UNIVERSITÀ DEGLI STUDI DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

CYCLE XXXIII

UNDERSTANDING AND MANAGING COMPLEX DATASETS

MARTIN BRUGNARA

Advisor:

Yannis Velegrakis
University of Trento, Trento - Italy
Utrecht University, Utrecht - the Netherlands

2022

“Bla bla bla. Love you. Thanks, bye!”

ABSTRACT

Nowadays, we are producing and collecting data at an unprecedented rate, measured in the order of petabytes per minute, together with a substantial increase in data volume, complexity, and variety. While tabular and unstructured data still dominate the scene, graphs are becoming ever more prominent, bringing new challenges. The size and complexity of graph datasets have increased, thus renewing the interest in graph databases and distributed graph processing. The current abundance of data and content complicates even simply accessing data. Web users are constantly overwhelmed by the availability of information and rely upon search engine and recommender systems to get a trusted personalized selection. Since people tend to prefer sources that reinforce their pre-existing beliefs, such systems optimize for whatever the users like. Whenever a controversial topic arises, users get more polarized as they see only a part of the reality and feel support from others who hold the same view. This feedback loop leads to "filter bubbles" and "echo chambers", new problems tackled by researchers in the social sciences. On the other hand, scientists and data analysts have a hard time navigating the different data lakes and data repositories with the data deluge. Thus, new tools need to help data scientists explore and understand data to maximize the value they can extract by processing them.

This thesis contributes to solving these issues by studying and evaluating the existing graph database technologies to reveal the implications of different design decisions. It offers a principled and systematic evaluation methodology based on microbenchmarks comprising tests for more than 51 classes of operations and graphs with up to 30M nodes and 178M edges. The framework has been materialized into an evaluation suite and executed against the major graph databases available today. The gathered results proved effective for better understanding graph databases systems' design choices, performances, and functionalities. Findings include analysis of the tradeoffs between native and hybrid graph database systems, their effect on important graph queries like traversals and pattern matching, and their current capability to handle highly heterogeneous graphs.

This thesis also contributes to the efficient processing of distributed graphs whose data is partitioned by other systems, like externally managed by graph databases. In particular, it provides a novel technique for k-core decomposition and maintenance. The solution has been implemented on top of AKKA and tested on various real and synthetic datasets. Results show that it efficiently exploits as much as possible the existing topology of the graph achieving shorter run-

ning time and higher scalability compared to existing sequential and distributed approaches.

To tackle news polarisation, this thesis proposes two novel recommender systems that account for different points of view expressed in a document and offer a holistic overview of the topic at hand. The first, Orthogonal-topics, focuses on the relationship of the topics, and it has been designed to generalize well on all datasets. The second, Sentimented-topics, focuses on the sentiment expressed by the documents on the different topics, and it has been designed to extract and exploit as much information as possible from text corpora that contain opinionated articles. Moreover, a new diversity-metric, MIN-BW, and a new optimization algorithm, FDLS, are provided to support these approaches in finding the most diverse set based on the metric mentioned above. For MIN-BW, a set of documents is modeled as a system of particles with repulsive forces, where the most diverse set is the one whose system requires less work to balance, *i. e.*, to make it statically stable. The results of a user study showed the superior quality of the recommendation of our approach, and further test on synthetic data showed the superior scalability of FDLS.

Finally, to aid researchers in navigating through data lakes, this thesis provides a new solution to the generations of compact and informative summaries of the contents of a dataset to enable a more systematic approach to data exploration. The task is modeled as a multi-objective optimization problem. We formally define the notion of a data description and the intuition behind the concept of goodness for such a description. Descriptions are modeled as sets of views over the datasets, where the views are defined as filtering clauses. Four factors determine the quality of a description: length, coverage on the dataset, overlap, and intricacy. We thus provide three algorithms that generate such descriptions given these four optimization objectives. Results showed the scalability and applicability of our approaches.

With this thesis, we have contributed to improving and scaling data management, processing, and exploration, which are fundamental tasks in big data and knowledge management both from a research and a business perspective.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[1] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. "Distributed k-core decomposition and maintenance in large dynamic graphs." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*. Ed. by Avigdor Gal, Matthias Weidlich, Vana Kalogeraki, and Nalini Venkasubramanian. ACM, 2016, pp. 161–168. DOI: 10.1145/2933267.2933299. URL: https://doi.org/10.1145/2933267.2933299.

[2] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. "Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation." In: *Proc. VLDB Endow.* 12.4 (2018), pp. 390–403. DOI: 10.14778/3297753.3297759. URL: http://www.vldb.org/pvldb/vol12/p390-lissandrini.pdf.

ACKNOWLEDGEMENTS

Voglio ringraziare mia madre Lorenza, mio padre Stefano, e mia sorella Giada per essermi stati vicini ed avermi supportato continuamente e incondizionatamente sin dal mio principio.

I want to thank my supervisor, professor Yannis Velegrakis, for advising me since my bachelor's and providing me with many opportunities to grow.

I am grateful to my friends
who walk with me through this adventure called life,
who help me through the hard times, and
who celebrate my achievements.

CONTENTS

1 INTRODUCTION 1

2 MICRO-BENCHMARKING GRAPH DATABASES 7
2.1 Related Work . . . 10
2.1.1 Evaluating Graph Processing Systems . . . 10
2.1.2 Evaluating Graph Databases . . . 10
2.1.3 Distribution & Cluster Evaluation . . . 11
2.1.4 Graph Benchmarks . . . 11
2.2 Graph Databases . . . 11
2.2.1 The Graph Data Model . . . 12
2.2.2 Implementing a Graph Database . . . 12
2.2.3 The Heterogeneity Problem in Graphs . . . 13
2.3 Test Operations: Queries . . . 14
2.3.1 [L] Load Operations . . . 16
2.3.2 [C] Create Operations . . . 16
2.3.3 [R] Read Operations . . . 17
2.3.4 [U] Update Operations . . . 17
2.3.5 [D] Delete Operations . . . 17
2.3.6 [T] Traversals . . . 18
2.3.7 [P] Pattern Matching . . . 19
2.3.8 Complex Query Set . . . 19
2.4 Evaluation Suite . . . 19
2.4.1 Requirements . . . 19
2.4.2 Technological Solutions . . . 20
2.4.3 Evaluation Framework . . . 21
2.5 Systems . . . 24
2.5.1 Native . . . 27
2.5.2 Hybrid . . . 28
2.5.3 RDF . . . 29
2.5.4 Query Processing and Evaluation . . . 30
2.6 Datasets . . . 30
2.6.1 Set A . . . 30
2.6.2 Set B . . . 31
2.6.3 Datasets Characteristics . . . 32
2.7 Experimental Setup . . . 33
2.7.1 Hardware . . . 33
2.7.2 GDB configuration . . . 34
2.8 Results . . . 34
2.8.1 Data Loading . . . 35
2.8.2 Complex Queries . . . 38
2.8.3 Micro-benchmark Results . . . 39
2.8.4 Fixed Traversal and Pattern Matching . . . 48

2.8.5 Progress across Versions . . . 51
2.8.6 Overall Evaluation and Insights . . . 51
2.9 Conclusion . . . 55

3 COMPUTING & MAINTAINING K–CORE 57
3.1 Related work . . . 58
3.1.1 Centralized algorithms . . . 58
3.1.2 Distributed algorithms . . . 59
3.1.3 k-core & dynamic graphs . . . 59
3.2 Problem formulation . . . 60
3.3 k-core computation . . . 61
3.4 Experiments . . . 65
3.4.1 Experimental data . . . 66
3.4.2 Experimental environment . . . 66
3.4.3 Experimental protocol . . . 67
3.4.4 Experimental results . . . 68
3.5 Conclusions . . . 71

4 TOPIC RECOMMENDATION: EXPAND YOUR HORIZON 73
4.1 Background . . . 75
4.1.1 Latent Dirichlet Allocation . . . 75
4.1.2 Diversification . . . 76
4.1.3 Unit-Hyper-Sphere . . . 78
4.2 Related . . . 81
4.3 Problem Statement . . . 82
4.4 Solution . . . 82
4.4.1 Approach 1: Orthogonal-topics . . . 82
4.4.2 Approach 2: Sentimented-topics . . . 83
4.4.3 Min Balancing Work . . . 84
4.4.4 FDLS . . . 87
4.5 Evaluation . . . 88
4.5.1 Quality . . . 89
4.5.2 Scalability & Performance . . . 92
4.6 Conclusions and Future Work . . . 95

5 ON DESCRIBING THE CONTENTS OF A DATASET 97
5.1 Motivating Example . . . 98
5.2 Problem Statement . . . 100
5.3 Identifying the best Description . . . 102
5.3.1 Naïve approach . . . 102
5.3.2 Vertical approach . . . 104
5.3.3 Adaptive . . . 112
5.4 Evaluation . . . 115
5.4.1 Results . . . 117
5.5 Related Work . . . 120
5.6 Conclusions . . . 122

6 CONCLUSIONS 123

BIBLIOGRAPHY 127

LIST OF FIGURES

1 INTRODUCTION 1

2 MICRO-BENCHMARKING GRAPH DATABASES 7
Figure 2.1 Example of Graph. 12
Figure 2.2 Evaluation Framework architecture. 22
Figure 2.3 Loading time (Set A). 36
Figure 2.4 Space occupancy (Set A). 37
Figure 2.5 Space occupancy (Set B). 38
Figure 2.6 Complex Query Performance on *ldbc*. 39
Figure 2.7 Timeouts (Set A). 40
Figure 2.8 Time for insertions. 41
Figure 2.9 Time for updates and deletions. 42
Figure 2.10 Time for searching by id. 43
Figure 2.11 Time for general selections. 43
Figure 2.12 Time for neighbors traversal. 44
Figure 2.13 Time for filtering on all nodes. 45
Figure 2.14 Time for BFS. 46
Figure 2.15 Time for SP on Fbr-*, fixed-lbl SP & BFS. . . . 47
Figure 2.16 Time for indexed property search. 47
Figure 2.17 Overall time (Set A). 48
Figure 2.18 Time for SP on *air-routes*. 49
Figure 2.19 Time for SP on *ldbc.10*. 49
Figure 2.20 Time for fixed traversals on *air-routes*. 49
Figure 2.21 Time for fixed traversals on *ldbc.10*. 49
Figure 2.22 Time for pattern matching on *air-routes*. 50
Figure 2.23 Time for pattern matching on *ldbc.10*. 50

