrieve all the possible actors born in Paris with their movie, but we also retrieve their children, if they exists, for those that have some, and without excluding those that do not have any.

The SPARQL query language is the language of choice for semantic web applications, but despite its widespread use, it still poses an important challenge to write SPARQL queries for users that are unfamiliar with it or with the graph they are querying [GFMPdlF11]. To provide a more effective way to pose SPARQL queries, two works[ADK16, DAB16b] propose an approach that is similar in spirit to the original Query-by-Example paradigm of [Zlo75]. Namely, given a set of example mappings, the system retrieve one or more SPARQL queries, that produce such mappings as subsets of the answer.

In particular, in the *SPARQLbyE* system, the user is asked to provide only a set of candidate variable mappings, i.e., $e1$:`?X`$\mapsto$“Emma Watson”, `?Y`$\mapsto$ “Paris”, and $e2$:`?X`$\mapsto$“Jim Carrey”. Moreover, the user is allowed to avoid specifying a mapping for some of the variables in some of the examples, e.g., without specifying a place of birth for “Jim Carrey” if they do not know it. This leaves room for the cases in which the user only knows partial examples, but also for **OPT** statements to be produced.

The task of reverse engineering SPARQL queries is in general computationally hard, although there are some exceptions, depending on whether we allow the full set of language constructs, and depending on what kind of examples we accept as input. In particular, the aforementioned work [ADK16] studies three different problem instantiations, namely, the case in which the user provides only the positive examples, the case where the user also provides negative examples (similar to [BCL15]), and finally the case in which the query should return only the example mappings, and no other triple. Moreover, they

also study the simpler problem of verifying whether a specific query satisfies a specific input.

In general, the work [ADK16] provides a set of proofs that concludes that both the verification and the reverse engineering task is solvable in polynomial time only for queries that use the **AND** operator and nothing else. Yet, they find that a polynomial time solution exists if the query can be restricted in the use of the **OPT** operator, and if the mappings provided present the convenient property called *tree*-like. This property holds when the sets of examples that map each variable can be arranged in a tree-shaped lattice under inclusion.

These results are then exploited to build system that is actually able to work on top of common SPARQL endpoint and allow for this solution to work in practice for common web use ([DAB16b]).

2.1.4 Search by Example Structures

We now survey methods that take in consideration general sub-graph structures of the graph both as input and output. These methods resemble in some way the case of reverse engineering SPARQL queries, but refer to more generic graph search via graph-homomorphism queries. In particular, we approach those cases in which the user need translates to one or more sample *structures* that are considered relevant, and the system performs the necessary computations in order to retrieve all other substructures in the graph that match the input. Moreover, these are the *exemplar-query* paradigm that we adopt and extends in this dissertation.

Graph Query via Entity-Tuples The first of these work is called **GQBE:** "Graph Query by Example" [JKL$^+$15], and which is specifically designed for the case of knowledge graph search. The method is inspired by the original Query by Example ([Zlo75]) and it's formulation is very similar to the previous method for reverse engineering SPARQL queries ([ADK16]).

In this case the user provides one or more *entity tuples*, where each entity tuple is an ordered list of entities in graph, e.g., $\langle$*Emma Watson*, *Paris*, *Harry Potter*$\rangle$. The output is then a set of entity tuples that subsume the same structure, in this case actors, places of birth, and movies.

To solve this kind of problem, there are two main challenges to overcome. The first is to identify which graph structures need to be taken into consideration, or in other words, which paths connect the entities in the tuple, and what other additional nodes not mentioned in the tuple should be taken in consideration. In a sense, there is the need to understand what additional aspects are relevant for the user (as it was happening in [MSS13]). The second task is to retrieve in the graph all other entities that take part in the same type of connections, i.e., to perform a subgraph-isomorphism search.

To understand the information-need behind the entity-tuple provided by the user, the concept of *Query Graph* is introduced [JKL+15], which is a weakly connected subgraph of the knowledge graph that (1) contains all the nodes in the entity tuple, and (2) contains at most m edges. In this case m is a user-provided parameter that limits the size of the query graph, so to avoid it to be the entire knowledge graph, for instance.

Then, the query graph obtained in this way is able to explain how the query entities are related one to the other, and to add also other aspects that might be important to determine what other answers are relevant. Yet, by this formulation, multiple query graphs may exists. Consequently, they first introduce the concept of *Maximal* Query Graph (MQG) [JKL+15]. To find the maximal query graph, the edges on the graph are weighted, and then only the graph that maximize the sum of the edge weights is kept. This definition is still problematic, so a series of relaxations and heuristics are applied. In the end, the final MQG is built via an heuristic approach that runs a number of depth-first visits from each query entity and keeps only the top weighted edges.

Weights are also assigned based on a mixture of scoring formulas that take in consideration the popularity of the edge label, both globally in the entire knowledge graph, and also locally around each query entity.

Finally, answers should be isomorphic-subgraphs of the final MQG obtained in the previous step. Yet, such MQG may be too selective, and as such only few substructures in the graph may be found matching it. For this reason, in the GQBE solution, a set of approximate answers are actually retrieved by generating a lattice of "simpler" query graphs.

In the lattice, the top node is the MQG itself, while all its children at the lower level are obtained by removing one-by-one its edges, but ensuring hat all the query nodes are

still present an that the graph is weakly connected.

The lattice is particularly important for the top-k processing of the query. As a matter of fact, query-answers are ranked based on the edge weights in the query graph, and based on the nodes they share with the entities in the query. When it's time for graph-query-processing then, the lattice is explored bottom-up, from the simpler queries up to the more complex. During this exploration, some nodes in the lattice are pruned either because already covered by some other nodes, or because their total weight cannot possibly surpass the weight of the answers already found.

Another important feature of the MQG computation is that it can accommodate for multiple entity tuples as input, as far as the match the same entity tuple template. In this way, a maximal query graph is generate for each entity tuple, and then those graphs are aligned and merged in one single query graph before building the lattice. In this way it is easier for the system to guess the edges and structures that match the user need.

Queries with Example Subgraphs. We now proceed to explain the last family of techniques that allow to perform search on graphs via examples. This technique is called *exemplar queries* and was introduced with this name by [MLVP14a] as a general concept which is further extended by this dissertation.

The general paradigm, as followed by all other methods presented here, accepts as input an example member of the answer set. The query engine, then, infers the full answer set based on the example and any additional information provided by the underlying database. The answers are all those elements in the database that satisfy a predefined *congruence relation*.

The work on *exemplar queries* [MLVP14a] has been the first to characterize in general terms this query paradigm, and proposed then a practical solution for the case in which the user example is a substructure (a subgraph) of the knowledge graph while providing as answers all other isomorphic subgraphs. Later, this approach has been extended [MLVP16a] in order to allow for a different congruence relation, namely, strong simulation [MCF+14].

The exemplar query work for knowledge graphs is specifically designed for very large datasets. The first contribution of that work is an exact pruning technique similar to many filtering approaches for graph search, called *neighborhood-based pruning*. The core

idea is to represent each node, both in the query and in the graph, with a vector that summarize the edges around it. Fast comparison among node vectors allow for effective pruning of candidate nodes.

In practice, a preprocessing step performs a BFS visit of each node in the graph, and counts at each level the number of edge labels with a specific label. Then, those are stored in compact integer vectors, following a predefined order. Those vectors are quickly computed on-line for the query instead. By comparing a vector for a query node, with the vector for the nodes in the graph it is possible to immediately discard any node that cannot possibly take part in any solution. This approach works both for isomorphism and for strong simulation, with only minimum tweaks in how the comparison is performed.

Yet, even when applying this type of pruning, the number of answers retrieved is usually extremely large, and for this reason hardly processable by the user. Hence, the work studies also an effective scoring and ranking technique that also offers great pruning power. In particular, according to what we have already seen in the beginning of this chapter and in other works, it is assumed that the structures that are located in the graph in the proximity of the user example, are also the most relevant for the user. The approach followed scores the nodes around the query with a personalized page rank computation based on some user-defined scoring threshold [MLVP14a, MLVP16a]. So that only those nodes with score above the threshold are taken into account.

In this dissertation we propose an extension of the exemplar query paradigm to allow for queries that contain multiple examples. In this case, in contrast with all other methods presented up to this point, each one of the examples provided by the user is assumed to present one distinct aspect of the desired result-set. Hence, each example per-se may be incomplete, and it is not even assured that those example will be part of the final answer-set.

2.2 Queries with Multiple-Examples

Our work provides a formal semantics for multi-examples queries moving apart from previous *by-example* methods in two directions: the structure of the results does not need to be known in advance, and the input examples might not be part of the query answer.

By-example methods. As we have seen earlier, the by-example paradigm exploits examples in order to find results without a fully specified query. In this way the user does not provide a list of specifications that the elements of interest should satisfy, rather an example of what such elements should look like. Query-by-Example (QBE) [Zlo75] describes a query interface for relational databases by means of a template tuple, where the attribute values are partially specified by the user. Other approaches [PDCC] accept tuples that should be included in the final desired result-set and the system infers the select-project-join queries that result in such tuples. Variations of this idea include the ability of the user to provide examples that are marked as relevant or irrelevant [DPD14], or tuples alongside explanations [DG16]. Previous work on relational and textual data has no straightforward adaptation to more complex structures such as graphs, and assumes that the provided examples must appear entirely in the answers.

As presented above, by-example works on graphs are divided into entity-based and structure-based. Entity-based approaches, like QBEES [MSS13], take entities (nodes) as examples and return other similar entities. Structured-based approaches take as input more complex examples [MLVP16a, JKL$^+$15]. Exemplar Queries [MLVP16a] define a general paradigm for searching by-example and is applied on knowledge graphs. GQBE [JKL$^+$15] instead considers tuples of entity mentions (such as $\langle$Barack Obama, USA$\rangle$) as input and finds other similar tuples. While in exemplar queries the user is able to provide complex structures, in GQBE only a list of entities forming a path is allowed as query. In this sense GQBE is a sub-problem of exemplar queries with a non-generic ranking function, and no solution for the case where the multiple examples in the input are only partial specifications. In contrast to the by-example works, in our work, the structure of the returned results does not need to be known in advance, and the input examples might not be part of the query answer.

Multiple queries on graphs. There are numerous methods [NW08, NW09] dealing with the optimization of single queries, but not for multiple small queries. A graph query can be seen as a multi-join query on the single edges. This has been considered especially in RDF databases [NW08, NW09]. The main limitation is that they require a fully specified query as input, which is not our case.

Other works consider the case of multiple query optimization in SPARQL queries [LKDL12].

SPARQL queries allow an optional part to be specified to generate queries with different structure. However, while in their case the number of options is limited, here we consider any possible structure combination in the results. The only solution would be to generate beforehand all the combinations as optional SPARQL parts, which is clearly impractical.

Finally, a recent work (PANDA) studies partial topology-based network search [XBCW17]. That is, to find the connections (paths) between structures node-label isomorphic to different user inputs. PANDA first materialize all isomorphic graphs, then groups them into connected components, and finally finds undirected shortest paths among them. Such semantics is mostly related to other solutions trying to find connections among disconnected nodes [KRS+09]. Moreover, as shown in the experiments for our multi-exemplar query solution (Section 3.8), the proposed exact solution does not scale to large graphs.

2.3 Interactive Query Suggestion

One of the focus of this work is on interactive query expansion of graph queries to support users in exploratory tasks.

Query expansion in document search. Query expansion in Information Retrieval (IR) has been studied to enhance the effectiveness of document search by including additional terms in the user's search [Har88, KH11, GXX13, HWZ11, NC10, QF93, CNGR08, BS96]. There exist two different types of query expansion techniques, namely the automatic (AQE) and the interactive query expansion (IQE) [CR12]. Both AQE and IQE concentrate on the disambiguation of the user intent by adding information to the user's query. However, AQE is one-shot approach which automatically includes more terms in the query, while IQE accounts for the user's feedback in the expansion process [CR12, TVFZ07]. As such, IQE aims at regarding the users as active players instead of guessing their intent thus excluding them from the expansion task.

Previous studies [Rut03] demonstrated the effectiveness of IQE compared to AQE when dealing with complex query statements and specific user needs. Our methodology includes the benefits of intuitive query expansions, additional explanations, and user feedback.

Query expansion based on Pseudo-Relevance Feedback. Two common methodologies to retrieve additional terms for query expansion are the Pseudo-Relevance Feedback (PRF) framework [CNGR08, TDH05] and the Language Models [LZ17, PC98, LC01]. The PRF framework defines a prior over the entire document collection and assumes that the documents retrieved at first when the query is issued are implicitly relevant for the user. Therefore, such documents are pseudo-relevant in a way that the user has expressed no explicit preference for them. Instead, the search engine is deemed as accurate. Subsequently, expansions are generated on the terms contained in the pseudo-relevant documents using language models [CNGR08, TDH05].

In particular, expansion terms that maximize the likelihood of being selected by the user are retrieved from the documents [PC98, LZ17, LC01, CNGR08]. Pseudo-Relevance Feedback and language models provide an effective solution for query expansion in documents; however, up to now, there has been no proper adaptation of such models on graphs. The only exception is Graph Relevance Feedback (GRF) [SYS+15], which however is based on explicit feedback on the results of a graph query and does not provide an effective reformulation criteria, nor an effective query-suggestion mechanism.

Therefore, none of the existing studies have managed to apply language models and pseudo-relevance feedback in the context of knowledge graph search. Also, to the best of our knowledge, our work is the first to formally study the case of graph-query expansion as an information-retrieval task.

Exploratory search in knowledge graphs. Exploratory search in knowledge graphs, such as DBPedia, WikiData, and Google Knowledge Vault (based on Freebase) usually empowers document search [SWW+11, ZCCW09, GXX13, PIW10] with richer semantics. A recent study [SVJ14, SVLV16] has shown how effective a knowledge graph is in assisting the user exploring unfamiliar topics and getting information about complex matters. For instance, entity linking allows the disambiguation of entity mentioned in a keyword search by exploiting the knowledge graph structure [HBB15, PIW10, NKZ16, ZCC+17].

As such, various exploratory search paradigms [MLVP16a, CZY13, JGK+14, MSS13, BCMT13, ZCC+17, NKZ16, HBB15, PIW10] have established methods for searching knowledge graphs in a more intuitive way than by formulating queries with declarative languages, such as SPARQL. In particular, in graph query by-example [JGK+14] and

exemplar queries [MLVP16a] the user can specify an example item of interest as a subgraph of the knowledge graph and the algorithm retrieves other similar subgraphs. The main limitation of such approaches is the assumption that the user is able to provide a complete specification of the example, which is unrealistic, especially in the case of exploratory search.

Our approach overcomes this limitation and describes a novel exploration scheme that takes advantage of both pseudo-relevance feedback and exemplar queries, leading to an interactive approach that requires minimal user feedback in order to return relevant answers in a knowledge graph.

2.4 Evaluations of Graph Data Management Systems

Evaluating Graph Processing Systems. There is a great deal of works on evaluating graph processing systems [HDA+14, LCYW14, CHI+15, ZCY+17, MLZ17]. Such systems are designed for computationally expensive algorithms that often require traversing all the graph multiple times to obtain an answer, like triangle counting, page rank, and community detection. Such systems are very different in nature from graph database systems, thus, it comes at no surprise that, in their evaluation, "needle in the haystack" queries like those that are of interest to us in this work are not considered. Of course, there are proposals for unified graph processing and database systems [FRP15], but this idea is in its infancy. Our focus is not on graph processing systems or their functionalities.

Evaluating Graph Databases. In contrast to graph processing systems, graph databases are designed for "needle in the haystack" operations, i.e., queries that identify and retrieve a small part of the data. Existing evaluation works [AG08, Ang12] for such systems are limited in describing the systems in terms of their implementation, data modeling and query capabilities, but provide no experimental evaluation. In particular, one of the earliest works [AG08] surveys graph databases in terms of their internal representation and modeling choices. It compares their different data-structures, formats and query languages, but provides no empirical evidence of their effectiveness. Another work [Ang12] compares 9 different systems and distinguishes them into graph databases and graph stores based on their general features, data modeling capabilities and support

for specific graph query constructs. Unfortunately, not even this work provides any experimental comparison, and like the previous one, it includes systems that have either evolved considerably since then, or have been discontinued.

A different group of studies [DSUBGVn$^+$10, JV13, KSM13] has focused on the empirical comparison of the performance of the systems, but even these studies are limited in terms of completeness, consistency, and currency of the results. The first of such works [DSUBGVn$^+$10] analyzes only 4 systems, two of which are no longer supported. Its experiments are limited both in dataset size as well as in number and type of operations performed. The systems were tested on graphs with at most 1 million nodes, and the operations supported were limited to edge and node insertion, edge-set search based on weights, subgraph search based on 3-hops BFS, and the computation of betweenness centrality. Update operations, graph pattern and path search queries are missing, alongside many scalability tests. A few years later, two empirical works [JV13, KSM13] compared almost the same set of graph databases over datasets of comparable small sizes, but agree only partially on the concluded results. In particular, the systems analyzed in the first study [JV13] were DEX[3], Neo4j, Titan, and OrientDB, while the second study [KSM13] considered also Infinite Graph. The results have shown that for batch insertion DEX[1] is generally the most efficient system, unless properties are attached to elements, in which case Neo4j is the fastest one [KSM13]. For traversal with breadth-first search, both works agree that Neo4j is the most efficient. Nonetheless, the second work claims, but without proving it, that DEX[1] would outperform Neo4j on bigger and denser graphs [KSM13]. In the case of computing unweighted shortest paths between two nodes, Neo4j performs best in both studies, but while Titan ends up being the slowest in the first study [JV13], it is one of the fastest in the second [KSM13]. For node-search queries, the first work [JV13] shows that both DEX[1] and OrientDB are the best systems when the selection is based on node identifiers, while the other [KSM13], which implements the search based on a generic attribute, shows Neo4j as the winner. Finally, on update operations, the two experimental comparisons present contradicting results, showing in one study favorable results for DEX[1] and OrientDB, while in the other for Neo4j. Due to these differences, these studies have failed to deliver a consistent picture of the available systems, and also provide no easy way of extending them with additional tests and systems.

[3]DEX is the old name for the Sparksee system

The benchmarks proposed in the literature to test the performance of graph databases are also of high significance [APPDSLP13, ABLP+14, EALP+15]. Benchmarks typically come with tools to automatically generate synthetic data, sampling techniques to be used on real data, and query workloads that are designed to pinpoint bottlenecks and shortcomings in the various implementations. These existing benchmarks are domain specific, i.e., RDF focused [APPDSLP13, ABLP+14] or social network focused [EALP+15], but despite the fact that we do not use any of them directly, the design principles and the datasets upon which they have been built have highly influenced our work.

Chapter 3

Multi-Exemplar Search

In this chapter we will present the first contribution of this work, namely, a query paradigm for exemplar query search that admits as input multiple incomplete examples.

In rich information spaces, it is often hard for users to formally specify the characteristics of the desired answers, either due to the complexity of the schema or of the query language, or even because they do not know exactly what they are looking for. Exemplar queries constitute a query paradigm that overcomes those problems, by allowing users to provide examples of the elements of interest in place of the query specification. As we have mentioned earlier (Chapter 2), example-based query paradigms have proved particularly effective to support complex search tasks, especially for assisting in exploratory search. Yet, they usually expect the user to be able to present one example (structure or template) that describes all the details of the user need (e.g., [PDCC, YGCC12, SCC$^+$, MLVP16a, JGK$^+$14]).

In contrast, the solution presented in this chapter relaxes this assumption, providing a more expressive, flexible, and permissive search paradigm. We propose a general approach, called *Multi-Exemplar Query Search*, where the user-provided example can comprise several partial specification fragments, where each fragment describes only one part of the desired result. We provide a formal definition of the problem, which generalizes existing formulations for both the relational and the graph model. We then describe exact algorithms for its solution for the case of information graphs, as well as top-k algorithms. To provide effective and efficient multi-exemplar query capabilities, we have developed efficient algorithms that selectively construct the solution by limiting

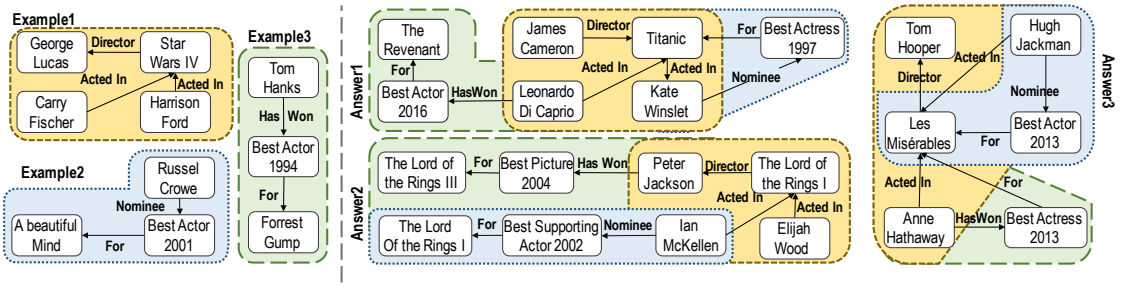

Figure 3.1: Multiple examples, and some of the possible answers proposed with multi-exemplar query semantics.

the number of isomorphic searches to be performed. Since the complete result-set may be too large to be consumed by the user and still too costly to compute, we developed a top-k solution based on a general family of ranking functions that takes into account weights on the nodes of the graph. To further reduce query-time we also studied an alternative approximate method based on a selective expansion order for candidate answers. This method is able to reduce the time complexity by drastically reducing the number of intermediate results to be computed, with only a minuscule reduction in answer-set completeness. Additionally, we present experiments on large real datasets that demonstrate the effectiveness and efficiency of the proposed approaches.

3.1 An Illustrative Example

Consider a movie aficionado consulting an on-line resource, such as the Google Knowledge Graph[1], for movies where directors or actors have been nominated for, or won a prize. They are aware of some examples that describe their interests (Figure 3.1, left). They know about *George Lucas* directing *Star Wars*, with actors *Carry Fischer* and *Harrison Ford*. They recall *Russel Crowe* being nominated for the *Best Actor Academy award* for his role in *A beautiful mind*. Moreover, they also remember about *Tom Hanks* winning *as Best Actor* with *Forrest Gump*.

Note that none of these examples by itself includes all the information that person is looking for. Although they could perform a separate search for each one of them, and then manually compare the three different result-sets to come up with a list of movies, directors and actors with all the required information, this would require an unacceptable amount of work.

[1] developers.google.com/knowledge-graph

Instead, with a multi-exemplar query semantics, the user could provide all three as input, and the system would retrieve various different answers (Figure 3.1, right). Those are just some of the different ways in which the aspects represented by the input could combine in a unique item in the database. In this example, an answer may be represented by a movie with one actor winning an award for another movie and the second one being instead nominated for a similar award (Answer1). A second answer presents a movie where the director has received a prize instead, while one of the featured actors has been nominated for another (Answer2). A third answer (Answer3) represents instead a movie where two actors have been nominated or won an award for the very same movie. Many other relevant answers exists also, and with many different structures.

3.2 Contributions

The task of multi-exemplar query answering requires answers with many different structures, which are not predefined as they are the result of a combination of different smaller examples. Such flexibility has never been considered before, especially for information graphs, and provides different computational challenges. Hence, the contributions of this chapter can be summarized as follows.

- We introduce and formalize the problem of answering *multi-exemplar queries* (Section 3.3).
- We propose *multi-exemplar queries* on graph-data with semantics that allow multiple combinations of non-homomorphic examples (Section 3.3).
- We describe an efficient exact method to answer exhaustively a *multi-exemplar query* on heterogeneous information graphs (Section 3.4).
- We present effective techniques for finding top-k answers given a generic relevance function defined on the nodes of the graph (Section 3.5).
- We study an approximate algorithm for multi-exemplar query search able to reduce intermediate computations with minimum loss in the completeness of the final answer-set (Section 3.6).
- We illustrate the efficiency and effectiveness of our solution at scale through extensive experiments on large real world information graphs (Section 3.8).

3.3 Multi-Exemplar Queries

An *Exemplar Query* [MLVP16a] is an example member of the answer set. In this query-paradigm, the query engine infers the full set of answers based on the example and any semantic annotation provided by the underlying database. Hence, the exemplar query search problem is formally defined as follows:

Definition 3.1 (Exemplar Query)**.** The evaluation of an *exemplar query* represented by the sample S on a database $\mathcal{D}$, denoted as $ExQ(S)$, is the set $\{A \mid S \approx A\}$; where A and S are elements in $\mathcal{D}$, and the symbol $\approx$ indicates a congruence relation between elements, i.e., it states whether two elements are similar or not [MLVP16a].

Here, we assume that the user presents a set of examples (also called *samples*) $\mathcal{S}$ with $|\mathcal{S}|>1$, where each one represents a partial instantiation of the features that the intended results should possess. A naïve solution for answering queries of this form is to evaluate each sample individually and return the union of the result-sets. However, this approach cannot retrieve answers that match (at the same time) more than one of the input query samples.

Therefore, we are in need of more expressive semantics: by providing several different samples, the user tries to describe results that match all their characteristics at once. We then obtain the following definition, when considering a set of (disjoint) user samples $\mathcal{S}$ in a database $\mathcal{D}$:

Definition 3.2 (Multi-Exemplar Query)**.** The result of a *Multi-Exemplar query* for the set of samples $\mathcal{S}$ on a database $\mathcal{D}$, i.e., $mExQ(\mathcal{S})$, is the set $\{A \mid \forall S \in \mathcal{S}. A \approx S\}$, where A and S are elements in $\mathcal{D}$, $\mathcal{S} \subseteq \mathcal{D}$, and the symbol $\approx$ indicates a congruence relation.

The above definition states that an answer to a multi-exemplar query is congruent to *all* the elements in the sample set $\mathcal{S}$ through a congruence relation ($\approx$). Hence, the choice of the congruence relation determines the characteristics and the nature of the answers. Note that, in the special case where $\approx$ is an equivalence relation and all the samples have the same characteristics (i.e., $\forall S_i, S_j \in \mathcal{S}. S_i \approx S_j$), the results are the same as those obtained by searching for elements similar to (any) one of the samples. Therefore, Definition 3.2 is a generalization of the definition of Exemplar Query [MLVP16a].

On the other hand, if the congruence relation adopted is the strict equivalence relation, Definition 3.2 may produce an empty result set when such condition among samples does not hold. Consider the example in Figure 3.1: any answer strictly equivalent (e.g., homomorphic) to the first sample is not congruent to the second or the third. Consequently, the choice for the congruence relation depends on the employed data model and on the intended results.

Following Definition 3.2, a multi-exemplar query requires that all the elements in the sample set are congruent to each result, thus enforcing the computation of all the answers at once (i.e., `AND` semantics). Therefore, answering such queries subsumes more flexible definitions, such as the `OR` semantics, or constraints on values for the answers. In this work, we aim at providing solutions for the most onerous semantics (according to Definition 3.2). However, we note that alternative semantics can also be captured via minimal adaptations to the proposed methods. We elaborate on this in Section 3.7.

We note that the semantics of Multi-exemplar Queries has no other constraints and does not dictate any specific implementation, which makes it relevant for any data model, and adaptable to many use-cases. For instance, in the case of relational data, the samples are tuples and the congruence relation might be the family of s-p-j queries that produces such tuples, whereas in the case of document-search, the samples are documents, or snippets, and the congruence relation a bag-of-words score, such as tf-idf.

3.3.1 Multi-Exemplar Queries on Graphs

Multi-exemplar Queries can be applied on a variety of data models and congruence relations. Existing approaches for web documents, relational tuples and entities can be seen as limited applications of this paradigm [ZW14, SCC$^+$, PDCC, MSS13]. In this study we direct our focus towards directed labeled graphs, which naturally model relational data [DVMT14], semistructured data [HBC15], knowledge graphs [BG14], and many other networks [PPP05, KRS10]. Formally, let $\mathcal{L}$ be a finite alphabet of vertex and edges labels, we adopt the following definition:

Definition 3.3 (Edge-Labeled Graph)**.** A labeled graph is a tuple $G = \langle V, E, \ell \rangle$ where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $\ell : V \cup E \to \mathcal{L}$ is a labeling function from each vertex in V and edge in E to $\mathcal{L}$.

The congruence relation adopted when querying graphs through examples is the graph isomorphism between the query-sample and the answer, which represents a strict bijection between both node and edge labels. In the case of knowledge graph search (refer to Figure 3.1), we are more interested in finding entities and concepts that have a specific relationship structure, i.e., that are connected by specific edge labels, hence the adopted congruence relation is usually *edge-preserving graph isomorphism* [JKL$^+$15, MLVP16a]. However, in the case of multiple samples, this idea cannot be directly applied, since answers need to be congruent to query elements with different topologies. A more appropriate choice for the congruence relation requires that an answer contains structures edge-isomorphic to each sample. Intuitively, an answer is a graph that reconciles in itself all the user samples.

We first define *edge-preserving graph isomorphism* (*graph isomorphism* in what follows) and *subgraph isomorphism*. Edge-preserving refers to searching for the same *structure* as the samples, yet dropping the node name identifiers.

Definition 3.4 (Edge-Labeled Graph Isomorphism)**.** A *graph isomorphism* between two graphs $G_1 = \langle V_1, E_1, \ell_1 \rangle$ and $G_2 = \langle V_2, E_2, \ell_2 \rangle$ is a bijective function $\mu : V_1 \to V_2$ such that for every $(u, v) \in E_1$, $(\mu(u), \mu(v)) \in E_2$ and $\ell_1(u, v) = \ell_2(\mu(u), \mu(v))$, and viceversa. If a graph isomorphism exists between G_1 and G_2, we say that G_1 and G_2 are *isomorphic*, and we write $G_1 \approx G_2$.

A *subgraph* of $G = \langle V, E, \ell \rangle$ is a graph $G' = \langle V', E', \ell \rangle$ such that $V' \subseteq V$ and $E' \subseteq E$. With a little abuse of notation we denote subgraphs as $G' \subseteq G$. Therefore, a sample S in the sample set $\mathcal{S}$ is a subgraph of G, i.e, $S \subseteq G$. If a graph isomorphism exists between G_1 and a subgraph G'_2 of G_2, we say that G_1 is *subgraph isomorphic* to G_2, and we denote it by $G_1 \sqsubseteq G_2$. Hence $G_1 \approx G'_2 \subseteq G_2 \iff G_1 \sqsubseteq G_2$. We can now define a valid answer to a multi-exemplar query on the user samples $\mathcal{S}$.

Definition 3.5 (Multi-Exemplar Answer on Graphs)**.** An *answer* A to a multi-exemplar Query represented by the user samples $\mathcal{S}$ on the database $G = \langle V, E, \ell \rangle$ is a subgraph $A \subseteq G$, such that $\forall S \in \mathcal{S}, S \sqsubseteq A$.

By comparing Definition 3.2 and Definition 3.5, we see that the latter satisfies the premises of the former: the input query is a set of exemplar graphs and the output is the set of answer graphs that are congruent by subgraph isomorphism to all of them, i.e., they contain all the query graphs as subgraphs.

3.3.2 Problem Definition

Note that Definition 3.5 does not constrain the subgraph size. However, with no bounds, even the entire graph may be a valid answer, which is useless in practice. On the same token, answers should not include information (in terms of nodes and edges) that is extraneous to the user request, and should represent a complete concept or situation.

Therefore, it is natural that the two properties below are satisfied: first, *connectedness*, meaning that the answer graph should be a single weakly connected component, i.e., the subgraphs that are isomorphic to each sample should all be connected; and second, *consistency* with the query, so that no additional node/edge is included into the answer graph, apart from those matching the samples in the query.

Formally, given an answer $A : \langle V_A, E_A, \ell \rangle$ on the sample set $\mathcal{S}$, the above two properties are stated as follows.

Property 3.6 (Connectedness)**.** For each two answer nodes $n_A, \bar{n}_A \in N_A$ there exists an undirected path that connects n_A to $\bar{n}_A$. Also, for each sample $S_i \in \mathcal{S}$, $S_i : \langle V_i, E_i, \ell \rangle$, there exists $S_j \in \mathcal{S}$, $S_j : \langle V_j, E_j, \ell \rangle$, $S_i \neq S_j$, with subgraph isomorphism mapping function μ_i and μ_j respectively, such that $\exists n_i \in N_i \ \exists n_j \in N_j$ for which $n'_A = \mu_i(n_i) = \mu_j(n_j)$, for some $n'_A \in N_A$. The node n'_A is called a *junction* node.

Property 3.7 (Consistency)**.** For each answer node $n_A \in V_A$ there exists at least one sample node $n_S \in V_S$, $S \in \mathcal{S}$, $S : \langle V_S, E_S, \ell_S \rangle$ with subgraph isomorphism mapping function μ, such that $\mu(n_S) = n_A$, and for each edge $(n'_A, n_A) \in E_A$ there exists an edge $(n_S, n_S) \in E_s$ such that $(\mu(n'_S), \mu(n_S)) = (n'_A, n_A)$.

Note that, by this definition, all answers are consistent with the query samples, since apart from the nodes/edges matching the samples, they contain no other additional node/edge (see Figure 3.1). Also, since subgraph-isomorphism is a bijection, each sample is matched by a single substructure in the answer, i.e., it contains only the minimal information to satisfy the user requirements.

Combining Definition 3.5 with the connectedness and consistency properties, we can now define our problem.

Problem 1 (Find all mExQ answers)**.** *Given a set of samples $\mathcal{S}$ on the database $G{:}\langle V, E, \ell\rangle$ find all answers $A \subseteq G$ such that $\forall S \in \mathcal{S}, S \sqsubseteq A$, and also A is both* connected *(Property 3.6) and* consistent *(Property 3.7).*

Top-K Problem formulation. Given a single query, the number of isomorphic graphs that exists within a large knowledge base is usually really large. Oftentimes, the user is interested only in the top-k answers, for a specific ranking function on the answers. In this regard, we propose a definition that can employ different ranking functions. We assume a weight function on each vertex $w : V \mapsto \mathbb{R}^+$. The weights can represent user preferences, value, or query relevance, and can be easily provided by contextual data, mined from query logs, or computed using ad-hoc functions [GGY$^+$]. In the following we consider a common use-case in which the score of an answer is given by a function $\rho : \mathcal{A} \mapsto \mathbb{R}^+$, defined as an average on the node weights.

$$\rho(A) = \frac{1}{|V_A|} \sum_{v \in V_A} w(v) \tag{3.1}$$

For instance, assuming a naïve weighting function $w_{deg} : V \mapsto \mathbb{N}$ that assigns to each node a score equal to that node's out-degree, in Figure 1, the score of Answer 1 would be 7/7, while for Answer 2 would be 7/8.

In general, any function ρ similar to the average may be used for ranking (such as max, or a simple sum); however, the choice in Equation 3.1 favors balanced answers in terms of their relative size. Nonetheless, the solutions we study in this work are efficient for the entire family of score functions that are monotonically increasing with the weights of the node scores(Section 3.5). To find answers that respect the multi-exemplar query semantics and retrieve only the top-k solutions that match the user interest we define the following problem.