3 COMPUTING & MAINTAINING K−CORE 57
Figure 3.1 System overview. 61
Figure 3.2 Average insertion/deletion time. 69
Figure 3.3 Amount of exchanged data. 70
Figure 3.4 #paritions *vs* amount of exchanged data. 70
Figure 3.5 #workers vs insertion/deletion time. 72

4 TOPIC RECOMMENDATION: EXPAND YOUR HORIZON 73
Figure 4.1 Thomson solution prefers *outer* points sacrificing distance and coverage. 86
Figure 4.2 The three different optimization problems have different optimal results. 86
Figure 4.3 User-study interface. 91
Figure 4.4 Scalability tests, avg running time. 93

5 ON DESCRIBING THE CONTENTS OF A DATASET 97
Figure 5.1 A Job Openings dataset. 99
Figure 5.2 Statements on the Jobs dataset. 100
Figure 5.3 Expansion highlights for running example. . . 106
Figure 5.4 Adaptive approach overview. 113
Figure 5.5 Real-World Datasets. 115
Figure 5.6 Number of $\mathcal{W}^1$ in synthetic datasets. 116
Figure 5.7 Execution failures % in **[V]**. 118
Figure 5.8 Mean running time of the 3 approaches on **[V]**. 118
Figure 5.9 Mean running time of the 3 approaches on **[S]**. 119
Figure 5.10 Mean running time of the 3 approaches on **[R]**. 119
Figure 5.11 Adaptive coverage *vs* Vertical. 120

6 CONCLUSIONS 123

LIST OF TABLES

1 INTRODUCTION 1

2 MICRO-BENCHMARKING GRAPH DATABASES 7
Table 2.1 Test Queries. 16
Table 2.2 Tested Systems. 26
Table 2.3 Datasets Characteristics (Set A). 33
Table 2.4 Datasets Characteristics (Set B). 33
Table 2.5 Evaluation Summary. 52

3 COMPUTING & MAINTAINING K−CORE 57
Table 3.1 Notation. 63
Table 3.2 Experiments data. 67
Table 3.3 Experiments results. 68

4 TOPIC RECOMMENDATION: EXPAND YOUR HORIZON 73
Table 4.1 Space used in Figure 4.1 to show Thomson *vs* MIN-BW different behavior. 85
Table 4.2 Example of a space for which each optimization problem yields a different solutions. . . . 86
Table 4.3 Statistics for the optimal solutions, in bold, computed accordingly to the three different optimization problems; they all differ. Recall, the first two maximize whereas the last minimizes. 87
Table 4.4 Mean score for each approach. 92

5 ON DESCRIBING THE CONTENTS OF A DATASET 97
Table 5.1 Domain Pruning on **[V]**: % of residual $|\mathcal{W}^1|$. . 117
Table 5.2 Domain Pruning on **[V]**: Dataset with a non-empty description. 117

6 CONCLUSIONS 123

1 INTRODUCTION

The world is producing data at an unprecedented rate measured petabytes per minute or in hundreds of zettabytes per year [59, 139]. The rate is only destined to grow as more of humanity not only gets connected to the internet but will start to rely on it for work, entertainment, and news. Moreover, the produced data vary widely in complexity and nature.

Data can be unstructured, semi-structured, or structured. Unstructured data can be seen as opaque blob identified by an id, they do not have a predefined schema and known data types, and it must be processed to extract structure and information. Examples of unstructured data include articles, text documents, web pages, and videos. On the contrary, structured data follow a schema that can be either implicit or explicit, and its data types are consistent within a dataset. Examples of structured data include anything with a predefined type, like census data, retail and e-commerce data, and transactional data, such as credit card payments, call logs, and purchase orders.

The data we are dealing with today was nothing like the past. Nevertheless, its evolution path was already being forecasted in 2001 by D. Laney with the definition of *Big Data* and its 3V: *Volume*, *Velocity*, and *Variety* [77]. The definition has since been extended multiple times, and one of the most important additions was Value [69] keyword. The value of the different datum kinds and instances depends upon the value of the information they carry. Research in data mining and knowledge management has focused on several aspects related to the extract of value from data.

From this perspective, one of the most valuable and fast-growing data source are graphs. Graphs, also known as networks, are a form of data used to model and track interactions that usually are non-linear. Graphs have become increasingly important for a wide range of applications [26, 87] and domains, including biological data [25], knowledge graphs [134], and social networks [54].

Moreover, if the value of the information is time-dependent, the value of the datum itself becomes time-dependent. For example, a news article reaches its peak value as soon as it is published and then diminishes as it gets less relevant. A support ticket is most relevant until the problem gets resolved, but its value can be maintained in time by incorporating it into a knowledge base. Instead, the value of sensor-generated monitoring data peeks two times, first while live monitoring for faulty equipment, then once it gets aggregated and

analyzed over time to enable and optimize, for example, preventive maintenance.

More generally, most data has two lives: as a single datum and then as part of a dataset. To extract the maximum value, the data must be: 1st) made easily and efficiently accessible by the primary consumer, being it a person for articles, a technician for support tickets, or the alarm system for sensor data; 2nd) securely and durably stored to allow aggregated processing and analysis at a later time. This dual problem of keeping the data available while also safely storing it, has been widely studied since the advent of the computer. Different, specialized solutions have been devised, each accounting for the different data types, velocity, scale, and requirements of the consuming services. Some solutions focus on the first life of the data where it is created, updated, and manipulated, often in an interactive setting (OLTP); others concern themselves only with the second one offering high performance, possibly distributed, batch processing (OLAP). Two data types of particular interest are tabular data (structural data organized in a table) and graph data. Tabular data is the most mature domain; for example, RDBMS have grown to support a greater volume of data and more complex workloads. Entire frameworks, completed with novel files systems and files formats, were born, *e.g.*, Apache Hadoop, Apache Spark, HDFS, and Parquet. On the contrary, support for graph data is still young, but it has recently received much attention as the number and complexity of available graphs increased. Most graph data management solutions are either strictly *graph processing systems* [46, 89, 93] or *graph databases* (GDB for short) [11]. Graph processing systems focus on complex batch analysis at a large scale, implementing computationally expensive graph algorithms. Most of them work only on a snapshot of the graph. They can, in some sense, be seen as the graph world parallel to OLAP systems, while graph databases the parallel to the OLTP systems. GDBs indeed focus on storage and querying tasks where the priority is on high-throughput and transactional operations.

Graph processing systems and their evaluation have received considerable attention [46, 57, 89, 93]. Instead, graph databases lag far behind. Since graph management systems are a relatively new technology, their features, performances, and capabilities are not yet fully understood nor agreed upon. Thus, there is a need for effective benchmarks to provide a comprehensive picture of the different systems. This is of major importance for practitioners in order to understand the capabilities and limitations of each system, for researchers to decide where to invest their efforts, and for developers to be able to evaluate their systems and compare with competitors.

Safely storing and managing graphs should not be a limiting factor for high-performance graph analysis and processing. Graphs are becoming so large that processing them on a single machine becomes

challenging. Even with graphs partitioned by distributed GDBs, or other solutions, efficiently processing them is complicated by the nature of most graph processing systems. Unfortunately, they assume sole control over the data, its data format, and its partitioning. This issue is usually handled by exporting and reimporting snapshots. However, continuously duplicating and shuffling data around is clearly suboptimal. There is a need for new algorithms and systems capable of working with externally managed and already partitioned data to bring graph data management and graph data processing on par with their tabular data counterpart.

While many of the interactions on the web are modeled as graphs, most of the content itself is text or can be processed as such, *e.g.*, videos by their captions. The source, quality, focus, and trustworthiness of the content vary. Governments are publishing official data, established journals are reposting articles, people are blogging their stories, users are reviewing products and services, and bad actors are spreading misleading information and fake news [35]. The amount of content produced every day means no human can dream of reading through all of it. Users rely on search engines, news feeds, and recommender systems to process all the data and provide a personalized content selection. People tend to prefer sources that reinforce their pre-existing beliefs [99]. Since such systems optimize for whatever the users like, whenever a controversial topic arises, they get more polarized as they see only a part of the reality and feel support from other users in the same position. While this phenomenon appears in many domains, it is most well documented in forums and social networks [30, 31, 128]. Nevertheless, studies showed it is possible to actively fight polarization and misinformation by providing the user with a comprehensive set of viewpoints [55, 82, 83, 105, 143]. Now, more than ever, there is a need for novel systems to provide holistic overviews of discussion topics.

Given the value of the raw data itself and the possibility of additional value from its processing and integrations with different data, many companies and organizations store much of the generated data whether they already have a plan for it or not. However, extracting novel information or insights from raw data is not trivial. Selecting the most appropriate analysis and processing to perform requires a good understanding of the datasets and their content. This knowledge is typically acquired through data exploration. In data exploration, the users start by looking at parts of the data and making their way to more specific or different parts until they find what is really of interest in it. Given the ever-increasing number of datasets and their growth in size, the effort required to skim through a data repository in this manner is colossal. We claim that there is a need for a more systematic data exploration approach. To achieve this, a user

must first obtain a compact and informative overview of the contents of the data.

CONTRIBUTIONS This thesis contributes to *Understanding and Managing Complex Datasets* in the following ways.

In Chapter 2, we provide a complete and systematic evaluation of existing graph databases, that is not provided by any other existing work to date. We test 51 classes of operations with both single queries and batch workloads, as opposed to the 4-13 that existing studies have done, and we scale our experiments up to 30 million nodes and 178+ million edges, as opposed to a few thousand nodes and few million edges of previous works. We provide a principled and systematic evaluation methodology based on microbenchmarks. We materialize it into an evaluation suite, designed with extensibility in mind and containing datasets, queries, and scripts. We apply this methodology on the major graph databases available today, using different real and synthetic datasets — from co-citation, biological, knowledge base, and social network domains — and discuss our findings.Our findings have illustrated the advantages microbenchmarks can offer practitioners, developers, and researchers and how they can help them better understand the graph database system's design choices, performances, and functionalities. As a result, we have presented several findings that help understand the trade-offs between native and hybrid graph database systems, their effect on important graph queries like traversals and pattern matching, and their current capability to handle highly heterogeneous graphs.