Problem 2 (Find top-k mExQ answers)**.** *Given a set of samples $\mathcal{S}$ on the database $G{:}\langle V, E, \ell\rangle$, a user-defined parameter k, a weight function $w{:}V \mapsto \mathbb{R}^+$ and the ranking function ρ, return the k answers $\mathcal{A}{:}\langle A_1, ..., A_k\rangle$, such that $\forall A_i \in \mathcal{A}$, all the following hold: (i) $A_i \subseteq G$, (ii) $\forall S \in \mathcal{S}. S \sqsubseteq A_i$, (iii) A_i is both* connected *(Property 3.6) and* consistent *(Property 3.7), and (iv) given any other answer A' that satisfies all the previous conditions (i, ii, and iii), $A' \notin \mathcal{A} \rightarrow \rho(A') \leq \rho(A_i)$.*

Algorithm 1 Multi-exemplar Answering

Input: Database $G : \langle V, E, \ell \rangle$
Input: Samples $\mathcal{S} : \langle S_1, ..., S_m \rangle$
Output: Answers $\mathcal{A}$
1: $\mathcal{G} \leftarrow$ PARTIAL$(G, \mathcal{S})$
2: $\mathcal{A} \leftarrow$ SEARCH$(\mathcal{G}, \mathcal{S})$
3: **return** $\mathcal{A}$

Algorithm 2 mQ-Naive

1: **function** PARTIAL$(G, \mathcal{S})$
2: **return** $\mathcal{G} : \{G\}$ ▷ Selectively pruned based on $\mathcal{S}$
3: **function** SEARCH$(\mathcal{G}, \mathcal{S})$
4: $\mathcal{A} \leftarrow \emptyset$
5: **for all** $G \in \mathcal{G}$ **do**
6: **for all** $S_i \in \mathcal{S}$ **do** ▷ Find isomorphic subgraphs
7: $\tilde{A}_i \leftarrow \{A \subseteq G | S_i \approx A\}$
8: $\mathbf{C} \leftarrow \arg\min_{\tilde{A}_i \in \langle \tilde{A}_1, ..., \tilde{A}_{|\mathcal{S}|} \rangle} |\tilde{A}_i|$
9: **while** $\mathbf{C} \neq \emptyset$ **do**
10: $c \leftarrow$ REMOVEONE$(\mathbf{C})$
11: **if** $\forall s \in \mathcal{S}.s \sqsubseteq c$ **then** ▷ Check sample in c
12: $\mathcal{A} \leftarrow \mathcal{A} \cup \{c\}$
13: **else** $\mathbf{C} \leftarrow \mathbf{C} \cup$ CONNECT$(c, \mathcal{S}, \langle \tilde{A}_1, ..., \tilde{A}_{|\mathcal{S}|} \rangle)$
14: **return** $\mathcal{A}$

3.4 Proposed Approach

We start with input a set of disconnected query-samples. Note that there exist already a number of methods to obtain graphs representations of the user requirements [KRS+09, KA11]. Moreover, we will present later (Chapter 4) an approach for interactive query specification that supports also query suggestions.

We design the task of Multi-Exemplar Queries Answering as a two step approach presented in Algorithm 1, these steps are represented by the PARTIAL and SEARCH functions described below. The first step takes as input the graph and detects a set of candidate subgraphs, or regions $\mathcal{G}$, that most probably contain multi-exemplar answers, i.e., it is a filtering step. The second step searches for answers in such candidate regions. In what follows, we describe algorithms for these two steps.

3.4.1 The Baseline Algorithm

As a baseline approach for Multi-Exemplar Query Answering (Problem 1), we extend the Exemplar Query approach [MLVP16a], and apply some additional optimizations. We refer to this algorithm as mQ-Naive (shown in Algorithm 2). Here, the PARTIAL step

Algorithm 3 CONNECT$^+$

Input: Candidate $c : \langle N_c, E_c, \ell_c \rangle$; HashTable $\mathbb{H}$
Output: Expanded candidates C^+
1: $C^+ \leftarrow \emptyset$
▷ Find candidate nodes contained in some answer
2: **for all** $n \in N_c$ **do**
3: **for all** $\tilde{A}_i \in \mathbb{H}(n)$ s.t. $S_i \not\sqsubseteq c$ **do**
4: $C^+ \leftarrow C^+ \cup \text{MERGE}(c, \tilde{A}_i)$
5: **return** C^+

returns only one candidate subgraph that corresponds to the whole graph, eventually pruned of edges that do not appear in the input.

The SEARCH function, instead, first finds the partial matches (line 7) and then joins them (line 13). In particular, it finds the graphs isomorphic to each sample individually (line 7 can involve any graph isomorphism algorithm and its optimizations, e.g., [Ull76, HLL13]), obtaining $|\mathcal{S}|$ sets, which are the *candidate partial answers* $\langle \tilde{A}_1, ..., \tilde{A}_{|\mathcal{S}|} \rangle$. Subsequently, each individual candidate partial answer is combined with the others (lines 13) to fulfill the *Connectedness Property* (Property 3.6, Section 3.3), keeping only those that can be merged into a complete answer (lines 11-12). To speed up the computation, we avoid the Cartesian product $\tilde{A}_1 \times \tilde{A}_2 ... \times \tilde{A}_{|\mathcal{S}|}$ of all possible combinations of individual sample answers for verifying when Property 3.6 holds. In our algorithm, instead, we progressively expand the smaller set of answers from one single sample S_i (line 8) and, by enforcing Property 3.6, we merge answers $\langle \tilde{A}_1, ..., \tilde{A}_{|\overline{\mathcal{S}}|} \rangle$ from other samples $\overline{\mathcal{S}} = \mathcal{S} \setminus \{S_i\}$ until no other merge is possible.

Baseline Optimizations. We now describe some optimizations we apply to our baseline solution. First, we avoid checking the entire graph in the PARTIAL function. Instead, we keep only the edges whose edge-labels appear in at least one of the samples. Additional pruning techniques are also possible, in particular we can filter nodes based on the neighborhood-filtering technique studied in [MLVP16a], using a node-vector representation (explained below) to further refine the set of nodes considered.

We also introduce a second optimization in the CONNECT function (line 13, Algorithm 2). The role of CONNECT is to expand a candidate answer c with all the possible answers from individual samples that share a node with c. A straight-forward implementation where we check all partial graphs, will lead to a very high computational cost. Algorithm 3 efficiently solves this problem, employing a hash-map $\mathbb{H}$ on the nodes of the answers to each sample. The hash-map is computed first: for all

nodes in the graphs that are isomorphic to (any of) the samples, it maps to the set of graphs that contain this node. That is, for a node n, $\mathbb{H}(n) = \langle \tilde{A}_1^{(n)}, ..., \tilde{A}_{|\mathcal{S}|}^{(n)} \rangle$, such that $\tilde{A}_i^{(n)} = \{A : \langle N_A, E_A, \ell_A \rangle | A \subseteq G, A \approx S_i, n \in V_A\}$. The hash-map stores only those nodes that appear in at least two graphs for two different samples. The algorithm then retrieves for each node in the candidate c only those answers that can be connected to it, and for which the corresponding sample is not already subgraph isomorphic to c (line 3). This is achieved by annotating c with the list of matching samples.

Complexity Analysis. We have $n{=}|\mathcal{S}|$ samples, and for each sample $S_i{\in}\mathcal{S}$, $|\tilde{A}_i|$ is the number of isomorphic answers to S_i. Then, the computational cost of the algorithm is at least the cost of solving n times subgraph isomorphism, which is NP-complete [Coo71], and then the worst case performance of all the calls to `CONNECT` sums up to $\mathcal{O}(\prod_i^n |\tilde{A}_i|)$. Yet, with our optimization we reduce it to $\mathcal{O}(|\tilde{A}_{min}| \times |\mathcal{S}|^3 \times \bigcup_i^n V_{\tilde{A}i})$, where $|\tilde{A}_{min}|$ is the smallest set of partial answers, and $V_{\tilde{A}i}$ the union of all the nodes among all the partial answers for S_i, which is in turn bounded by the set V of all the nodes in the graph. The $|\mathcal{S}|^3$ comes from $\sum_i^{|\mathcal{S}|-1} (|\mathcal{S}| - i)(i)$, which is due to the fact that we need to try every possible way to join each partial answer with the structures matching the remaining samples.

3.4.2 Finding Answers Efficiently

`mQ-Naive` has two main bottlenecks: (1) the computation of all the individual sample answers, and (2) the need to compute and store all the possible partial answers that are built during the incremental expansion of candidates.

We propose here a more efficient (exact) algorithm (Algorithm 4), called `mQ-Fast`, which first selects subgraphs matching one single sample, and then selectively expands these subgraphs in search for complete answers. This approach can reduce the number of isomorphism evaluations, and the number of graphs kept in memory at each step. In particular we show here the modified implementation of `PARTIAL` function for the retrieval of candidate subgraphs.

The expansion step starts from a sample (with the minimum number of expected appearances in the graph, line 2) and retrieves the nodes of its matching subgraphs (line 3),

these are partial answers. To select the initial sample, we estimate the number of matching subgraphs for a graph by exploiting edge statistics as explained later.

From the nodes of the partial answers computed, we start a constrained expansion (EXPAND - line 11-25). The expansion includes neighboring nodes, in breadth-first fashion, while retaining only those that potentially belong to one of the partial answers for the other samples, until no other neighbor node is added (line 5-8). This exploration exploits a compact representation of the edge-labels in the neighborhood of each node at some distance d (usually at most 3 [MLVP16a]), called *node-vectors*. The expansion procedure compares the node-vectors of the samples to the node-vectors in the current candidate. Thus, we can exclude non-matching nodes by comparing vectors instead of graph structures. The candidate subgraphs $\mathcal{G}$ obtained at the end of this procedure are then passed to the SEARCH procedure in Algorithm 2 (which we described earlier for `mQ-Naive`).

Node-vectors. The node-vectors representations are computed as follows. We assume the labels to be ordered, i.e., $l_1, ..., l_{|\mathcal{L}|}$. Given a node n, and a maximum distance value d from n, we compute vector $\mathbf{v}^{(n)} = [v_1^n, ..., v_M^n]$, where $M{=}d|\mathcal{L}|$ represents the number of entries in the vector (i.e., one for each label at each distance). An entry in the vector is set to $v_i^n = 1$ iff there is at least one edge labeled l_t, where $t = (i \mod |\mathcal{L}|) + 1$ at distance $\lfloor i/d \rfloor$ (see the upper part of Figure 3.2, where 0s are replaced by "$-$" for readability). We note experimentally that the number of edge-labels in real large graphs with more than 10^7 nodes is usually below 10^5, and, given the high connectivity of the graph, considering distances above 3 hops provides limited gain. Consequently, the size of these vectors is also limited. Moreover, these vectors are usually sparse, allowing for a considerable space reduction. For instance, in a real graph with $4k$ labels [Goo14] each node has on average around $10d$ bits set to 1.

The vectorial representation provides an effective way to compare a node from a candidate answer with a node from a sample. A graph node is a candidate matching for a sample node if the two vectorial representations are *compatible*. The vectorial representations are compatible if the candidate matching node has a 1 in the same positions as the sample node vector. This is assessed through fast bitwise AND operations between the negation of the node vector and the sample node vector. More formally, given the vectorial representation $\mathbf{v}^{(n)}$, we denote as $\bar{\mathbf{v}}^{(n)}$ the bitwise-negated version of $\mathbf{v}^{(n)}$, i.e.,

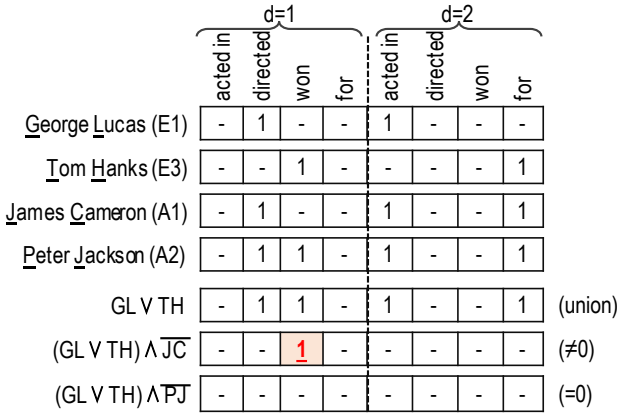

Figure 3.2: Node Binary Vector Matching.

$\bar{v}_i^n = 1 - v_i^n$. We also write $\mathbf{v}^{(n_1,n_2)} = \mathbf{v}^{(n_1)} \vee \mathbf{v}^{(n_2)}$ as the union vector between n_1 and n_2, where $\vee$ is the bitwise OR (similarly $\wedge$ indicates the bitwise AND). Hence, a node n is a candidate node matching the sample node n_1 if $\mathbf{v}^{(n_1)} \wedge \bar{\mathbf{v}}^{(n)} = \mathbf{0}$ (i.e., the zero vector). Then, using the distributive property of logical conjunction over disjunction we have an effective method to assess if a node might connect two or more samples. In particular, if a node n is a candidate node shared between two samples nodes n_1, n_2 from sample s_1 and s_2, respectively, the following equation holds.

$$(\mathbf{v}^{(n_1)} \vee \mathbf{v}^{(n_2)}) \wedge \bar{\mathbf{v}}^{(n)} = \mathbf{0} \tag{3.2}$$

Hence, we check if a node can be a junction node without any false negatives (i.e., we never discard nodes that are part of an answer).

A simplified example is shown in Figure 3.2, with reference to Figure 3.1. We see a vectorial representation, $d = 2$, of the nodes *George Lucas* (GL) from *Example*1 (E1), *Tom Hanks* (TH) from *Example*3 (E3), *James Cameron* (JC) from *Answer*1 (A1) and *Peter Jackson* (PJ) from *Answer*2 (A2). The union vector of GL and TH is GL∨TH. We then see that node JC is not a joint for the two, because it cannot match an edge labeled *won* for $d = 1$, so the result of the bitwise operation $(\mathsf{GL} \vee \mathsf{TH}) \wedge \overline{\mathsf{JC}}$ is not the zero vector $\mathbf{0}$. On the other hand, the node vector for PJ can match all the necessary edge labels, thus $(\mathsf{GL} \vee \mathsf{TH}) \wedge \overline{\mathsf{PJ}} = \mathbf{0}$, and PJ is identified as a junction node.

The correctness of the approach is based on the fact that any matching node possesses the above property, which is formalized in the following theorem:

Algorithm 4 `mQ-Fast`

function PARTIAL($G, \mathcal{S}$)
  $S^* \leftarrow$SELECT($\mathcal{S}$) ▷ Choose the best starting sample
  $\mathbf{C} \leftarrow \{A \subseteq G | s^* \approx A\}$
  $\mathcal{G} \leftarrow \emptyset$
  while $\mathbf{C} \neq \emptyset$ **do**
    $c \leftarrow$REMOVEONE($\mathbf{C}$)
    if $\mathcal{S} \setminus \{\mathcal{S}_c\} \neq \emptyset$ **then**
      $\mathbf{C} \leftarrow \mathbf{C} \cup$EXPAND($c, \mathcal{S} \setminus \mathcal{S}_c, G$)
    else $\mathcal{G} \leftarrow \mathcal{G} \cup \{c\}$
  return $\mathcal{G}$
function EXPAND($c, \overline{\mathcal{S}}, G$) ▷ Add matching nodes to c
  $toVis \leftarrow Vis \leftarrow V_c \leftarrow$MAPS($V_c, \overline{\mathcal{S}}$)
  $\mathcal{L}_{\mathcal{S}} \leftarrow \{\ell(e_S) \mid \forall S \in \overline{\mathcal{S}}\ \forall e_S \in E_S\}$
  while $toVis \neq \emptyset$ **do** ▷ Pruning BFS
    $n_c \leftarrow$REMOVEONE($toVis$)
    $V_t \leftarrow \{x | (n_c, x) \in E, \ell(n_c, x) \in \mathcal{L}_{\mathcal{S}}, x \notin Vis\}$
    $V_c \leftarrow V_c \cup$MAPS($V_t, \overline{\mathcal{S}}$)
    $Vis \leftarrow Vis \cup V_t$; $toVis \leftarrow toVis \cup V_t$
  $V_t \leftarrow \{v | v \in \bigcup_{s \in \overline{\mathcal{S}}} V_S, \exists n \in V_c$ s.t. $\mathbf{v}^{(v)} \wedge \bar{\mathbf{v}}^{(n)} = \mathbf{0}\}$
  if $\bigcup_{s \in \overline{\mathcal{S}}} V_S \setminus V_t = \emptyset$ **then**
    return $\{G[V_c]\}$ ▷ Subgraph of G induced by V_c
  return $\emptyset$
function MAPS($V_t, \mathcal{S}$) ▷ Filter non matching nodes in V
  $V_c \leftarrow \emptyset$
  for all $v \in V_t$ **do**
    for all $n_S \in \bigcup_{S \in \mathcal{S}} N_S$ s.t. $\mathbf{v}^{(n_S)} \wedge \bar{\mathbf{v}}^{(n_c)} = \mathbf{0}$ **do**
      $V_c \leftarrow V_c \cup \{v\}$
  return V_c

Theorem 3.8. *Let $A : \langle N_A, E_A, \ell_A \rangle$ be a multiple exemplar answer for the two samples $S_1{:}\langle N_1, E_1, \ell_1 \rangle$, $S_2{:}\langle N_2, E_2, \ell_2 \rangle$ in a database $G : \langle N, E, \ell \rangle$. If $n \in N_A$ is a junction node shared among s_1 and s_2, then exist two nodes $n_1 \in N_1, n_2 \in N_2$ such that $\mathbf{v}^{(n_1,n_2)} \wedge \bar{\mathbf{v}}^{(n)} = \mathbf{0}$.*

Proof. (sketch) If n is the junction node between n_1 and n_2 by Property 3.6 it belongs to the subgraph isomorphism relations of S_1 and of S_2 to A. Therefore, both the structures surrounding n_1 and n_2 are included in the neighbors of n, i.e., we can follow any undirected-path starting from either n_1 or n_2 in the respective samples, and we will find, at any hop distance, a path with the same labels starting from n in A. Consequently, given the binary vectors $\mathbf{v}^{(n_1)}$ and $\mathbf{v}^{(n_2)}$, it holds that $\mathbf{v}^{(n_1)} \wedge \bar{\mathbf{v}}^{(n)} = \mathbf{0}$ and $\mathbf{v}^{(n_2)} \wedge \bar{\mathbf{v}}^{(n)} = \mathbf{0}$. Hence, it follows that it must hold true

$$((\mathbf{v}^{(n_1)} \vee \mathbf{v}^{(n_2)}) \wedge \mathbf{v}^{(n)}) = (\mathbf{v}^{(n_1)} \vee \mathbf{v}^{(n_2)}) \implies (\mathbf{v}^{(n_1)} \vee \mathbf{v}^{(n_2)}) \wedge \bar{\mathbf{v}}^{(n)} = \mathbf{0}$$

□

Cardinality Estimation. We now describe our solution for cardinality estimation of isomorphic subgraphs (Algorithm 4, Line 2), in order to return the sample with the minimum number of expected matches. Existing models for selectivity of graph queries and cardinality estimation of their results [GTK01, WBTS14] are designed to capture complex interdependencies between labels and nodes and to estimate the size of the results of specific graph queries. In addition, these approaches heavily exploit attributes in nodes and edges, hence they do not easily adapt to our case, where edge-label connectivity is the only information that we can exploit.

To compute our estimation we first decompose a graph into a set of star structures, as it is also done for graph query answering [YHWY16], i.e., trees with a single node and n children at depth 1. Computing cardinality-upperbounds for those small trees is easy, as we can exploit the frequency of co-occurence of label-pairs. Given the maximum match cardinality for each star, we approximate the number of matching based on the combination of those upperbounds.

We maintain two cardinality indexes to quickly estimate the selectivity of edge labels and their co-occurrence: $\mathbb{I}_{pair}$ and $\mathbb{I}_{star}$. The first one, $\mathbb{I}_{pair}$, maintains the number of occurrences of patterns composed by just two edges. Thus, for each pairs of labels $l_1, l_2 \in \mathcal{L}$ the index stores $\mathbb{I}_{pair}(l_1, l_2)$=$\{G' \subseteq G | G' : \langle V', E', \ell\rangle,\ E' = \{(v_1, v_2),\ (v_2, v_3)\}$ s.t. $\ell(v_1, v_2) = l_1,\ \ell(v_2, v_3) = l_2\}|$. The edge labels are hashed to speedup retrieval, and the entire data structure can fit in memory since its size is limited by $\mathcal{O}(|\mathcal{L}|^2)$. Note that the number of pairs is usually much smaller, as not all combinations exist in the graph. Also, in real graphs $|\mathcal{L}|$ is less than 10^5.

The second index, denoted as $\mathbb{I}_{star}$, stores the number of occurrences of a star subgraph containing a label $l \in \mathcal{L}$ and having a predefined size c>0. Therefore the index $\mathbb{I}_{star}(l, c) = |\{G' \subseteq G | G' : \langle V', E', \ell\rangle$ is a star $\wedge |E'| = c \wedge \exists (v_1, v_2) \in E'$ s.t. $\ell(v_1, v_2) = l\}|$. The size of this index is bounded by the number of labels $|\mathcal{L}|$ and the maximum c, which in our case is determined by a parameter C_{max}. Hence, the index size is $\mathcal{O}(C_{max}|\mathcal{L}|)$, but for all practical cases $\mathcal{O}(|\mathcal{L}|)$, since usually $C_{max} \ll |\mathcal{L}|^2$.

The cardinality estimation works as follows. If the sample is just an edge, then the frequency of the label is the correct estimation. If the sample is a 2-edges path, then

[2] In our experiments, C_{max}=10 takes into account the average node-degree and the structure of the expected queries.

the index $\mathbb{I}_{pair}(l_1, l_2)$ stores the correct frequency of such graphs. To estimate an upper-bound for the cardinality of a star-shaped sample $G^* : \langle V^*, E^*, \ell^* \rangle$ we first compute the maximum number of stars that can exists with $|E^*|$ edges and at least one of them with label l, which is computed as follows $Stars(l) = \sum_{c=|E^*|}^{C_{max}} \mathbb{I}_{star}(l, c) * \binom{c}{|E^*|}$

The summation takes into account that $\mathbb{I}_{star}$ contains values for different numbers of edges ($c \in [1, C_{max}]$). For $c = |E^*|$ then $\mathbb{I}_{star}(l, c)$ is the number of stars with exactly $|E^*|$ edges. Then we take into account stars that are formed by selecting a subset from a star with any $c > |E^*|$. In this case, we consider the number of subsamples of size $|E^*|$ out of c elements.

By selecting a label $l_1 = \arg\min_{l \in \mathcal{L}_{G^*}} Stars(l)$, we know that $Stars(l_1)$ is an upper-bound estimation for the number of subgraphs isomorphic to G^*. To obtain a much tighter upper-bound, although approximate this time, we exploit pair-label frequencies once more. We select a second label to be $l_2 = \max_{l \in \mathcal{L}_{G^*}} \mathbb{I}_{pair}(l_1, l)$, i.e., the label that more often appears paired with the previously selected. We estimate the selectivity of G^* as the number $Stars(l_1)$ scaled by the conditional probability of finding l_2 given l_1. This is justified by the fact that not all the stars that have the correct size and contain the label l_1 also contain l_2, while both are required by G^*. The final (estimated) selectivity of G^* is then

$$Stars(l_1) * \frac{\mathbb{I}_{pair}(l_1, l_2)}{\sum_{l \in \mathcal{L}} \mathbb{I}_{pair}(l_1, l)}. \tag{3.3}$$

For more complex structures, we estimate the selectivity of the graph as the lowest selectivity among its stars. We experimentally demonstrate the accuracy of this estimation (Section 3.8).

Complexity Analysis. The `mQ-Fast` algorithm does not discard any correct answers (Theorem 3.8). In terms of time complexity, the most demanding tasks are `Maps`, `Expand`, and subgraph isomorphism. `Maps` compares the node-vectors in V with each sample node-vector, which takes $\mathcal{O}(d|\mathcal{L}| \sum_{S \in \mathcal{S} \setminus \bar{\mathcal{S}}} |N_S|)$. The `Expand` procedure instead performs a traversal of the graph for nodes matching a single sample. This procedure is then repeated at every cycle. In the worst case lines 5-10 in `Partial` scan the entire graph, leading to a complexity of $\mathcal{O}(d|\mathcal{L}| \sum_{S \in \mathcal{S} \setminus \bar{\mathcal{S}}} |N_S| \times |N|^2 |\mathbf{C}|)$.

Algorithm 5 `mQ-Fast`$^+$

```
1: function PARTIAL(G, S)
2:     G ← ∅
3:     s* ← SELECT(S)
4:     for all n ∈ V_{s*} do
5:         M(n) ← {v ∈ V | v^(v) ∧ v̄^(n) = 0}
6:     n* ← arg min_{n∈V_s} |M(n)|                    ▷ Node with min-matchings
7:     C ← M(n*)
8:     while C ≠ ∅ do
9:         c ← REMOVEONE(C)
10:        c+ ← EXPAND(c, S, G)
11:        C ← C \ {c1 ∈ C | c1 ⊑ c+}                  ▷ Remove redundancy
12:        G ← G ∪ c+
13:    return G
```

3.4.3 Avoiding Redundant Computations

We observe that the `mQ-Fast` algorithm performs several expensive operations when generating candidate answers, at the very beginning of the PARTIAL function.

We now present the `mQ-Fast`$^+$ algorithm (Algorithm 5), which introduces further optimizations, while still producing all answers. First, we observe that, thanks to the expansion process, retrieving all possible answers for a sample is not necessary, as long as one single node on the candidate sample is considered. In the `mQ-Fast`$^+$ algorithm, the PARTIAL function finds the (candidate) node matchings between sample nodes of a selected sample (line 3) and graph nodes using the node-vectors (line 4-5). In particular, it chooses the node of the selected sample with the minimum number of matches (line 6-7) and uses each one of those matching as seeds for expansion. This allows to avoid performing the expensive isomorphic search at the beginning. Second, we note that the candidate returned in the EXPAND function may be generated multiple times, if some multi-exemplar answers overlap. This occurs when one candidate includes the answers of another candidate that has already been processed. To prevent this, we add an extra condition, which removes from the list a candidate that overlaps with another one (line 11). The EXPAND function checks if we have found all the matchings for all the samples as in `mQ-Fast` (line 23, Algorithm 4).

Complexity Analysis. This optimization does not require any extra indices, but all the the optimizations proposed for `mQ-Naive` and `mQ-Fast` can be used in `mQ-Fast`$^+$, as well. The complexity remains the same as before, as lines 4 to 7 iterate over each node in the graph, with $\mathcal{O}(d|\mathcal{L}| \times |V|)$ operations.

3.5 Finding Top-k Answers

In this section, we consider the problem of returning only the k best answers to a multi-exemplar query, given a generic scoring function on the graph nodes (Problem 2). This is fundamental in the context of large graphs, where too many results would overload the user, while only few of them contain entities that are relevant to them. Note that the algorithm proposed in this section is exact, i.e., the returned answers have the k highest scores.

In order to compute the score of an answer, Section 3.3 describes a reasonable instance for the score function that computes the average between the weights of the elements to prevent the side effect of skewing the result-set in favor of larger results. Yet, we note that any other analogous function, which can be bounded by monotonically higher ranking scores for answers containing higher scoring nodes, can be used.

Here, we introduce an early termination method, based on the upper bound for the ranking function in Equation 3.1 that can be computed in any given part of the graph. Thus, we avoid SEARCH computing isomorphic graphs in all the areas selected by PARTIAL where answers are bound to a ranking score that is too low. The procedure can stop searching as soon as the kth lower scoring answer A_k found has a score $\rho(A_k)$ higher than any upperbound $\overline{\rho}$ for the remaining portions of the database. Therefore, given a portion of the graph G, we aim at pairing each sample node with a graph node, such that the resulting scoring function is maximized. The optimization version of this problem can be seen as weighted formulation of the hitting set problem [ADP80], which makes a tight estimation impractical. Instead, we propose the upperbound $\overline{\rho}$ computed according to the following theorem.

Theorem 3.9. *Given the set of graph samples $\mathcal{S}$, and answers A_1 and A_2, $\forall S \in \mathcal{S}. S \sqsubseteq A_i$, via the isomorphism function $\mu^S_{A_i}$, the node weighting function w, and the ranking ρ (Equation 3.1). It holds:*

$$\overline{\rho}(A_2) = \frac{\sum_S^{\mathcal{S}} \sum_v^{V_S} w(\mu^s_{A_2}(v))}{max_{S \in \mathcal{S}} |V_S|} < \rho(A_1) \rightarrow \rho(A_2) < \rho(A_1) \tag{3.4}$$

Proof. (sketch) (1) Some $v \in N_{A_2}$ are junction nodes and match more than one sample, we have that $\sum_S^{\mathcal{S}} \sum_v^{V_S} w(\mu^s_{A_2}(v)) > \sum_v^{V_{A_2}} w(v)$. (2) Then, given that an answer contains all samples, we know that $\forall S{\in}\mathcal{S}.|N_{A_2}|{\geq}|N_S|$, and in particular that the minimum number of nodes it contains is at least the size of the larger sample within $\mathcal{S}$. It follows that for sure $\overline{\rho}(A_2){\geq}\rho(A_2)$. Consequently we conclude that if $\rho(A_1){\geq}\overline{\rho}(A_2)$, then $\rho(A_1){\geq}\overline{\rho}(A_2){\geq}\rho(A_2)$ □

Hence, given a portion of the graph $G'$for which to estimate the upperbound of the scoring function, for each node v_S in each sample $S \in \mathcal{S}$, we select the candidate matching $n \in N_{G'}$ with the highest weight $w(n)$. Note that, given Theorem 3.8, we can use the vector representation of each sample and each candidate node to recognize which node could be used for the mapping. We compute the score for a node v_S as $max_{\mathbf{v}^{(v_S)} \wedge \overline{\mathbf{v}}^{(n)} = \mathbf{0}} w(n)$

We change the SEARCH procedure described earlier to compute in advance such upper bound, and search first within the region that has the largest one. The optimization proposed here is implemented in a modified SEARCH function. Note that, the difference between the exhaustive solutions `mQ-Fast` and `mQ-Fast`$^+$ is the implementation of PARTIAL. Hence, the SEARCH, is applicable to both of them, obtaining in this way `mQ-Fast-topK` and `mQ-Fast-topK`$^+$.

Weight functions. Even though the study of the best ranking is out of the scope of this chapter, to showcase the flexibility of our approach, we implemented a set of traditional measures adopted in top-k algorithms for different use-cases. In particular, we consider (1) a *degree-based* ranking function, which uses the node degree as the weight for each node so that the popular nodes are those that are the highest probability to be found [Bar09]; (2) *structural similarity* computing the jaccard similarty of the sets of labels at distance d from each node, or alternatively computing the maximum cosine similarity between the vectorial representations of the neighborhood of each node [GGY$^+$]; finally (3) a *random-walk* similarity measure provides higher score to those nodes where most probably a random surfer will end up when starting from the nodes in the query. In other cases we readily exploit precomputed weights, e.g., from query-logs or other statistics [Bas14].

3.6 Expansion Order Optimization

In the previous sections we explored exact techniques for finding multi-exemplar query answers. The main optimizations presented above aim at reducing the number of candidate answers to the input samples to be retrieved, without losing any final answer. Such optimizations rely on the observation that the graphs matching the user samples are sufficiently separate one another. Consequently, the proposed algorithms will remove candidate answers that are clearly unconnected because too far apart. This allows for an early removal of portions of the graph that do not contain complete answers.

However, the are several cases in which the majority of isomorphic structures lay near each other to form large connected components. Hence, the solutions presented earlier will not save any computation. In particular, `mQ-Naive`, `mQ-Fast`, and `mQ-Fast`$^+$, do not save subgraph-isomorphism computations and also generate large amounts of intermediate redundant results in the SEARCH function.

In this section we present a technique that is able to drastically reduce the number of intermediate results to be computed, and consequently the time complexity at a price of a minuscule reduction in completeness (as demonstrated by the experiments in Section 3.8.2).

Expansion Order. Our approach establishes an expansion order on the individual sample answers that avoids redundant computations. Unfortunately, there is no way to know before-hand what partial answers can be expanded into complete answers. Moreover, when more than 2 samples are provided, different expansion orders may produce different subset of answers. Consequently, the only way to ensure the completeness of the answer-set, would be to materialize all the possible expansions orders.

The approach called `mQ-ExpOrder` is presented in Algorithm 6 as an alternative SEARCH function, and showed (simplified) in Figure 3.3. The core of the algorithm determines a total ordering for the samples in $\mathcal{S}$. Such ordering induces the sequence in which sample answers are merged into larger structure to form multi-exemplar answers.

The order is defined as follows. First, we construct a graph J with a node representing each sample in the query (Algorithm 6, line 7). The graph contains an edge between two nodes i, j if at least one of the structures isomorphic to S_i shares one node with one

Algorithm 6 `mQ-ExpOrder`

```
1: function SEARCH(𝒢, 𝒮)
2:     𝒜 ← ∅
3:     for all G ∈ 𝒢 do
4:         J ← new GRAPH                                  ▷ prepare ExpansionOrder Graph
5:         for all S_i ∈ 𝒮 do                  ▷ Find isomorphic subgraphs & set data structures
6:             Ã_i ← {A ⊆ G | S_i ≈ A}
7:             N_i ← ⋃_{A_i ∈ Ã_i} N_{A_i}
8:             J.ADDNODE(i)
9:         for all S_i, S_j ∈ 𝒮 ∧ j > i do          ▷ Create ExpansionOrder Graph Edges
10:            J.ADDWEIGHTEDEDGE(i, j, |N_i ∩ N_j|)
11:        start ← arg min_{1≤i≤|𝒮|} |Ã_i|
12:        C ← Ã_start
13:        T ← MST(J)                  ▷ Expansion Order based on Maximum Spanning Tree
14:        for all i ∈DFS(start, T) do
15:            C ← MERGE(C, Ã_i)
16:        𝒜 ← 𝒜 ∪ C
17:    return 𝒜
```

structure isomorphic to S_j (line 10). The edges in J are undirected, and are weighted according to the number of joint nodes for the two samples they connect. More precisely, the weight for (i, j) is the number of nodes that appear both in structures isomorphic to S_i and in structures isomorphic to S_j.

The first sample is selected to have the minimum number of subgraph-isomorphic structures in the graph (line 11). These represent the initial set of partial answers, as in the previous algorithms (line 12). Finally, the order in which samples are chosen determines at each iterations which set of partial answers is candidate for merging. The order is determined via a DFS visit of the maximum spanning tree on the graph J (lines 13 and 14).

Complexity Analysis. The time complexity of `mQ-ExpOrder` depends on the choice of the PARTIAL function, while the expansion order influences only the cost of the SEARCH function.

In general, as in `mQ-Naive`, for each one of the n=$|\mathcal{S}|$ samples, $|\tilde{A}_i|$ is the number of isomorphic answers to S_i. Then, the computational cost of the algorithm is still at least the cost of solving n times subgraph isomorphism. Yet, with our optimization he complexity is linear on $|S|$ as opposed to the previous cubic, i.e., we reduce it to $\mathcal{O}(|\tilde{A}_{min}| \times |\mathcal{S}| \times \bigcup_i^n V_{\tilde{A}i})$. In practice, this can reduce of some orders of magnitude the size of the intermediate results computed at each iteration (Section 3.8.2).

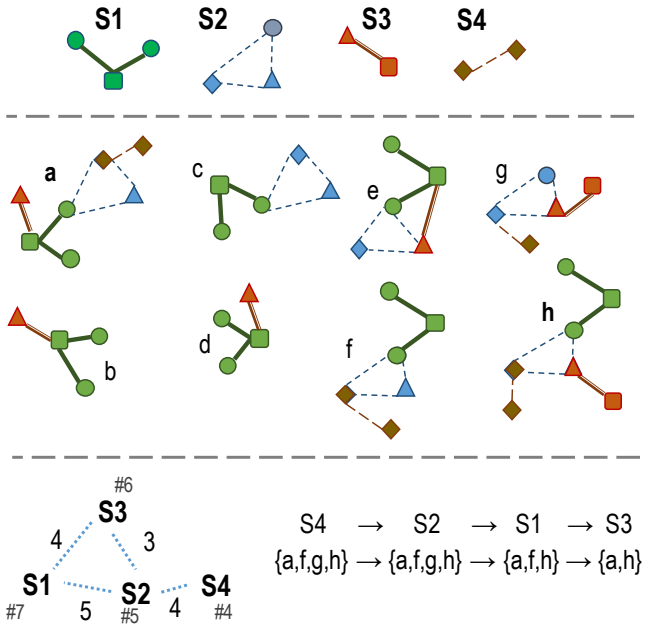

Figure 3.3: Simplified version of the steps to identify an expansion-order between candidate fragments. Top: query samples. Middle: candidate partial answers. Bottom: expansion-order graph and final expansion order.

3.7 Alternative semantics

We previously focused on finding all, or top-k results that are congruent to *all* the samples at the same time. This section presents immediate adaptations to the proposed techniques to accept alternative semantics, yet preserving Connectedness (Property 3.6) and Consistency (Property 3.7) of the answers. Although an exhaustive study of alternative semantics is out of the scope of the current work, we describe two extensions that fit a vast number of use cases: optional samples, and fixed node labels.

Optional samples. Optional samples refer to the situation in which the user can specify whether the samples should be part of the multi-exemplar answers, or can be optional. This case reflects the `OR` and the `OPTIONAL` clauses in SPARQL queries [DAB16a]. The `OR` clause requires that at least one of the samples in the clause is in the answer. The `OPTIONAL` clause additionally allows the case that none of the samples in the clause are in the answer. Moreover, any combination of `OR`, `OPTIONAL` and `AND` (our proposed semantics) clauses is also taken into account.

To adapt our current framework to these more flexible semantics, we need to consider the logic formula $\varphi_{\mathcal{S}}$ expressed as `AND` and `OR` clauses over the samples $\mathcal{S}$. For instance,

if the user requires sample s_1 and one among samples s_2 and s_3, the formula over $\mathcal{S} = \{s_1, s_2, s_3\}$ is $\varphi_{\mathcal{S}} := s_1 \wedge (s_2 \vee s_3)$. `OPTIONAL` clause are not considered in the formula since they are not required for consistency. Then, `mQ-Naive` (Algorithm 2) instead of considering valid candidates c in Line 11 that contain all samples, should check whether c satisfies formula $\varphi_{\mathcal{S}}$. We change our efficient `mQ-Fast` and `mQ-Fast`$^+$ algorithms preventing an early pruning of potentially good candidates in Algorithm 4, Line 20. A simple adaptation removes the pruning condition in Line 20, while a more elaborate solution first converts the formula into Conjunctive Normal Form (CNF), and then checks whether the sample is at least in one `AND` clause.

Fixed node values. The fixed node semantics allow the specification of fixed nodes, or edge values in the samples. For instance, the user might be interested in movies, where the director is always *George Lucas*. Such constraints can be easily included in the current solution by means of additional conditions in the initial filtering and in the graph isomorphism. More specifically, the number of candidate answers for each sample (refer to Algorithm 2, Line 7), is conditioned to the values expressed by the user, exclusively. This additional condition can substantially speed up the computation of answers for multi-exemplar queries.

3.8 Experimental Evaluation

We focus first (Section 3.8.1) on the efficiency of the proposed optimizations for the exact search, both for the computation of the complete result set (Problem 1) as well as for the top-k answers (Problem 2). We also report on the quality of our selectivity estimation, compare the efficiency of our optimization to a solution for partial topology matching, and demonstrate the expressiveness of the paradigm with some results of multi-exemplar queries over a real knowledge graph.

In the second part (Section 3.8.2) we study instead approximate-search solutions. There we present both results in terms of running time, and results in terms of completeness of the result-set and precision at top-k.

Datasets: We tested our algorithm on two of the largest existing knowledge graphs: Freebase [Goo14] and Yago [RSH$^+$16]. We downloaded both graphs in their latest version and removed the unnecessary metadata (e.g., users informations and multilingual

names). We obtained for **Yago3** (YG from now on) 2.9M nodes (comprising entities and taxonomy) and 16.7M edges with 38 edge labels. **Freebase** (FB in the following) instead is much larger, it contains a graph of 76M nodes and 314M edges, with about 4.5K distinct edge labels. We also compared to PANDA [XBCW17], on one of the datasets used in their evaluation, **Cit-HepPh** [LK14]: a citation network of papers, with a total of $\sim 30K$ papers (nodes) and $\sim 35K$ citation links (edges). Each paper is a node with the publication month and year as a label. The original dataset had 122 node labels; we assigned to each edge a label obtained by concatenating the two labels of the edge vertices. In this way, we obtained 8114 distinct edge labels without changing the structure of the graph.

Queries: Since no existing real-world benchmark is available for the problem of multiple-example graph queries, we collected query samples via a user study asking 20 users to create multiple-example queries on different topics, such as, movies, countries, politics, and so on. To this end, users were proposed a prototype search engine (running on FB) and asked to search any entity and connection among them. The users were partially volunteers and partially hired through a crowd-sourcing platform[3]. The queries obtained as described represent the first real dataset for Multiple Exemplar Query, which we now make available[4].

Based on the structure and size of the obtained real queries, we generated a workload of single connected subgraphs of size 1 to 6 edges, based on both the YG and FB knowledge graphs. Following previous works [MLVP16a, XBCW17], this was done via random-walk sampling. We then combined these subgraphs in sets of different multi-exemplar queries. We generated multi-exemplar queries of each sample size between 2 and 5 samples, starting from 5-samples queries and repeatedly subsetting the samples to obtain the smaller sets, resulting in 160 queries for FB and 120 for YG. In our experiments, we report results based on a subset of 100 queries (25 for each one of the four different sample sizes) that all algorithms can fully process in memory. The size of this query-workload is among the largest in this field [MLVP16a, XBCW17, JKL+15], and we make it available online[4]. We did the same for the *Cit-HepPh* dataset, for which we additionally built queries with 6 and 8 subgraphs.

[3] www.crowdflower.com

[4] disi.unitn.eu/~lissandrini/files/mexq-queries.zip

Experimental Setup: We implemented all algorithms presented in this chapter in Java 1.8, on an Intel Xeon $E52440$ (12 Cores 2.40GHz, 188Gb RAM) server running Linux $v3.13.0$. Regarding PANDA [XBCW17], we obtained the code from the authors, and with their feedback we applied the changes described in their paper in order to allow also for answers similar to our semantics. Similar to other approaches, the knowledge graphs and all relevant indexes are memory resident [MLVP16a].

3.8.1 Results for Exact Search

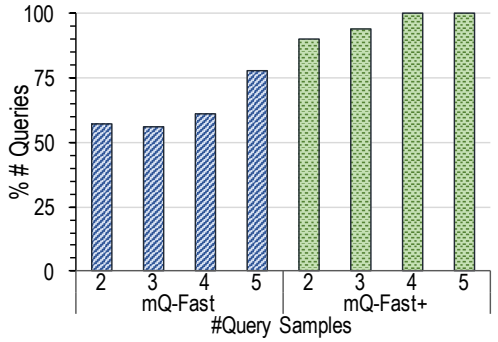

Figure 3.4: Portion of queries where `mQ-Fast` and `mQ-Fast`$^+$ compute less isomorphism than `mQ-Naive` as a function of the number of fragments;

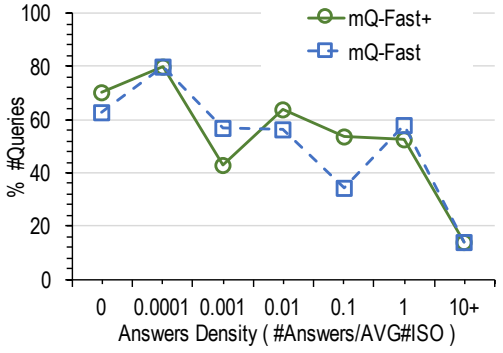

Figure 3.5: Portion of queries where `mQ-Fast` and `mQ-Fast`$^+$ compute less isomorphism than `mQ-Naive` as a function of the ratio of final answer over the number of isomorphic subgraphs in the graph.

Selectivity estimation: First, we evaluate our selectivity estimation method (described in Section 3.4.2) with respect to the real number of subgraph isomorphic structures, since the selectivity estimation takes negligible time ($<$ 10ms), but is an important component in our optimizations. To this end, we compare the cardinalities of all the generated samples to the estimates produced by our method. Both estimated and actual cardinalities are sorted by the number of answers. The closer the two rankings are,

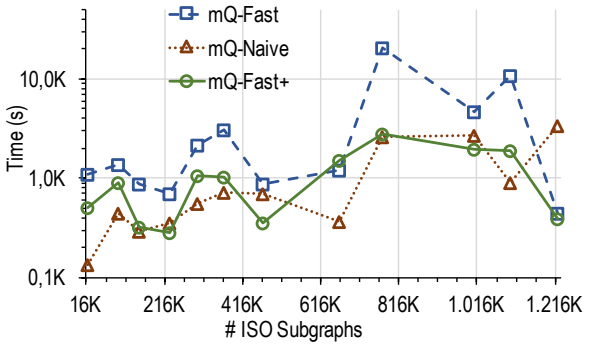

Figure 3.6: Comparison of Running time w.r.t. number of isomorphic graphs existing in the graph.

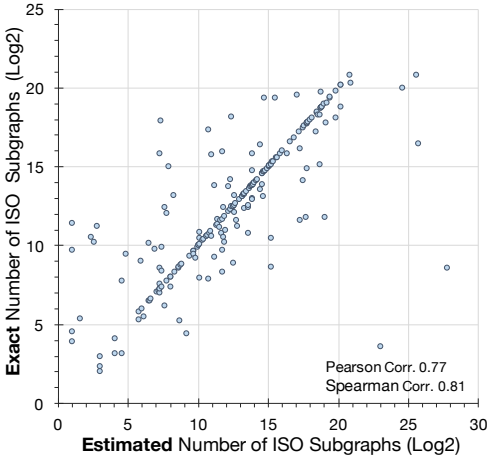

Figure 3.7: Correlation between the estimated and the exact number of isomorphic subgraphs.

the more likely a pairwise comparison between samples provides the correct minimum. Therefore, we measure the Spearman's rank correlation [HT11] between the two ranked lists: a high correlation value would mean that our method obtains a ranking similar to the real one. The result of the experiments (Figure 3.7) shows a Spearman rank correlation of 0.81 with p-value $< 10^{-50}$ (a value near +1 indicates a perfect association of ranks). This means that in most cases our estimate is able to identify the best sample to select for our algorithms.

Evaluation of Complete Search: Considering that `mQ-Naive` retrieves all the subgraphs isomorphic to all samples, we test how many of those `mQ-Fast` and `mQ-Fast`$^+$ computes. In Figure 3.4, we show the percentages of queries (on `FB`) in which our optimized algorithms actually compute less isomorphisms than `mQ-Naive` as a function of the number of samples. The results show that `mQ-Fast` computes the same (or more) number of isomorphisms than `mQ-Naive` in 43% of cases, while `mQ-Fast`$^+$ is more efficient

in $> 90\%$ of the cases. Those are cases in which some structures are shared by many answers, hence they appear in many different candidate regions of $\mathcal{G}$ when computed by PARTIAL, in these particular cases `mQ-Fast` is actually wasting some computations. Note that, despite more subgraphs are computed, they are generated in different iterations, so that, at any time, only a portion of those is in memory. `mQ-Fast`$^+$, on the other hand, will never waste computation since the regions $\mathcal{G}$ identified by the optimized PARTIAL (Algorithm 5) never overlap. Hence, it computes at most the exact same number of subgraphs computed by `mQ-Naive`, but never more. This proves that our optimizations effectively reduce the memory requirements of the algorithms, leading to better scalability than `mQ-Naive`.

In Figure 3.5, we present the percentages of queries (on `FB`) in which the two algorithms, `mQ-Fast` and `mQ-Fast`$^+$ compute faster than `mQ-Naive` as a function of the ratio between the number of all final answer (without top-k) and the average number of isomorphic graph per sample. Therefore, in the presence of few multi-exemplar answers, even if there are many candidate fragments, the optimizations are faster in more than 50% of the cases, and can still be faster in less favorable situations. Indeed, when there are more multi-exemplar answers than fragments (and the ratio is > 1), it means that few fragments combine altogether in many different ways, so it is better to compute the few fragments and then compute their combinations. This is also shown in Figure 3.6, where we show the running time on `FB` as a function of the sum of isomorphic subgraphs present in the knowledge-base (here, points summarize intervals of approximately $60K$). This substantiate the choice of the `mQ-Fast`$^+$ for large knowledge graphs with rich informations, where `mQ-Naive` would not cope with the number of candidates to handle and `mQ-Fast` has higher risk to waste computations. Also, this suggests that we could study the application of strong-simulation [MCF$^+$14] in place of isomorphism as the congruence relation, so that many solutions would be merged in one single answer [MLVP16a, XBCW17].

Evaluation for Top-K Search: We first tested the performance obtained to compute the top-3 answers in the `FB` dataset with `mQ-Fast` and `mQ-Fast`$^+$, and compared that to the baseline `mQ-Naive`. We report values only for the *structural similarity* weight function since the behavior of the other functions are comparable. Figures 3.8 and 3.9 show the median number of isomorphisms and the median and average running-time as a function of the number of samples, respectively. Note that the query time accounts for

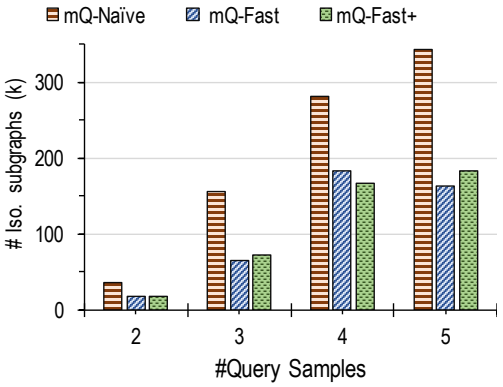

Figure 3.8: Median number of isomorphisms performed as a function of the number of query samples (top-3 answers, `Structural similarity`).

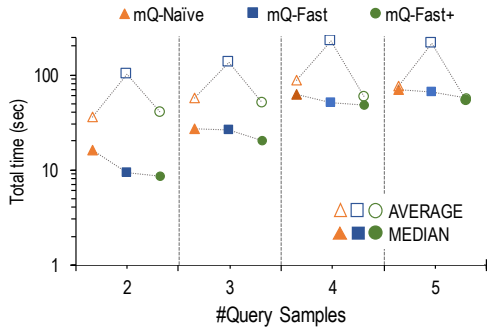

Figure 3.9: Average and median query time as a function of the number of samples in the query (top-3 answers, `Structural similarity`).

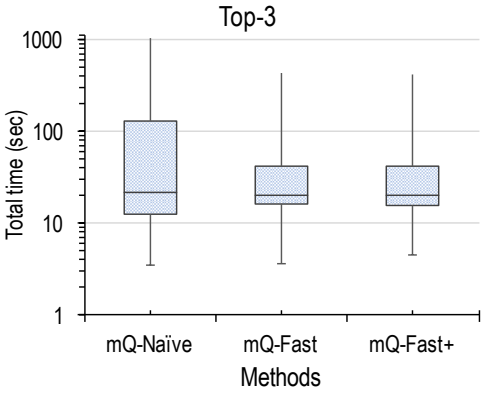

Figure 3.10: Comparison of Running Time on YAGO

the complete process of retrieval and ranking of the answers. While query time is biased towards the specific implementation, the number of isomorphisms is implementation-independent.

The median number of isomorphisms (Figure 3.8) shows that both optimizations reduce the number of computations most of the times. As seen before, on average (not showed in figure) `mQ-Fast` computes many more isomorphisms than `mQ-Naive`. This is also

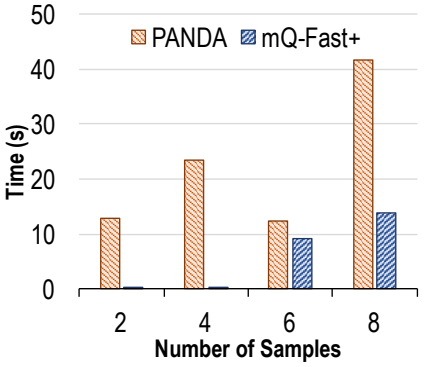

Figure 3.11: Running Time Comparison of Fast+ vs. PANDA on *Cit-HepPh*.

reflected on the average running time (Figure 3.9). The different behavior of `mQ-Fast` on average and median reflects once again the larger sensitivity of the method to the graph topology. As a matter of fact, `mQ-Fast`$^+$ performs up to two times better than `mQ-Naive` in terms of number of isomorphisms and time. Therefore, for these queries `mQ-Fast`$^+$ is the only choice. Finally, we report that experiments on `YG` obtained similar results (Figure 3.10). Yet, on `YG`, the difference between `mQ-Fast` and `mQ-Fast`$^+$ is rather negligible, while the gain of `mQ-Fast`$^+$ is much larger on `FB`. This shows that (1) the algorithms keep similar performances on the larger and the small graph, but also (2) that it is not just the size of the graph but also the number of isomorphic subgraphs to connect that makes the problem challenging.

Note that the query time reported, although impractical for real time scenarios, refers to multi-exemplar exact answers. While approximate schemes [MLVP16a, XBCW17, JKL$^+$15] can be employed and represent a possible extension to this work.

Comparison to PANDA [XBCW17]: We compared the running time of PANDA and Multiple Exemplar Queries with `mQ-Fast`$^+$ on *Cit-HepPh*. The experiment with queries containing between 2 to 8 samples (Figure 3.11) showed that `mQ-Fast`$^+$ is much faster for this task. We also tried to run queries on `YG` and `FB`, but since PANDA has to first build all isomorphic answers (similar to `mQ-Naive`) without employing any optimization for this task, the PANDA approach did not terminate within 1 hour, and we were not able to compare on larger datasets. Moreover, their approach does not enumerate all the answers, but rather stops when finding portions of the graphs that contain them. Therefore, when considering their running time, the time needed for this extra step should be taken into consideration.

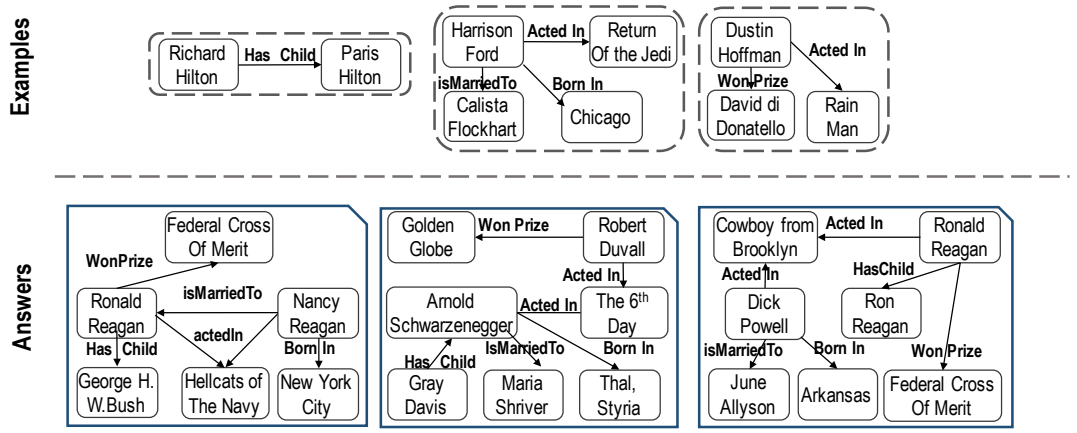

Figure 3.12: Top: input examples of actors and interesting biographical informations. Bottom: Answers using multi-exemplar paradigm.

Expressive Power: Although an analysis of the quality of the algorithms largely depends on the choice of the weight function, we show some non-trivial result found with multi-exemplar queries. In this case multi-exemplar queries can be employed for cinema journalism to quickly retrieve facts on actors and movies and their biographical informations. For instance, in Figure 3.12 the samples describe notable actors and facts about prize won, spouse, father-child relationships and in which movie they appeared. We run this query on `YG`, which is not complete, so the samples are not part of the results, since none of the examples have all the relationships required. Since none of the examples are part of an answer, with only this information as input any other query paradigms will fail. First we note that in `YG` the notion of *child* and *successor* are somehow collapsed, so that *George H.W. Bush* is listed as *child* of Ronald Reagan. Nonetheless, we find the 40th president of the U.S. has been an actor, and his wife as well acted in the same movie. We retrieve a similar case for *A. Schwarzenegger*, and one more for *Ronald Reagan*, now with the actual son *Ron*. Note how the results are non isomorphic one another.

3.8.2 Results for Approximate Expansion Order

We now present a set of experiments to test quality and performance of some approximate multi-exemplar search algorithms. In particular we compare running time, completeness of the results, and quality at top-k of `mQ-Naive` with `mQ-ExpOrder` and also with an approximation scheme based on Personalized Page Rank [JW03] (PPR) as done for the single exemplar query search [MLVP16a]. In this second method we filter out

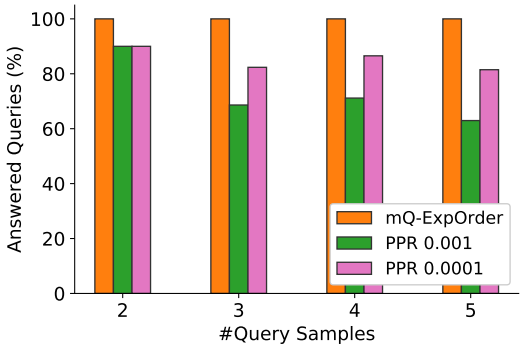

Figure 3.13: Percentage of Queries for which at least an answer is retrieved (full-search).

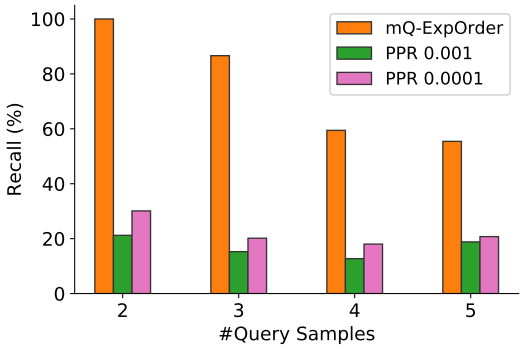

Figure 3.14: Average Recall as a function of the number of samples in the query(full-search).

nodes with a PPR value lower than user defined threshold $\tau \in [0, 1]$ when using the user samples as seeds. In this way we reduce the search space to structures with high proximity to any of the user samples. In our experiments we compare both methods to a larger set of queries (180 instead of 100), and for PPR we use values of τ 0.001 and 0.0001.

The experiments show that `mQ-ExpOrder` provides a noticeable reduction in running time for queries with 4 and 5 samples, while being able to retrieve the majority of answers and without any loss in quality in the task of top-k retrieval. The filtering scheme based on PPR, instead, to provide the same or faster running times requires sacrificing both recall and ranking precision.

Completeness and Quality. Figure 3.13 compares the percentage of queries for which `mQ-ExpOrder` and the PPR filtering fail to retrieve answers. This is the number of times that they incorrectly report that no answer is found while at least one answer exists. In this test we see that `mQ-ExpOrder` never fails to find at least some answers, while PPR

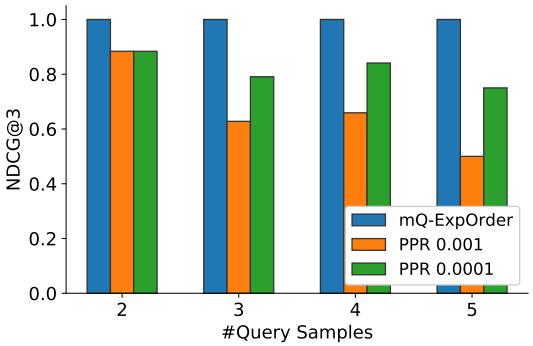

Figure 3.15: NDCG at top3 as a function of the number of samples in the query (Ranking score [Bas14]).

with $\tau = 0.0001$ fail between 10% and 20% of times, with $\tau = 0.001$, i.e., with an even more restrictive filtering, the PPR technique fails up to 40% of queries.

We also compare average recall for the aforementioned methods in Figure 3.13. Here, we see that while `mQ-ExpOrder` is always retrieving at least one answer when there exist one, on average it retrieves between 80% and 60% of all the existing answers. For the PPR filtering, the recall is much lower, with an average value of 30% at most. Yet, we recall that in most cases, there exists thousands and even millions of answers. Hence, we focus now on the actual precision in the retrieval of the top-k. Intuitively, since the user would be interested in only few results, we can afford to disregard results in the long-tail.

We study the quality of the retrieved top-k results in terms of normalized discounted cumulative gain at top-3, top-5 and top-10 in Figure 3.15, 3.16, and 3.17. Given the absence of query logs in terms of graph search, in these experiments - for all the search methods - we use a real score based on entity name popularity in a web corpus as computed in the FreebaseEasy dataset [Bas14]. First we report that `mQ-ExpOrder` obtains nearly perfect performance in all cases (NDCG of 0.9999). This confirms it as a method able to reduce unnecessary computations, without sacrificing the quality of the results. Regarding the filtering with PPR, instead, we obtain NDCG scores between 0.5 and 0.9, with higher loss in quality for queries with 4 and 5 samples and when using PPR threshold 0.001.

Running Times Finally, we compare the performance gain obtained by applying the aforementioned methods, namely `mQ-ExpOrder` and the PPR filtering. The distribution of query running times are reported in Figure 3.18 in seconds for each method and

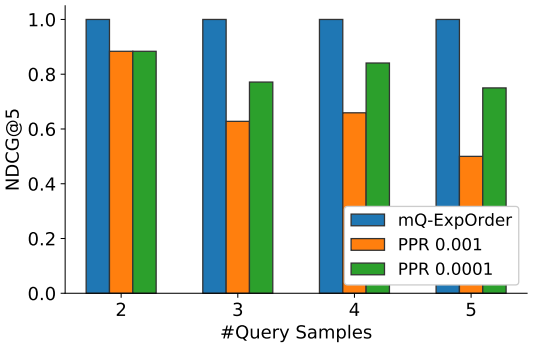

Figure 3.16: NDCG at top5 as a function of the number of samples in the query (Ranking score [Bas14]).

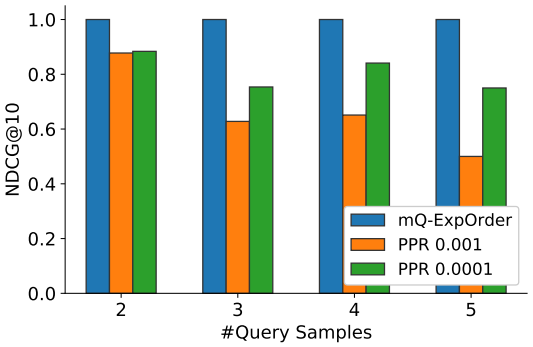

Figure 3.17: NDCG at top10 as a function of the number of samples in the query (Ranking score [Bas14]).

compared to the performance of `mQ-Naive`. When compared to the running times of the `mQ-Naive` algorithm, we see that `mQ-ExpOrder` provide shorter running time for queries with 4 and 5 samples. Especially with 5 samples the gain is up to an order of magnitude. The PPR filter provides a large speedup with $\tau = 0.001$, while with $\tau = 0.0001$ in many cases we obtain worse performances than with `mQ-Naive`. In this case the time lost is due to our implementation of the PPR computation, which doesn't exploit any advanced techniques or precomputed partial results [Cha07], and with such low threshold performs a large number of traversals of the graph.

We conclude that the `mQ-ExpOrder` optimization provide solid guarantees both in terms of running times and quality of the results, while the PPR filter, to provide sufficient performance gain, will sacrifice too much in terms of answer quality.

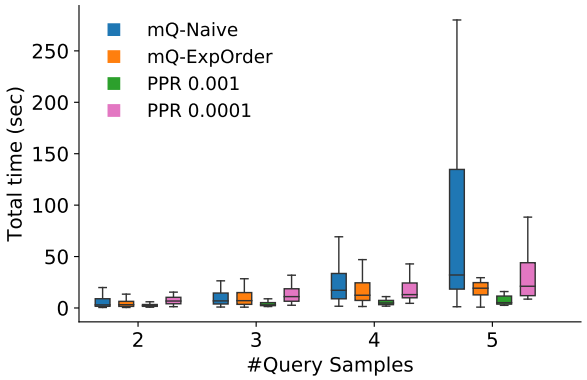

Figure 3.18: Distribution of query time as a function of the number of samples in the query (full-search).

3.9 Summary

By-example methods have been proven useful to users that are not able to formulate a query that describes the features of the intended results. Yet, existing by-examples methods are limited to accept a single structure.

In this chapter, we propose multi-exemplar queries, a novel query paradigm that identifies elements that are similar to a *set* of examples provided by the user, without enforcing the complete structure of the answer in advance. We describe effective exact solutions, and introduce a generic formulation to efficiently return top-k answers. Moreover, we present a study of approximate solutions that reduce query time by limiting the amount of intermediate results to be computed. The experiments show the efficiency and effectiveness of our approach, especially in the pursue of reducing the subgraph-isomorphism computations.

Chapter 4

Interactive Graph-Query Suggestion

In the previous chapter, we have introduced the multi-exemplar query-paradigm as an easy tool for posing queries that represent complex information needs. Yet, such a solution still requires the user to describe some structure of interest. This task may still be hard when the user is not familiar with the content of the knowledge graph they are exploring.

In this chapter, we study the problem of graph-query suggestion and expansion, in order to help the user in exploring a knowledge graph and in expressing their intent more easily. We note that there is no study that provides a principled approach for Interactive graph-Query Expansion (IQE). Hence, to the best of our knowledge, this is the first study about methods to provide IQE functionality to knowledge graph search engines. We present an interactive query suggestion and refinement solution that helps the user identify and investigate the portion of a knowledge graph describing their need. The end goal is to help the user build a graph-query through an expansion/refinement process.

In particular, we study models for graph-query suggestion within a pseudo-relevance feedback framework and study the application of language modeling approaches to the case of graph queries. Moreover, we examine their effectiveness in proposing facts and entities to include in a given query. For the case of exploratory navigation, such refined query can represent the information the user was looking for, helping the user to enrich their knowledge and understanding of the domain. Furthermore, the expanded queries can be used as actual graph-queries, like exemplar queries, to retrieve other similar

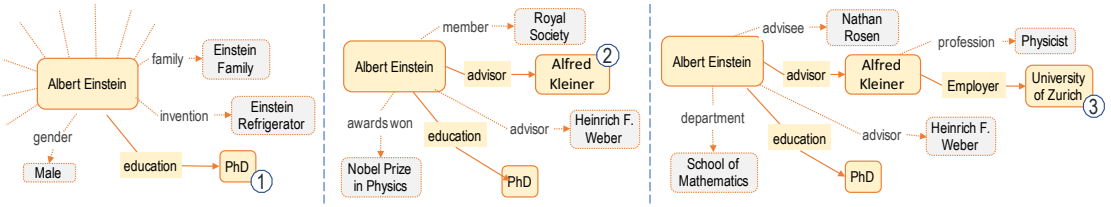

Figure 4.1: Composing a graph query: Einstein Academic Education.

situations from the knowledge graph. In the following, we present an illustrative example of such use-case.

4.1 An Illustrative Example

A typical search task on knowledge graphs is the search for similar or related cases to those currently known [MBL15, MLVP16a, MSS13, ZCC+17]. This allows users to search for elements or situations of interest by providing as query one of those objects that are already known to be relevant. As we have seen, this is typically useful in the case of a novice user tapping into the rich information of a knowledge graph, and trying to find insights for specific topics.

Consider a student looking for information about the education of famous scientists. A popular example would be Albert Einstein. If the user was just to search for entities similar to Einstein, they would have found Nobel Prize winners, inventors, or physicists. To obtain information about their education, it is necessary to specify in the query something about it, so for instance, that Alfred Kleiner was Einstein's advisor and that Einstein himself had a PhD. A graph query engine [MLVP16a, JGK+14, CZY13] will then retrieve all other similar situations that include those aspects. A student not familiar with the topic will be oblivious to these details; they would only know Einstein's name and the fact that Einstein is a good subject for their search.

In our example, the user starts with the query "Albert Einstein". The system we envision should then suggest possible additional information, e.g., the facts that he invented the "Einstein Refrigerator", or that he had a PhD. The user will then select the most suitable suggestion among the options, e.g., "education-PhD", as depicted in Figure 1.2(left).

After the user choice, the system should then respond with a new set of suggestions to expand the query even further, with the suggestions of each subsequent step of the

interaction being more focused towards the direction that the user wants to move. For example, after the user selects "education-PhD", the system proposes more options related to education and academics (i.e., advisors, awards, etc.), as shown in Figure 1.2(middle); finally, after the user selects "advisor-Alfred Kleiner", the system focuses more on advisors and their affiliations, allowing the user to complete the query by selecting "employer-University of Zurich", depicted in Figure 1.2(right). Therefore, the system helps the user to quickly arrive at the intended query. Without such a system, the user would be forced to a laborious and cumbersome search within the entire list of available edges, which may be hundreds or thousands.

4.2 Contributions

The task of query-expansion requires a way to assess the likelihood a candidate expansion represents the actual user need. Graph query suggestions have not been studied formally so far, and traditional IR approaches have not been applied in this domain.

In this chapter we compare several models and expansion techniques on a real-world knowledge graph at web-scale and provide insights in their effectiveness in terms of helping the user describe their information needs, and of building appropriate graph queries. More specifically, we present the following contributions:

- We formally define the problem of Interactive Graph-Query Suggestion (Section 4.3).
- We solve the problem of computing graph-query suggestions and expansions, by extending and adapting language-modeling and relevance feedback from traditional IR theory to the case of knowledge graph search (Section 4.4).
- We compare two different approaches, one estimating the likelihood of an expansion to be relevant based on edges that are more frequent in general, and another favoring instead edges that are unexpectedly frequent around the query.
- Our approach is implemented in a pseudo-relevance framework that is extensible to multiple scoring techniques (Section 4.5).
- We study their effectiveness with a series of user-studies on queries from the 7th Open Challenge on Question Answering over Linked Data and with real-world, large knowledge graph (Section 4.6).

4.3 Problem Formulation

We recall, as seen in the previous chapters, that knowledge graphs are commonly represented as directed labeled graphs, in which nodes model entities, edges relationships among them, and labels represent the names of entities and relationships. Hence, a *knowledge graph* is an edge-labeled graph defined by a triple $\mathcal{K} : \langle V, E, \ell \rangle$ where V is a set of entities represented as nodes, $E \subseteq V \times V$ is a set of relationships (or facts) among entities represented as edges, and $\ell : V \cup E \to \mathcal{L}$ is a labeling function on entities and relationships (see also Definition 3.3 in the previous chapter). Knowledge graphs are also known as heterogeneous information networks or simply information graphs.