In Chapter 3, we tackle the problem of processing existing large graphs that are already stored in a distributed way. We developed a technique that can exploit as much as possible the existing topology of the graph data and perform the k-core decomposition in a cooperative way among the distribution nodes. Our solution is based on the idea of recomputing the coreness only for those nodes of the graph that are affected by the graph updates. The propagation of the effect is done first inside the partition that exists in a single node and then across partitions by considering the *cut edges*, *i. e.*, edges between nodes of different partitions. We present our algorithm, encompassing initial k-core decomposition and *coreness* maintenance strategy, and implement it on top of AKKA [148], a framework for building distributed and resilient message-driven applications. By running experiments on a variety of both real and synthetic datasets, we show that the proposed method is interesting in the case of very large graphs with a very satisfactory performance and scalability for large graphs.

In Chapter 4, we address the polarization problem in content recommendation systems. We are interested in building a system whose recommended articles cover, depending on the text corpora at hand, the different perspectives or the different opinions and sentiments for

the subject of the query document and its closely related topics. We thus propose two new approaches. The first, *Orthogonal-topics*, focuses on the relationship of the topics, and it has been designed to generalize well on all datasets. The second, *Sentimented-topics*, focuses on the sentiment expressed by the documents on the different topics, and it has been designed to extract and exploit as much information as possible from text corpora that contain opinionated articles. To support these approaches, we also propose a new diversity-metric, MIN-BW, and a new optimization algorithm, FDLS, to find the most diverse set based on the metric mentioned above. To evaluate our solution we first perform a user study where we ask the users to compare the quality of the recommendations generated by our two approaches (orthogonal-topics + MIN-BW and sentimented-topics + MIN-BW) against the state-of-the-art recommender from Abbar *et al.* [1] on real-world data. The results show that both our approaches outperform in all aspects, on average, the solution from Abbar *et al.* [1], especially in diversity and usefulness. We then validate the scalability and precision of FDLS by performing an extensive set of tests on synthetic data. We compare its running time and solution quality (accordingly to MIN-BW score) with the state-of-the-art approximation algorithms for similar diversity-metrics [116]. The results demonstrated that the FDLS algorithm is a good approximating for MIN-BW and scales far beyond what the competition can do; it handles the most complex test cases in less than 175ms, whereas the other algorithms exceed 20s.

In Chapter 5, we streamline the exploration of data lakes and data warehouses. Several tools and techniques have been proposed in the past with the purpose of helping the user understand the data [3, 36, 66, 70, 124, 147]. We focus on the problem of generating concise and informative data summaries; in particular, we see it as a multi-objective optimization problem. We formally define the notion of a data description and the intuition behind the concept of *goodness* for such a description. We propose three solutions for selecting the best description, namely, the Naïve, which considers all the different descriptions and evaluates them; the Vertical, which exploits a smart exploration strategy and heavy pruning; and the Adaptive, which builds on the Vertical and brings in an auto-tuning capability for its parameters. We then evaluate the solutions' scalability and performance on both synthetic and real-world datasets. Results show the effectiveness of our pruning strategies in Vertical and the increased scalability of Adaptive.

DISSERTATION ORGANIZATION The remaining of this thesis is organized as follows. In Chapter 2 we present our Graph Databases Evaluation framework and the insights gathered testing the existing systems with it. In Chapter 3 we present our approach at comput-

ing and maintaining the k-core decomposition of a distributed graph such that the process exploits the existing topology of the graph and executes cooperatively among the distribution nodes. In Chapter 4 we present two new approaches to recommend articles that provide a holistic view of the discussion topic at hand. In Chapter 5 we present our solution to generate concise and complete datasets descriptions. Finally, we conclude with a summary of our results and a possible future work in Chapter 6.

2

MICRO-BENCHMARKING GRAPH DATABASES

Graphs have become increasingly important for a wide range of applications [26, 87] and domains, including biological data [25], knowledge graphs [130], and social networks [54]. As graph data is becoming prevalent, larger, and more complex, the need for efficient and effective graph management is becoming apparent. Since graph management systems are a relatively new technology, their features, performances, and capabilities are not yet fully understood or agreed upon. Thus, there is a need for effective benchmarks to provide a comprehensive picture of the different systems. This is of major importance for practitioners in order to understand the capabilities and limitations of each system, for researchers to decide where to invest their efforts, and for developers to be able to evaluate their systems and compare with competitors.

There are two categories of graph management systems that address two complementary yet distinct sets of functionalities. The first is that of *graph processing systems* [46, 89, 93], which analyze graphs to discover characteristic properties, *e.g.*, average connectivity degree, density, and modularity. They also perform batch analytics at large scale, implementing computationally expensive graph algorithms, such as PageRank [112], SVD [39], strongly connected components identification [135], and core identification [17, 28]. Those are systems like GraphLab, Giraph, Graph Engine, and GraphX [150]. The second category is that of *graph databases* (GDB for short) [11]. Their focus is on storage and querying tasks where the priority is on high-throughput and transactional operations. Examples in this category are Neo4j [106], OrientDB [110], Sparksee [129] (formerly known as DEX), Titan[136] (recently renamed to JanusGraph), ArangoDB[16] and BlazeGraph [121]. To make this distinction clear, graph processing systems can, in some sense, be seen as the graph world parallel to OLAP systems, while graph databases as the parallel to OLTP systems.

Graph processing systems and their evaluation have received considerable attention [46, 57, 89, 93]. Instead, graph databases lag far behind. Our focus is specifically on graph databases aiming to reduce this gap, with a two-fold contribution. First, we introduce a novel evaluation methodology for graph databases that complements existing approaches, and second, we apply it to gather several insights on the performance of the existing GDBs. Some experimental comparisons of graph databases do exist [37, 68, 74]. However, they test a limited set of features providing a partial understanding of the systems, with

experiments at a small scale making assumptions not verifiable at a larger scale [68, 74], sometimes provide contradicting results, and fail to pinpoint the fine-grained limitations that each system has.

Motivated by the above, we provide a complete and systematic evaluation of existing graph databases, that is not provided by any other existing work to date. We test 51 classes of operations with both single queries and batch workloads, as opposed to the 4-13 that existing studies have done, and we scale our experiments up to 30 million nodes and 178+ million edges, as opposed to a few thousand nodes and few million edges of previous works. Our tests cover all the types of insert-select-update-delete queries that have so far been considered and, in addition, cover a whole new spectrum of use-cases, data types, and scales. Our extended spectrum of tests includes pattern matching queries (for these systems that support them) and fixed and dynamic traversal queries. Our selection of datasets is spanning in all dimensions: their size ranges from 2.3k/7.1k up to 30M/178M nodes/edges; some are synthetic, while some are real; their nature ranges from co-authorship networks to airlines-routes networks, from social networks to heavenly heterogeneous knowledge bases.

Moreover, the sheer number of systems, systems versions, queries, and datasets we planned to test made clear we needed a solution for reliably and repeatably evaluating the systems autonomously and automatically. This need was exacerbated by the velocity with which GDBs are being developed, which also lead us to evaluate multiple versions of some systems. We then designed and implemented a flexible evaluations suite capable of automatic scheduling and performing the different kinds of tests (one-shot or batch)in complete isolation, and where systems, queries, and datasets can be added with minimum effort, The evaluation suit also handles transient and permanent failures of the database systems, ensuring each has the best conditions to run.

The fact that GDBs are still a relatively new and volatile technology makes even more apparent the need for evaluation frameworks.

MICRO-BENCHMARKING. In designing the evaluation methodology, we follow a principled *micro-benchmarking* approach. To substantiate our choice, we look at the test queries provided by the popular LDBC Social Network benchmark [45], and show how the produced results are ambiguous and limited in providing a clear picture of the advantages of each system. So, instead of considering queries with such a complex structure, we opt for a set of primitive operators. The primitive operators are derived by decomposing the complex queries found in LDBC, the related literature, and some real application scenarios. By testing primitive operators, we can better pinpoint underperforming opaque system components. Furthermore, the performance of any complex query can be explained by the performance

of the primitive operations it is composed of and the components supporting them. Query optimizers may change the order of the basic operators or select among different implementations, but the primitive operator performance is always a significant performance factor. This evaluation model is known as *micro-benchmarking* [23] and is similar to the principles that have been successfully followed in the design of benchmarks in many other areas [6, 37, 64, 65, 68, 74]. Note micro-benchmarking is not intended to replace macro-benchmarks. Macro-benchmarks are equally important in order to evaluate the overall performance of query planners, optimizers, and caches. They are, however, limited in identifying underperforming operators at a fine grain.

Our evaluation provides numerous specific insights. Among them, three are of particular importance: (i) we highlight the different insights that micro and macro benchmarks can provide; (ii) we experimentally demonstrate limitations of the tested hybrid systems when dealing with localized traversal queries that span across multiple long paths, such as the breadth-first search, and with vertex-centric pattern-matching queries; and (iii) we identify the trade-offs between the logical and physical data organizations, supporting the choice of the native graph databases we study to separate structural information from the actual data. For example, we found that the most effective organization for typical graph queries is storing nodes and edges as records directly linked to each other and with pointers to off-loaded structures for node attributes.

Note that the current work does not consider any distribution features, and the focus is on single-machine installation.

CONTRIBUTIONS. Our specific contributions are as follows:

- We explain the limitations of the existing graph database evaluations and clarify the motivations for the current evaluation study;
- We provide an extensive list of primitive operations (queries) that graph databases should support;
- We introduce the first thorough experimental evaluation methodology based on the micro-benchmarking model for Graph Databases;
- We materialize the methodology into an open-source testing suite [1], based on software containers and Apache TinkerPop [14], that automates the addition of new systems, tests, and datasets;
- We provide a technical analysis of the state-of-the-art systems we evaluate;

1 https://graphbenchmark.com

- We apply this methodology on the major graph databases available today, using different real and synthetic datasets — from co-citation, biological, knowledge base, and social network domains — and discuss our findings.

The remainder of this chapter is organized as follows. First, the related studies are presented in Section 2.1, followed by a description of the graph data model and the challenges of its implementation (Section 2.2). A complete list of queries is provided in Section 2.3. The evaluation suite is described in Section 2.4. A technical analysis of the evaluated state-of-the-art systems is presented in (Section 2.5). The datasets are introduced alongside their characteristics in Section 2.6. Then the results of the test are presented and discussed in Section 2.8. Finally, the chapter terminates with our conclusions 2.9.

2.1 RELATED WORK

2.1.1 *Evaluating Graph Processing Systems*

There are plenty of works on evaluating graph processing systems [27, 57, 90, 94, 153]. Such systems are designed for computationally expensive algorithms that often require traversing the complete graph multiple times to obtain an answer, like page rank, or community detection. Such systems are very different in nature from graph database systems; thus, in their evaluation, "needle in the haystack" queries like those that are typical of transactional workloads are not considered. Of course, there are proposals for unified graph processing and database systems [46], but this idea is in its infancy. Our focus is not on graph processing systems or their functionalities.

2.1.2 *Evaluating Graph Databases*

In contrast to graph processing systems, graph databases are designed for transactional workloads and "needle in the haystack" operations, *i. e.*, queries that identify and retrieve a small part of the data. Existing evaluation works [9, 11] for such systems are limited in describing the systems implementation, data modeling, and query capabilities but provide no experimental evaluation. A different group of studies provides an experimental comparison but is incomplete and fails to deliver a consistent picture. In particular, one work [37] analyzes only four systems, two of which are no longer supported, with small graphs and a restricted set of operations. Two other empirical works [68, 74] compared almost the same set of graph databases over datasets of comparable small sizes but agreed only partially on the concluded results. Moreover, all existing studies do not test with graphs at a large scale and with rich structures. Our work comes to fill precisely this gap in graph database evaluation by providing the most

extensive evaluation of the state-of-the-art systems in a thorough and principled manner.

2.1.3 *Distribution & Cluster Evaluation*

In the era of Big Data, it is important to understand the abilities of graph databases in exploiting parallel processing and distributed architectures. This has already been done in graph processing systems [57, 90, 140]. However, distributed data processing is out of the scope of the current work for several reasons. First, not all the systems support distribution in the same way, *i. e.*, partitioning, replication, or sharding. Second, an evaluation of distribution capabilities to be complete would require considering additional parameters like the number of nodes and concurrency level. Third, despite the popularity of distributed graph management systems, single machine installations are still a highly popular choice [122]. For these reasons, we consider the study of distribution as our natural follow-up work since the question about which system is able to scale out better may only come after the understanding of its inherent performance [97, 120].

2.1.4 *Graph Benchmarks*

There is already a number of benchmarks [2, 10, 12, 45] for evaluating systems for RDF or social data. Yet, those benchmarks are application-specific. For instance, RDF benchmarks [2] focus only on finding structures that match a set of RDF triples. While another graph benchmark, LDBC [45], simulates queries on a social graph. We have used such benchmarks, among others, to create our list of test queries. Moreover, in our experiments, we illustrate the limitations of complex benchmarks. Our goal is not to replace such benchmarks but to enhance them with the extra insights that our own methodology can bring.

2.2 GRAPH DATABASES

Graph databases adopt the *attributed graph model* [11]. Graph data is data consisting of nodes (also called vertexes) and connections between them, called edges. Edges have labels, and every node or edge has a set of attributes or properties, *i. e.*, a set of name-value pairs. In the implementation of such a model, graphs and edges are typically first-class citizens and are assigned internal identifiers.

2.2.1 *The Graph Data Model*

Formally, we can 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 from the set $\mathcal{N}\times\mathcal{A}$. A *graph* is 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.*, $E\subseteq V\times V$, $l:\{V\cup E\}\rightarrow\mathcal{N}$ is a labeling function, and $p:\{V\cup E\}\rightarrow 2^{\mathcal{N}\times\mathcal{A}}$ is a property assignment function on edges and nodes.

Note that, in the model above, different nodes and edges are allowed to have exactly the same set of properties. Systems thus need to extend the implementation of the above model with unique identifiers to distinguish the different nodes and edges. In particular, they consider a countable set $\mathcal{O}$ of unique values and a function $id:\{V\cup E\}\rightarrow\mathcal{O}$ that assigns a unique identifier to each node and edge. Since nodes and edges are fundamental building blocks of graph data, they are typically implemented as atomic *objects* in the systems and may be referred to as such in some parts of the text that follows.

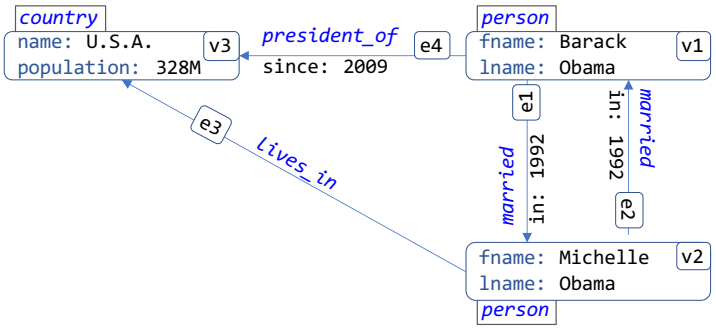


Figure 2.1: Example of Graph.

Figure 2.1 illustrates a portion of some graph data where one can see the identifiers as a boxed number, the properties as pairs separated by the colon symbol " : ", and the labels in blue and italics.

2.2.2 *Implementing a Graph Database*

There are two ways to implement a graph database system (GDBMS). One is to design and build it from scratch so that nodes and edges are first-class citizens. Such systems are referred to as *native*. The other approach is to delegate some functionalities to an external system, often designed for a different model. Examples of used third-party external systems include document stores and relational DBMS. The systems designed using this architecture are referred to as *hybrid*. The main challenges faced in both the native and the hybrid systems are the efficient and effective storage of the graph structures as well as the retrieval of those parts of the graph that satisfy the specifications of a query at hand.

When implementing a native GDBMS, since the nodes and the edges have a more central role in the model and in the query processing than that of the attributes and labels, it is often the case that the latter are kept in separate structures, speeding-up the traversal and retrieval operations. In practice, each atomic object (*e.g.*, a node) is annotated with pointers to other objects connected to it (*e.g.*, incoming and outgoing edges). As a result, when processing a query, operations on the structures are translated into a series of direct pointer traversals, which for atomic objects have a constant-time cost. Labels are usually the only additional information stored with nodes and edges. In the simplest solutions, they are stored as explicit annotations in the objects themselves; in more sophisticated solutions, they are used to cluster nodes and edges, with the implicit intuition that objects with the same label should have a similar structure. Not embedding the properties helps with traversal queries based only on labels but complicates queries that access both structure and properties because the two information resides in two different structures and must then be joined; nevertheless, efficient implementations exist.

When implementing a hybrid architecture nodes and edges are usually stored as single tuples. Like for the native architecture, the properties can either be inlined in the object's entry or get their own tuple in a different table. The first approach requires processing more pages for structural queries, while the second avoid joining the labels tables for mixed or non-structural queries. In both cases, the tables are then usually partitioned per label either explicitly or implicitly (dedicate table or table partition). Moreover, this architecture usually makes use of heavy indexing to speed up both property-based search and structural queries.

When using document stores as the underlying system, nodes and edges are represented as documents storing the various properties of the objects. Structural connections are modeled as document attributes. For simple traversal queries, *i.e.*, queries that do not access object properties, ad-hoc indexes are added to navigate the graph's structure. In general, search and filtering by properties in hybrid systems is usually a highly optimized operation, while traversing the structure is less efficient.

2.2.3 *The Heterogeneity Problem in Graphs*

A significant application of the graph model is that of modeling Knowledge Graphs [107]. Knowledge graphs have historically adopted the RDF model and been stored in triplestores [149]. Recently much interest has been shown in modeling RDF [13] as property graphs. Thus, we would like to understand the challenges of managing KGs using graph databases. A typical characteristic of knowledge graphs is that they model an open domain. By studying well-known knowl-

edge graphs, like DBpedia, Yago, and WikiData, one can see that the structures used model entities such as people, places, things, organizations, locations, work of art, events, publications, fields of study, and so forth [103, 134, 142]. As a result, modeling a knowledge graph through a property graph would require a very large number of types which poses certain strains in storage and querying. Furthermore, due to the open domain nature, entities of the same type may be highly heterogeneous. Attributes with the same name of entities of the same type may have different value types. For example, the price of a product may be expressed for in an instance as a sole number for the default currency, *e. g.*, `price:15`, whereas in another can be expressed as the local price in a different currency, *e. g.*, `price:'13eur'`. This heterogeneity poses significant challenges on how the system can specialize the storage space and optimize the queries.

2.3 TEST OPERATIONS: QUERIES

The set of queries selected for our tests adhere to a micro-benchmark approach [23] that has been repeatedly used in many cases [43, 47, 138]. The *query list* (Table 2.1) is the result of an extensive study of the literature and many practical scenarios. Furthermore, it also encompasses the elementary operations that we have identified by decomposing the many complex scenarios we found. We obtained in this way a set of common operations that are independent of the schema and the semantics of the underlying data; hence, they enjoy 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 their properties. Specifically for the creation, we treat the initial loading of the dataset and the individual object creations as separate cases. The reason is that the first happens in bulk mode on an empty instance, while the second at runtime with data already in the database. The category of Reads operations comprises statistical operations, content search, and filtering content.

We consider next the **T**raversal operations across nodes and edges, and **P**attern matching queries, which are characteristic for graph databases. We recall that operations like finding the centrality or computing strongly connected components are typical in graph processing systems and not in graph databases. The categorization we follow is aligned to the one found in other similar works [9, 68, 74] and benchmarks [45]. The complete list of queries can be found in Table 2.1 and is briefly presented next.