Following also the model introduced for exemplar queries on graphs (Section 3.3), a graph-query is a subgraph of the knowledge graph that presents a set of features that are of interest for the user. Hence, with the exemplar queries semantics for the user's query, the user issues queries on a knowledge graph $\mathcal{K}$ by means of a subgraph $Q \sqsubseteq \mathcal{K}$.

Assuming that a user is able to operate on complex graphs and provide a complete subgraph representing an example of the intended results is in many cases unrealistic. Moreover, in case of incomplete (or underspecified) examples the algorithm returns a large number of potentially irrelevant answers.

We instead consider a more natural settings where the user is aided by a query suggestion system. In these settings the user initially provides a partial specification Q_0 of the example and the system gradually recommends additional constraints to be added to such example to restrict the results.

We assume, without loss of generality, that the initial query Q_0 provided by the user is a single node (or entity). At each iteration i the system provides an *expansion* Q_i of the the previous query Q_{i-1}. Such expansion adds *one extra edge* to the previous Q_{i-1}.

Definition 4.1 (Graph Query Expansion)**.** An *expansion* $Q_i : \langle V_i, E_i, \ell \rangle$ of a query $Q_{i-1} : \langle V_{i-1}, E_{i-1}, \ell \rangle$ is a connected subgraph such that it holds that $Q_{i-1} \sqsubseteq Q_i$ and $|E_i| = |E_{i-1}| + 1$.

The expansion process terminates when, at some iteration n, the expanded query Q_n satisfies the user need or when the user abandons. This termination criteria, assumes

that (1) the initial query of the user is a partial description of an exemplar query, i.e., it does not contain irrelevant information, and (2) that at each step the user is able to choose some relevant edge.

However, while the former assumption is reasonable in most practical scenarios as we require the user to just provide one entity of interest, to make the latter possible the user should be able to inspect, every time, all the potential expansions of a query. Indeed, the straightforward approach that shows all possible relationships around the initial query, overloads the user with tens, hundreds or even thousands of expansions, i.e., connections to neighbor entities. For instance, the entity *Albert Einstein* in Freebase has more than 500 outgoing relationships with other entities excluding attributes like age, or height, and such relationships add up to more than forty different relationship types. As such, we require an intelligent way to select only those relationships that are the most likely to describe the type of information the user has in mind.

We formulate such problem in a ranking-retrieval fashion by defining a relevance function ρ on the possible relationships that can be added to the current expansion Q_i to obtain Q_{i+1} on the knowledge graph $\mathcal{K} : \langle V, E, \ell \rangle$.

Definition 4.2 (Expansion Relevance)**.** The *relevance* of the candidate expansions for a query Q_i is represented by a function[1] $\rho : \mathcal{P}_{\sqsubseteq}(\mathcal{K}) \times E \mapsto \mathbb{R}^+$ on a knowledge graph $\mathcal{K} : \langle V, E, \ell \rangle$ that assigns a score to each expansion edge e given the current query Q_i. The relevance function is such that for each $e_1, e_2 \in E$, $\rho(Q_i, e_1) = 0$ means that the edge e_1 does not belong to Q_{i+1}, and $\rho(Q_i, e_2) > \rho(Q_i, e_1)$, means that the expansion edge e_2 is more-likely to belong to Q_{i+1} than the edge e_1.

At each step i we denote the set of expansion edges as $E_\delta = E_{i+1} \setminus E_i$ on query $Q_i : \langle V_i, E_i, \ell \rangle$ and expansion $Q_{i+1} : \langle V_{i+1}, E_{i+1}, \ell \rangle$. Hence, once presented with the set of expansion edges E_δ, the user either selects one expansion or interrupts the process if no further expansion is required. When an expansion is selected, the expanded query will be provided as input for ρ in the next step. Finally, the problem we tackle in this chapter is the following.

Problem 3 (Graph Query Suggestion)**.** *Given a knowledge graph $\mathcal{K} : \langle V, E, \ell \rangle$ and a query Q_i to be expanded, retrieve a set of k edges $E_\delta \subset E_\mathcal{K}$, $|E_\delta| = k$, such that $\forall e \in E_\delta . \rho(Q_i, e) > 0$, and $\forall \bar{e} \in E_\mathcal{K} \setminus E_\delta$, $\exists e \in E_\delta$ such that $\rho(Q_i, e) \geq \rho(Q_i, \bar{e})$.*

[1] In this chapter we denote as $\mathcal{P}_{\sqsubseteq}(G)$ the set of all subgraphs of G.

In other words, given a knowledge graph $\mathcal{K}$ and the current query $Q_i{:}\langle V_i, E_i, \ell\rangle$, we retrieve the set of edges that are not part of the query, but that are connected to it, i.e., $E_\delta : \{\langle v_1, v_2\rangle \notin E_{Q_i} \wedge (v_1 \in V_i \vee v_2 \in V_i)\}$. Since there may be a large number of edges, we will keep only the top-k by ranking such edges based on their score as given by ρ. The elements from E_δ are there returned as candidate expansion edges.

Through this process, we can start from an initial query Q_0 and proceed through a set of expanded queries Q_i until reaching a query Q_n that satisfies the user need and that includes all the intermediate suggestions as subgraphs, i.e., $Q_0 \sqsubseteq Q_1 \sqsubseteq Q_i ... \sqsubseteq Q_n$

4.4 Graph-Query Suggestion

In this section, we describe the modeling of relevance function ρ which represents the implicit preference of the user. The core of the application, as defined above, is to retrieve a set of expansion edges E_δ that can be added to the current user query Q_i. Since the relevance function solely depends on the current step, as to simplify the notation we drop the index i and use $Q : \langle V_Q, E_Q, \ell\rangle$ to indicate the current query.

The challenge is to exploit the limited information provided by the query, alongside what we know about the contents of the knowledge graph, in order to guess what the user intent is. In the following, we establish a model that given a query allows us to estimate the likelihood that an expansion edge is of interest to the user. To this effect, we first estimate a probabilistic model for a graph-query, and then expand such a model so we could estimate the likelihood of a candidate expansion.

4.4.1 Bag-of-Labels Model for Graph-Query

We treat the task of graph-query-suggestion similarly to expansion-term-selection for keyword-queries. The user intent is defined by a probabilistic model from which the user draws the contents of their query. The goal is to estimate such a model treating the query as a set of observations. The solution we present exploits the results from the existing approaches in information retrieval. In particular we design a model that has similar characteristics to the language models used in keyword query expansion for document search.

Language models have been initially proposed to estimate the relevance of a document given a keyword query [PC98]. From a high-level perspective, the language modeling approach tries to estimate a multinomial model $\hat{p}(w|D)$ over each keyword w for each document D in a collection C [PC98]. As such, it is able to assign a likelihood to a query $\mathbf{q} = \langle w_1, w_2 ... w_n \rangle$, so that given a document prior $\hat{p}(D)$, one can rank each document according to $\hat{p}(D|\mathbf{q}) \propto \hat{p}(\mathbf{q}|D)\hat{p}(D)$. This assumes that the probability that document D is the target document, given that the user chooses $\mathbf{q}$, is proportional to the probability that the user would use $\mathbf{q}$ to describe D. Such assumption is based on the intuition that the user is describing some kind of prototype document with a sample of words that should appear in such document.

To draw a parallel between keyword-search on documents and graph-search, we first simplify our model for a graph. To do so we build on the intuition that, in a graph query, the user is describing a set of relationships which are fragment of a larger knowledge graph, and those relationships are described by their respective edge labels. Hence we represent a graph as a bag of edge labels, corresponding to both edges within the graph itself and edges connecting the nodes within the graph and the other nodes in the knowledge graph. Given this, a graph can be modeled as follows:

Definition 4.3 (Bag of Labels Model of a Graph)**.** A subgraph $G : \langle V_G, E_G, \ell \rangle$ of a knowledge graph $\mathcal{K}$, i.e., $G \sqsubseteq \mathcal{K}$, has a simplified representation as a multiset (or bag) that is defined as $\overline{G}$:$\big\{\ell(\langle e_1, e_2 \rangle) | \langle e_1, e_2 \rangle \in E_{\mathcal{K}} \wedge (e_1 \in V_G \vee e_2 \in V_G)\big\}$

Under this definition, two graphs are considered similar if they can be translated to similar bags of edge labels. Note that $\bar{G}$ is a multiset, so duplicate labels are maintained. Furthermore, we do not include only edges that appear in the graph but also edges on the fringe that connects the graph-nodes with the surrounding portion of the knowledge graph. Intuitively, this choice has two positive effects: (1) enriches the description of $\bar{Q}$ and $\bar{G}$, and (2) allows the model to be applicable even when Q contains just a single node.

Limits and Possible extensions. Another important note to make is that, by this definition, we are effectively disregarding a great deal of information about the structure of the query, i.e., its topology. Yet, the experiments we present later will prove its effectiveness.

Nonetheless, the model we propose could be easily extended. What we propose here is the equivalent of the unigram model for keyword-queries. In the classical unigram model, the queries "*Alice called Bob*" and "*Bob called Alice*" would be treated in the same way, as they both would be represented by the set {"*Alice*", "*Bob*", "*called*"}. In the bi-gram model instead, the query "*Bob called Alice*" would be translated into {"*Bob called*", "*called Alice*"}. Similarly, models similarly to the bi-gram (or n-gram) are applicable to our case. Here, for instance, we could also consider all the pairs of edge labels that are incident on the same node, or consider paths at with a predefined length. In the following, we limit ourselves to the simpler, so-called unigram model; other expansions will be left for future work.

4.4.2 Baseline Scoring-Functions

Given the model described above, we rank edges in E_δ according to some score computed based on their label l. We envision two baseline techniques in order to rank the set E_δ given a query $\bar{Q}$, one favoring *frequent* labels and the other favoring *distinctive* labels.

Maximum Likelihood Estimation. The first score is based on a simple maximum likelihood estimation (MLE), where the score of a label l is proportional to its relative frequency around the graph-query and in the entire repository, i.e., frequent labels are estimated to be more likely to be part of a query. In practice, the model $M_{\bar{Q}}$ from which the query $\bar{Q}$ has been drawn, is estimated with the help of Dirichlet smoothing [ZL01a, ZL01c] as

$$\hat{p}(l|M_{\bar{Q}})_{MLE} = \frac{|E^l_Q| + \epsilon\hat{p}(l|\mathcal{K})}{|E_Q| + \epsilon} \tag{4.1}$$

Where ϵ is the Dirichlet prior[2], $|E^l_Q|$ is the number of edges in Q with label l, $|E_{\bar{Q}}|$ is the total number of edges in Q, and $\hat{p}(l|\mathcal{K})$ should represents the probability of l appearing in any graph in the collection of target graphs. Yet, since we do no have such collection, but only one large graph and a target graph could be any of its subgraphs, in our case $\hat{p}(l|\mathcal{K})$ is approximated by the relative frequency of the edge label, i.e., $|E^l_\mathcal{K}|/|E_\mathcal{K}|$. Note that, were we investigating the same problem within a graph database (where multiple distinct graphs are stored), or in case we had access to a graph-query log, $\hat{p}(l|\mathcal{K})$ could be estimated in a more classical way.

[2] A system-wide constant usually between 1000 and 2000 [ZL01a, ZL01c].

KL-Divergence. The second technique favors distinctive labels. Those are labels which are frequent around the query, but infrequent in the dataset. This score is based on the KL-divergence between the probability distribution of labels around the query and their distribution in the rest of the graph (similarly to what's done by Lafferty and Zhai [LZ17]), as follows

$$\hat{p}(l|M_{\bar{Q}})_{KL} \propto exp\left(\frac{1}{(1-\lambda)}\log\left(\hat{p}(l|M_{\bar{Q}})\right) - \frac{\lambda}{(1-\lambda)}\log\left(\hat{p}(l|\mathcal{K})\right)\right) \tag{4.2}$$

The parameter $\lambda \in [0, 1)$ is a weighting parameter that depends on the frequency of labels in the graph and on the application. As anticipated, these methods are quite simple, and can be used as baselines for the problem of graph query suggestion.

4.4.3 Exploiting Pseudo-Relevance Feedback

We presented above some baseline methods to score expansion edges that consider only the query to be expanded and the frequency of labels in the knowledge graph. Next, we describe instead a more advanced model-based approach, which is based on the pseudo-relevance feedback framework [Roc71]. With this approach, we estimate the likelihood of an edge to be a relevant expansion, based on its relative frequency within the pseudo-relevance set, which is the set of graphs that satisfy the original query (before expansion).

Model-based approaches start by estimating a generative model for each document (i.e., M_D for each $D \in C$) so that a document is viewed a sample from such model. In our case we estimate a model $M_{\bar{G}}$ for each graph $G \in \mathcal{P}_{\sqsubseteq}(\mathcal{K})$[3]. We now have two generative models $\hat{p}(\bar{Q}|M_{\bar{Q}})$ and $\hat{p}(\bar{G}|M_{\bar{G}})$ that represent respectively the likelihood of $\bar{Q}$ to be drawn from $M_{\bar{Q}}$ and the likelihood of $\bar{G}$ to be drawn from $M_{\bar{G}}$.

In the (pseudo-)relevance framework [LC01, ZL01b, CNGR08] an additional element is take into consideration, namely the so called *pseudo-relevant set*. This is the set of documents $\mathbb{D}_{rel}$ matching the query (before expansion). They are referred to as a *pseudo*-relevant when their relevance is assumed just because they match the query, and hence those documents are more likely to be actually relevant to the query.

[3]Note that, although $\mathcal{P}_{\sqsubseteq}(\mathcal{K})$ is an intractably large set of graphs, we will not need to actually consider all of them.

Given the pseudo-relevance set $\mathbb{D}_{rel}$, a new model is estimated (M_{rel}), and this model is usually used to re-weight the query model. In the re-weighted query-model, according to the (pseudo-)relevance model, estimation is done without smoothing as follows:

$$\hat{p}(l|M_{\bar{Q}})_{REL} = (1-\lambda)\frac{|E^l_{\bar{Q}}|}{|E_{\bar{Q}}|} + \lambda\hat{p}(l|M_{rel}). \tag{4.3}$$

The re-weighted query model is also called a mixture model, because it partially draws from a simple model with (empirical) maximum likelihood estimation (MLE) in the first component, and it also draws from the pseudo-relevance model (M_{rel}).

In our case M_{rel} does not refer to a set of pseudo-relevant documents, but to a set of answer graphs. In practice, we process the current graph-query Q and obtain some set $\mathcal{G}_{rel} = \{G_1, ..., G_m\}$. These answer graphs may be the result of entity queries [SCSS15], or graph exemplar-queries [MLVP16a]. This $\mathcal{G}_{rel}$ is our pseudo-relevant set.

MLE with Pseudo Relevance Feedback. Once we've obtained the set $\mathcal{G}_{rel}$ of (pseudo-)relevant graphs – with $\overline{\mathcal{G}}_{rel}$ the corresponding set of bags of labels – the relevance model is obtained through maximum likelihood estimation as

$$\hat{p}(l|M_{rel})_{MLE} \approx \sum_{\bar{G}\in\bar{\mathcal{G}}_{rel}} \hat{p}(l|M_{\bar{G}})\hat{p}(\bar{Q}|M_{\bar{G}}), \tag{4.4}$$

where $\hat{p}(\bar{Q}|M_{\bar{G}}) \propto \prod_{l\in\bar{Q}} \hat{p}(l|M_{\bar{G}})$, and each $\hat{p}(l|M_{\bar{G}})$ is computed according to Equation 4.1. Note that we can multiply $\hat{p}(l|M_{\bar{G}})$ by $\log(\hat{p}(l|\mathcal{K})^{-1})$ to limit the contribution of terms (labels) that are frequent in the graph [Met07].

Now we have a new way to rank the expansions edges to be presented to the user, which is based on their score $\hat{p}(l|M_{rel})_{MLE}$, with l being the label of the edge to rank.

KL-Divergence with Pseudo Relevance Feedback. As we observed earlier (Equation 4.2), one could use alternatively the intuition behind the KL-divergence scoring model [ZL01b]. In practice, the scoring should assign a high probability to expansions that are common among the pseudo-relevant graphs, but not so common within the rest of the graph. Hence, we estimate the the likelihood of a candidate expansion label l as:

$$\hat{p}(l|M_{rel})_{KL} \propto exp\left(\frac{1}{(1-\lambda)}\frac{1}{|\bar{\mathcal{G}}_{rel}|}\sum_{\bar{G}}^{\bar{\mathcal{G}}_{rel}}\log\left(\hat{p}(l|M_{\bar{G}})\right) - \frac{\lambda}{(1-\lambda)}\log\left(\hat{p}(l|\mathcal{K})\right)\right) \tag{4.5}$$

4.4.4 Surprise-based Heuristic

Finally, we complete our study including a scoring technique for expansions based on the concept of "surprise" [SBDK09]. Such heuristic has been successfully employed by Sarkas et al. [SBDK09] to provide query-driven and domain-independent keyword-query expansion in the context of document search. The spirit behind the notion of surprise would favor expansions that are *unexpectedly* common within the set of search results. This method adopts the same intuition seen earlier for the case of the KL-divergence scoring model, i.e., the expansion terms that obtain a higher score are those with a relative frequency higher in the result-set than in the rest of the dataset. The intuition is formalized as follows: given a set of terms T=$\{t_1, t_2, ..., t_n\}$, $\hat{p}(t_i)$ is the probability of term t_i to appear in one document, and $\hat{p}(t_1, t_2, ..., t_n)$ is the probability of all terms to occur together in one document. Under the independence assumption, i.e., assuming that the terms $t_1, t_2, ..., t_n$ are unrelated and appear in documents independently, the expectation would be for the following to hold $\hat{p}(t_1, t_2, ..., t_n) = \hat{p}(t_1)\cdot...\cdot\hat{p}(t_n)$ [SBDK09]. The surprise is then measured for the set of terms T by the ratio

$$Surprise(T) = \frac{\hat{p}(t_1, t_2, ..., t_n)}{\hat{p}(t_1)\cdot\hat{p}(t_2)...\cdot\hat{p}(t_n)} \tag{4.6}$$

It follows that, given the query Q composed by the terms $\{q_1, q_2, ..., q_n\}$, we can score an expansion term q' by considering the increment in the surprise score obtained by $Surprise(T) - Surprise(T \cup \{q'\})$. Note that, opposed to the earlier models, this score does not take into consideration the frequency of a term within a single document, but just their frequency of appearance within some set (in our case, the pseudo-relevance set).

As done previously, we port this model to the case of graph-query expansion by considering edge-labels in place of terms, and extended-graphs (as in Definition 4.3) instead of documents. Moreover, since all members of our relevance set contains, by definition,

at least the same edge labels of the query, it follows that we can compute the score of the candidate expansion label l as follows:

$$Surprise(l) \propto \frac{\hat{p}(l|M_{rel})}{\hat{p}(l|\mathcal{K})}. \tag{4.7}$$

4.5 Retrieving Candidate suggestions

The relevance functions proposed above (Section 4.4) form the basis of our interactive suggestion algorithm that proposes, at each step, a list of k most relevant expansions given the current query Q. Algorithm 7 shows in pseudocode one of such steps. The core part of the suggestion algorithm is the model estimation in Line 6. In our framework that step can accept any relevance (or score) function of those in Equation 4.1, 4.2, 4.4, 4.5, or 4.7. Such flexibility allows potentially any other score to be included, as well as combinations of scores.

Initially, the suggestion algorithm retrieves the candidate set of expansions E_ε (Lines 1–3). Such set of candidate expansions is constructed on the edges incident to any query node, excluded those already in Q (Line 4). Recall that when we defined our bag-of-labels model (Definition 4.3), we included also the edge labels in the fringe. This means that by exploring the neighborhood of the query, we know already what candidates are possible additions. From E_ε we compute the set of all edge-labels L_ε in any edge $e \in E_\varepsilon$ (Line 5). Finally, after the model relevances are retrieved, we return the k expansions with the highest score.

Computing the relevance on the fly might be inefficient, especially when a large number of edges has to be considered. However, the overall frequency of the edge-labels is readily computed offline and stored in a hashed-index. To compute the pseudo-relevant feedback in Equations 4.4, 4.5, and 4.7 we require instead efficient query-answering methods. Hence, we may prefer to retrieve a limited set of answer graphs with approximate top-k techniques [MLVP16a].

Result presentation. As the goal of a graph query is primarily to describe structures of interest, the algorithm will produce a ranked list of the top-k edge-labels. Although the study of appropriate interfaces is out of the scope of this work, we describe below alternative result presentations that fit most of the use cases.

Algorithm 7 Graph-Query Suggestion

Input: Knowledge graph $\mathcal{K} : \langle V, E, \ell \rangle$
Input: Current query $Q : \langle V_Q, E_Q, \ell \rangle$
Input: Current answers $\mathcal{A}_Q$
Input: Model M ▷ One defined by Eq 4.1, 4.2, 4.4, 4.5, or 4.7
Input: Number of expansions k
Output: Expansions $\langle l_1, l_2, ..., l_k \rangle$

$E_\varepsilon \leftarrow \emptyset$
for each $v_i \in V_Q$ **do**
    $E_\varepsilon \leftarrow E_\varepsilon \cup$ GETEDGES(v, E)
$E_\varepsilon \leftarrow E_\varepsilon \setminus E_Q$
$L_\varepsilon \leftarrow \{\ell(e) | e \in E_\varepsilon\}$
ESTIMATEMODEL$(M, E_\varepsilon, L_\varepsilon, \mathcal{A}_Q)$
Scores ←new $Dict()$
for each $l \in L_\varepsilon$ **do**
    Scores $\leftarrow \{l : \rho_M(Q, l)\}$
Scores $\leftarrow sort(Scores)$
return $Scores.get(k)$

- *Only edge-labels:* The system shows only the edge labels of the top-k expansions. In this way, the user is not overloaded with information and can execute expansions as, for instance, SPARQL queries with the "*" wildcard.

- *Single representative:* Among all the possible edges having a specific edge-labels, the system shows one single edge (or a small set of representative edges). Such presentation provides some sort of explanations for the meaning of the proposed edge-labels.

- *Faceted search*: The system aggregates the suggestions (edges or labels) in a topical or structural manner forming a facet that is presented to the user. Each facet represents a coherent group of suggestions (e.g., academic information or business-related). Facets could be obtained by grouping edge labels that frequently appear together, or by exploiting meta-paths [MCM+15].

4.6 Experimental Evaluation

We report results in terms of quality of our expansion framework comparing five scoring functions. To obtain real user feedback we performed a user study where we evaluated the quality of the suggestions proposed by the different scores.

4.6.1 Experimental Setup

Dataset: Experiments have been executed on top of the full Freebase data-dump [Goo14], which is one of the largest and most commonly used knowledge graphs in the contest of knowledge-graph search [JGK+14, MLVP16a]. We downloaded Freebase in their latest version and removed the unnecessary metadata (e.g., users informations and multilingual names). After such preprocessing, we obtained a graph of 76M nodes and 314M edges, with about 4.5K distinct edge labels.

Implementation: The dataset is loaded into the Neo4j 3.3.0 [neo] graph database to enable fast node search. The choice of the system has been made based on the results of the experimental evaluation presented in the next chapter. The suggestion algorithms are implemented in Python 3.6.4 and run on an Intel Xeon $E5-2420$ (12 Cores 1.90GHz, 128Gb RAM) server running Linux $v3.13.0$.

Queries: We obtained 65 graph queries by manually translating questions and answers from the 7th Open Challenge on Question Answering over Linked Data (QALD-7) [UNH+17]. The queries in the challenge describe questions about various topic of interest for which an answers had to be retrieved from a knowledge-graph. In particular we selected questions and answers that we were able to map to an entity, an edge, or 2 or more edges in Freebase. Each of those have been treated in turn as a graph-query in our application. For instance, the first query in the QUALD-7 dataset is "doctoral supervisor, Albert Einstein", this can be translated into the single entity Albert Einstein, or in the edge Albert Einstein, advisor, Alfred Kleiner (see Figure 1.2).

We then assigned to each graph-query a descriptive sentence to describe an exploratory information need. As an example, to the query above we assigned the topic Academic information about Albert Einstein.

User study: For each query in our dataset we computed with each method the top-20 edge expansion suggestions. Hence, we obtained five top-20 edge expansion lists, one for each scoring function: the maximum likelihood estimation (MLE – Equation 4.1), the score based on the KL-divergence (KL – Equation 4.2), the maximum likelihood estimation extended with pseudo-relevance feedback (MLE-rel – Equation 4.4), the same

for the KL-divergence (KL-rel – Equation 4.5), and the score based on *Surprise* (Srp - Equation 4.7).

We submitted each query alongside the topic description and each expansion list to a crowdsourcing platform[4]. For each query and each candidate expansion, we collected human judgments assessing their relevance/interestingness on a four point scale: irrelevant (0), uninteresting (1), fairly interesting (2), really interesting (3). We obtained in this way approximately 25 thousands distinct judgments collecting *at least* three judgments for each query-suggestion pair. We note here that, on average, for each query we found approximately five fairly interesting or really interesting suggestions, among all methods. This first number is a testament to the complexity of the task, as it support the intuition that, in any case, only few of edge suggestions are relevant expansions.

4.6.2 Results

Quality of the ranking. In the following we present Normalized Discounted Cumulative Gain at top 3, 5, 10 and 20 for the case of queries composed by one single entity (Figure 4.2), one single edge (Figure 4.3), and 2 or more edges (Figure 4.4). In each chart, we include the score obtained with a random sorting of the list (Rnd).

The first interesting thing to note is that when the query is just one entity mention (Figure 4.2 and Figure 4.5); our result is not better than guessing. As one could expect, knowing just the entity of interest is too little an information to predict the user interest. This case equates to guessing from the word "apple" whether the user is looking for companies or fruits. In such cases, we can conclude, a more effective strategy would be to maximize the suggestion-list diversity to cover many different aspects and help the user disambiguate their intention.

The outcome is different, instead, with queries composed by one edge (Figure 4.3). On one side we see that the scoring function that exploits the KL-divergence and the pseudo-relevance feedback, as well as the Surprise heuristics, can produce much better rankings. On the other side, we see instead that all other methods are quite under-performing.

Both of the two best-performing methods are favoring edge-labels that are more frequent in the pseudo-relevant set than in the rest of the graph, and this seems to better capture

[4] crowdflower.com

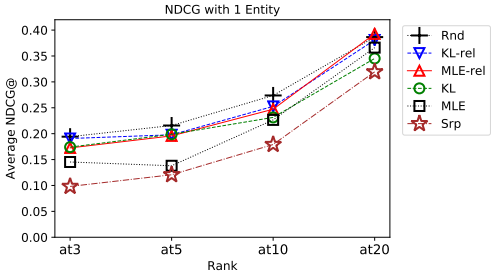

Figure 4.2: Variation of NDCG score at top-3,-5,-10, and top-20 when the starting query is 1 Entity

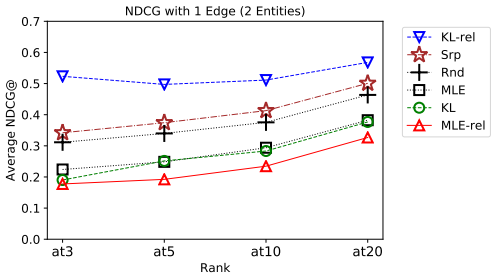

Figure 4.3: Variation of NDCG score at top-3,-5,-10, and top-20 when the starting query is 1 Edge (2 Entities)

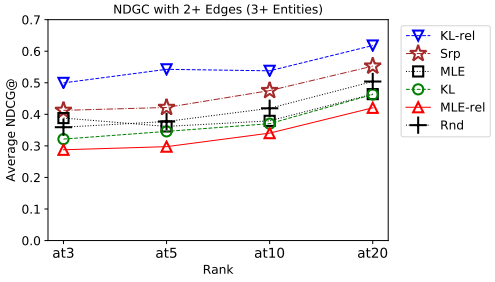

Figure 4.4: Variation of NDCG score at top-3,-5,-10, and top-20 when the starting query is 2 or more Edges (3+ Entities)

the user need. The probabilistic model provides a far better estimation of the correct score. On the other hand, we see that favoring really frequent edge labels is actually counterproductive. As a matter of fact, those scoring functions (MLE, and MLE-rel) are favoring really generic aspects.

The observations above are confirmed also by the experiments with queries containing 2 or more edges (Figure 4.4), where once more the KL-divergence with pseudo-relevance feedback outperforms all the alternatives.

Moreover, by comparing in both experiments the result of the scoring based on KL-divergence with and without the inclusion of the pseudo relevance-feedback, we see that PRF is of vital importance to improve the precision of the model

Comparison with different query-sizes. Finally, we study the effect of the increase in the size of the query to the ability of predicting which graph-query expansions are relevant (in this case we consider all suggestions that obtained an average score larger than 2 on the scale [0–3]). We present the average precision at the top-5 for all the methods, comparing different query sizes (Figure 4.5). The chart shows both the better precision provided by the KL-methods, but also the fact that, as the user provides additional information, the score is able to understand better the user intention and provide more relevant information. As a matter of fact, consider that, as the size of the query increases, it also increase the number of possible expansions, making the task even harder, yet the precision increases, proving that the additional information is effectively exploited .

Measuring user effort. We validated the set of test queries on the amount of effort spared on retrieving the desired graph query, assuming such query exists. In particular, we assume each query's target graph is the one obtained by selecting those edges with the highest score in the user study. As initial query, we consider the fact indicated as answer in the QUALD-7 dataset (e.g., for the query asking for the advisor of *Albert Einstein*, which we translated to the need Academic information about Albert Einstein, we select as initial query Albert Einstein, advisor, Alfred Kleiner).

The overall effort compares the number of suggestion required in order to retrieve all the edges to add to the initial query, to the total number of possible edge types that a user would be required to inspect without our system (i.e., the number of edge-types for each entity). For instance, for the query above, the final query graph would contain information about the advisor, the PhD degree, and the department and university of affiliation. Considering, for instance, that the entity Albert Einstein has 41 edge-types around it, and presenting to the user just one instance of each type, the user would be required to inspect all of them in order to retrieve the desired edges. With our best scoring (KL-rel), the first relevant fact (Albert Einstein, education, PhD) was within the top-5 suggestions, the information about the department and the university of affiliation, among the top-6 in the subsequent set of suggestions. This allows the user to limit the inspection for this example to just 11 edge types, instead of a total of 50.

We repeated the process for all queries, and we computed that, with our suggestion system, the user will have to inspect 60% less edge-types, significantly reducing the effort

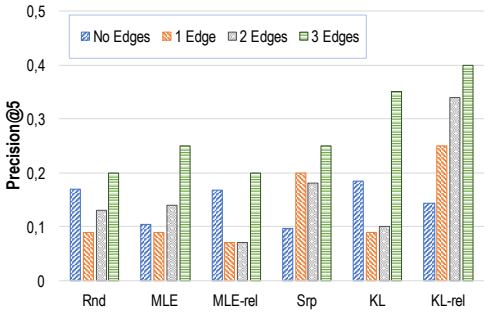

Figure 4.5: Variation of Precision at top-5 as the sizes of the query increase.

for the user in formulating their queries. Therefore, our system makes the difference between a task that users will most probably never complete (using traditional tools), and a task they can easily carry out (using the suggestions of the proposed system).

4.7 Summary

Searching a large knowledge graph of entities and relationships can be a cumbersome and discouraging task for users, especially novices. Although knowledge graphs offer a significant support for textual and semantic search, interactive exploratory methods for knowledge graphs are still in their infancy. Yet, they are of great importance, especially to help users build graph queries.

In this chapter, we assumed the perspective of the user, and devised an interactive query expansion method that helps them navigate towards the answers of interest. Given the scarce feedback and lack of query logs (normally accessible for document search), we developed a pseudo-relevant feedback solution, which embodies language models and results obtained by example-based search on the graph. We provided an extensive experimental evaluation with real users on Freebase, one of the largest knowledge graphs. Our simple and expressive framework outperformed traditional models and query expansion techniques in both expressiveness and quality. The results demonstrate the effectiveness of our approach and the usefulness of the proposed expansions.

Chapter 5

Evaluating Graph-Databases

In the previous chapters, we have studied how to provide users with an effective query paradigm for graphs and how to help them in performing exploratory search on top of graph data. Those studies spawn from the fact that graph data, in general, has become increasingly important for a wide range of applications [LMP+15, SKW07, GSSZ15].

The increase of importance of graph data has also led to the development of many *graph data management systems*. Among those, in this chapter we focus on the study of Graph Databases (GDBs). While there exist the so-called *graph processing systems* [HDA+14], which focus on batch workloads and long-running business-intelligence analytic pipelines, GDBs constitute a different category since they focus on storage and querying tasks where the priority is the high-throughput interrogations of the data, and the execution of transactional operations.

Despite the increasing interest in graph databases their requirements and specifications are not yet fully understood, leading to a high variation in the supported functionalities and the achieved performances. In this chapter we provide a comprehensive study of the existing such systems. The results of this study can help decide which graph database to pair and extend with our exploratory techniques, in order to fully enable access to and exploration of information graphs. For instance, the results of this work guided our decision to choose Neo4j [neo] as the graph database to employ in the back-end of our graph-suggestion system presented in the previous chapter (Chapter 4).

As seen earlier (Section 2.4), existing graph database evaluations have serious limitations, not only in terms of systems and datasets tested, but more importantly in the scope

and type of operations employed in their experiments, and in the interpretability of their results. Here we overcome such limitations by introducing a novel micro-benchmarking framework for their assessment, and provide detailed insights on their performance. As a result, we support the broadest spectrum of test queries and conduct the evaluations on both synthetic and real data at a scale much larger than what has been considered so far. Finally, we materialized our evaluation framework in an open-source suite that can be easily extended with new datasets, systems, or queries.

Summary of findings. The results of this work (see Section 5.5) (i) substantiate the intuition that there is no one system that perform best over the others, yet (ii) only few systems provide good performance in general. In particular, (iii) we see in practice the difference in performance between hybrid-systems and pure system, i.e., systems that provide graph functionality by applying a graph processing layer on top of another non-graph-based system and systems that instead have been built specifically for graph data management. Moreover, (iv) we conclude that in most graph database systems there are currently many opportunities of improvement in their implementation of the Gremlin query language [Rod15], which is currently the only common language that all of them are able to process. Finally, (v) we demonstrate in practice the ability of our micro-benchmarking approach to effectively lead to the identification of under-performing operators and to identify more precisely the limitations of the systems under scrutiny.

5.1 Contributions

To provide a deep understanding of graph database technologies, it is important to identify an exhaustive set of operations to put under scrutiny, and also to understand how these relate to the internal structure of the system. Hence, the specific contributions of this work are the following:

- We survey the most well-known graph database systems, both old and new, the features that each one provides, and highlight the implementation choices that characterize them (Section 5.2);

- Based on a systematic study of the existing literature, we provide an extensive list of fundamental primitive operations (queries) that graph databases should support (Section 5.3);

- We introduce an exhaustive experimental evaluation methodology for graph databases, driven by the micro-benchmarking model. The methodology consists of queries identified previously and a number of synthetic and real-world datasets at different scales, complexity, and other characteristics (Section 5.4). For fairness across systems, the methodology adopts a standard application interface, the Gremlin query language, which allows the testing of each system using the same set of operations;

- We materialize the methodology into a testing suite based on software containers, which is able to automate the installation and investigation of different graph databases. The suite allows for future extensions with additional tests, and is available online as open source, alongside a number of datasets;

- We apply this methodology on the state-of-the-art graph databases that are available today, and study the effect that different real and synthetic datasets, from co-citation, biological, knowledge base, and social network domains has on different queries (Section 5.5), along with a report on our experience with the set-up, loading, and testing of each system.