In addition to those queries, we also run a set of complex queries in order to compare the insights they provide with the results of the other operators, as well as to test the query optimization capabilities

of the systems. These queries are taken from a social network application benchmark [10].

#	QUERY	DESCRIPTION	CAT
1.	readGraph("/path"), g	Load dataset into the graph 'g'	**L**
2.	g.addVertex(l)	Create new node with label *l*	**C**
3.$^{+}$	g.addVertex(l).property(p,val)	*Same as Q.2* adding property *p=val*	
4.	v.addEdge(l , v2)	Add edge l from *v* to *v2*	
5.	v.addEdge(v2 , l , p, val)	*Same as Q.3*, adding property *p=val*	
6.	v.property(p, val)	Add property *p=val* to node *v*	
7.	e.property(p, val)	Add property *p=val* to edge *e*	
8.	g.addVertex(...); g.addEdge(...)	Add a new node, and then edges to it	
9.	g.V.count()	Total number of nodes	**R**
10.	g.E.count()	Total number of edges	
11.	g.E.label.dedup()	Existing edge labels (no duplicates)	
12.$^{+}$	g.V.label.dedup()	Existing node labels (no duplicates)	
13.	g.V.has(p, val)	Nodes with property *p=val*	
14.$^{+}$	g.V.has(l, p, val)	*Same as Q.13* limited to node label *l*	
15.	g.E.has(p, val)	Edges with property *p=val*	
16.	g.E.hasLabel(l)	Edges with label *l*	
17.$^{+}$	g.V.hasLabel(l)	Nodes with label *l*	
18.	g.V(id)	The node with identifier id	
19.	g.E(id)	The edge with identifier id	
20.	v.property(p, val)	Update property *p=val* for vertex *v*	**U**
21.	e.property(p, val)	Update property *p=val* for edge *e*	
22.	v.remove()	Delete node *v*	**D**
23.	e.remove()	Delete edge *e*	
24.	v.property(p).remove()	Remove node property *p* from *v*	
25.	e.property(p).remove()	Remove edge property *p* from *e*	
26.	v.inE()	Edges incoming to *v*	**T**
27.	v.outE()	Edges outgoing from *v*	
28.	v.both('l')	Edges adjacent to *v* with label *l*	
29.	v.inE.label.dedup()	Labels of in coming edges of *v* (no dupl.)	
30.	v.outE.label.dedup()	Labels of outgoing edges of *v* (no dupl.)	
31.	v.bothE.label.dedup()	Labels of edges of *v* (no dupl.)	
32.	g.V.where(inE.count()>=k)	Nodes of at least k-incoming-degree	
33.	g.V.where(outE.count()>=k)	Nodes of at least k-outgoing-degree	
34.	g.V.where(bothE.count()>=k)	Nodes of at least k-degree	
35.	g.E.inV.dedup()	Nodes having an *incoming* edge	
36.	v.repeat(both().simplePath()).until(loops().is(k))	Undirected Breadth-first traversal from *v*	
37.	v.repeat(both(ls).simplePath()).until(loops().is(k))	Undirected Breadth-first traversal from *v* on labels ls	
38.	v.repeat(both().simplePath()).until(v2)	Unweighted Shortest Path from *v* to *v2*	
39.	v.repeat(both(ls).simplePath()).until(v2)	*Same as Q.38* but only following labels ls	
40.$^{+}$	v.out().out().out().where(v2)	Paths of length 3 from *v* to *v2*	
41.$^{+}$	v.both().both().both().where(v2)	*Same as Q.40* but undirected	
42.$^{+}$	v.out(l1).out(l2).out(l3).where(v2)	*Same as Q.40* with labels l1, l2, l3	
43.$^{+}$	v.bothE(l1).bothE(l2).bothE(l3).where(v2)	*Same as Q.41* with labels l1, l2, l3	
44.$^{+}$	v.out(l1).out(l2).out(l3)	All Paths from *v* with labels l1, l2, l3	
45.$^{+}$	v.both(l1).both(l2).both(l3)	*Same as Q.44* but undirected	

#	QUERY	DESCRIPTION	CAT
46.+	.match(.as(a).out(l).as(b) .as(b).out().as(c) .as(a).out().as(c))	Find triangles containing edge label l	
47.+	.match(.has('label',l) .as(a).out().as(b) .as(b).out().as(c) .as(a).out().as(c))	Find triangles for node label l	
48.+	.match(.hasId(v) .as(a).out().as(b) .as(b).out().as(c) .as(a).out().as(c))	Find triangles in node v	
49.+	.match(.as(a).out(l).as(b) .as(b).out().as(c) .as(a).out().as(d)) .as(d).out().as(c))	Find squares containing edge label l	**P**
50.+	.match(.has('label',l) .as(a).out().as(b) .as(b).out().as(c) .as(a).out().as(d)) .as(d).out().as(c))	Find squares for node label l	
51.+	.match(.hasId(v) .as(a).out().as(b) .as(b).out().as(c) .as(a).out().as(d)) .as(d).out().as(c))	Find squares in node v	

g is the graph; v and e are node/edges, ls is a list of labels.
+ Hetereogenity support (C, R), fixed tracersal (T), pattern matching (P) – Test-Set B.

Table 2.1: Test Queries by Category (in simplified Gremlin 3 syntax).

2.3.1 *[L] Load Operations*

Data loading is a fundamental operation. Given the size of modern datasets, understanding the speed and complexity of this operation is crucial for evaluating a system. The specific operator (Query 1) reads the graph data from a GraphSON [15] file. Additional operations may be needed for loading, *e.g.*, to deactivate indexing, but in general, they are vendor-specific, *i.e.*, not found in the Gremlin specifications.

2.3.2 *[C] Create Operations*

Create operators may be for nodes, edges, or even properties (on pre-existing nodes or edges). To create a complex object, *i.e.*, a node with many connections to other existing nodes, these operators must often be called multiple times. We tested the insertion of nodes alongside some initial properties (Queries 2, and 3), the insertion of edges with and without properties attached (Queries 4, and 5), the insertion of properties on top of existing nodes or edges (Queries 6, and 7), and finally, the insertion of a new node, alongside several edges that con-

nect it to other nodes already in the database (Query 8). Note that one does not create an edge label without an edge, so edge labels are instantiated with the edge instantiation. In some of these (and other queries below) the node (or the edge) is explicitly referred through its unique id, and thus no search task is involved, as the lookup for the object is performed before the time is measured.

2.3.3 *[R] Read Operations*

GRAPH STATISTICS. (Queries 9, 10, 11, and 12) The evaluation set includes four operations that require scanning the entire graph dataset. The first one scans and counts all the nodes; the second one counts all edges; the last two counts the number of distinct edge-labels and node-labels, respectively. Performing the last two operations also tests the ability of the system to maintain intermediate information in memory since it requires eliminating duplicates before reporting the results.

SEARCH BY PROPERTY. (Queries 13, 14, and 15) These are the basic operators used for content filtering since they search for nodes (or edges) with a specific property. The name and the value of the property are provided as arguments.

SEARCH BY LABEL. (Queries 16, and 17) This task is similar to the previous but filters nodes and edges with a given label. Labels are fundamental components in a graph, and probably, for this reason, the syntax in Gremlin *3* has distinct operators for labels and properties, while in *2.6*, they are treated equally.

SEARCH BY ID. (Queries 18, and 19) As it happens in almost any other kind of database, a fundamental search operation is searching by a key, *i.e.*, ID. These two queries retrieve a node and an edge, respectively, via their unique identifier.

2.3.4 *[U] Update Operations*

Graphs structure updates consist only of insertions and deletions. As such, here, we test only updates on properties. Queries 20 and 21 test the ability of a system to change the value of a property of a specific node or edge.

2.3.5 *[D] Delete Operations*

We include four types of deletions: the deletion of a node (Query 22), which implicitly requires also the elimination of all its properties and edges; the deletion of an edge and its attached properties (Query 23);

and the deletion of a property from a node or an edge (Queries 24, and 25).

2.3.6 *[T] Traversals*

DIRECT NEIGHBORS. (Queries 26, 27, and 28) A popular operation is retrieving all the nodes directly reachable from a given node (1-hop), *i.e.*, those that can be found by following either an incoming or an outgoing edge. Finally, a specific query performs a 1-hop traversal only through edges with a specific label, allowing more advanced filtering.

NODE EDGE-LABELS. (Queries 29, 30, and 31) Given a node, we often need to know the labels of the incoming, outgoing, or both types of edges. This set of three queries performs these three kinds of retrieval, respectively.

K-DEGREE SEARCH. (Queries 32, 33, 34, and 35) 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 first three queries retrieve nodes with at least k edges. They differ in which type of edges they consider: incoming, outgoing, or both. The fourth query identifies nodes with at least one incoming edge and is often used when retrieving a hierarchy.

BREADTH-FIRST SEARCH. (Queries 36, and 37) Some search operations prefer nodes found in close proximity and are better implemented with a breadth-first search *(BFS) from a given node. These two queries test the support for BFS. In particular, the second one is a special case of the former that considers only edges with a specific label.

SHORTEST PATH. (Queries 38, and 39) Another traditional operation on graphs is the discovery of the path between two nodes that contains the smallest number of edges. Thus, we include these two queries, with the second query being the special case that considers only edges with a specific label.

FIXED TRAVERSALS. (Queries 40, 41, 42, 43, 44, and 45) In contrast to these operations that require recursive traversal, there are traversals of fixed length. Here we implement six variations of this query. They search for paths of length 3, with or without a specific destination node, an exact sequence of labels to traverse, or the directionality of the traversal.

2.3.7 *[P] Pattern Matching*

Queries [46-51] test searching for structural patterns. In particular, Queries 46, 47, and 48 look for triangles, while the 49, 50 and 51 look for square shapes in the graph.

2.3.8 *Complex Query Set*

In order to compare the insights obtained using the micro-benchmark approach with those using a macro-benchmark and to test the ability of the systems to optimize complex queries, we also created a workload of 13 queries based on the LDBC Social Network benchmark [10, 45]. These queries mimic the tasks that may be performed by a new user in the system, from the creation of an account (creating a new node with attributes) and fill-up of the profile (connecting to nodes representing the school, place of birth, and workplace), to the task of retrieving recommendations of items or other users. For these operations, we include in the workload queries composed of multiple primitive operators, multiple join predicates, sorting, top-k, and max finding [86].

2.4 EVALUATION SUITE

2.4.1 *Requirements*

Building a successful evaluation suite is a challenging task. It is not enough to have the list of tests to perform, the graph database that needs to be evaluated, and the data on which the evaluation tasks will run. There is a need for a complete suite that facilitates the execution of the tests in a systematic way and the collection of all the necessary information.

One of the first requirements is the understanding of the *use of the different resources* during the execution of the tests. Typically, this can be achieved by embedding the necessary messages that report the resource usage in the right format and the right time into the code of the graph database. Unfortunately, this is not always possible since the graph databases are handled as black boxes and do not always provide such functionality. Thus, there is a need for an alternative solution.

The second requirement is *reproducibility and fairness*. All the systems need to be tested under the same conditions, on the same datasets, and be provided with the same input. We should avoid situations in which the experimental results are affected by a cold start, a system has more information than another, or some execution benefits by the presence of meta information like indexes. Furthermore, the results

of any test execution should be easily and consistently reproducible; otherwise, they will be of minimal value.

A third requirement is *extensibility*. It is not always possible to predict all the possible scenarios that may become of interest in the future. For this reason, any evaluation suite should be able to be easily enhanced with additional tests, datasets, or metrics.

Evaluation action may be specialized for each dataset. We refer to this as *adaptability*. Several tests may require vertexes, edges, or paths with specific properties. The systems shall automatically identify suitable samples or outright refuse to execute the test if the necessary conditions cannot be met.

Last but not least, the results of the tests should become available in an *interchangeable and user-friendly format* to allow for easy communication.

2.4.2 *Technological Solutions*

To achieve the above requirements, a number of technological solutions have been employed.

GREMLIN QUERY LANGUAGE. For fairness, all the systems should be able to understand in the same way the task that is to be performed, which means that there should be a common formal query language. For this, the Apache TinkerPop - Gremlin [2] has been adopted. The Gremlin [117] graph traversal language is a component of the Apache TinkerPop framework, an open-source, vendor-agnostic, graph computing framework. The project aims to provide the missing layer of compatibility between existing graph databases and graph processing solutions. It is the query language with the most widespread support across graph databases, and it can be seen as the SQL of the graph database systems [60] aspiring to be what ODBC is for relational databases and much more. There are Gremlin clients for groovy, python, or java. An application connects and interacts with the database by sending "commands" in a standard Gremlin format, which are then translated to native commands and sent for execution to the database engine.

DOCKER CONTAINERS. We employed Linux containers through the Docker application to control the environment in which an evaluation test is running and to guarantee its portability, reproducibility, and extensibility. A Linux container is a combination of Linux namespaces and control groups (*cgroups*) that allow for packaging and isolating applications with their entire runtime environment, effectively acting as operating-system-level virtualization. Linux cgroups is a feature that allows limiting, tracking, and isolating process resources

2 http://tinkerpop.apache.org

(CPU, memory, I/O, network, ...). At the same time, namespaces make it possible to restrict the set of resources a process sees [119]. While this set of capabilities may resemble those of virtual machines (VM), containers have a distinctive property: the hosted applications, *i.e.*, the applications inside a container, share their kernel with the host. As such, containers are much more lightweight than VMs.

The Docker platform dramatically simplifies the usage of containers by providing tools and automation to manage all aspects of a container life, from its definition to monitoring it while running. It simplifies the process of creating the container's initial snapshot and environment, providing commands to generate the *image* from a file. It abstracts the complex process of spawning a new container that includes cloning the initial environment, setting up cgroups, namespaces, virtual networks, and routing rules behind a single command.

JUPYTER NOTEBOOKS. To present the results of the different evaluations systematically and dynamically, Jupyter Notebooks looked to be an appealing solution. Notebooks are an old technology that has recently seen a new wave of interest, thanks in part to the IPython project. Introduced by Wolfram Mathematica in 1988, they consist, in their simpler form, in a document like interface where text, code, and the result of the code execution can be arbitrarily mixed. The Jupyter Notebook is a client-server application composed of three main components the *notebooks documents*, the *kernels*, and the *notebook web application*.

A notebook document is composed of different building blocks called cells; they may either be text (markdown), source code, or results. The document is, on disk, a JSON file, where the binary data, if any, is encoded in base64 and has a characteristic extension `.ipynb`. The kernels are server-side processes responsible for executing code blocks and generating result blocks. The notebook documents are manipulated through a web interface. The interface makes use of the latest web technologies to seemingly interlace rendered markdown, highlighted editable source code, and rendered results blocks. Results block supports a variety of content like plain text, images, charts, and HTML. Moreover, code cells can be edited on the fly, and the corresponding result blocks updates live. Additional features include auto code completion and export-import of documents and cells.

2.4.3 *Evaluation Framework*

We developed an end-to-end evaluation framework to run a sequence of experiments systematically and automatically. The overall architecture of the framework is illustrated in Figure 2.2.

The framework consists of three main logical components, one for each of the three necessary steps to perform a complete evaluation:

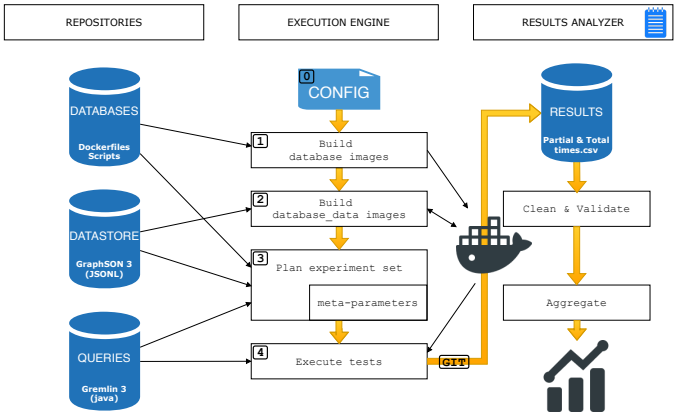


Figure 2.2: The Graph Database Evaluation Suite Architecture

the tests specification, the tests execution, and the analysis of the results.

REPOSITORY. The *Repository* consists of the *Database Engine Repository*, the *Query Repository*, and of the *Dataset Repository*.

The *Database Engine Repository* contains a Dockerfile for each Graph Database that will be involved in the comparison. The Dockerfiles are plain text files containing a list of directives to build the docker images. In other terms, a list of operations defining how to prepare a database installation alongside the necessary software. These files are usually provided by the companies or the community behind the systems. In the remote case of a missing Dockerfile, one can be derived from the installation instruction. Moreover, to facilitate the setup of a standard set of libraries and support binaries, one can exploit the *inheritance* capabilities of Dockerfiles.

The *Query Repository* contains the set of queries (tests) that the suite needs to execute against the database systems. Each query is expressed in Gremlin and modeled as a java class or jar file. It implements, however, a specific interface that allows it to be called in a standard way by the other components.

The *Dataset Repository* is where all datasets that will be used as a testbed for the evaluation reside. Each dataset is expected to be a single file in GraphSON 3 format. This is a JSONL file, where each line is a vertex with its list of incoming and outgoing edges.

EXECUTION ENGINE. The *Execution Engine* (engine) executes the tests from the *Repository* and provides the raw results to the *Result Analyzer*. The test procedure consists of a set of steps, the most important of which are summarized in the center column of Figure 2.2. The first step is identifying the databases, datasets, and queries to test. By default, the unit tests all possible combinations of the databases, datasets, and queries available from the *Repository*; however, it also accepts a configuration file that can restrict the tests to a subset of

the combinations or define more complicated test scenarios like, for example, the one comparing indexed data and the effect of persisting cache and intermediate data. (Step 1) The engine first builds a docker image, the *database images*, for each database involved. (Step 2) Then it proceeds to create a *database_dataset image* for each database-dataset combination. Before starting the test phase, the systems must ensure suitable samples are available for the selected dataset. A query may indeed require additional parameters; for example, the shortest path query required `source-node` and `max-depth`. Since the range of values to test for `max-depth` are equally defined for all datasets, the engine must ensure the sample contains vertexes that can generate paths of at least `max-depth` length, or the general results would be misleading. (Step 3) The engine then checks for the availability and conformity of such samples; in case they are not available, it generates one.

(Step 4) Finally, the engine executes each test, for each valid combination of parameters, in its own isolated and controlled environment and monitors the progress. Once an execution completes, the results are logged in a CSV file versioned into a git repository. The git repository serves as the communication channel between the execution engine and the result analyzer.

RESULT ANALYZER. The *Result Analyzer* provides basic analytics, monitoring, and a simple but powerful tool to explore and study the results. It consists of a relational database and three Jupyter Notebooks: the first one ingests the data and monitors the comparison progress; the second one computes basic analytics and prepares frequently used intermediate views; the last one acts as a scratchpad while offering general-purpose results exploration functions. Before ingesting any results, a schema is created in the relational databases. It contains four tables, three that define the experiments and one that holds the results. The three tables representing the databases, the datasets, and the queries, will be used to check the consistency of the experiment and monitor the progress. Moreover, they can be used in the analysis to group by properties of the tests, *e. g.*, to compare the mean performance on real datasets against synthetic ones. The *result table* is modeled according to the structure of the results CSV files. The CSV files contain time records of the execution, as well as the parameters under which the queries were executed.

After the schema has been set up, the results are pulled from the git repository, parsed, and inserted into the relational tables. Some of the tests may have failed to execute, which means that the results may contain invalid data. For this reason, we employ a verification, and a clean-up task takes place to remove those entries. One of the notebooks' advantages is that the user may now monitor the evaluation results as they are produced and does not have to wait until the

whole experiments terminate. This feature is of critical importance when dealing with large-scale experiments. It allows detecting early on the scalability limits of the database/machines (swap trashing) and pathological queries, which will consistently lead to timeouts. The user may then decide to exclude these systems/datasets/queries test combinations and dedicate the computational resources to other experiments.