We note that the focus of this study is on single machine installations, even though some systems may support clusters, since single-machine installations are still highly popular [GGL+13]. Our goal was, as a first step, to understand how the graph databases perform in a single-machine installation. The question about which system is able to scale-out better, may only come after the understanding of its inherent performance [MIM15, RND+12].

5.2 Graph Databases

In the previous chapters we have dealt with edge-labeled graphs, i.e., graphs that are characterized by the presence of different edge types. In this chapter we consider an even richer model, where we still have edge labels, but also every node and edge has a

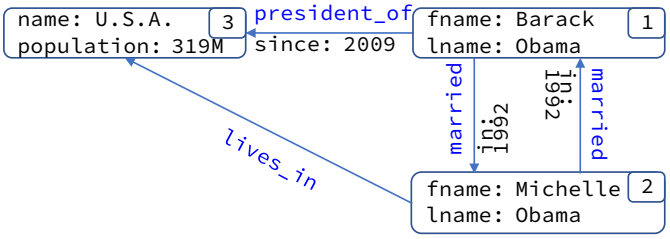

Figure 5.1: A portion of graph data

set of attributes that describes its characteristic properties. This model is referred to as the *property graph model*.

Formally, we axiomatically assume the existence of an infinite set of names $\mathcal{N}$ and an infinite set of values $\mathcal{A}$. A *property* is an element in the set $\mathcal{N}\times\mathcal{A}$ of name-value pairs.

A *property graph* is then a tuple $G=\langle V, E, l, p\rangle$ where V is a set of nodes, E is a set of edges between them, i.e., $E\subseteq V\times V$, $l{:}E\to\mathcal{N}$ is an edge labeling function, and $p{:}\{V\cup E\}\to 2^{\mathcal{N}\times\mathcal{A}}$ is a property assignment function on edges and nodes.

Note that the above model allows different nodes to have exactly the same properties, and different edges to have exactly the same label and set of properties. To be able to distinguish the different nodes or edges, systems extend the implementation of the above model by means of unique identifiers. In particular, they consider a countable set $\mathcal{O}$ of unique values and a function $id{:}\{V\cup E\}\to\mathcal{O}$ that assigns to each node and edge a unique value as its identifier. Furthermore, the nodes and edges, as fundamental building blocks of graph data, are typically implemented as atomic *objects* in the systems and are referred to as such.

Figure 5.1 illustrates a portion of graph data in this model. The annotations containing the colon symbol " **:** " are the properties, while the others are the labels. The number on each node indicates its identifier. For presentation reasons we have omitted the ids on the edges.

5.2.1 Systems

For a fair comparison we need all systems to support a common access method. Tinkerpop [tin], an open source, vendor-agnostic, graph computing framework, is currently the only common interface in most graph databases. TinkerPop-enabled system are able to

process a common query language: the Gremlin language. Thus, we chose systems that support some version of it through officially recognized implementations. Furthermore, we consider systems with a licence that permits the publication of experimental comparisons, and also those that were made available to us to run on our server without any fee. Table 5.1 summarizes the characteristics of the systems we consider in our study. Among others, we show the query languages that these systems support (other than Gremlin). We would've also included GraphDB [gra] and InfiniteGraph [inf], but licensing issues of the first did not allow us to publish any performance verification results, while the second has been discontinued.

Table 5.1: Features and Characteristics of the tested systems

System	Type	Storage	Edge Traversal	Gremlin	Query Execution	Access	Languages
ArangoDB (2.8)	Hybrid (Document)	Serialized JSON	Hash Index	v2.6	AQL, Non-optimized	REST (V8 Server)	AQL, Javascript
BlazeGraph (2.1.4)	Hybrid (RDF)	RDF statements	B+Tree	v3.2	Programming API, Non-optimized	embedded, REST	Java, SPARQL
Neo4J (1.9, 3.0)	Native	Linked Fixed-Size records	Direct Pointer	v2.6 / v3.2	Programming API, Non-optimized	embedded, WebSocket, REST	Java, Cypher,
OrientDB (2.2)	Native	Linked Records	2-hop Pointer	v2.6	Mixed, Mixed	embedded, WebSocket,REST	Java, SQL-like
Sparksee (5.1)	Native	Indexed Bitmaps	B+Tree/Bitmap	v2.6	Programming API, Non-optimized	embedded	Java, C++,Python, .NET
SQLG (1.2) / Postgres (9.6)	Hybrid (Relational)	Tables	Table Join	v3.2	SQL, Optimized(*)	embedded (JDBC)	Java
Titan (0.5, 1.0)	Hybrid (Columnar)	Vertex-Indexed Adjacency List	Row-Key Index	v2.6 / v3.0	Programming API, Optimized	embedded, REST	Java

ArangoDB. ArangoDB [ara] is a multi-model database. This means that it can work as a document store, a key/value store and a graph database, all at the same time. With this model, objects like nodes, edges or documents, are treated the same and stored into special structures called collections. Apart from Gremlin, it supports its own query language, called AQL, *ArangoDB Query Language*, which is an SQL like dialect that supports multiple data models with single document operations, graph traversals, joins, and transactions. The core, which is open-source (Apache License 2.0), is written in C++, and is integrated with the V8 JavaScript Engine (`github.com/v8/v8`). That means that it can run user-defined JavaScript code, which will be compiled to native code by V8 on the fly, while AQL primitives are written in C++ and will be executed as such. Nonetheless, the supported way of interacting with the database system is via REST API and HTTP calls, meaning that there is no direct way to embed the server within an application, and that every query will go through a TCP connection.

It supports ACID transactions by storing data modification operations in a write-ahead log, which is a sequence of append-only files containing every write operations executed on the server. While ArangoDB automatically indexes some system attributes (i.e., internal node identifiers), users can also create additional custom indexes. As a consequence, every collection (documents, nodes or edges) has a default primary index, which is an unsorted hash index on object identifiers, and, as such, it can be used neither for non-equality range queries nor for sorting. Furthermore, there exists a default edge index providing for every edge quick access to its source and destination. ArangoDB can serve multiple requests in parallel and supports horizontal scale-out with a cluster deployment using Apache Mesos [mes].

BlazeGraph. Blazegraph [bla] is open-source and available under GPLv2 or under a commercial licence. It is an RDF-oriented graph database entirely written in Java. Other than Gremlin, it supports SPARQL 1.1, storage and querying of reified statements, and graph analytics.

Storage is provided through a journal file with support for index management against a single backing store, which scales up to $50B$ triples or quads on a single machine. Full text indexing and search facility are built using a key-range partitioned distributed B+Tree architecture. The database can also be deployed in different modes of replication or distribution. One of them is the federated option that implements a scale-out

architecture, using dynamically partitioned indexes to distribute the data across a cluster. While updates on the journal and the replication cluster are ACID, updates on the federation are *shard-wise ACID*. Blazegraph uses Multi-Version Concurrency Control (MVCC) for transactions. Transactions are validated upon commit using a unique timestamp for each commit point and transaction. If there is a write-write conflict the transaction aborts. It can operate as an embedded database, or in a client-server architecture using a REST API and a SPARQL end-point.

Neo4J. Neo4j [neo] is a database system implemented in Java and distributed under both an open source and commercial licence. It provides its own unique language called Cypher, and supports also Gremlin, and native Java API. It employs a custom disk-based native storage engine where nodes, relationships, and properties are stored separately on disk. Dynamic pointer compression expands the available address space as needed, allowing the storage of graphs of any size in its latest version. Full ACID transactions are supported through a write-ahead log. A lock manager applies locks on the database objects that are altered during the transaction.

Neo4j has in place a mechanism for fast access to nodes and edges that is based on IDs. The IDs are basically offsets in one of the store files. Hence, upon the deletion of nodes, the IDs can be reclaimed for new objects. It also supports *schema indexes* on nodes, labels and property values. Finally, it supports full text indexes that are enabled by an external indexing engine (Apache Lucene [luc]), which also allows nodes and edges to be viewed and indexed as "key:value" pairs. Other Neo4J features include replication modes and federation for high-availability scenarios, causal cluster, block device support, and compiled runtime.

OrientDB. OrientDB [ori] is a multi-model database, supporting graph, document, key/value, and object data models. It is written in Java and is available under the Apache licence or a Commercial licence. Its multi-model features Object-Oriented concepts with a distinction for classes, documents, document fields, and links. For graph data, a node is a document, and an edge is mapped to a link. Various approaches are provided for interacting with OrientDB, from the native Java API (both document-oriented and graph-oriented), to Gremlin, and extended SQL, which is a SQL-like query language.

OrientDB features 3 storage types: (i) *plocal*, which is a persistent disk-based storage accessed by the same JVM process that runs the queries; (ii) *remote*, which is a network access to a remote storage; and (iii) *memory-based*, where all data is stored into main memory. The disk based storage (also called Paginated Local Storage) uses a page model and a disk cache. The main components on disk are files called *clusters*. A cluster is a logical portion of disk space where OrientDB stores record data, and each cluster is split into pages, so that each operation is atomic at page level. As we will discuss later (Section 5.5), the peculiar implementation of this system provides a good performance boost but poses an important limitation to the storing of edge labels.

OrientDB supports ACID transactions through a write ahead log and a *Multiversion Concurrency Control* system where the system keeps the transactions on the client RAM. This means that the size of a transaction is bounded by the JVM available memory. OrientDB also implements SB−Tree indexes (based on B-Trees), hash indexes, and Lucene full text indexes. The system can be deployed with a client-server architecture in a multi-master distributed cluster.

Sqlg/Postgresql. Sqlg [Mar] is an implementation of Apache TinkerPop on a relational DBMS. Postgresql [PS+08] is one among those RDBMS supported, and the one we chose for our experiments. Sqlg provides Java API to the gremlin language, and the underlying implementation maps graph semantics to that of the RDBMS. It is possible to also send standard SQL queries directly to the back-end relational database, although this is not often convenient. For graph data, a vertex label is modeled by a table, containing all vertices with that label, and all the vertex's properties. An edge label is modeled as a many-to-many join-table between the vertices, Similarly to vertices, edge labels are mapped to tables. containing the vertex ID of the two edge endpoints alongside the edge properties. Indexes, transactions and parallelization are inherited from the underlying database system.

Sparksee. Sparksee [spa, MBALMM+12], formerly known as DEX [MBGVEC11], is a commercial system written in C++ optimized for out-of-core operations. It provides a native API for Java, C++, Python and .NET platforms, but it does not implement any other query language apart from Gremlin.

It is specifically designed for labeled and attributed multi-graphs. Each vertex and each edge are distinguished by permanent object identifiers. The graph is then split into multiple lists of pairs and the storage of both the structure and the data is partitioned into different clusters of bitmaps for a compact representation. This data organization allows for more than 100 billion vertices and edges to be handled by a single machine. Bitmap clusters are stored in sorted tree structures that are paired with binary logic operations to speedup insertion, removal, and search operations.

Sparksee supports ACID transaction with a *N-readers* and $1-writer$ model, enabling multiple read transactions with each write transaction being executed exclusively. Both search and unique indexes are supported for node and edge attributes. In addition a specific neighbor index can also be defined to improve certain traversal operations. Finally, Sparksee provides horizontal scaling, enabling several slave databases to work as replicas of a single master instance.

Titan. Titan [tit] is available under the Apache 2 license. The main part of the system handles data modeling, and query execution, while the data-persistence is handled by a third-party storage and indexing engine to be plugged into it. For storage, it can support an in-memory storage engine (not intended for production use), Cassandra [cas], HBase [hba], and BerkeleyDB [ber]. To store the graph data, Titan adopts the adjacency list format, where each vertex is stored alongside the list of incident edges. In addition, each vertex property is an entry in the vertex record. Titan adopts Gremlin as its only query language, and Java as the only compatible programming interface. No ACID transactions are supported in general, but are left to the storage layer that is used. Among the three available storage backends only Berkeley DB supports them. Cassandra and HBase provide no serializable isolation, and no multi-row atomic writes.

Titan supports two types of indexes: *graph centric* and *vertex centric*. A graph index is a global structure over the entire graph that facilitates efficient retrieval of vertices or edges by their properties. It supports equality, range, and full-text search. A Vertex-centric index, on the other hand, is local to each specific vertex, and is created based on the label of the adjacent edges and on the properties of the vertex. It is used to speed up edge traversal and filtering, and supports only equality and range search. For more complex indexing external engines like Apache Lucene or ElasticSearch [ela] can

be used. Due to the ability of Cassandra and HBase to work on a cluster, Titan can also support the same level of parallelization in storage and processing.

5.2.2 System Architectures and Query Processing

There are two ways to implement a graph database. One is to build it from scratch (*native* databases) and the other to achieve the required functionalities through other existing systems (*hybrid* databases). In both cases the two challenges to solve are how to store the data and how to traverse these stored structures.

5.2.2.1 Native System Architectures

For data storage, a common design principle is to separate information about the graph structure (nodes and edges) from other they may have, e.g., attribute values, to speed-up traversal operations.

Neo4J has one file for node records, one file for edge records, one file for labels and types, and one file for attributes. **OrientDB** stores information about nodes, edges and attributes similarly, in distinct records. In both systems, node and edge records contain pointers to other edges and nodes, and also to types and attributes, but the organization is different in the two systems. In Neo4J nodes and edges are stored as records of fixed size and have unique IDs that correspond to the offset of their position within the corresponding file. In this way, given the id of an edge, it is retrieved by multiplying the record size by its id, and reading bytes at that offset in the corresponding file. Moreover, being records of fixed size, each node record points only to the first edge in a doubly-linked list, and the other edges are retrieved by following such links. A similar approach is used for attributes. In OrientDB, on the other hand, record IDs values are not linked directly to a physical position, but point to an append-only data structure, where the logical identifier is mapped to a physical position. This allows for changing the physical position of an object without changing its identifier. In both cases, given an edge, to obtain its source and destination requires constant time operations, and inspecting all edges incident on a node, hence visiting the neighbors of a node, has a cost that depends on the node degree and not on the size of the graph.

Sparksee decomposes data into separate data-structures: one structure for objects, which refers to both nodes and edges, two for relationships which describe which nodes and edges are linked to each other, and a data-structure for each attribute name. Each of these data-structures is in turn composed by a map from keys to values, and a bitmap for each value [MBALMM+12]. In each data-structure objects are identified by IDs generate sequentially, and each ID is linked as key trough the map to one single value. In particular, in the data-structure for objects (i.e., containing both nodes and edges), each ID links to a label or a type, so given a node (or an edge) an index points to its type (or label). Also, each value links to a bitmap, where each bit corresponds to an object ID, and the bit is set if that object has that value. Given a label, one can scan the corresponding bitmap to identify which edges share the same label. For relationships, one data-structure links edge-IDs to their head (the ID of the destination node) and the other data structure to their tail (the source). Furthermore, each bitmap identifies all edges incident to a node. For the attributes a similar mechanism is used. The main advantage of this organization is that many operations become bitwise operations on bitmaps. Key-value maps and bitmaps are organized in B+Trees, and graph structure navigation is quite fast, although it has no constant time guarantees.

5.2.2.2 Hybrid System Architectures

ArangoDB is based on a document store. Each document is represented as a self contained JSON object (serialized in a compressed binary format). To implement the graph model, ArangoDB materialize JSON objects for each node and edge. Each object contains links to the other objects to which it is connected, e.g., a node lists all the IDs of incident edges. A specialized hash index is in place, in order to retrieve the source and destination nodes for an edge. **BlazeGraph** is an RDF database and stores all information into Subject-Predicate-Object (SPO) triples. Each statement is indexed three times by changing the order of the values in each triple, i.e., a B+Tree is built for each one of SPO, POS, OSP. BlazeGraph stores attributes for edges as reified statements, i.e., each edge can assume the role of a subject in a statement. Hence, traversing the structure of the graph may require more than one accesses to the corresponding B+Tree. In **Sqlg** the graph structure consists of one table for each edge type, and one table for each node type. Each node and edge is identified by a unique ID, and connections between nodes and edges are retrieved through joins. The limitation of this approach

is that unions and joins are required even for retrieving the incident edges of a node. Finally, **Titan** represents the graph as a collection of adjacency lists. With this model the system generates a row for each node, and then one column for each node attribute and each edge. Hence, for each edge traversal, it needs to access the node (row) ID index first.

5.2.2.3 Query Processing and Evaluation

All the systems we considered support Gremlin. A Gremlin query is a series of operations. Consider, for instance, query 28 in Table 5.2, which selects nodes with at least k incoming edges. It first filters nodes (`g.V.filter`$\{\dots\}$) and then the incoming edges are counted (`it.inE.count()`) for every node in the output of the filter. In **ArangoDB** each step is converted into an AQL query and sent to the server for execution, so the above Gremlin query will be executed as a series of two independent AQL queries implementing its two parts. ArangoDB does not provide any overall optimization of these parts. Note that Gremlin is a touring-complete language and can describe complex operations that declarative languages, like AQL or Cypher, may not be able to express in one query. The only other query system that translates all operations to a declarative query language is **Sqlg**. Where possible, the system tries to conflate operators in a single query, which is some form of query optimization. All the other systems translate Gremlin queries directly into a sequence of low-level operators with direct access to their programming API, evaluate every operator, and pass the result to the next in the sequence. In **OrientDB**, in particular, some consequent operators may get translated into queries and then the processed with the programming API, resulting in some form of optimization for a part of the query. **Titan**, which has Gremlin as the only supported query language, features also some optimization during query processing.

5.3 Queries

To generate the set of queries to run on the systems we follow a micro-benchmark approach [BD84]. The list is the results of an extensive study of the literature and of many practical scenarios. Of the many complex situations we found, we identified the very basic operations of which they were composed. We eliminated repetitions and

Table 5.2: Test Queries by Category (in Gremlin 2.6)

#	Query	Description	Cat
1.	g.loadGraphSON("/path")	Load dataset into the graph 'g'	**L**
2.	g.addVertex(p[])	Create new node with properties p	**C**
3.	g.addEdge(v1 , v2 , l)	Add edge l from $v1$ to $v2$	
4.	g.addEdge(v1 , v2 , l , p[])	*Same as Q.3*, but with properties p	
5.	v.setProperty(Name, Value)	Add property *Name*=*Value* to node v	
6.	e.setProperty(Name, Value)	Add property *Name*=*Value* to edge e	
7.	g.addVertex(...); g.addEdge(...)	Add a new node, and then edges to it	
8.	g.V.count()	Total number of nodes	**R**
9.	g.E.count()	Total number of edges	
10.	g.E.label.dedup()	Existing edge labels (no duplicates)	
11.	g.V.has(Name, Value)	Nodes with property *Name*=*Value*	
12.	g.E.has(Name, Value)	Edges with property *Name*=*Value*	
13.	g.E.has('label',l)	Edges with label l	
14.	g.V(id)	The node with identifier *id*	
15.	g.E(id)	The edge with identifier *id*	
16.	v.setProperty(Name, Value)	Update property *Name* for vertex v	**U**
17.	e.setProperty(Name, Value)	Update property *Name* for edge e	
18.	g.removeVertex(id)	Delete node identified by *id*	**D**
19.	g.removeEdge(id)	Delete edge identified by *id*	
20.	v.removeProperty(Name)	Remove node property *Name* from v	
21.	e.removeProperty(Name)	Remove edge property *Name* from e	
22.	v.in()	Nodes adjacent to v via incoming edges	**T**
23.	v.out()	Nodes adjacent to v via outgoing edges	
24.	v.both('l')	Nodes adjacent to v via edges labeled l	
25.	v.inE.label.dedup()	Labels of in coming edges of v (no dupl.)	
26.	v.outE.label.dedup()	Labels of outgoing edges of v (no dupl.)	
27.	v.bothE.label.dedup()	Labels of edges of v (no dupl.)	
28.	g.V.filter{it.inE.count()¿=k}	Nodes of at least k-incoming-degree	
29.	g.V.filter{it.outE.count()¿=k}	Nodes of at least k-outgoing-degree	
30.	g.V.filter{it.bothE.count()¿=k}	Nodes of at least k-degree	
31.	g.V.out.dedup()	Nodes having an *incoming* edge	
32.	v.as('i').both().except(vs) .store(j).loop('i')	Nodes reached via breadth-First traversal from v	
33.	v.as('i').both(*ls).except(j) .store(vs).loop('i')	Nodes reached via breadth-First traversal from v on labels ls	
34.	v1.as('i').both().except(j).store(j) .loop('i'){!it.object.equals(v2)} .retain([v2]).path()	Unweighted Shortest Path from $v1$ to $v2$	
35.	*Shortest Path on 'l'*	*Same as Q.34*, but only following label l	

* [] denotes a Hash Map; g is the graph; v and e are node/edges.

ended up with a set of common operations that are independent from the schema and the semantics of the underlying data, hence, they enjoy a generic applicability.

In the query list we consider different types of operations. We consider all the "CRUD" kinds, i.e., **C**reations, **R**eads, **U**pdates, **D**eletions, for nodes, edges, their labels, and for their properties. Specifically for the creation, we treat separately the case of the initial

loading of the dataset from the individual object creations. The reason is because the first happens in bulk mode on an empty instance, while the second at run time with data already in the database. We also consider *traversal* operations across nodes and edges, which is characteristic in graph databases. Recall that operations like finding the centrality, or computing strongly connected components are for graph analytic systems and not typical in a graph database. The categorization we follow is aligned to the one found in other similar works [KSM13, JV13, Ang12] and benchmarks [EALP+15]. The complete list of queries can be found in Table 5.2 and is analytically presented next. The syntax is for Gremlin 2.6, but the syntax for gremlin version 3 is quite similar.

5.3.1 Load Operations

Data loading is a fundamental operation. Given the size of modern datasets, understanding the speed of this operation is crucial for the evaluation of a system. The specific operator (Query 1) reads the graph data from GraphSON[1] file. In general it's bound to the speed with which objects are inserted, which will be affected by any index in place and any other consistency check. In some cases GDBs have in place special methods or configurations to allow bulk loading, e.g., to deactivate indexing, but in general they are vendor specific, i.e., not found in the Gremlin specifications.

5.3.2 Create Operations

The first category of operations (**C**) includes operators that create new structures in the database. In this group we consider anything that generates new data-entries. Creation operators may be for nodes, edges, or even properties on existing nodes or edges. Often, to create a complex object, e.g., a node with a number of connections to existing nodes, many different such operators may have to be called. Among the others, we also consider a special composite workload where we first insert a node and then also a set of edges connecting it to other existing nodes in the graph.

Insert Node (Query 2) The operator creates a new node in the database with the set of properties that are provided as argument, but without any connection (edges) to other nodes.

[1] A JSON-based format `tinkerpop.apache.org/docs/current/reference/#graphson-io-format`

Insert Edge (Queries 3, and 4) This operator creates a new edge in the graph between the two nodes specified as arguments, and with the provided label. In a second version, the operator can also take a set of properties as additional argument. In the experiments performed we randomly select nodes among those in the graph, we choose a fresh value as label, and a custom property name and value pair.

Insert Property (Queries 5, and 6) These two operators test the addition of a new property to a specific node and to a specific edge, respectively. The node (or the edge) is explicitly stated, i.e., referred, through its unique id, and, there is no search task involved since the lookup for the object with the specific identifier is performed before the time is measured. In this case the operation are applied directly to the node and edge (`v` and `e`).

Insert Node with Edges (Query 7) This operation requires the insertion of a new node, alongside a number of edges that connect it to other nodes already existing in the database.

5.3.3 Read Operations

The category of read operations comprises queries that locate and access some part of the graph data stored in the system that satisfy certain conditions. Sometimes, such part may end up being the entire graph.

Graph Statistics (Queries 8, 9, and 10) Many operations often require a scan over the entire graph datasets. Among the queries of this type, three operators were included in the query evaluation set. One that scans and counts all the nodes, one that does the same for all edges, and one that counts the unique labels of the edges. The goal of the last operation is also to stress-test the ability of the system to maintain intermediate information in memory, since it requires to eliminate duplicated before reporting the results.

Search by Property (Queries 11, and 12) These two queries are typical selections. They identify nodes (or edges) that have a specific property. The name and the value of the property are provided as arguments. There may be a unique object satisfying the condition of having the specific property, or there may be more than one.

Search by Label (Query 13) The search by label task is similar to the search by property, but has only one operator since labels are only on edges. Labels are fundamental components of almost every graph dataset, and this is probably the reason why the syntax in Gremlin 3.x distinguishes between labels and properties with a special provision, while in 2.6, they were treated equally. In a graph database edge labels have a primary role, also usually, labels are not optional and are immutable, hence searching edges based on a specific label should receive special attention.

Search by Id (Queries 14, and 15) As it happens in almost any other kind of database, a fundamental search operation is the one done by reference to a key, i.e., ID. Those are *system defined*, and in some cases based on the internal data organization of the system. These two queries have been included, to retrieve a node and an edge via their unique identifier.

5.3.4 Update Operations

Data update operators are typical of dynamic data, and graph data is no exception. Since edges are first class citizens of the system, an update of the structure of the graph, i.e., on the connectivity of two or more nodes, requires either the creation of new edges or deletion of existing. In contrast, updates on the properties of the objects are possible without deletion/insertion, as it happens in other similar databases. Thus, we have included Queries 16, and 17 to test the ability of a system to change the value of a property of a specific node or an edge. In this case, as above, we do not consider the time required to first retrieve the object to be updated.

5.3.5 Delete Operations

To test how easily and efficiently data can be removed from a graph database, we included three types of deletions: one for a node, one for an edge and one for a property.

Delete Node (Query 18) Deleting a specific node requires the elimination of all its properties, all its edges, as well as the node itself. It may result to a very costly operation when many different data-structures are involved.

Delete Edge (Query 19) Similarly to the node case, deleting an edge requires the prior removal of its properties. This operation is probably one of the most common delete operations in continuously evolving graphs.

Delete Property (Queries 20, and 21) The last two queries eliminate a property from an node or an edge, respectively. As the structure of a node or edge is not fixed, it may happen that either element lose a property.

5.3.6 Traversals

The ability to conveniently perform traversal operations is one of the main reason why graph models are preferred to others. A traversal means moving across different nodes that are connected in a consecutive fashion through edges.

Direct Neighbors (Queries 22, 23) A popular operation is the one that, given a node, retrieves those directly reachable from it (1-hop), i.e., those that can be found by following either an incoming or an outgoing edge.

Filter Direct Neighbors (Query 24) The specific query performs a traversal of only one hop, and for edges having a specific label. The reason why it is considered separately from other traversals is because it is very frequent and involves no recursion, and as such, it is often subject to separate efficient implementation by the various systems.

Node Edge-Labels (Queries 25, 26, and 27) Given a node, there is often the need to know the labels of the incoming, outgoing, or both edges. These three kinds of retrieval is exactly what this set of three queries perform, respectively.

K-Degree Search (Queries 28, 29, 30, and 31) For many real application scenarios there is a need to identify nodes with many connections, i.e., edges, since this is an indicator of the importance of a node. The number of edges a node has is called the degree of the node, and nodes with high degree are usually hubs in the network. The first three queries identify and retrieve nodes with at least k edges. They differ from each other in considering only incoming edges, only outgoing, or both. The fourth query identifies nodes with at least one incoming edge and is often used when a hierarchy needs to be retrieved.

Breadth-First Search (Queries 32, and 33) A number of search operations give preference to nodes found in close proximity, and they are better implemented with a breadth-first search from a given node. This ability is tested with these two queries, with the second being a special case of the first that considers only edges with a specific label.

Shortest Path (Queries 34, and 35) Another traditional operation on graph data is the identification of the path between two nodes that contain the smallest number of edges. For this we included these two queries, with the second query being a special case of the first that considers only edges with a specific label. In particular, given an unweighted graph, they determine the shortest path between two nodes via a BFS-like traversal.

5.3.7 Complex Query Set

In order to test the ability of the systems to optimize complex queries, i.e., collectively optimize multiple primitive operators, we also created a workload of 12 queries based on queries provided by the LDBC Social Network benchmark [EALP+15]. The queries mimic the tasks carried out for a new user in the system, from the point of creating an account (creating a new node with attributes) and filling up her profile (connecting to nodes representing the school, the place of birth and the workplace), to the point of making recommendations of topics and other users. For these operations there are queries in the workload with composition of multiple primitive operators, multiple joins predicates, group by, sorting, max finding, and top-k. Since these queries are heavily dependent on the schema of the dataset, they can be run solely on the *ldbc* dataset presented below (Section 5.4).

Max-search To add a new user we should assign an unique identifier to it. In the *ldbc* dataset objects have two different properties, one is '*oid*' and the other is the '*iid*', the first is a string the second is an integer. Although in real applications this is handled different, here we search for the maximum value for both (queries '*max-iid*' and '*max-oid*'), and the we will increment these values and use them when creating a new node for a user.

User Creation We create a new node for a user, we take all the attributes that a user has and attach those attribute to the node created. We will also assign to it the two values for '*iid*' and '*oid*' obtained previously.

User Profile Once the node for the user is created, we connect it to other nodes signifying some personal details. In particular we issue a query for finding places, companies and universities with name starting with some combination of letters. For each of those we take the first in alphabetical order and add an edge between the node of the user and the node of the place. Those represent the city where she lives, the company where she works at, and the university in which she studied.

User Friends We also search and add as friends other existing users. Users to be added as friends (i.e., connected via an edge labeled 'knows') are selected based on a selection on their first name ('*friend1*'), and on a selection based on both first and last name ('*friend2*'). All results are sorted in ascending order based on the name and only the top-10 results are returned.

Recommending Tags Tags are similar to labeled topics, and users are connected to the nodes representing the topics they like. We first select among the tags liked by the user friend, the top 10 tags with highest number of likes ('*friend-tags*'), and then connect those tags to the user ('*add-tags*').

Friend Search and Recommendation The last part of the workload explored the neighborhood of a user node searching for friends. In the first query ('*friend-of-friend*') we find up to 10 people with a given first name that the user is connected to by at most 3 steps via 'knows' relationships. For the retrieved persons we return their personal information, including workplaces and places of study. The retrieved persons are sorted by their distance from the user.

In the second query we recommend instead friend of friends, that are not already friend of the user, simulating in this way a first iteration of triangle closure.

5.4 Evaluation Methodology

Fairness, reproducibility, and extensibility have been three fundamental principles in our evaluation of the different systems. In particular, a common query language and input format for the data has been adopted for all the systems. For the query executions, it has been ensured that they have been performed in isolation so that they have not been affected by external factors. Any random selection made in one system (e.g., a random

selection of a node in order to query it) has been maintained the same across the other systems. Furthermore, all experiments have been performed on the same machine to avoid any effect caused by hardware variations. The goal is to perform a comparative evaluation and not an evaluation in absolute terms. Both real and synthetic datasets have been used, especially on large volumes in order for the experiments to be able to highlight the differences across the systems. Finally, our evaluation methodology has been materialized in a test suite and is available on-line [LBV17] It contains scripts, data and queries, and is extensible to new systems and queries.

5.4.1 Common Query Language.

We have opted for a common query language across all the systems to ensure that the semantics of the queries we run are interpreted in the same way by the different systems. In particular, we selected as application layer the Apache TinkerPop[tin] framework and the expressive query language Gremlin [Rod15], which is the most supported language across graph databases. In the context of graph databases, TinkerPop acts as a database-independent connectivity layer, while Gremlin is the analogous to SQL in relational databases [HP13]. All the graph databases we tested have adapters for Gremlin already implemented and supported by the various database vendors.

5.4.2 Software Containers.

To ensure full control over the environment in which each system runs, and to facilitate reproducibility, we opted for installing and running each graph database within a dedicated software container [Boe15]. A popular solution is Docker [doc], an open source software that creates a level of "soft" virtualization of the operating system, which allows an application within the environment to access machine resources directly without the overhead of interacting with an actual virtual machine. Furthermore, thanks to the so called *overlay file-system* (AUFS [Oka]), it is possible to create a "snapshot" of a system and it's files, and then share the entire computational environment. This allows the sharing of our one-click installation scripts for all the databases and our testing environment, so that the experiments can be repeated elsewhere.

5.4.3 Hardware.

For the experiments we used a machine with a 24-core CPU, an Intel Xeon E5-2420, 1.90GHz processor, 128 GB of RAM, 2TB hard disk with 20000 rpm, Ubuntu 14.04.4 operating system, and with Docker 1.13, configured to use AUFS on *ext4*. Each graph database was configured to be free to use all the available machine resources, e.g., for the JVM we used the option -Xmx120GB. For other parameters we used the settings recommended by the vendor. The latter applies also to Apache Cassandra that was serving as the back-end for Titan.

5.4.4 Evaluation Approach.

The Gremlin queries are called for execution via Groovy[2] scripts. For the systems supporting different major versions of Gremlin, we tested both. The reason is that since the latest version came out recently, we would like to illustrate the difference in performance that has been achieved and help stakeholders having an old system in in operation to decide whether it is worth the extra step of upgrading them. Furthermore, the difference between the versions illustrates the space for improvement that exists, an area that our current work can help significantly.

All system were tested using the *embedded mode*, where direct Java calls are sent to the system, and the application runs within the JVM of the engine. The only exception was ArangoDB and Sqlg. They may receive Gremlin commands through the Java API, but in the back-end perform REST and/or JDBC calls to the underlying storage engine. Nonetheless, since all storage systems operate locally, there is no delay due to network routing or latency.

Note that Gremlin has no specification for indexes. Some systems create indexes automatically in a way agnostic to the user while others require explicit commands in their own language. We considered both the default behavior of not taking any action and letting the system go with its default indexing policy, and the case of explicitly creating the needed indexes.

In the case of queries with parameters, for fair comparisons, the parameter values are decided in advance and kept the same for all the systems. For instance, query 14 needs

[2] A superset of Java: `groovy-lang.org`

the ID of the node to retrieve. If a different node is retrieved in every system, then it wont be possible to compare them. For this reason, when we need to evaluate a query, we first decide the parameters to use and then start the executions on the different systems. The same applies on queries that need to start from a node or an edge, e.g. query 22 needs to know the node v. For these queries, the node is decided first and then the query is run for that same node in all the systems. Naturally, the time needed to identify the node (or edge) that will be used in the query and retrieve its id, is not part of the time reported as execution time for the respective queries. A similar approach is followed also for the multi-fold evaluation. When we perform k runs of the same query on the same system and dataset (to select the average response time as the indicative performance), we first select k parameter values, nodes, or edges to query (usually through random sampling), and then perform each of the k runs with one of these k parameters (or nodes, or edges). Here each query is executed k =10 times.

In the scalability studies of queries 11 and 12 that are performing selection based on a property value, it is important that the performance variation observed when running the same query on datasets of different sizes is due to the size of the data and not due to the cardinality variation of the answer set. To achieve this goal, we select to use properties that not only exist in all the datasets of different sizes, but also have the same cardinality in all of them. In case such properties do not exist, we create and assign at loading time two different properties with predefined names to 10 random edges and 10 random nodes in each dataset and then use these property names for queries 11 and 12. (The different case of the same type of query run on the same dataset producing results of different cardinalities is covered by the different parameter values that are decided in the k-fold experiments.)