The last notebook contains the code for a holistic and clean presentation of the final results. Moreover, it can serve as a comprehensive standalone report of the evaluation campaign.

The complete framework can be freely downloaded at the project web page https://graphbenchmark.com .

2.5 SYSTEMS

For a fair comparison, we need all the systems to support a common access method. For this, we considered systems that support the Gremlin query language [14] through officially recognized implementations. Gremlin [117] is the query language with the most widespread support across graph databases and can be seen as the SQL of the Graph Database Systems [61]. We also required that the systems we consider have a license permitting the publication of experimental comparisons and their operation on a server without a fee.

Given these requirements, we have applied our graph database evaluation suite on a number of state-of-the-art graph database systems, both native and hybrid, and also on two RDF databases. We have not included systems that were not supporting Gremlin, which is why Cayley [3] and DGraph [4] or systems built for RDF, like Apache Jena [5], Virtuoso [6], AllegroGraph [7], and Stardog [8] have not been considered.

Table 2.2 provides a summary of the main characteristics of the systems we consider. Note that for some systems we considered two versions (ThinkerPop 2 and 3.*). We do so because we want to illustrate the degree of progress that has been achieved in these systems from one version to another and also help stakeholders decide whether it is worth the effort to upgrade. The most recent versions have adopted a newer version of Gremlin with cleaner semantics, less overloaded operators, and a richer operator set. Gremlin is mainly a syntax; thus, any performance variation observed across the different versions of

3 https://cayley.io/
4 https://dgraph.io/
5 https://jena.apache.org/
6 https://virtuoso.openlinksw.com/
7 https://franz.com/agraph/allegrograph/
8 https://www.stardog.com

the same system will most likely be due to more effective implementation and not the actual language per se.

SYSTEM	TYPE	STORAGE	EDGE TRAVERSAL	GREMLIN
ArangoDB(2.8, 3.6′)	Hybrid (Document)	Serialized JSON	Hash Index	v2.6 / 3.4.2
BlazeGraph (2.1.4)	Hybrid (RDF)	RDF statements	B+Tree	v3.2
Neo4J (1.9, 3.0, 3.4′)	Native	Linked Fixed-Size records	Direct Pointer	v2.6 / v3.2 / 3.4.2
OrientDB (2.2, 3.0′)	Native	Linked Records	2-hop Pointer	v2.6 / 3.4.2
Sparksee (5.1)	Native	Indexed Bitmaps	B+Tree/Bitmap	v2.6
SQLG (1.2, 2.0′) / Postgres (9.6)	Hybrid (Relational)	Tables	Table Join	v3.2 / 3.4.2
Titan (0.5, 1.0) - Janus (0.4′)	Hybrid (Columnar)	Vertex-Indexed Adjacency List	Row-Key Index	v2.6 / v3.0/ 3.4.2

SYSTEM	QUERY EXECUTION	ACCESS	LANGUAGES
ArangoDB(2.8, 3.6′)	AQL, Non-optimized	REST (V8 Server)	AQL, Javascript
BlazeGraph (2.1.4)	Programming API, Non-optimized	embedded, REST	Java, SPARQL
Neo4J (1.9, 3.0, 3.4′)	Programming API, Non-optimized	embedded, WebSocket, REST	Java, Cypher,
OrientDB (2.2, 3.0′)	Mixed, Mixed	embedded, WebSocket,REST	Java, SQL-like
Sparksee (5.1)	Programming API, Non-optimized	embedded	Java, C++,Python, .NET
SQLG (1.2, 2.0′) / Postgres (9.6)	SQL, Optimized(*)	embedded (JDBC)	Java
Titan (0.5, 1.0) - Janus (0.4′)	Programming API, Optimized	embedded, REST	Java

′ System version used for advanced tests (Set B).

Table 2.2: Features and Characteristics of the tested systems.

2.5.1 *Native*

NEO4J (V.3.0). Neo4j [9] is implemented in Java. It stores nodes and edges natively but separately and supports some schema and textual indexes. It uses one file for node records, one file for edge records, one file for labels and types, and one file for attributes. Nodes and edges are stored as fixed-size records 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 double-linked list, and the other edges are retrieved by following such links. A similar approach is used for attributes. Given an edge, obtaining its source and destination requires constant time operations, and inspecting all edges incident to a node, *i.e.*, visiting a node neighborhood, has a cost that depends on the node degree and not on the graph size. Neo4J has its own query language, called Cypher, which is translated to a set of Java operators. Gremlin queries, instead, are directly describing a sequence of low-level operators with direct access to their programming API. Each operator is evaluated one at a time and passes the result to the next in the sequence. Therefore, our micro-benchmarking approach allows us to investigate the direct performances of each operator without having to take into account query-translation time.

ORIENTDB. OrientDB [10] is implemented in Java. It is a multi-model database where nodes are modeled as documents and edges as links. Hence, it is designed to support graph storage and querying natively. It supports SB-Trees, hash, and Lucene full-text indexes for node search. Information about nodes, edges, and attributes is stored in distinct records that have an ID. Record IDs 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. Thanks to this approach, the cost of getting the nodes of a given edge depends on the node degrees and not on the overall size of the graph. Interaction is done through native Java API, Gremlin, and extended SQL, which is a SQL-like query language. For complex queries, a series of operations, after having been translated into native queries, may have their result processed through the programming API, which can be seen as some sort of query optimization.

9 http://neo4j.com
10 http://orientdb.com/orientdb/

2.5.2 *Hybrid*

ARANGODB. ArangoDB [11] is an open source engine based on the document model and using RocksDB [12], a log structured key-value store, as storage layer. To implement the graph model, it materializes JSON objects for each node and edge and stores them serialized in compressed binary format in documents. Each object contains links to the other objects to which it is connected, *i. e.*, a node contains all the IDs of its incident edges. A specialized hash index is in place that allows the retrieval of the source and destination nodes of an edge, accelerating traversals. While the system indexes automatically the edge endpoints and some attributes, *e. g.*, the internal node identifiers, users can also create additional custom indexes. Nodes and edges are stored into special structures called collections that are treated as *shards*. This poses a strict limitation since the total number of collections is limited by design to 2048. Alternatively, one can coerce the system to store each node and edge in one collection (*i. e.*, all nodes have the same label, and all edges have the same label) and then encode the node or edge label in an attribute. However, this solution will hinder most of the traversal optimization put in place. Apart from Gremlin, ArangoDB supports its own query language, called AQL, *ArangoDB Query Language*, which is an SQL-like dialect supporting graph traversals, joins, and transactions. Interaction happens via a REST API and HTTP calls over TCP connections. Arango supports Gremlin at a driver lever. At query time, all Gremlin pipelines are thus translated into more AQL queries and are sent to the server for execution. Note that Gremlin is a Turing-complete language and can describe complex operations that declarative languages, like AQL, may not be able to express in one query. For instance, the Gremlin query Q32 in Table 2.1, which selects nodes with at least k incoming edges will be translated into a selection of nodes (`g.V`), and for each node applying a filter (`.where(...)`) by counting its incoming edges (`inE.count()`). ArangoDB does not provide any overall optimization of these parts.

SQLG/POSTGRESQL. Sqlg [13] is an generic implementation of Apache TinkerPop on top of RDBMS; Sqlg in particular, is based on the relational database Postgresql [114].

Each node and edge is identified by a unique ID as the primary key, and connections between nodes and edges are retrieved through joins. It models every vertex type as a separate table and edge labels as many-to-many join tables. The indexes it supports are those inherited by the relational engine. It provides Java API to the Gremlin language,

11 https://www.arangodb.com/
12 https://rocksdb.org
13 http://www.sqlg.org/

and the underlying implementation maps graph semantics to that of the RDBMS. Hence, this approach requires unions and joins even for retrieving the incident edges of a node. Wherever possible, it tries to conflate operators in a single query, which is some form of query optimization.

JANUS / TITAN. Janus Graph [14] is another hybrid system. The name Janus Graph is the latest rebranding of Titan. The main part of the system handles data modeling and query execution, while data persistence is delegated to a third-party storage and indexing engines. For storage, it supports primarily Cassandra [15] and HBase [16], then it also supports BerkeleyDB [17] and an in-memory storage engine (both not intended for production use). The graph is stored with the adjacency list model, where each vertex is stored alongside the list of incident edges. In addition, each vertex property is an entry in the vertex record. 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. Janus adopts Gremlin as its only query language and Java as the only compatible programming interface.

2.5.3 *RDF*

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 the edges attributes as reified statements, *i. e.*, each edge can assume the role of a subject in a statement. Hence, traversing the graph's structure may require more than one access to the corresponding B+Tree.

SPARKSEE employs separate data structures: one structure for objects, 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 itself composed of a map from keys to values and a bitmap for each value [96]. In each data structure, the objects are identified by sequential IDs, and each ID is linked as a key through the map to one single value. 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. For instance, given a label, one can scan the corresponding bitmap to identify which edges share the same label. Furthermore, bitmaps identify all edges incident

14 https://janusgraph.org/
15 http://cassandra.apache.org
16 http://hbase.apache.org
17 http://www.oracle.com/technetwork/products/berkeleydb

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, although operations like edge traversals have no constant time guarantees.

2.5.4 *Query Processing and Evaluation*

A Gremlin query is a series of operations. Consider, for instance, Q32 in Table 2.1, which selects nodes with at least `k` incoming edges. For every node (`g.V`), it applies the filter (`.filter{...}`) by counting the incoming edges (`it.inE.count()`). In **ArangoDB** each step is converted into an AQL query and sent to the server for execution so that the above Gremlin query will be executed as a series of two independent AQL queries implementing the outer and the inner part, respectively. ArangoDB does not provide any overall optimization of these parts. Note that Gremlin is a Turing-complete language and can describe complex operations that declarative languages, like AQL or Cypher, may not be able to express in one query. **Sqlg** translates all operations to a declarative query language. Moreover, Sqlg, where possible, tries to conflate operators in a single query, which is some form of query optimization. In **OrientDB** some consequent operators may get translated into queries and then their result 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, also features some optimization during query processing. **BlazeGraph**, **Neo4J**, and **Sparksee**, instead, 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.