Unfortunately, almost all the databases, when loading the data, create and assign their own internal identifiers. This creates a problem when we later need to run across all the systems the same query that is using the identifier as a parameter. For this reason, when loading the data, we add to every node a property $\langle \text{“}objectID\text{”}, id \rangle$ where the id is the node identifier in the original dataset. As a result, even if the system decides to replace the identifier of the node with an internal one, we still have a way to find the node using the $objectID$ property. So before starting the evaluation of query $g.V(id)$, for instance, on the graph database system S, where id is the node identifier in the original dataset, we first search in the system S and retrieve the internal identifier iid

of the node with the attribute $\langle$"$objectID$", $id\rangle$. Then, we execute the query $g.V(iid)$ instead of the $g.V(id)$, and report its execution time as the time for the evaluation of Q.14.

The k times that a query execution is repeated are performed first in isolation and then in batch mode. For the isolation, we turn the system on, run the single query with one of the parameters, then shut the system off, and reset the file-system. Then repeat again with the next parameter. In this way, each run is unaffected by what has run before. In batch mode, we turn the system on, run the query with the first parameter, then with the second, then the third, and so forth. At the end we shut down the system. The isolation mode makes no sense to be repeated for the queries 8, 9, 10, 28, 29, 30 and 31 since they have no graph-dependent parameters, thus, every isolation mode repetition will be identical to the others. Thus, these queries are evaluated only once in isolation and not in batch. In queries 28, 29 and 30, the k has been considered a threshold and not a parameter, and the fixed value k=50 has been considered throughout the experiments. In total, for every evaluation of a specific system with a specific dataset, 337 query executions are taking place. To these we add the 120 queries from the LDBC benchmark.

5.4.5 Test Suite.

We have materialized the evaluation procedure into a software package (a test suite) and have made it available on-line [LBV17], enabling repeatability and extensibility. The suite contains the scripts for installing and configuring each database in the Docker environment and for loading the datasets. The queries themselves are also contained in the suite. There is also a python script that instantiates the Docker container and provides the parameters required by each query. To test a new query it suffices to write it into a dedicated script, while in order to perform the tests on a new dataset one only needs to place the dataset in GraphSON format in a JSON file in the directory from where the data are loaded.

5.4.6 Datasets.

We have tested our system on both real and synthetic datasets. One dataset (*MiCo*) describes co-authorship information crawled from the CS Microsoft Academic portal [EASK14]. Nodes represent authors while edges represent co-authorship between two authors and have as a label the number of co-authored papers. Another dataset (*Yeast*) is a protein interaction network [BM06]. Nodes represent budding yeast proteins (*S.cerevisiae*) [BZC+03] and have as labels the short name, a long name, a description, and a label based on its putative function class. Edges represent protein-to-protein interactions and have as label the two classes of the proteins involved. A third real dataset is Freebase [Goo14], which is one of the largest knowledge bases nowadays. Nodes represent entities or events, and edges model relationships between them. We have taken the latest snapshot [Lis17, MLVP16b] and have considered four subgraphs of it of different sizes.

Despite the fact that the raw data dump contains 1.9B triples [Goo14], those comprise duplicate, technical, or experimental meta-data and links to other sources that are commonly removed [Bas14, MLVP16b], leaving a clean dataset of 300M facts. The size of the samples were chosen to ensure that all engines had a fair chance to process them in reasonable times, and on the other hand to show the system scalability at levels higher than those of previous works.

One subgraph (*Frb-O*) was created by considering only the nodes related to the topics of organization, business, government, finance, geography and military, alongside their respective edges. Furthermore, we randomly selected 0.1%, 1%, and 10% of the edges from the complete graph, which alongside the nodes at their endpoints created 3 graph datasets, the *Frb-S*, *Frb-M*, and *Frb-L*, respectively.

For a synthetic dataset we used the data generator[3] provided by the Linked Data Benchmark Council[4] (LDBC) [EALP+15] to produce a graph that mimics the characteristics of a real social network with power-law structure, and real-word characteristics like assortativity based on interests or preferences (*ldbc*). The generator was instructed to produce a dataset simulating the activity of 1000 users over a period of 3 years. The *ldbc* is the only dataset with properties on both edges and nodes. The others have properties only on the nodes.

[3] `github.com/ldbc/ldbc_snb_datagen`
[4] `ldbcouncil.org`

Table 5.3: Dataset Characteristics

				Connected Component				Degree		
	|V|	**|E|**	**|L|**	**#**	**Maxim**	**Density**	**Modularity**	**Avg**	**Max**	**Δ**
Yeast	$2.3K$	7.1K	167	101	$2.2K$	$1.34{*}10^{-3}$	$3.66{*}10^{-2}$	6.1	66	11
MiCo	$100K$	1.1M	106	$1.3K$	$93K$	$1.10{*}10^{-6}$	$5.45{*}10^{-3}$	21.6	$1.3K$	23
Frb-O	$1.9M$	4.3M	424	$133K$	$1.6M$	$1.19{*}10^{-6}$	$9.82{*}10^{-1}$	4.3	$92K$	48
Frb-S	$0.5M$	0.3M	1814	$0.16M$	$20K$	$1.20{*}10^{-6}$	$9.91{*}10^{-1}$	1.3	$13K$	4
Frb-M	$4M$	3.1M	2912	$1.1M$	$1.4M$	$1.94{*}10^{-7}$	$7.97{*}10^{-1}$	1.5	$139K$	37
Frb-L	$28.4M$	31.2M	3821	$2M$	$23M$	$3.87{*}10^{-8}$	$2.12{*}10^{-1}$	2.2	$1.4M$	33
ldbc	$184K$	1.5M	15	1	$184K$	$4.43{*}10^{-5}$	0	16.6	$48K$	10

Table 5.3 provides the characteristics of the aforementioned datasets. It reports the number of nodes ($|V|$), edges ($|E|$), labels ($|L|$), connected components (#), the size of the maximum connected component (Maxim), the graph density (Density), the network modularity (Modularity), the average degree of connectivity (Avg), the max degree of connectivity (Max), and the diameter (Δ).

As shown in the table, the *MiCo* and the *Frb* are sparse, while the *ldbc* and *Yeast* are one order of magnitude denser, which reflects their nature. The *ldbc* is the only dataset with a single component, while the *Frb* datasets are the most fragmented. The average and maximum degree are reported because large hubs become bottleneck in traversals.

5.4.7 Evaluation Metrics.

For the evaluation we consider the disk space, the data loading time, the query execution time, but we also comment on the experience with installing and running each system.

5.5 Experimental Results

In the tests we run we noticed that *MiCo* and *ldbc* were giving results similar to the *Frb-M* and *Frb-O*. The *Yeast* was so small that didn't highlight any particular issue, especially when compared to the results of *Frb-S*. We also tried to load the full freebase graph *Frb-F* (with 314M edges and 76M nodes), but only Neo4J, Sparksee, and Sqlg managed to do so without errors, and only Neo4J (v.3.0) to successfully complete all the queries, making it the most scalable. Moreover, the running times recorded on the full dataset respected the general trends witnessed with the other subsamples. Thus, in what follows, we will talk mainly about the results of the *Frb-S*, *Frb-O*, *Frb-M*, and *Frb-L*

and only when there is some different behavior from the others we will be mentioning it. Additional details about the experimental results on the other datasets can be found below (Section 5.5.9).

Regarding the documentation, Neo4J, Sqlg, and OrientDB provide in-depth information. Sparksee, Titan and ArangoDB are limited in some aspects, yet clear for basic installation, configurations and operational needs. The Titan documentation is the less self-contained, especially on how to be configured with Cassandra. Finally, the BlazeGraph documentation is largely outdated.

In terms of system configuration Neo4J doesn't require any specific set-up. OrientDB instead supports a default maximum number of edge labels equal to 32676 divided by the number of CPU cores, and requires disabling a special feature for supporting more. Sqlg has limits on the maximum length of nodes and edge labels (inherited from Postgresql). ArangoDB requires two configurations, one for the engine, and one for the V8 javascript server for logging. With only default values this system generated 40 GB of log files in about 24 hours of activity, with a single active client and it is not able to allocate more than 4GB of memory. For Titan instead the most important configurations are for the JVM Garbage Collection and for the Cassandra backend. Moreover, for large datasets, it is necessary to disable automatic schema creation, and create it manually before data loading.

Finally, the systems based on Java, namely, BlazeGraph, Neo4J, OrientDB and Titan, are sensitive to the JVM garbage collection, especially for very large datasets that require large amount of main-memory. As a general rule, the option `-XX:+UseG1GC` for the *Garbage First* (G1) garbage collector is strongly recommended.

5.5.1 Data Loading

The Task. For many systems, loading the data simply by executing the Gremlin query 1 was causing system failures or was taking days. For OrientDB and ArangoDB we are forced to load the data using their native routines. With Gremlin, ArangoDB sends each node and edge insertion instruction separately to the server via a HTTP call making it prohibitively slow even for small datasets. For OrientDB, limited edge-label storing features and long loading times required us to pass through some server-side

Table 5.4: Evaluation Summary

Task	★★★	★★	★
Load	N1 · N3 · S	A · O · P	T0 · T1 · B
Insertions	S · O · N1 · A	P · N3 · T0 · T1	B
Graph Statistics	S · N3	N1 · O · P	T0 · T1 · A · B
Search by Property	P · N3 · N1	O	S · T0 · T1 · A · B
Search by Label	P · N3 · N1	O · S	T0 · T1 · A · B
Search by Id	S · N1 · O	A · P · N3 ·T0 · T1	B
Updates	S · A · O · N1 · P	N3 · T0 · T1	B
Delete Node	A · N1	S · O · T0 · T1 · N3 · P	B
Other Deletions	S · A · O · N1 · P	T0 · T1 · N3	B
Direct Neighbors	N1 · O · N1	A · S · T0 · T1	B · P
Node Edge-Labels	N1 · O · N3	A · T0 · T1,S	B · P
K-Degree Search	N3 · N1	O · T0 · T1	A · B · P
Breadth-First Search	N3 · O · N1	T0 · T1	A · S · B · P
Shortest Path	N3 · N1	O · T0	A · T1 · S · B · P
SN Queries	P · N3	O · N1 · S · T1	A · T0 · B

A=ArangoDB; B=BlazeGraph; N1=Neo4J (v.1.9); N3=Neo4J (v.3.0); O=OrientDB; S = Sparksee; T0 =Titan (v.0.5); T1 =Titan (v.1.0); P=Sqlg

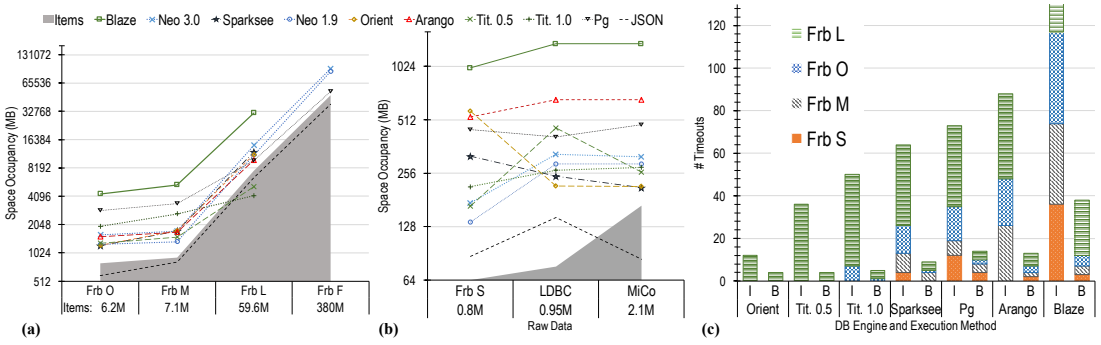

Figure 5.2: Space occupancy on disk required by the systems on the various datasets compared to the size of the original Json file and number of elements in the dataset ((left) and (center)) and number of Time-Outs for Isolation (I) and Batch (B) modes (right)

implementation-specific commands in order to load the datasets. BlazeGraph required the explicit activation of the *bulk loading* feature otherwise we were facing loading times in the order of days. Titan, for any medium to large size dataset requires disabling the automatic schema creation during loading, otherwise its storage back-end (Cassandra) would get swamped with extra consistency check operations. This means that the complete schema of the graph should be known to the system prior to the insertion of the data and is immutable unless implementation specific directives are issued to update it. Sqlg, instead, has a limit on the maximum length of labels (due to Postgresql). In the Gremlin implementation in all other systems those operations are transparent to

the user. As a result, only Neo4J and Sparksee managed the loading through the gremlin API with no issues, and they did so in times comparable to those achieved by the built-in scripts provided by ArangoDB. Consequently, since (for the loading alone) the different systems did not go through exactly the same procedures, discussions regarding the loading times need to be taken with this information in mind.

The time. For the *Yeast*, which is the smallest dataset, loading times vary from a couple of seconds (with ArangoDB) to a minute (with Titan (v.1.0)). With the *Frb-S* dataset, loading times range from 16 seconds (with ArangoDB), 5 minutes (Titan and OrientDB), 16 minutes (BlazeGraph), up to 42 minutes (Sqlg). Titan and OrientDB are the second slowest, requiring around 5 minutes. Neo4J is usually the second fastest in all loadings tasks being only ten seconds slower than ArangoDB. This ranking stays similar with *MiCo* and *ldbc*, with the only exception of Sqlg being much more faster.

Using the *Frb-O*, *Frb-M*, *Frb-L*, we observed that loading time increases proportionally to the number of elements (nodes and edges) within each dataset. With the largest dataset (*Frb-L*) ArangoDB has the fastest loading time (∼19 min) and only Neo4J (v.3.0) is just few minutes slower, followed by Neo4J (v.1.9) (∼38 min), and Sparksee (∼48 min). OrientDB, instead, took almost 3 hours, while both versions of Titan approximately 4.5 hours. BlazeGraph instead took almost 4.45 hours to load *Frb-M* and around 4 days to load *Frb-L*. These tests showed that BlazeGraph, Sqlg and OrientDB, given they internal storage structure, are very sensitive to the edge label cardinality.

The Space. We exploited the docker utilities to measure the disk size occupied by the data in each system. The results are illustrated in Figures 5.2(a) and 5.2(b). For each system, we obtained the size of the docker image with the system installed and its required libraries, then we measured again the size of said image after the data loading step. The difference gives a precise account of all files that the loading phase has generated.

Loading *Yeast*, not reported in figure, leaves the image size almost unchanged for both Neo4J (v.1.9) and Titan (v.0.5), and only 10, 20, 30, 60 and 70MB are added for Neo4J (v.3.0), Sparksee, Titan (v.1.0), Sqlg, and OrientDB, respectively. Instead, ArangoDB generates almost 150MB of additional disk space, and BlazeGraph more than 830MB,

the latter due to the size of journal and automatic-indexing that, when generated, are multiples of a fixed size.

With the *Frb-O* dataset, as Figure 5.2 illustrates, Sparksee, OrientDB, Neo4J (v.1.9), and Titan (v.0.5) are all equally *compact*, with a delta on the disk image size of about 1.2GB. For *Frb-M*, though, Neo4J (v.1.9) and Titan (v.0.5) are equally effective in disk size and a little better than the others, requiring respectively 1.3GB and 1.5GB to store a dataset of 816MB and 7.1 million elements.

Titan (v.1.0) has, on both medium size datasets (*Frb-O* and *Frb-M*), the third worst performance (the worst being BlazeGraph and the second worst Sqlg), with three to four times the space consumption of the original dataset in plain text. Instead, for the *Frb-L*, Titan (v.1.0) scales the best, compressing 6.4GB of raw data into 4.1GB, followed by Titan (v.0.5) taking 5.1GB. The remaining databases are almost equivalent, taking from 10 to 14GB. Exception is the BlazeGraph, on all the datasets, requiring on average three times the size of any other system. This shows that the compression strategy of Titan is the most compact at larger scales.

The comparison between the disk space required by the systems to store *Frb-S*, *MiCo* and *ldbc* (Figure 5.2(b)) reveals a peculiar behavior for Sparksee and OrientDB, where the space occupied on disk is smaller for the two larger datasets. As a matter of fact, the *ldbc* dataset stored as plain text file occupies twice more space on disk than the *Frb-S* file, and contains 2 hundred thousands more elements. Nonetheless Sparksee requires for *ldbc* about 25% less space, and OrientDB less than half the space occupied on disk by the corresponding image with *Frb-S*. *MiCo* has comparable size, in plain text, to *Frb-S*, and contains twice the objects of *Frb-S*, but still the respective docker images of OrientDB and Sparksee for *MiCo* are almost half the size of their images containing the *Frb-S*. These disproportions can be explained by the fact that *Frb-S* contains almost ∼2K different edge labels, while *MiCo* 106, and *ldbc* only 15. Apparently these systems store different data-structure for different edge labels, causing a certain amount of overhead for datasets with many such labels.

It is important to note here that we have tried also much larger datasets, but we were not able to load them on a large number of systems so we could not have comparison across all the systems and we have not reported them.

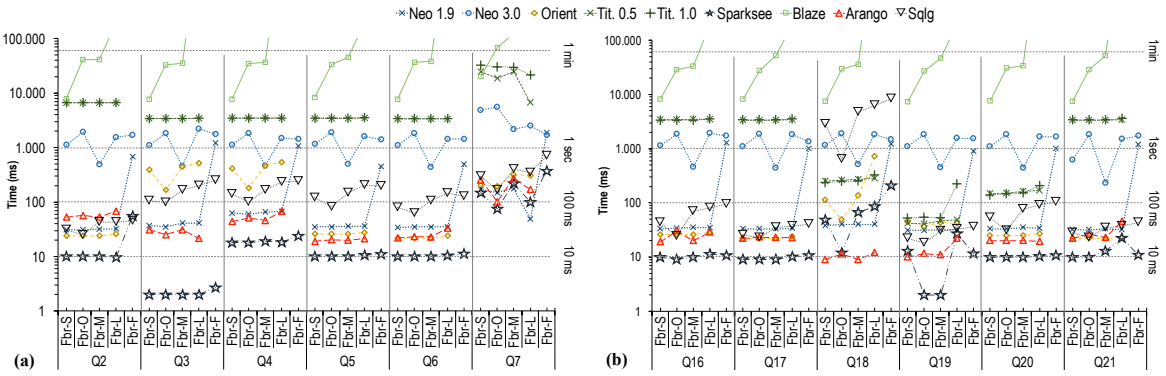

Figure 5.3: Time required for (a) insertions and (b) updates and deletions.

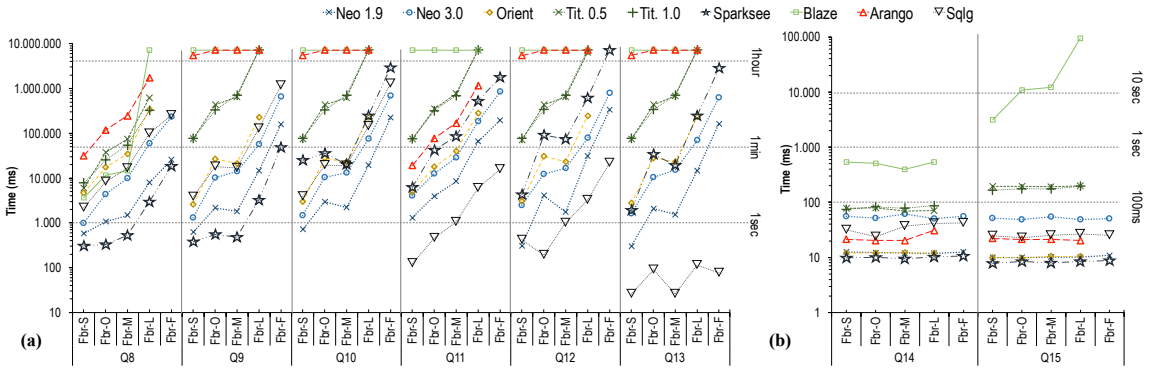

Figure 5.4: Selection Queries: The Id-based (right) perform orders of magnitude better than the rest (left)

5.5.2 Complex Queries

Our complex queries are based on those of the LDBC benchmark applied on their *ldbc* dataset, and executed in the various systems in the presence of indexes. The results of their execution are illustrated in Figure 5.10. BlazeGraph is not reported in the figure because the queries timed-out. ArangoDB and Titan (v.0.5) were in general the slowest, which indicates that they could not effectively exploit the index structures and neither employ any advanced optimization. Titan (v.1.0) was very fast for queries involving short joins and with single-label selections. This is explained by Cassandra being able to provide fast access to edges with specific labels in each node adjacency list, yet when many more hops where necessary it performed much worse. Neo4J (v.3.0) maintains a consistent performance throughout all the queries, and is the second best in half of the cases, suggesting that the system is adequate for applications that are not specialized on a single type of operation. Sqlg is the fastest in almost half the queries. These are the queries that can be translated to conditional join queries, with no recursion and short join

chains. For these queries Sqlg is able to analyze the sequence of Gremlin operators and optimize them by combining sequences of them into single relational operators, instead of executing each one separately. Hence, the system takes advantage of the relational optimizer to push down selection predicates and exploit indexes. Yet, we see that there are cases (e.g., the last query) where Sqlg is much slower than the competition. Those cases are queries that, for instance, traverse many edges and do not filter on a single edge label, and thus generate large intermediate results.

Next, we will see, that most of these results account for only a portion of the use-cases of graph databases, and they fail to cover other problems that can emerge when other types of queries are performed, e.g., queries used when exploring a large knowledge graph, where the edge label cardinality is much larger. For instance, we will see that in many unbounded traversals, Sqlg is proven to be among the slowest.

5.5.3 Completion Rate

Since graph databases are often used for on-line applications, ensuring that queries terminate in a reasonable time is important. We count the queries that could not complete within 2 hours, in isolation or in batch mode, and illustrate the results in Figure 5.2(c). Note that, if one instantiation of one query fails to run within the alloted amount of time in isolation, when executed in a batch it will cause the failure of the entire batch as well.

Neo4J, in both version, is the only system which completed successfully all tests with all parameters on all datasets (omitted in the figure). OrientDB is the second best, with just few timeouts on the large *Frb-L*. BlazeGraph is at the other end of the spectrum, collecting the highest number of timeouts. It reaches the time limit even in some batch executions on *Yeast*, and almost on all queries on *Frb-L*. In general the most problematic queries are those that have to scan or filter the entire graph, i.e., queries Q.9 and Q.10. Some shortest-path searches, and some bread first traversal with depth 3 or more in most databases reach the timeout on *Frb-O*, *Frb-M* and *Frb-L*. Filtering of nodes based on their degree (Q.28, Q.29, and Q.30) and the search for nodes with at least one incoming edge (Q.31) are proved to be extremely problematic almost for all databases apart from Neo4J and Titan (v.1.0). In particular for Sparksee these queries cause the system to exhaust the entire available RAM and swap space on all Freebase subsamples (this has

been linked to a known problem in the gremlin implementation). BlazeGraph fails also these last queries on all the Freebase datasets, while ArangoDB fails it only on *Frb-M* and *Frb-L*, and OrientDB instead only on *Frb-L*.

5.5.4 Insertions, Updates and Deletions

For operations that add new objects (nodes, edges, or properties), tests show extremely fast performances for Sparksee, Neo4J (v.1.9), and ArangoDB, with times below 100ms, with Sparksee being generally the fastest (Figure 5.3(a)). Moreover, with the only exception of BlazeGraph, all databases are almost unaffected by the size of the dataset. We attribute this to the use of write-ahead logs, and the internal configuration of the adopted data-structures. BlazeGraph is instead the slowest with times between 10 seconds and more than a minute. Both versions of Titan are the second slowest with times around 7 seconds for insertion of nodes, and 3 seconds for insertion of edges or properties, while for the insertion of a node with all the edges (Q.7) it takes more than 30 seconds. Sparksee, ArangoDB, OrientDB, Sqlg, and Neo4J (v.1.9) complete the task in less than a second. OrientDB is among the fastest for insertions of nodes (Q.2) and properties on both nodes and edges (Q.5 and Q.6), but is much slower, with inconsistent behavior, for insertion of edges. Neo4J (v.3.0), is more than an order of magnitude slower than its previous version, with a fluctuating behavior that does not depend on the size of the dataset. Sqlg is among the fastest for insertions of nodes, and nodes alongside edges, while is much slower for all other queries. Similar results are obtained for the update of properties on both nodes and edges (Q.16, and Q.17), and for the deletion of properties on edges (Q.21).

The performance of node removal (Q.18) for OrientDB, Sqlg, and Sparksee seems highly affected by the structure and size of the graphs (Figure 5.3(b)). On the other hand, ArangoDB and Neo4J (v.1.9) remain almost constantly below the 100ms threshold, while Neo4J (v.3.0) completes all the deletions between 0.5 and 2 seconds. Finally, for the removal of nodes, edges, and node properties, Titan shows almost one order of magnitude improvement.

For creations, updates and deletions, as a whole, the fastest are Neo4J (v.1.9), with constant times below 100ms, and then Sparksee, but with quite a scale-sensitive behavior for edge-deletion, that is shared with OrientDB. ArangoDB is also consistently among

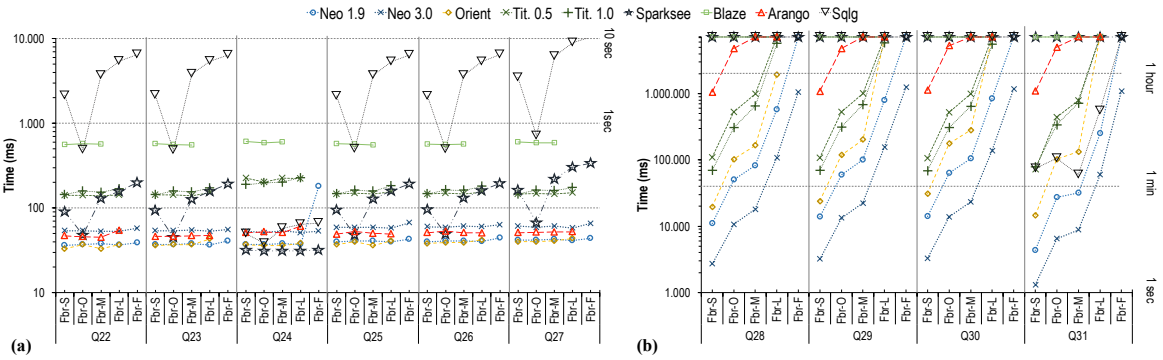

Figure 5.5: Time required for traversal operations: (a) local access to node edges, and (b) filtering on all nodes

the fastest, but its interaction through REST calls, and the fact that it does not support transactions, constitutes a bias on those results in its favor since the time is measured on the client side.

5.5.5 General Selections

With *read* queries, some heterogeneous behaviors start to show up. The search by ID (Figure 5.4(b)) differs significantly from all other queries in this class. BlazeGraph is again the slowest, with performances around 500ms for the search of nodes, and instead 4 seconds or more for the search of edges. All other systems take less than 400ms to satisfy both queries, with Titan the slowest among them. Here Sparksee, OrientDB and Neo4J (v.1.9) return in about 10ms, hinting to the fact that, given the ID, they are able to jump immediately to the right position on disk where to find it.

In counting nodes and edges (Q.8, and Q.9), Sparksee has the best performance followed by Neo4J (v.3.0). As a matter of fact Sparksee and Neo4J (v.3.0) complete the two tasks in less than 10 seconds on all sizes of Freebase, while Neo4J (v.1.9) take more than an minute on the *Frb-L*. For BlazeGraph and ArangoDB, node counting is one of the few queries in this category that complete before timeout. In particular in Q.8 BlazeGraph is faster than ArangoDB, but then it hits the time limit for Q.9 on all Freebase subsamples, while ArangoDB, at least for *Frb-S* it's able to get the answer in time also on the other queries. Edge iteration, on the other hand, seems hard for ArangoDB that rarely completes within 2 hours for the Freebase datasets.

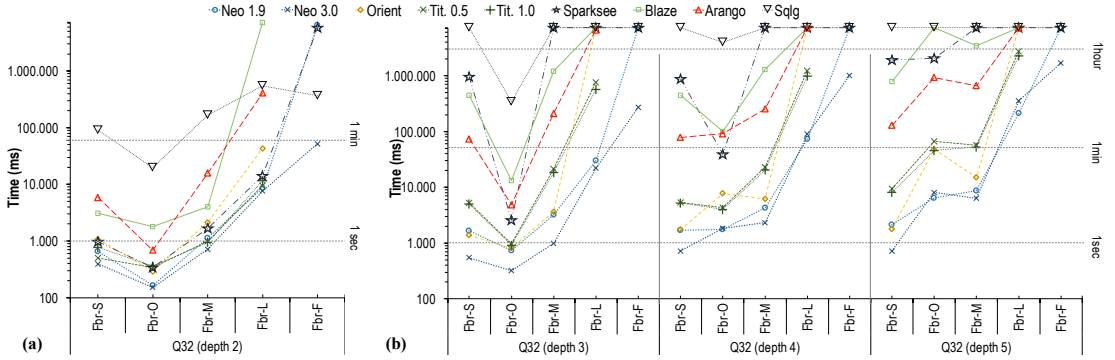

Figure 5.6: Time required for breadth-first traversal (a) at depth= 2, and (b) at depth>= 3

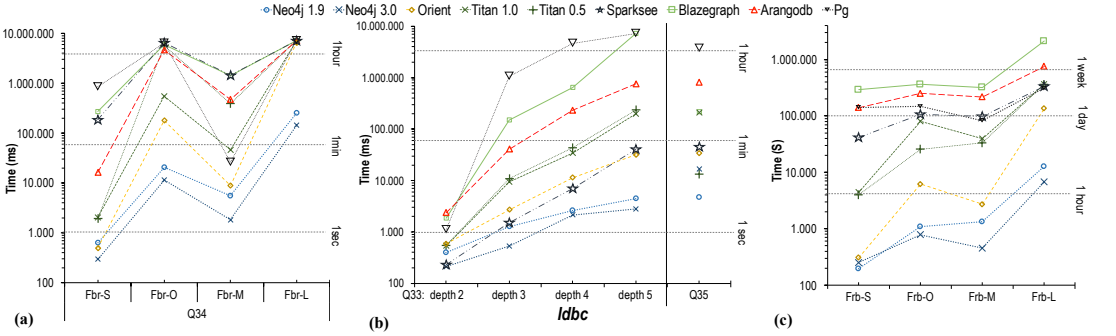

Figure 5.7: Performance of (a) Shortest Path, (b) label-constrained BFS and Shortest Path, and (c) Overall

Computing the set of unique labels (Q.10) changes a little the ranking. Here, the two versions of Neo4J are the fastest databases, while Sparksee gets a little slower. The search for nodes (Q.11) and edges (Q.12) based on property values performs similar to the search for edges based on labels (Q.13), for almost all databases. These 3 are some of the few queries where the RDBMS-backed Sqlg works best, with results an order of magnitude faster than the competition. Among the others, Neo4J (v.3.0) gives the shortest time, with the Q.13 performing slightly faster than the others, getting an answer in a little more than 10 seconds on the larger dataset, while Neo4J (v.1.9), Sparksee, and OrientDB are at least one order of magnitude slower. Only for Sparksee we notice relevant differences between Q.12 and Q.13. Hence, equality search on edge labels is not really optimized in the various systems.

5.5.6 Traversals

As mentioned above, the most important class of queries that GDBs are called to satisfy regards the *traversal* of graph structures. In the performance of traversal queries that access the direct neighborhood of a specific node (Q.22 to Q.27), we observe (Figure 5.5(a)) that OrientDB, Neo4J (v.1.9), ArangoDB, and then Neo4J (v.3.0) are the fastest, with response times below the 60ms, and being robust to the size and structure of the dataset. Sparksee seems to be more sensitive to the structure and size of the graph, requiring around 150ms on *Frb-L*. The only exception for Sparksee is when performing a visit of the direct neighborhood of a node filtered by the edge labels, in which case it is on par with the former systems. BlazeGraph is again an order of magnitude slower (∼600ms) preceded by Titan (∼160ms). We notice also that Sqlg is the slowest engine for these queries, unless a filter is posed on the label to traverse, in which case Sqlg becomes much faster.

When comparing the performance of queries Q.28 to Q.31 that traverse the entire graph filtering nodes based on the edges around them, as shown in Figure 5.5(b), the clear winner is Neo4J (v.3.0), with its older version being the second fastest. Those two are also the only two engines that complete the query on all datasets. In particular Neo4J (v.3.0) completed each query on *Frb-L* in less than two minutes on average, while Neo4J (v.1.9) took at lest 10 minutes for the same dataset. All tested systems are obviously affected by the number of nodes and edges to inspect. Sparksee is unable to complete any of these queries on Freebase due to the exhaustion of the available memory, indicating probably a problem in the implementation, as this never happens in any other case. BlazeGraph as well hits the timeout limit on all samples, while ArangoDB is able to complete only on *Frb-S* and *Frb-O*. Finally Sqlg is able to complete only Q.31, although with time comparable to Neo4J (v.1.9). Nevertheless, all systems complete the task on *Yeast*, *ldbc* and *MiCo*.

We study breadth-first searches (Q.32 and Q.33) and shortest path searches (Q.34 and Q.35) separately from the other traversal operations. The performance of the unlabeled version of breadth-first-search, shown in Figure 5.6, highlights once more the good scalability of both versions of Neo4J at all depths. Although Neo4J (v.3.0) is the only system to complete the task before timeout even on the *Frb-F*. OrientDB and Titan give the second fastest times for depth 2, with times 50% slower than those of Neo4J. For depth

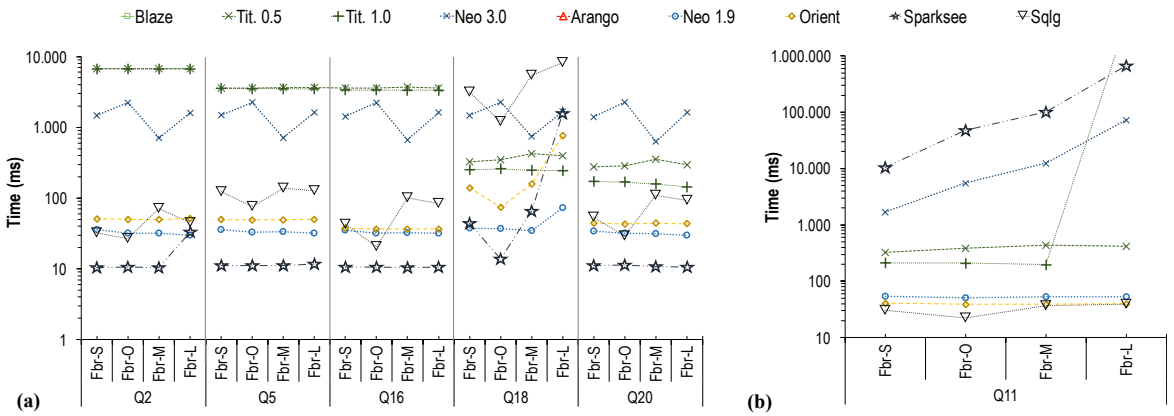

Figure 5.8: Effect of indexing on the Time required for Q.2, Q.5, Q.16, Q.18, Q.20, and Q.11

3 and higher, as Figure 5.6(b) illustrates, OrientDB is a little faster than Titan. On the other hand, in these queries we observe that Sqlg and Sparksee are actually the slowest engines, even slower than BlazeGraph. For query Q.34 in Figure 5.7(a), which is the shortest path with no label constraint, the performance of the system is similar to the above, BlazeGraph and Sparksee are in this case very similar, and Sqlg still the slowest.

The label-filtered version of both the breadth first search and the shortest path query on the Freebase samples (not shown in a figure) were extremely fast for all datasets because the filter on edge labels cause the exploration to stop almost immediately. Running the same queries on *ldbc* we still observe (Figure 5.7(b)) that Neo4J is the fastest engine, but also Sparksee is the second fastest in par with OrientDB for the breadth-first search, while on the shortest path search filtered on labels, Titan (v.1.0) gets the second place.

5.5.7 Effect of Indexing.

We built node-attribute indexes on the graphs in the various systems to evaluate the effect of indexing on the system performance (Figure 5.8, and 5.9). BlazeGraph has been excluded since it does not allow any custom index (and the system already builds its own). ArangoDB showed no difference in running times, so we suspect some defect in the gremlin implementation. For insertions, updates, and deletions, we noticed longer running times, as expected since the indexes had to be updated, but was no more than 10% in most cases. The only exception to this trend are Neo4J (v.3.0) and OrientDB, with delays of about 30% and 100% respectively. Despite the increase in time, Neo4J

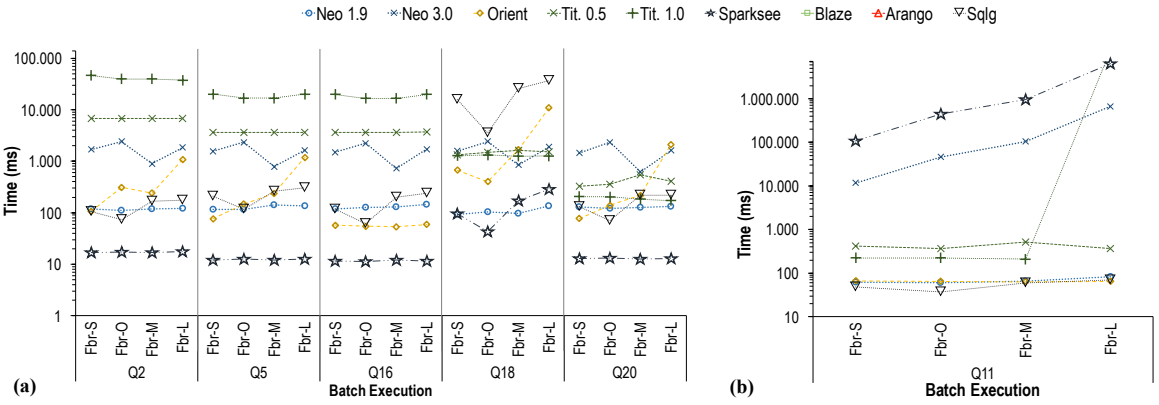

Figure 5.9: Effect of indexing on the Time required in Batch Mode for Q.2, Q.5, Q.16, Q.18, Q.20, and Q.11

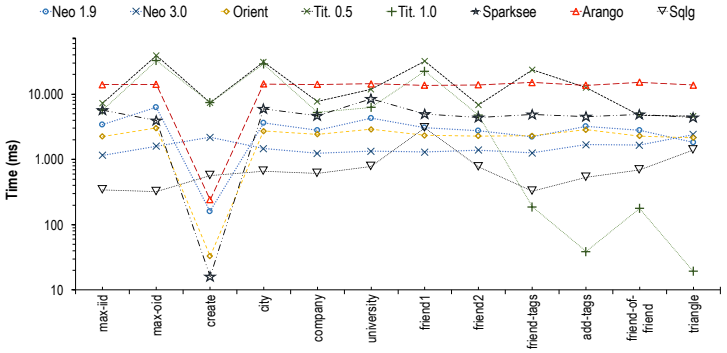

Figure 5.10: Complex Query Performance

(v.1.9), Sparksee, and OrientDB remained the fastest systems for *CUD* operation. For search queries (Q.11), the presence of indexes gave to Neo4J (v.1.9), OrientDB, Titan (v.0.5), and Titan (v.1.0) an improvement of 2 to 5 orders of magnitude (depending on the dataset size), while Sqlg witnessed up to a 600x speed up. Sparksee and Neo4J (v.3.0) instead were not able to take advantage of the indexes. As a matter of fact, both system support labels not only for edges but also for nodes. For this reason, for both systems, indexes are tied to a specific node label, and hence only queries specifying a selection on a node label can then exploit the index for the attribute. This means that indexes play a significant role in most of the systems, and are taken seriously into consideration in query execution.

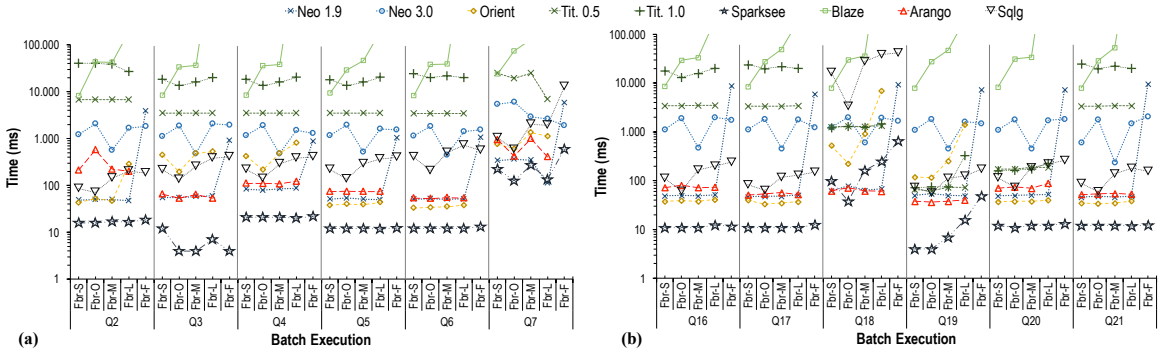

Figure 5.11: Time required in Batch Mode for (a) insertions and (b) updates and deletions.

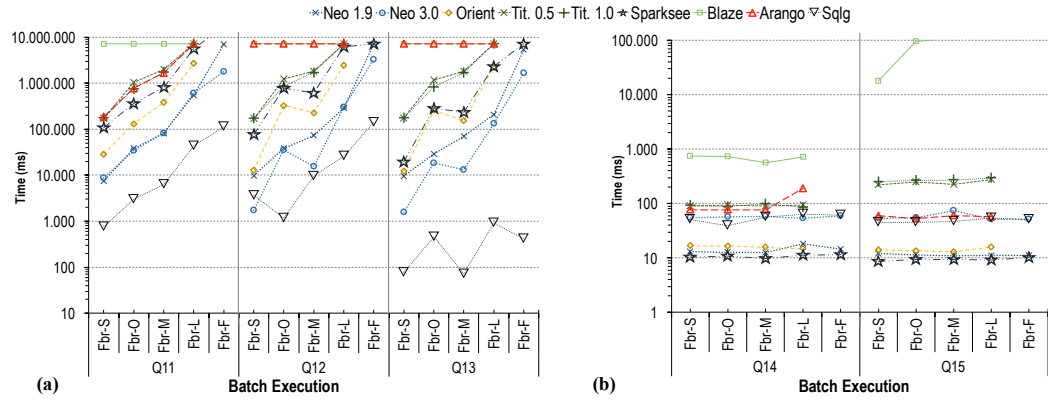

Figure 5.12: Selection Queries in Batch Mode: The Id-based (right) perform orders of magnitude better than the rest (left), and compared to the isolation mode they take the same amount of time

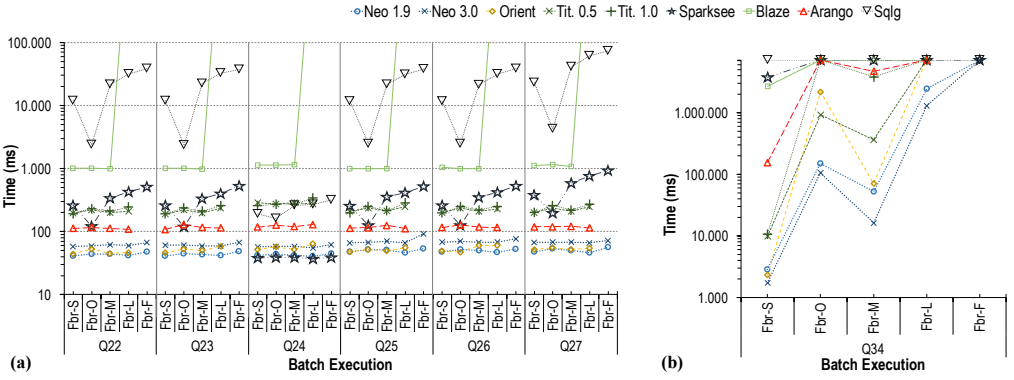

Figure 5.13: Time required for traversal operations in Batch Mode: (a) local access to node edges, and (b) for shortest path search

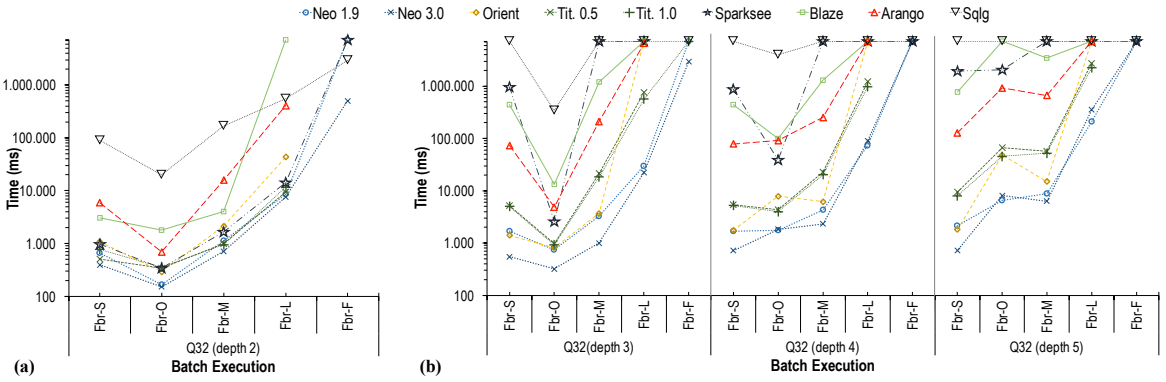

Figure 5.14: Time required for breadth-first traversal in batch mode (a) at depth= 2, and (b) at depth>= 3

5.5.8 Single vs Batch Execution.

We looked at the times differences between single executions (run in isolation) and batch. We report times for each batch execution for *Frb-S*, *Frb-O*, *Frb-M*, and *Frb-L* in Figures 5.11, 5.12, 5.13, and 5.14. Running the queries in batch mode does not create any major changes in the way the systems compare to each other. For the retrieval queries, the batch requests of the 10 queries were taking exactly 10 times the time of one iteration, i.e., no benefit obtained from the batch execution. Exception is for queries 14 and 15 (Figure 5.12 b), here times to retrieve 10 nodes by their internal IDs are almost exactly the same as for retrieving one single node (see Figure 5.4 above). Such behavior suggests that the systems load the data into main memory at the first call, and then retrieves everything from there.

Instead, for the create, update and delete operations, the batch is less than 10 times the time needed for one iteration, meaning that in single mode most of the time we measure is some initiation set-up time for the operation. For traversal queries the batch executions only stressed the differences between faster and slower databases.

5.5.9 *Yeast*, *MiCo*, and *ldbc*

In the following we report on the results of the tests performed on the *Yeast*, *MiCo*, and *ldbc* datasets, which are generally smaller than the Freebase samples, and also have a much smaller number of edge labels. Results for queries in isolation mode are reported in Figure 5.15, 5.17, 5.19, and 5.21, while results for the batch mode execution are in

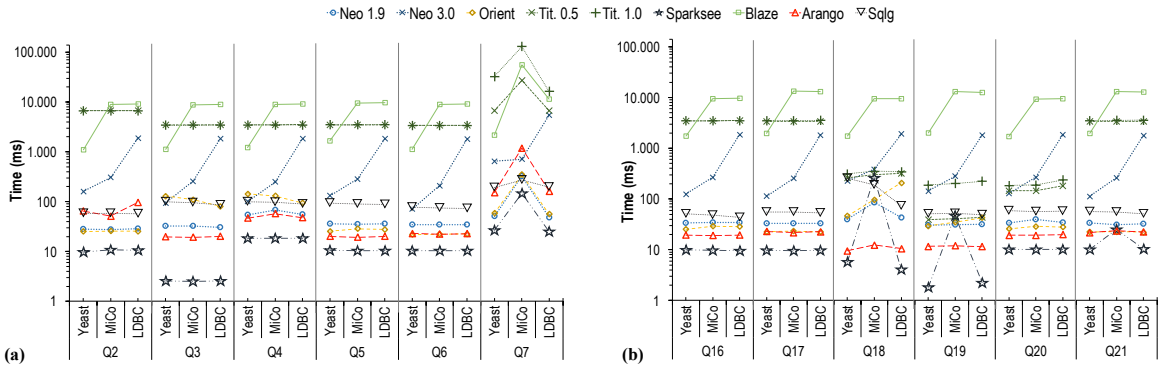

Figure 5.15: Time required on *Yeast*, *ldbc*, and *MiCo* for (a) insertions and (b) updates and deletions in isolation mode.

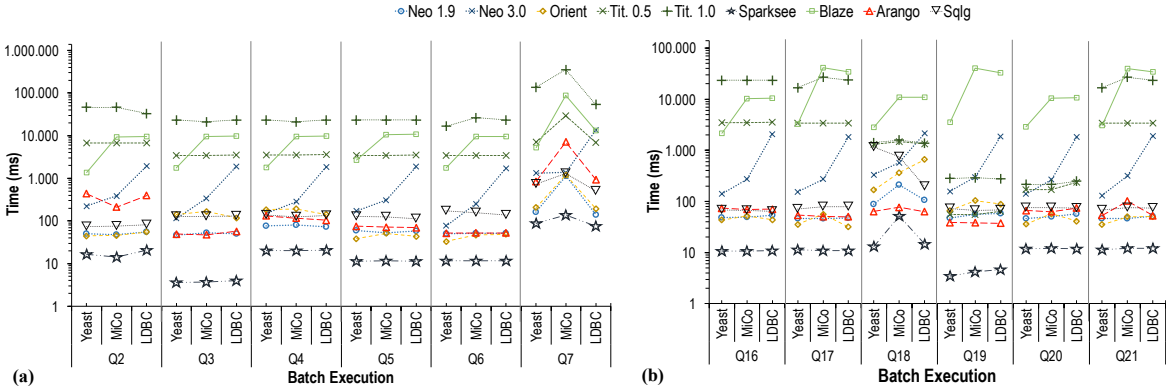

Figure 5.16: Time required for (a) insertions and (b) updates and deletions in batch mode for *Yeast*, *ldbc*, and *MiCo*.

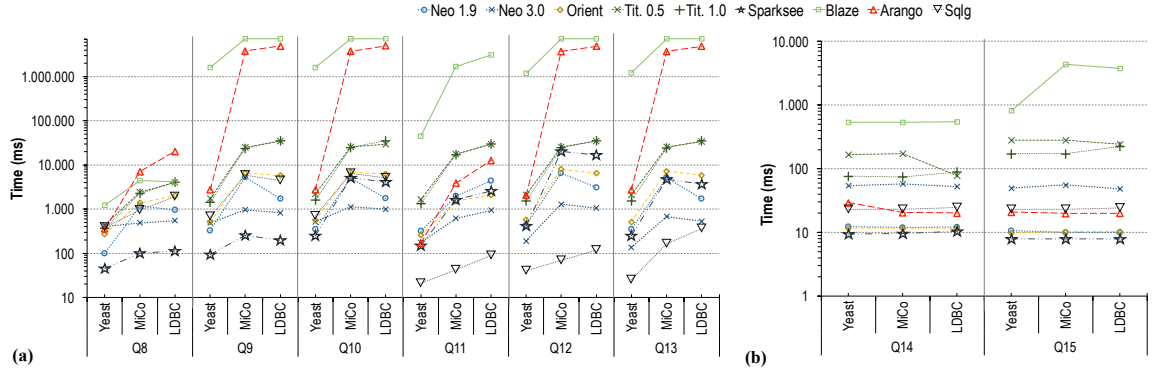

Figure 5.17: Selection Queries in isolation mode for *Yeast*, *ldbc*, and *MiCo*: The Id-based (right) perform orders of magnitude better than the rest (left)

Figure 5.16, 5.18, 5.20, and 5.22. Experiments on these datasets, as noted above, show again similar relative performances compared to the results on the Freebase samples described earlier. In general we see Sparksee performing among the fastest databases more often. ArangoDB's performance as well is much more similar to the other systems. BlazeGraph instead is usually the slowest also on those datasets. As a matter of fact,

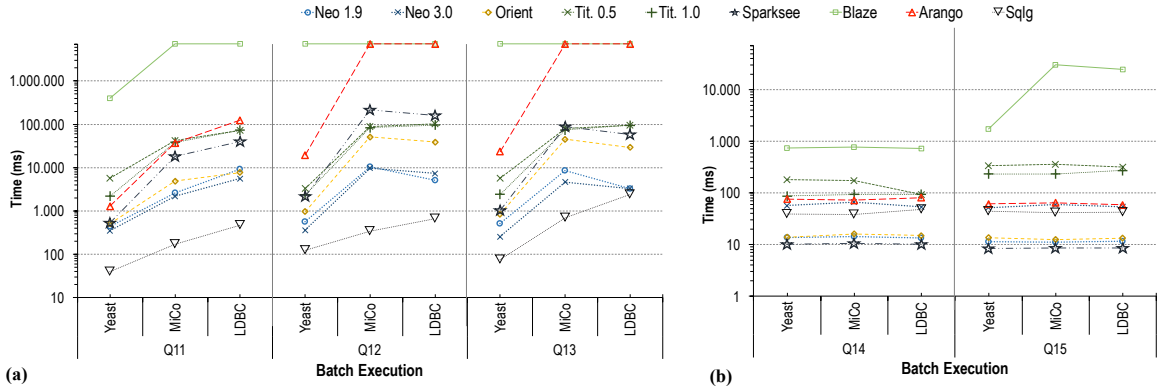

Figure 5.18: Selection Queries in Batch Mode for *Yeast*, *ldbc*, and *MiCo*: The Id-based (right) perform orders of magnitude better than the rest (left), and compared to the isolation mode they take the same amount of time

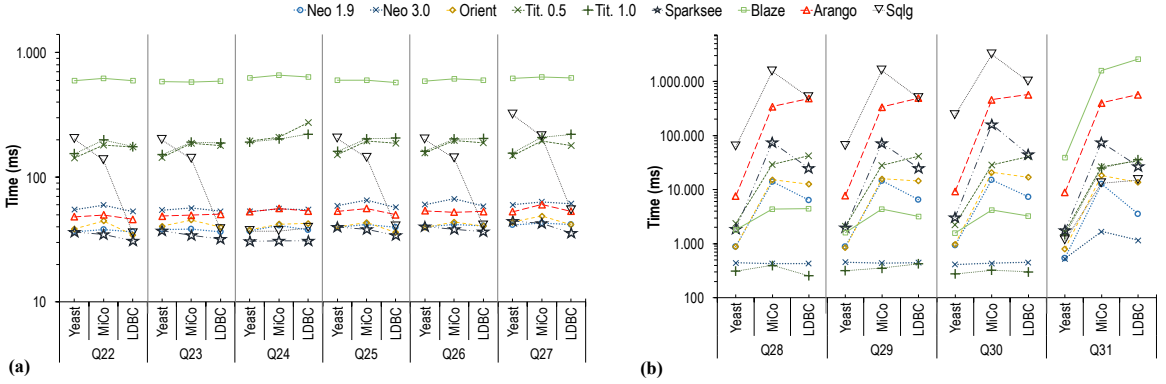

Figure 5.19: Time required for traversal operation for *Yeast*, *ldbc*, and *MiCo*: (a) local access to node edges, and (b) global filtering of nodes based on degree.

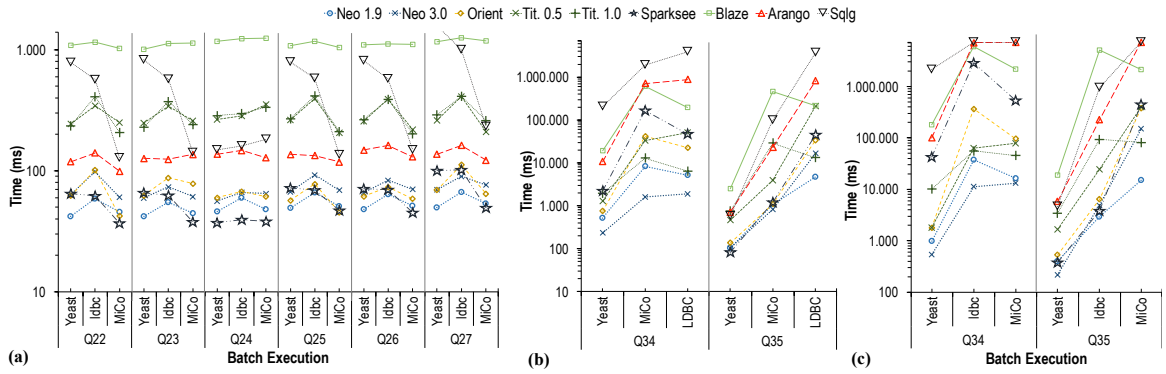

Figure 5.20: Time required for traversal operations for *Yeast*, *ldbc*, and *MiCo*: (a) local access to node edges in batch mode, and for shortest path search (b) in isolation, and (c) in batch mode.

even in tests with *Yeast*, BlazeGraph is not always able to terminate queries within the timeout limit, which indicates some serious implementation problems for some of the selection queries (Figure 5.17).

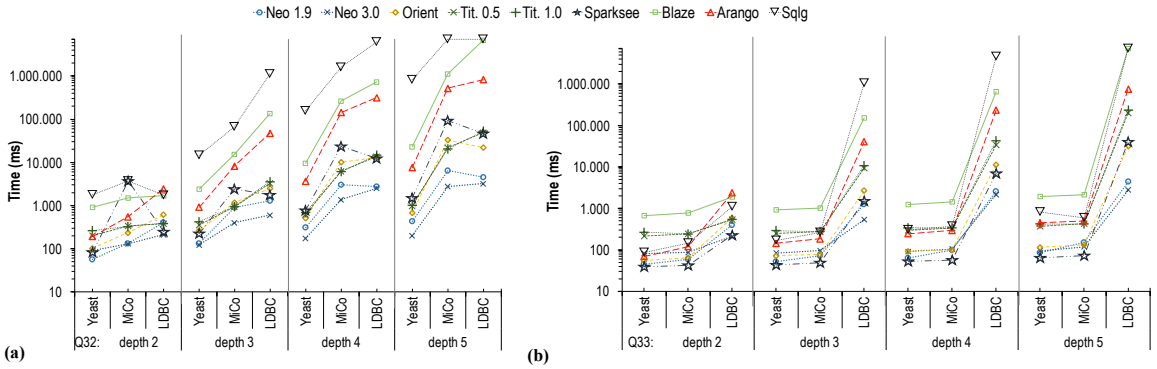

Figure 5.21: Time required for breadth-first traversal with (a) and without (b) label filtering for *Yeast*, *ldbc*, and *MiCo*.

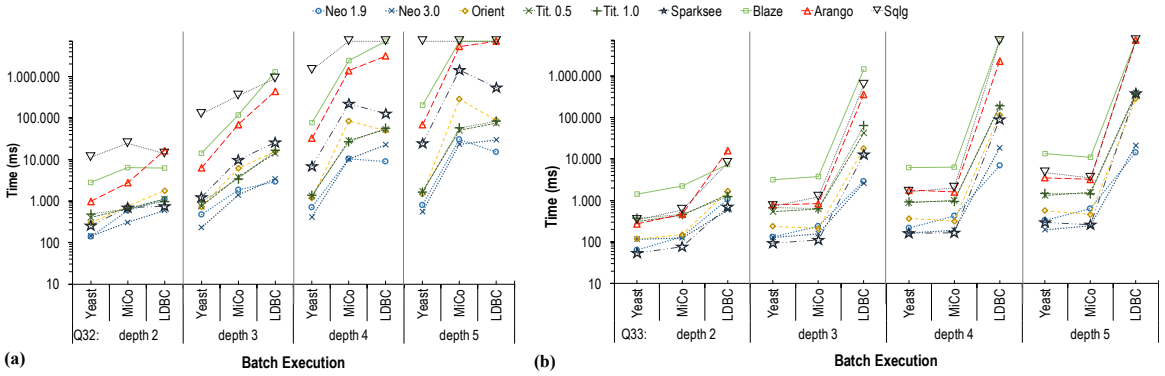

Figure 5.22: Time required for breadth-first traversal with (a) and without (b) label filtering in batch mode for *Yeast*, *ldbc*, and *MiCo*.

5.5.10 Overall Evaluation and Insights

When looking at the cumulative time taken by each system to complete the entire set of queries in both single and batch executions (Figure 5.7(c)), Neo4J is the system with the shortest execution time. Pretty good running times have also been recorded for OrientDB, which is often on par with Neo4J, and in some cases is better than one of its two versions. However, it doesn't perform well when large portions of the graph have to be processed and kept in memory, e.g., with *Frb-L*. Titan results quite often one order of magnitude slower than the best engine. It shows difficulties in create and update operations, while it is much better in deletions, most likely due to the *tombstone* mechanism, where it marks an item as removed instead of actually removing it. Sparksee gives almost consistently the best execution time in create, update and delete operations. Although it is not very fast with deletions of nodes having lots of edges, it is still better than many of the others. It performs better also in edge and

node counts, as well as in the retrieval of nodes and edges by ID, thanks to its internal compressed data structures. Nevertheless, it performs worse than the others for the remaining queries due to suboptimal filtering and memory management. Finally, it gives a lot of timeouts on the degree-based node search queries. ArangoDB excels only in few queries. For creations, updates and deletions, it ranks among the best. This is also because all updates are kept in main memory and synced on disk. For retrievals and search, its performance is in general poor. This is due to the way Gremlin primitives are translated into the engine, where ArangoDB has to materialize all the objects in order to iterate through them. Exception is when searching by ID, which is expected since it is a key-value store at the core, while for traversals it has a narrow lead over Sparksee and BlazeGraph demonstrating good effectiveness of its edge-specific hash index.

Sqlg shows the expected low performance for all the traversal operations, due to the need to traverse the graph via relational joins instead of via direct links to node/edges. For queries containing 1 or 2 hop traversals restricted to a single edge-label, like some of the complex queries, however, it performs extremely well. In these cases it takes advantage of the ability to conflate multiple operations in a single query and filter using foreign key indexes for specific edge-label tables. BlazeGraph results also in a generally poor performance. The indexes it builds automatically do not help much. This is most likely because each single step is executed against some specific graph API, instead of having the Gremlin query translated into SPARQL and executed as such. This graph API implementation do not exploit any of the optimization implemented by the SPARQL query engine.

Ranking. A summary of the ranking of the systems for each query type is presented in Table 5.4. The characterization here is an overall aggregation over the ranking from fastest (3 out of 3 stars) to slowest (1 out of 3) for each type of query. Systems are in the first class when their performance is approximately in the same order of magnitude to the best, while they are in the last when they incur in timeouts or have the worst performance. The middle class is for mixed results. Within each group systems are ranked from the best to the worst.

Query importance. Although we considered all the queries equally important, for different applications some may be playing a more important role. We have identified three main uses of graph databases: interactive, exploratory and business analytics.

From an informal survey with some system developers, they confirmed us that all the operators covered in our micro-benchmark are important still with different priority, from case to case, and that no single is more important than others in all cases. The weight of each query for each such use, as produced in collaboration with some system developers, can be found in Table 5.5. This table can guide the evaluation of the performance results of a system depending on the final use-case.

Hybrid and Native systems. One of the main findings of the experiments is that hybrid and native systems perform differently in some important operations. For a limited set of use cases hybrid systems perform equally well as the native, but for traversal queries, like finding the connectivity between two nodes, unbounded traversals and the enumeration of structures, the hybrid systems under-perform significantly. The use case and sets of typical queries should then be well understood before committing to a hybrid system, while a native system is generally a safer choice. This indicates that design choices made in native systems, e.g., the separation of the attribute values from nodes/edges, are the right way to go, but also highlights the need for further development in terms of indexing and filtering.

Query language. Regarding the query language, Gremlin is the one supported by all the systems. Nevertheless, each system offers its own native query language and performs all the optimizations on it. As explained earlier, in many systems Gremlin queries have to be translated step-to-step to the native primitives, thus loosing many of the possible optimizations. Gremlin may be used as a standard, but is not the first priority of the systems. The fact that, in some cases, data loading was not possible through Gremlin but only through native calls, and the problems with large intermediate results are another indication of this priority.

Micro-benchmarking. By comparing the methodologies of the micro and macro benchmarking, it became clear the importance of studying individual operators in a context-agnostic way. The micro-benchmark evaluation has pointed out many specific problems that could successfully be communicated to the vendors, and that either did not surface using the macro-benchmarking or for which isolating the actual cause required a deeper analysis.

Table 5.5: Relative importance of queries.

Q.#	Interactive	Exploratory	Business Analytics
LOADING			
1.	3	**4**	1
CREATE			
2.	2	1	3
3.	3	1	**4**
4.	2	1	3
5.	**4**	1	**4**
6.	2	1	3
7.	**4**	1	**4**
READ			
8.	3	**4**	1
9.	3	3	1
10.	3	3	2
11.	**4**	**4**	**4**
12.	2	2	2
13.	2	3	2
14.	**4**	2	**4**
15.	1	1	3
UPDATE			
16.	**4**	2	**4**
17.	2	**4**	3
DELETE			
18.	3	1	3
19.	**4**	2	**4**
20.	**4**	2	3
21.	2	1	2
TRAVERSALS			
22.	**4**	**4**	**4**
23.	**4**	**4**	**4**
24.	**4**	**4**	**4**
25.	3	3	2
26.	3	3	2
27.	3	3	2
28.	1	3	2
29.	1	3	2
30.	1	3	2
31.	2	3	1
32.	**4**	3	2
33.	**4**	**4**	2
34.	2	3	2
35.	3	**4**	2

1 = Low importance, 4=Critical Importance

5.6 Experiences

In general our experiences cover a large spectrum of issues, technical challenges that we faced, and areas of improvement that are related to the usability of the various systems.

5.6.1 Installation, Configuration, Documentation and Support.

First we stress that the only 2 systems that we were able to install and run as expected were Neo4J and Sparksee. For those, after downloading the relevant binaries and following the instructions provided on the respective websites, we were almost immediately able to load our datasets, at all sizes, and run some queries. For the others, as mentioned earlier (and below) we had to overcome some difficulties in importing the datasets, configuring the systems properly, and understanding the errors raised when running some of the queries. As a result, for those system that are open-source and hosted on a public repository, we reported those problems and bugs found as issues. In total we issued 8 support request (comprising bug issues) for ArangoDB, 4 for OrientDB, 2 for Titan, 2 for Sqlg, and 1 for BlazeGraph.

For ArangoDB and OrientDB some of those bugs have been fixed in official releases of the software or have been included in the development road-map. Instead those regarding Titan and BlazeGraph didn't receive any reply from the developers (in many months) and, where possible, were either fixed or circumvented in our local installs. This also describes the level of support received by the respective development teams.

Regarding the documentation, we note that Neo4J, Sqlg and OrientDB are provided with pretty in-depth informations for developers. Sparksee, Titan and ArangoDB have some documentation, limited in some aspects, but still clear for basic installation, configurations and operational needs. Among those Titan manual contains a lot of confusion among the various existing software versions, and in some cases, the provided instructions and example-code are not actually self-contained. Also, given the reliance on Cassandra for the storage, it is to note the reduced amount of information on how to properly configure this system and how to tackle the various problems arising with it. BlazeGraph's documentation, instead, is largely outdated. Also, even though the system

relies a lot on the user for proper configuration, the information provided is generally cryptic.

Regarding the configuration of the other systems, we report that Neo4J doesn't require any specific configuration. OrientDB instead supports by default a number of edge labels at most equal to 32676 divided by the number of cores in the machine (e.g., 4084 edge labels on a 8 cores machine), for supporting more labels, it requires a special feature to be disabled. ArangoDB requires two configurations, one for the engine, and one for the V8 javascript server, the second regards the level of logging of the system. Without proper configuration (with only default values) this system generated 40 GB of log files in about 24 hours of activity, with a single active client. For Titan instead the most important configurations are for the JVM Garbage Collection and for the Cassandra backend. Additionally, with large datasets, it is necessary to disable automatic schema creation, and to create instead the schema manually before loading the data.

All systems based on Java, were also extremely sensitive to the effect of the garbage collection routines. When dealing with data-intensive applications and a large amount of main-memory, it is necessary to provide a customized configuration to the JVM, yet, none of the systems provide clear instructions on how to tune it properly for their needs, but they only propose generic advices.

Finally we report on the Tinkerpop/Gremlin documentation. For version 2.6 the list of supported methods with some examples are provided [5], for version 3 the official manuals are much more extended [6], although not to the benefit of clarity. In this sense, we also hope that the code of the queries implemented in this study serve as more concrete tutorial for understanding the basics of the Gremlin language.

5.6.2 Loading problems.

As already mentioned, we encountered a great deal of issues when trying to load the datasets in some of the databases tested. ArangoDB in particular, when using Gremlin for loading, sends each node and edge insertion instruction separately to the server (in a HTTP call). This method results too slow, even for small datasets, so that we were forced to use some routines provided by the back-end system itself. For BlazeGraph, with

[5] `gremlindocs.spmallette.documentup.com`
[6] `http://tinkerpop.apache.org/docs/current/reference/`

the exception of the smallest datasets, we had to activate a specific *bulk loading* feature otherwise we were facing loading times in the order of days. OrientDB as well required us to pass through some server-side implementation-specific commands in order to load the datasets. In particular, it didn't support non-alphanumeric characters in edge-label, and for the Freebase samples we had to disable some features that were limiting the maximum number of edge-labels. Also for Sqlg we re-encoded all edge labels to unique hashes that did not exceed the 63 characters limit that Postgresql imposes, after that the loading proceeded without any major issue. Finally, Titan (in both versions) for any medium to large sized datasets requires disabling the automatic schema creation during loading, otherwise its storage back-end (Cassandra) would get swamped with extra consistency check operations. This means that the complete schema of the graph, in terms of node and edge labels and properties, should be known to the system prior to the insertion of the data, the same way one should declare the schema in a relational database before loading any data. This required us to issue a set of instructions, before loading the data, to create such schema.

5.6.3 Queries, Groovy, and Gremlin.

Last, we report that using Groovy as support language for Gremlin was quite problematic in some cases. As a matter of fact the Groovy language has dynamic types, and uses type inference along with peculiar handling of variable scope. As a result, explicit type casting is needed when providing the values to queries in some systems, especially with numbers. For example, in Sparksee if one attribute is of *Long* type and size (i.e., larger than a 32 bit number), then all values for the attributes with the same name need to be passed and queried as *Long* values, otherwise values compatible with the *Integer* type will be treated as such, and the search will result in a mismatch, independently of the value they represent. For Titan, instead, when not provided by the schema declared a priori, each value should be inserted as the smaller available type, i.e., if a number is within the integer range, it should be converted to the integer type. With the other systems instead, types are handled transparently for the user, and work without explicit type casts.

A second problem with Gremlin 2.0 is the lack of explicit operations for pattern-matching queries and shortest paths queries. In the new version (Gremlin 3.0) a new '*match*' operator is introduced, but there is still no operator for the shortest-path search. Both types of queries could be implemented with the composition of basic constructors (although for weighted shortest path the implementation would be extremely hard), while would be better to have an abstract operator in the language and leave to the engine the implementation of advanced and optimized algorithms.

Finally, Gremlin doesn't provide a way to handle indexes, this as well is a limitation of the language that requires for the user to access directly the back-end system.

5.7 Summary

We provided a principled and systematic evaluation methodology based on micro-benchmarking that contains 35 different operations. In terms of operations, this is the first micro-benchmarking approach for graph database systems, and the most complete to date. We performed an extensive experimental evaluation of the state-of-the-art graph databases. Furthermore, we scaled to graph sizes that have not been considered before, and included systems that have not been previously considered.

This evaluation allowed us to identify important differences in the performance of hybrid and pure system, to understand advantages and limitations of various implementation choices, and to gauge the performance of the state-of-the-art implementations of the Gremlin query language.

We also described the challenges we faced in loading the large datasets and running the queries, and how we overcame these challenges. We materialized our methodology into a suite that we made available on-line [LBV17]. It includes, scripts, datasets, and queries, among any other interesting material. Apart from the direct benefits, our work can complement studies on the different (but highly related) graph analytic systems.

Chapter 6

Conclusions

In this dissertation we studied new advancements in *by-example* methods for exploratory search tasks. Since exploratory tasks originate from open-ended information needs within an unfamiliar environment, we argued about the high utility of flexible query paradigms that do not require from the user a complete understanding of the conditions that the items of interest should satisfy. Moreover, we also proposed to relax a common assumption in many other by-example search paradigms, i.e., that the examples proposed by the user should all respect the same structure and should in this sense require a predefined combination of characteristics. Hence, in this work we extended the exemplar query paradigm allowing for multiple incomplete examples.

We ported this query formulation to the case of information graphs. In particular, we proposed a way to retrieve structures of interest that are composed in many different ways by smaller subgraphs, each one describing a specific and partial aspect of the final result. We explored both exact and approximate techniques for this task. The algorithms studied proved effective on large real world knowledge graphs.

Proceeding from the efficacy of the exemplar query paradigm for graphs, we recognized the necessity to also provide assistance to the user in the formulation of their queries. Especially in exploratory settings for large and heterogeneous graphs, we studied how to assist search tasks for which the initial input is as simple as a single entity. To this end, we provided the first principled study of an interactive graph query suggestion framework that incorporated techniques from traditional information retrieval models and ported them for the first time in the context of graph search. Among the techniques

we studied, we identified a promising method that effectively exploited both pseudo-relevance feedback and language models.

Finally, since we envision our proposed methods to complement existing graph search systems, we embarked on the task of comparing and understanding existing graph databases. The proposed methodology allows, among other things, to select the most appropriate graph database for the task at hand. In our case, for instance, it informed the choice of Neo4j [neo] as the graph database to employ in the back-end of our graph-suggestion system presented earlier. To achieve a comprehensive understanding of the capabilities of the state of the art graph data management systems, we elaborated the first micro-benchmarking approach for such systems. As a result we proposed both a vast and punctual list of graph-query operators and an effective evaluation methodology that allowed us to identify a series of insights in the functionality of the tested systems. In particular we shed more light on the difference between hybrid and pure systems, we identified important limitations in some of the technical choices currently implemented, and provided also an interesting survey of the current implementations of the Gremlin query language.

6.1 Extensions and Open Problems

The study conducted for this work and the results obtained prompted us with some additional avenues for future studies.

Approximate Search and Summarization. The first important problem that results from the application of many exemplar query paradigms is the necessity to accommodate for approximate answers. It is still possible that some example are unnecessary detailed and for this reason will be translated into queries that are too restrictive. For these cases, the query engine should be able to prioritize the features listed in the exemplar-query, in order to retrieve answers that match only a portion of them. Additionally, we could study substructures with similar meaning, so to allow for answers that match the semantics of the example even when a strict structural similarity is not found.

On the other side of the spectrum, we may encounter examples that are too generic, and for this reason they may result in a result-set that is too large to be consumed by

the user. Yet, while we have proposed top-k approaches, we should also study summarization techniques, that do not discard any answer, but instead try to group them in topical clusters, and provide to the user a complete overview of all the existing answers. Techniques like faceted search, or frequent pattern mining can help in these situations.

Learning to Rank and Advanced Suggestions. Our study for interactive graph query suggestion confirmed the applicability of interactive IR approaches to the task of knowledge graph search. Yet, the methods we proposed could be further extended in a number of ways. The research directions in this area comprise, among others, the study of adaptive machine learning techniques in order to incrementally learn the user preference from the interactions with the system. In this sense, we envision the applicability of learning-to-rank techniques, like reinforcement learning, that could more accurately capture the user preferences in a dynamic context. Moreover, since our approach is currently limited in proposing single-edge expansions, we aim to study approaches that are able to present the user with more complex structures instead. To this end, by mining meta-paths, or frequent patterns, we could both reduce the number of required interactions with the system, and also provide more rich suggestions.

Evaluating Distribution and Concurrency. Regarding the graph databases, we aim to extend our evaluation methodologies in two directions: distribution and concurrency. These are important extensions that are necessary in order to understand also the scale-out capabilities of the existing systems. In particular, we aim at implementing a set of concurrent workloads of many micro-benchmarking operations that should be executed at different levels of parallelism, i.e., by simulating a number of concurrent clients querying the system at the same time. We also plan to extend our use of containers to enable the set-up, of the systems that allow it, on a cluster of machines. This will also help to understand at which scale distribution can become necessary or helpful.

The Graph Exploration System. Finally, another goal is to integrate the querying techniques proposed in this work with the existing graph databases systems, with a unified client endpoint. The desired outcome would be a complete system that should enable efficient and effective explorations of any information graph. This has implications at the application level, at the language level, and also at the implementation level. To this end, we will need to overcome a number of technical challenges in order to match their current capabilities to the needs of the techniques studied in this work. In

particular, the system should be able to easily accept and understand keyword queries or natural language exemplar queries, provide useful query suggestions, learn user preferences, quickly process and return results with on-line performances, provide useful visualizations and summarizations of the results, and allow to refine and process them with traditional graph query techniques when needed. Such *graph exploration system*, whose core components have been studied in this dissertation, would constitute an invaluable tool to allow access to and exploration of information graphs, for both novice and expert users.

List of Figures

List of Tables

Bibliography

[ABLP+14] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. The linked data benchmark council: A graph and rdf industry benchmarking effort. *SIGMOD Rec.*, 43(1):27–31, May 2014. (Cited on page 38.)

[ACL06] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 475–486. IEEE, 2006. (Cited on page 22.)

[ADK16] Marcelo Arenas, Gonzalo I Diaz, and Egor V Kostylev. Reverse engineering sparql queries. In *WWW*, pages 239–249, 2016. (Cited on pages 28 and 29.)

[ADP80] Giorgio Ausiello, Alessandro D'Atri, and Marco Protasi. Structure preserving reductions among convex optimization problems. *JCSS*, 21(1), 1980. (Cited on page 56.)

[AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008. (Cited on pages 11 and 36.)

[Ang12] Renzo Angles. A comparison of current graph database models. In *ICDEW*, pages 171–177, 2012. (Cited on pages 36 and 105.)

[APPDSLP13] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. Benchmarking database systems for social network applications. In *GRADES*, pages 15:1–15:7, New York, NY, USA, 2013. ACM. (Cited on page 38.)

[ara] Arangodb. `https://www.arangodb.com/`. (Cited on pages 11 and 97.)

[ATV08] B. Alexe, W. C. Tan, and Y. Velegrakis. Stbenchmark: towards a benchmark for mapping systems. *PVLDB*, 1(1):230–244, 2008. (Cited on pages 13 and 14.)

[Bar09] Albert-László Barabási. Scale-free networks: a decade and beyond. *science*, 325(5939):412–413, 2009. (Cited on page 57.)

[Bas14] Bast, Hannah and Baurle, Florian and Buchhold, Bjorn and Haussmann, Elmar. Easy access to the freebase dataset. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 95–98, New York, NY, USA, 2014. ACM. (Cited on pages 57, 70, 71, 115, 145, and 146.)

[BB13] Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 175–188. ACM, 2013. (Cited on page 25.)

[BCL15] Angela Bonifati, Radu Ciucanu, and Aurélien Lemay. Learning path queries on graph databases. In *EDBT*, 2015. (Cited on pages 25, 26, 27, and 28.)

[BCMT13] Roi Blanco, Berkant Barla Cambazoglu, Peter Mika, and Nicolas Torzec. Entity recommendations in web search. ISWC '13, pages 33–48, 2013. (Cited on page 35.)

[BD84] Haran Boral and David J. Dewitt. A methodology for database system performance evaluation. In *Proceedings of the International Conference on Management of Data*, pages 176–185, 1984. (Cited on pages 13 and 103.)

[ber] BerkeleyDB. `http://www.oracle.com/technetwork/products/berkeleydb`. (Cited on page 100.)

[BG14] Antoine Bordes and Evgeniy Gabrilovich. Constructing and mining web-scale knowledge graphs: Kdd 2014 tutorial. In *KDD*, pages 1967–1967, 2014. (Cited on page 43.)

[bla] Systap, llc., blazegraph. `https://www.blazegraph.com/`. (Cited on pages 11 and 97.)

[BM06] Vladimir Batagelj and Andrej Mrvar. Yeast, pajek dataset. `http://vlado.fmf.uni-lj.si/pub/networks/data/`, 2006. `http://vlado.fmf.uni-lj.si/pub/networks/data/`. (Cited on page 115.)

[Boe15] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015. (Cited on page 111.)

[BS96] Richard C. Bodner and Fei Song. Knowledge-based approaches to query expansion in information retrieval. In *Canadian AI*, 1996. (Cited on page 34.)

[BZC+03] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, et al. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic acids research*, 31(9):2443–2450, 2003. (Cited on page 115.)

[cas] Apache Cassandra. `http://cassandra.apache.org`. (Cited on page 100.)

[CDSE05] Michael J Cafarella, Doug Downey, Stephen Soderland, and Oren Etzioni. Knowitnow: Fast, scalable information extraction from the web. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 563–570. Association for Computational Linguistics, 2005. (Cited on page 5.)

[Cha07] Soumen Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW*, pages 571–580, 2007. (Cited on page 71.)

[CHI+15] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. Graphalytics: A big data benchmark

for graph-processing platforms. In *GRADES*, pages 7:1–7:6, New York, NY, USA, 2015. ACM. (Cited on pages 11 and 36.)

[CKCO11] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Ozsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011. (Cited on page 11.)

[CNGR08] Guihong Cao, Jian-Yun Nie, Jianfeng Gao, and Stephen Robertson. Selecting good expansion terms for pseudo-relevance feedback. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 243–250, New York, NY, USA, 2008. ACM. (Cited on pages 4, 9, 34, 35, and 81.)

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Symposium on Theory of Computing*, 1971. (Cited on page 49.)

[CR12] Claudio Carpineto and Giovanni Romano. A survey of automatic query expansion in information retrieval. *ACM Comput. Surv.*, 44(1):1:1–1:50, January 2012. (Cited on page 34.)

[CZY13] Jiefeng Cheng, Xianggang Zeng, and Jeffrey Xu Yu. Top-k graph pattern matching over large graphs. In *ICDE*, pages 1033–1044. IEEE, 2013. (Cited on pages 2, 5, 7, 8, 9, 10, 35, and 74.)

[DAB16a] Gonzalo Diaz, Marcelo Arenas, and Michael Benedikt. SPARQLByE: Querying RDF data by example. *PVDLB*, 9(13), September 2016. (Cited on pages 4 and 60.)

[DAB16b] Gonzalo Diaz, Marcelo Arenas, and Michael Benedikt. Sparqlbye: Querying rdf data by example. *Proceedings of the VLDB Endowment*, 9(13):1533–1536, 2016. (Cited on pages 28 and 29.)

[DFK+04] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition. *Mach. Learn.*, 56(1-3):9–33, June 2004. (Cited on page 11.)

[DG16] D. Deutch and A. Gilad. Qplain: Query by explanation. In *ICDE*, pages 1358–1361, 2016. (Cited on pages 5 and 33.)

[DGH+14] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. KDD '14, pages 601–610, 2014. (Cited on pages 2 and 5.)

[DlH10] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010. (Cited on page 27.)

[doc] Docker inc., docker. `https://www.docker.com/`. (Cited on page 111.)

[DPD14] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-example: An automatic query steering framework for interactive data exploration. In *SIGMOD*, pages 517–528. ACM, 2014. (Cited on pages 5 and 33.)

[DSUBGVn+10] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, pages 37–48, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on pages 12, 13, 14, and 37.)

[DVMT14] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. R2g: a tool for migrating relations to graphs. In *EDBT*, pages 640–643, 2014. (Cited on page 43.)

[EALP+15] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015. (Cited on pages 13, 38, 105, 109, and 115.)

[EASK14] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, March 2014. (Cited on page 115.)

[EB11] Shady Elbassuoni and Roi Blanco. Keyword search over rdf graphs. In *Proceedings of the 20th ACM international conference on Information*

and knowledge management, pages 237–242. ACM, 2011. (Cited on page 2.)

[ela] Elasticsearch. `http://www.elastic.co/products/elasticsearch`. (Cited on page 100.)

[FRP15] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015. (Cited on pages 11, 12, and 36.)

[GFMPdlF11] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. In *USEWOD Workshop-WWW*, 2011. (Cited on pages 27 and 28.)

[GGL+13] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. ACM, 2013. (Cited on page 93.)

[GGY+] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. Top-k interesting subgraph discovery in information networks. In *ICDE'14*. (Cited on pages 46 and 57.)

[GMU15] Aristides Gionis, Michael Mathioudakis, and Antti Ukkonen. Bump hunting in the dark: Local discrepancy maximization on graphs. In *ICDE*, pages 1155–1166, 2015. (Cited on pages 17, 19, and 20.)

[Goo14] Google. Freebase data dumps. `https://developers.google.com/freebase/data`, 2014. (Cited on pages 50, 61, 86, and 115.)

[gra] Ontotext graphdb. `http://graphdb.ontotext.com/`. (Cited on page 95.)

[GSSZ15] Oshini Goonetilleke, Saket Sathe, Timos Sellis, and Xiuzhen Zhang. Microblogging queries on graph databases: An introspection. In *GRADES*, pages 5:1–5:6, New York, NY, USA, 2015. ACM. (Cited on pages 11 and 91.)

[GTK01] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *SIGMOD Record*, volume 30, pages 461–472, 2001. (Cited on page 53.)

[GXX13] Jianfeng Gao, Gu Xu, and Jinxi Xu. Query expansion using path-constrained random walks. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 563–572, New York, NY, USA, 2013. ACM. (Cited on pages 5, 34, and 35.)

[Har88] D. Harman. Towards interactive query expansion. In *Proceedings of the 11th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '88, pages 321–331, New York, NY, USA, 1988. ACM. (Cited on page 34.)

[hba] Apache Hbase. `http://hbase.apache.org`. (Cited on page 100.)

[HBB15] Faegheh Hasibi, Krisztian Balog, and Svein Erik Bratsberg. Entity linking in queries: Tasks and evaluation. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, ICTIR '15, pages 171–180, New York, NY, USA, 2015. ACM. (Cited on page 35.)

[HBC15] Mokhtaria Hacherouf, Safia Nait Bahloul, and Christophe Cruz. Transforming xml documents to owl ontologies: A survey. *Journal of Information Science*, 41(2):242–259, 2015. (Cited on page 43.)

[HDA+14] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Ozsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, August 2014. (Cited on pages 11, 12, 36, and 91.)

[HLL13] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, 2013. (Cited on page 48.)

[HP13] Florian Holzschuher and René Peinl. Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j.

In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 195–204, New York, NY, USA, 2013. ACM. (Cited on page 111.)

[HRW92] Frank K Hwang, Dana S Richards, and Pawel Winter. *The Steiner tree problem*, volume 53. Elsevier, 1992. (Cited on page 19.)

[HT11] Jan Hauke and Kossowski Tomasz. Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data. *Quaestiones Geographicae*, 30(2):87–93, 2011. (Cited on page 64.)

[HWZ11] Guangjun Huang, Shuili Wang, and Xiaoguo Zhang. Query expansion based on associated semantic space. *Journal of Computers*, 6(2), 2011. (Cited on page 34.)

[inf] Infinitegraph. `http://www.objectivity.com/products/infinitegraph`. (Cited on page 95.)

[IRV13] E. Ioannou, N. Rassadko, and Y. Velegrakis. On generating benchmark data for entity matching. *J. Data Semantics*, 2(1):37–56, 2013. (Cited on pages 13 and 14.)

[jan] The linux foundation, janusgraph. `http://janusgraph.org/`. (Cited on page 11.)

[JCE+07] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD*, 2007. (Cited on page 4.)

[JGK+14] Nandish Jayaram, Mahesh Gupta, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. GQBE: Querying knowledge graphs by example entity tuples. *ICDE*, pages 1250–1253, 2014. (Cited on pages 8, 9, 10, 35, 39, 74, and 86.)

[JKL+15] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. Querying knowledge graphs by example entity tuples. *TKDE*, 27(10), 2015. (Cited on pages 5, 6, 7, 29, 30, 33, 44, 62, and 67.)

[JV13] Salim Jouili and Valentin Vansteenberghe. An empirical comparison of graph databases. In *Proceedings of the 2013 International Conference*

on Social Computing, SOCIALCOM '13, pages 708–715, Washington, DC, USA, 2013. IEEE Computer Society. (Cited on pages 12, 13, 14, 37, and 105.)

[JW03] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *WWW*, 2003. (Cited on pages 17 and 68.)

[KA11] Mehdi Kargar and Aijun An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011. (Cited on page 47.)

[KH11] Vitaly Klyuev and Yannis Haralambous. Query expansion: Term selection using the ewc semantic relatedness measure. *CoRR*, abs/1108.4052, 2011. (Cited on page 34.)

[KK14] Isabel M Kloumann and Jon M Kleinberg. Community membership identification from small seed sets. In *KDD*, 2014. (Cited on pages 16, 17, and 18.)

[KRS+09] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, and Gerhard Weikum. Star: Steiner-tree approximation in relationship graphs. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *ICDE*. IEEE Computer Society, 2009. (Cited on pages 34 and 47.)

[KRS10] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. Route skyline queries: A multi-preference path planning approach. In *ICDE*, pages 261–272, 2010. (Cited on page 43.)

[KRSW09] Gjergji Kasneci, Maya Ramanath, Fabian Suchanek, and Gerhard Weikum. The yago-naga approach to knowledge discovery. *SIGMOD Rec.*, 37(4):41–47, March 2009. (Cited on page 5.)

[KSM13] Vojtěch Kolomičenko, Martin Svoboda, and Irena Holubová Mlýnková. Experimental comparison of graph databases. In *IIWAS*, pages 115:115–115:124, 2013. (Cited on pages 12, 13, 14, 37, and 105.)

[LBG+12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A

framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012. (Cited on pages 11 and 12.)

[LBV17] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. The Trento GDB Test Suite. `https://disi.unitn.it/~lissandrini/gdb.html`, 2017. (Cited on pages 111, 114, and 140.)

[LBV18] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. A micro-benchmarking approach for comparing and understanding graph databases. In *Under Submission*, 2018. (Cited on page 4.)

[LC01] Victor Lavrenko and W. Bruce Croft. Relevance based language models. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, pages 120–127, New York, NY, USA, 2001. ACM. (Cited on pages 35 and 81.)

[LCYW14] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292, November 2014. (Cited on pages 11 and 36.)

[Lis17] Matteo Lissandrini. Freebase exq data dump. `https://disi.unitn.it/~lissandrini/notes/freebase-data-dump.html`, 2017. (Cited on page 115.)

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014. (Cited on page 62.)

[LKDL12] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *ICDE*, pages 666–677. IEEE, 2012. (Cited on page 33.)

[LMP+15] Matteo Lissandrini, Davide Mottin, Themis Palpanas, Dimitra Papadimitriou, and Yannis Velegrakis. Unleashing the power of information graphs. *SIGMOD Rec.*, 43(4):21–26, February 2015. (Cited on pages 3, 5, 11, and 91.)

[LMPV18a] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. Interactive graph ery expansion for exploratory search on knowledge graphs. In *Under Submission*, 2018. (Cited on page 4.)

[LMPV18b] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. X2q: Your personal example-based graph explorer. In *Under Submission*, 2018. (Cited on page 4.)

[LMVP18] Matteo Lissandrini, Davide Mottin, Yannis Velegrakis, and Themis Palpanas. Simple multi-example search in complex information spaces. In *ICDE*, 2018. (Cited on pages 3, 8, and 9.)

[luc] Apache Lucene. `http://lucene.apache.org`. (Cited on page 98.)

[LZ17] John Lafferty and Chengxiang Zhai. Document language models, query models, and risk minimization for information retrieval. *SIGIR Forum*, 51(2):251–259, August 2017. (Cited on pages 35 and 81.)

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010. (Cited on pages 11 and 12.)

[Mar] Pieter Martin. Sqlg. `http://www.sqlg.org/`. (Cited on page 99.)

[MBALMM+12] Norbert Martínez-Bazan, M. Ángel Águila Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L. Larriba-Pey. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, IDEAS '12, pages 110–119, New York, NY, USA, 2012. ACM. (Cited on pages 99 and 102.)

[MBGVEC11] Norbert Martinez-Bazan, Sergio Gomez-Villamor, and Francesc Escale-Claveras. Dex: A high-performance graph database management system. In *ICDEW*, pages 124–127, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 99.)

[MBL15] Iris Miliaraki, Roi Blanco, and Mounia Lalmas. From "selena gomez" to "marlon brando": Understanding explorative entity search. WWW '15, pages 765–775, 2015. (Cited on pages 1, 8, and 74.)

[MCF+14] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Strong simulation: Capturing topology in graph pattern matching. *TODS*, 39(1):4, 2014. (Cited on pages 31 and 65.)

[MCM+15] Changping Meng, Reynold Cheng, Silviu Maniu, Pierre Senellart, and Wangda Zhang. Discovering meta-paths in large heterogeneous information networks. In *WWW*, pages 754–764, 2015. (Cited on page 85.)

[mes] Apache Mesos. `http://mesos.apache.org`. (Cited on page 97.)

[Met07] D. Metzler. Beyond bags of words: Effectively modeling dependence and features in information retrieval. phd, University of Massachusetts Amherst, 2007. (Cited on page 82.)

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association. (Cited on page 93.)

[MLVP14a] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014. (Cited on pages 3, 5, 31, and 32.)

[MLVP14b] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Searching with xq: the exemplar query search engine. In *SIGMOD*, pages 901–904. ACM, 2014. (Cited on pages 3, 4, and 5.)

[MLVP16a] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: a new way of searching. *VLDB J.*, pages 1–25, 2016. (Cited on pages 3, 5, 6, 7, 8, 9, 10, 31, 32, 33, 35, 36, 39, 42, 44, 47, 48, 50, 62, 63, 65, 67, 68, 74, 82, 84, and 86.)

[MLVP16b] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: A new way of searching. *The VLDB Journal*, 25(6):741–765, December 2016. (Cited on page 115.)

[MLVP17] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. New trends on exploratory methods for data analytics. *Proceedings of the VLDB Endowment*, 10(12):1977–1980, 2017. (Cited on pages 4 and 6.)

[MLZ17] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 631–643. USENIX Association, 2017. (Cited on page 36.)

[MSS13] Steffen Metzger, Ralf Schenkel, and Marcin Sydow. Qbees: query by entity examples. In *CIKM*, pages 1829–1832, 2013. (Cited on pages 5, 23, 24, 30, 33, 35, 43, and 74.)

[NC10] Vuong M. Ngo and Tru H. Cao. Ontology-based query expansion with latently related named entities for semantic text search. In *IJIIDS*. 2010. (Cited on page 34.)

[NDC15] Thi-Nhu Nguyen, Duy-Thanh Dinh, and Tuan-Dung Cao. Empowering exploratory search on linked movie open data with semantic technologies. In *Proceedings of the Sixth International Symposium on Information and Communication Technology*, SoICT 2015, pages 296–303, New York, NY, USA, 2015. ACM. (Cited on page 5.)

[neo] Neo technology, inc., neo4j. `http://neo4j.com`. (Cited on pages 11, 86, 91, 98, and 142.)

[NKZ16] Fedor Nikolaev, Alexander Kotov, and Nikita Zhiltsov. Parameterized fielded term dependence models for ad-hoc entity retrieval from knowledge graph. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '16, pages 435–444, New York, NY, USA, 2016. ACM. (Cited on page 35.)

[NW08] Thomas Neumann and Gerhard Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008. (Cited on page 33.)

[NW09] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD*, pages 627–640. ACM, 2009. (Cited on page 33.)

[Oka] Junjiro Okajima. Aufs: Advanced multi layered unification filesystem. `http://aufs.sourceforge.net/`. (Cited on page 111.)

[ori] Orient technologies, orientdb. `http://orientdb.com/orientdb/`. (Cited on pages 11 and 98.)

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009. (Cited on page 27.)

[PAISM14] Bryan Perozzi, Leman Akoglu, Patricia Iglesias Sánchez, and Emmanuel Müller. Focused clustering and outlier detection in large attributed graphs. In *KDD*, pages 1346–1355, 2014. (Cited on page 21.)

[PBMW] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. TR 1999-66, Stanford InfoLab, Nov. (Cited on page 11.)

[PC98] Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, pages 275–281, New York, NY, USA, 1998. ACM. (Cited on pages 9, 35, and 79.)

[PDCC] Fotis Psallidas, Bolin Ding, Kaushik Chakrabarti, and Surajit Chaudhuri. S4: Top-k spreadsheet-style search for query discovery. In *SIGMOD'15*. (Cited on pages 5, 6, 33, 39, and 43.)

[PIW10] Jeffrey Pound, Ihab F. Ilyas, and Grant Weddell. Expressive and flexible access to web-extracted data: A keyword-based structured query language. SIGMOD '10, pages 423–434, 2010. (Cited on pages 5 and 35.)

[PPP05] Stephen R Proulx, Daniel EL Promislow, and Patrick C Phillips. Network thinking in ecology and evolution. *Trends in Ecology & Evolution*, 20(6):345–353, 2005. (Cited on page 43.)

[PS+08] Eric Prud'Hommeaux, Andy Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008. (Cited on page 99.)

[QF93] Yonggang Qiu and Hans-Peter Frei. Concept based query expansion. In *SIGIR*, 1993. (Cited on page 34.)

[RBGS+15] Natali Ruchansky, Francesco Bonchi, David García-Soriano, Francesco Gullo, and Nicolas Kourtellis. The minimum wiener connector problem. In *SIGMOD*, pages 1587–1602, 2015. (Cited on pages 17, 18, and 19.)

[RND+12] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using hadoop on a cluster. In *HotCDP*, pages 2:1–2:5, 2012. (Cited on page 93.)

[Roc71] Joseph John Rocchio. Relevance feedback in information retrieval. *The Smart retrieval system-experiments in automatic document processing*, 1971. (Cited on page 81.)

[Rod15] Marko A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *DBPL*, pages 1–10, 2015. (Cited on pages 92 and 111.)

[RSH+16] Thomas Rebele, Fabian M. Suchanek, Johannes Hoffart, Joanna Biega, Erdal Kuzey, and Gerhard Weikum. YAGO: A multilingual knowledge base from wikipedia, wordnet, and geonames. In *ISWC*, 2016. (Cited on pages 2 and 61.)

[Rut03] Ian Ruthven. Re-examining the potential effectiveness of interactive query expansion. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, pages 213–220, New York, NY, USA, 2003. ACM. (Cited on page 34.)

[SBDK09] Nikos Sarkas, Nilesh Bansal, Gautam Das, and Nick Koudas. Measure-driven keyword-query expansion. *Proceedings of the VLDB Endowment*, 2(1):121–132, 2009. (Cited on page 83.)

[SCC+] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. In *SIGMOD'14*. (Cited on pages 6, 39, and 43.)

[SCSS15] Grzegorz Sobczak, Mateusz Chochół, Ralf Schenkel, and Marcin Sydow. iqbees: Towards interactive semantic entity search based on maximal aspects. In *Foundations of Intelligent Systems*, pages 259–264. Springer, 2015. (Cited on pages 2, 5, 8, 23, 24, 27, and 82.)

[SKW07] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007. (Cited on pages 2, 11, and 91.)

[spa] Sparsity technologies, sparksee. `http://www.sparsity-technologies.com/`. (Cited on pages 11 and 99.)

[SVJ14] Bahareh Sarrafzadeh, Olga Vechtomova, and Vlado Jokic. Exploring knowledge graphs for exploratory search. In *Proceedings of the 5th Information Interaction in Context Symposium*, IIiX '14, pages 135–144, New York, NY, USA, 2014. ACM. (Cited on pages 5, 8, and 35.)

[SVLV16] Bahareh Sarrafzadeh, Alexandra Vtyurina, Edward Lank, and Olga Vechtomova. Knowledge graphs versus hierarchies: An analysis of user behaviours and perspectives in information seeking. In *Proceedings of the 2016 ACM on Conference on Human Information Interaction and Retrieval*, CHIIR '16, pages 91–100, New York, NY, USA, 2016. ACM. (Cited on pages 5 and 35.)

[SWW+11] Yangqiu Song, Haixun Wang, Zhongyuan Wang, Hongsong Li, and Weizhu Chen. Short text conceptualization using a probabilistic knowledgebase. In *IJCAI*, 2011. (Cited on pages 5 and 35.)

[SYS+15] Yu Su, Shengqi Yang, Huan Sun, Mudhakar Srivatsa, Sue Kase, Michelle Vanni, and Xifeng Yan. Exploiting relevance feedback in

knowledge graph search. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1135–1144, New York, NY, USA, 2015. ACM. (Cited on page 35.)

[Tar72] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING*, 1(2), 1972. (Cited on page 11.)

[TDH05] Jaime Teevan, Susan T. Dumais, and Eric Horvitz. Personalizing search via automated analysis of interests and activities. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '05, pages 449–456, New York, NY, USA, 2005. ACM. (Cited on page 35.)

[TF06] Hanghang Tong and Christos Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 404–413. ACM, 2006. (Cited on pages 17 and 18.)

[tin] Apache tinkerpop. `http://tinkerpop.apache.org/`. (Cited on pages 94 and 111.)

[tit] Thinkaurelius, titan. `http://titan.thinkaurelius.com/`. (Cited on pages 11 and 100.)

[TVFZ07] Bin Tan, Atulya Velivelli, Hui Fang, and ChengXiang Zhai. Term feedback for information retrieval with language models. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 263–270, New York, NY, USA, 2007. ACM. (Cited on page 34.)

[Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), January 1976. (Cited on page 48.)

[UNH+17] Ricardo Usbeck, Axel-Cyrille Ngonga Ngomo, Bastian Haarmann, Anastasia Krithara, Michael Röder, and Giulio Napolitano. *7th Open Challenge on Question Answering over Linked Data (QALD-7)*, pages 59–69. Springer International Publishing, Cham, 2017. (Cited on page 86.)

[WBTS14] Andreas Wagner, Veli Bicer, Thanh Tran, and Rudi Studer. Holistic and compact selectivity estimation for hybrid queries over rdf graphs. In *ISWC*, pages 97–113, 2014. (Cited on page 53.)

[Wie47] Harry Wiener. Structural determination of paraffin boiling points. *Journal of the American Chemical Society*, 69(1):17–20, 1947. (Cited on page 18.)

[WKW+10] Jörg Waitelonis, Magnus Knuth, Lina Wolf, Johannes Hercher, and Harald Sack. The path is the destination – enabling a new search paradigm with linked data. In *In Proc. of the Workshop on Linked Data in the Future Internet at the Future Internet Assembly, Dec 16–17, 2010, Ghent, Belgium, CEUR Workshop Proc*, 2010. (Cited on pages 1, 5, and 8.)

[Woo12] Peter T Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, 2012. (Cited on page 25.)

[WR09] Ryen W. White and Resa A. Roth. *Exploratory Search: Beyond the Query-Response Paradigm*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan and Claypool Publishers, 2009. (Cited on pages 1, 6, and 8.)

[XBCW17] Miao Xie, Sourav S. Bhowmick, Gao Cong, and Qing Wang. PANDA: Toward partial topology-based search on large networks in a single machine. *VLDBJ*, 26(2), April 2017. (Cited on pages 6, 34, 62, 63, 65, and 67.)

[XJRN03] Eric P Xing, Michael I Jordan, Stuart J Russell, and Andrew Y Ng. Distance metric learning with application to clustering with side-information. In *Advances in neural information processing systems*, pages 521–528, 2003. (Cited on page 21.)

[YBT+16] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, and James Cheng. Big graph analytics systems. In *SIGMOD*, pages 2241–2243, 2016. (Cited on page 11.)

[YGCC12] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. Infogather: Entity augmentation and attribute discovery

by holistic matching with web tables. SIGMOD '12, pages 97–108, 2012. (Cited on pages 6 and 39.)

[YHWY16] Shengqi Yang, Fangqiu Han, Yinghui Wu, and Xifeng Yan. Fast top-k search in knowledge graphs. In *ICDE*. IEEE, 2016. (Cited on page 53.)

[ZCC+17] Xiangling Zhang, Yueguo Chen, Jun Chen, Xiaoyong Du, Ke Wang, and Ji-Rong Wen. Entity set expansion via knowledge graphs. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 1101–1104, New York, NY, USA, 2017. ACM. (Cited on pages 5, 8, 35, and 74.)

[ZCCW09] Ce Zhang, Bin Cui, Gao Cong, and Yu-Jing Wang. A revisit of query expansion with different semantic levels. In *DASFAA*, 2009. (Cited on pages 5 and 35.)

[ZCY+17] Qizhen Zhang, Hongzhi Chen, Da Yan, James Cheng, Boon Thau Loo, and Purushotham Bangalore. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 40–51, New York, NY, USA, 2017. ACM. (Cited on page 36.)

[ZH10] P. Zhao and J. Han. On graph query optimization in large networks. *VLDB J.*, 3(1-2):340–351, 2010. (Cited on pages 2, 5, 8, and 9.)

[Zha02] Liecai Zhang. *Knowledge Graph Theory and Structrural Parsing*. Twente University Press, 2002. (Cited on page 2.)

[ZL01a] Chengxiang Zhai and John Lafferty. The dual role of smoothing in the language modeling approach. In *Proceedings of the Workshop on Language Models for Information Retrieval (LMIR) 2001*, pages 31–36, 2001. (Cited on page 80.)

[ZL01b] Chengxiang Zhai and John Lafferty. Model-based feedback in the language modeling approach to information retrieval. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 403–410, New York, NY, USA, 2001. ACM. (Cited on pages 81 and 82.)

[ZL01c] Chengxiang Zhai and John Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, pages 334–342, New York, NY, USA, 2001. ACM. (Cited on page 80.)

[Zlo75] Moshé M. Zloof. Query by example. In *AFIPS NCC*, pages 431–438, 1975. (Cited on pages 28, 29, and 33.)

[ZW14] Mingzhu Zhu and Yi-Fang Brook Wu. Search by multiple examples. In *WSDM*, 2014. (Cited on pages 5 and 43.)