2.6 DATASETS

We used seven different datasets (*Set A*), both real and synthetic, to test the systems general capabilities.

We then used three more datasets (*Set B*) to stress-test fixed traversal & pattern matching queries, support for heterogeneous data, and scalability. These tests have been highlighted with a $^{+}$ in Table 2.1. Since these operations and datasets are much more challenging, we used them to test only systems that support the pattern matching operations (Thinkerpop 3) and which demonstrated good results on *Set A*.

2.6.1 *Set A*

mico. The first dataset (*MiCo*) describes co-authorship information crawled from the CS Microsoft Academic portal [44]. Nodes represent

authors, while edges represent co-authorships between authors and have as a label the number of co-authored papers.

yeast. The second dataset (*Yeast*) is a protein interaction network [20]. Nodes represent budding yeast proteins (*S.cerevisiae*) [25] 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 labels the respective protein classes.

FRB-*. The third dataset is Freebase [53], which is one of the largest knowledge bases freely available for download nowadays. Nodes represent entities or events, and edges model relationships between them. We took the latest snapshot, cleaned it, and considered four subgraphs of it of different sizes [85, 104]. The raw data dump contains 1.9B triples [53], many of which are duplicates, technical or experimental meta-data, and links to other sources that are commonly removed [19, 104], thus leaving a clean dataset of 300M facts. The sizes of the samples were chosen to ensure that all the engines had a fair chance to process them in a reasonable time, but, on the other hand, to show also the system scalability at levels higher than those of previous works.

In this study, we created one subgraph (*Frb-O*) by considering only the nodes related to the topics of organization, business, government, finance, geography, and military, alongside their respective edges. Furthermore, we created other 3 graph datasets by randomly selecting 0.1%, 1%, and 10% of the edges from the complete graph, resulting in the *Frb-S*, *Frb-M*, and *Frb-L* datasets, respectively.

ldbc. We generated a synthetic dataset [86] using the data generator provided by the Linked Data Benchmark Council (LDBC) [45], which produces graphs that mimic the characteristics of a real social network with power-law structure, and real-world characteristics like assortativity based on interests or preferences (*ldbc*). We selected this in place of any available social network dataset because it is richer in attribute types, edge types, and relationships. It is the only dataset of *Set A* with attributes on the edges. The generator was instructed to produce a dataset simulating the activity of 1000 users over a period of 3 years.

2.6.2 *Set B*

We then specifically selected three more datasets to stress-test fixed traversal & pattern matching queries, support for heterogeneous data, and scalability.

air-routes. To test reachability and long traversal queries, we selected *air-routes*, a real-world dataset that describes the world airlines route network [18]. Vertexes represent airports, countries, and continents, while edges describe routes between airports and continents. Both vertexes and edges have attributes.

dbpedia. To verify the support for heterogeneous data, we used *DBpedia*. The popular knowledge graph extracted mainly from Wikipedia [78], another real-world dataset. We took the latest official dump and converted it to a property graph with an official Neo4J (v.3.0) library designed for this task. We converted RDF URIs into a node of type *Resource* and translated the predicates connecting to resources into edge labels and those connecting to literals into properties. This gave a very heterogeneous graph with 12 thousand different edge types and 47 thousand distinct node properties.

ldbc.10. Finally, to further test scalability, we generated another synthetic dataset using the data generator provided by the Linked Data Benchmark Council (LDBC) [45]. We selected this in place of any available social network dataset because it is richer in attribute types, edge types, and relationships. This time we instructed the generator to go for a much larger scale. The generator was instructed to produce a dataset with a scale factor of 10, simulating the activity of 73.000 users over a period of 3 years.

2.6.3 *Datasets Characteristics*

Table 2.3 provides the characteristics for the first set of datasets (*Set A*). It reports the number of nodes (|V|), edges (|E|), labels (|L|), connected components (#), the size of the maximum connected component (CC/-max), the graph density (Density), the network modularity (Modularity), the average degree of connectivity (Avg), the max degree of connectivity (Max), and the diameter (⊘).

As shown in the table, *MiCo* and *Frb* are sparse, while *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 statistics include the average and the maximum degree because large hubs become bottlenecks in traversals.

Table 2.4 provides a summary of the characteristics for the second set of datasets (*Set B*). It reports the number of nodes (|**V**|), edges (|**E**|), labels on nodes and edges (|$\mathbf{L_v}$| and |$\mathbf{L_e}$|), as well as number of properties on nodes and edges (|$\mathbf{P_v}$| and |$\mathbf{P_e}$|).

.

18 https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html

				CC				DEGREE		
	|V|	|E|	|L|	#	MAX	DENSITY	MODULARITY	AVG	MAX	⌀
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 2.3: Datasets Characteristics (Set A).

	$\|V\|$	$\|E\|$	$\|L_v\|$	$\|L_e\|$	$\|P_v\|$	$\|P_e\|$
ldbc.10	30M	178M	11	15	18	4
DBpedia	16M	27M	1	12K	47K	0
air-routes	3742	57K	4	2	15	1

Table 2.4: Datasets Characteristics (Set B).

2.7 EXPERIMENTAL SETUP

We used our framework (Section 2.4) to subject selected systems (Section 2.5) with our micro-benchmarking oriented test methodology (Section 2.3) on a variety of dataset (Section 2.6.)

As already discussed, thanks to our evaluation framework: all systems were tested with the very same queries, which were implemented just once in the Gremlin common query language; each test had dedicated resources and was completely isolated (Linux containers); all systems were tested with exactly the same set of pseudo-random samples (selected once and then re-mapped for each database).

Our goal is to perform a comparative evaluation. Nevertheless, we strive to have numbers as close to a production-like installation as possible. Thus, we measure each system bootstrap time, *i.e.*time to execute a blank query, and subtract it from the total running time, so that results include only the actual query execution time.

2.7.1 *Hardware*

We used a machine with a 24-core CPU, an Intel Xeon E5-2420 1.90GHz, 128 GB of RAM, 2TB hard disk (20.000 rpm), Ubuntu 14.04.4 operating system, and with Docker 1.13, configured to use AUFS on *ext4*. During the tests, we reserved 8GB of main memory for the host operating systems.

2.7.2 *GDB configuration*

The system configuration is important since the different parameters significantly affect its performance. Neo4J does not need any special configuration to run. 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 in order to support more. ArangoDB requires configurations for the engine and for its V8 javascript server for logging. With default values, it generates approximately 40 GB of log files in 24 hours, and it is impossible to force it to allocate more than 4GB of memory. In Titan, the most critical configuration is that of the JVM Garbage Collection and of the Cassandra backend. The other systems that are also based on Java, namely, BlazeGraph, Neo4J, OrientDB and Titan, are equally sensitive to the garbage collection, especially for very large datasets that require large amounts of main memory. As a general rule, the option `-XX:+UseG1GC` for the *Garbage First* (G1) garbage collector is strongly recommended. Finally, Sqlg has a limit on the maximum length of labels (due to Postgresql), which requires special handling.

As a general observation, it seems that Neo4J is a mature system in which the developers have paid attention to both usability and automatic tuning. The rest of the systems are heavily restricted by their underlying technology, which significantly affects the system performance if not well-tuned.

2.8 RESULTS

We provide first an overview of the results of the experimental evaluation performed for the individual types of operations, and then, in Section 2.8.6 and Table 2.5, we provide an overall evaluation and the main insights.

Throughout our tests, we noticed that *MiCo* and *ldbc* were giving results similar to *Frb-M* and *Frb-O*. *Yeast* was so small that it did not highlight any particular issue, especially when compared to the results of *Frb-S*. We also tried to load the full freebase graph (with 314M edges and 76M nodes), but only Neo4J, Sparksee, and Sqlg managed to do so without errors, and only Neo4J (v.3.0) successfully completed all the queries. Furthermore, the running times recorded on the full dataset respected the general trends witnessed with its subsamples. Thus, for the initial set of tests (Set A, Table 2.3), we focus only on the results of *Frb-S*, *Frb-O*, *Frb-M*, and *Frb-L* and make reference to the other samples only when they show a behavior different from the one of Freebase (Section 2.8.3). Additional details about the experimental results that are not mentioned here can be found in our technical report [86].

For the tests on fixed traversal & pattern matching queries, support for heterogeneous data, and scalability (Set B, Table 2.4, Section 2.8.4), we can report results for all systems only over the *air-routes* dataset (Section 2.8.4). Since ArangoDB failed the mapping setup for *ldbc* – *i. e.*, it was not able to extract the node and edge information required to instantiate the queries (the operation timed out) – and only Neo4J and Janus managed to load *DBpedia* and were able to run at least some of the queries on it.

We first discuss the procedure employed to load the data (Section 2.8.1). We then introduce the results and analysis for Complex Queries (Section 2.8.2). We follow up with the two sets (A and B) of micro-benchmark style tests in Section 2.8.3 and Section 2.8.4, respectively. We thus present our remarks on the systems' progress over their versions 2.8.5. Finally, we conclude with a summary of the results, insights, and takeaways gathered throughout the whole evaluation summary (Section 2.8.6).

2.8.1 *Data Loading*

SET A

EXECUTION. Many systems were failing or taking days to load the data through Gremlin (using query 1). The Gremlin implementation of ArangoDB was sending each node and edge insertion instruction separately to the server via HTTP calls, making it prohibitively slow to load with this method, even with small datasets. OrientDB had a similar limitation and was, in addition, performing a lot of bookkeeping tasks for each edge-label it was loading. For both, we used implementation-specific scripts and commands, bypassing the Gremlin library, in order to load the datasets. To load the data in BlazeGraph, we had to explicitly activate a *"bulk loading"* option. Without it, the system processes each label and node separately and updates its meta-data structures after the insertion of each such item. In Titan, the delays were higher. That was due to consistency checks and schema inference tasks. Disabling automatic schema inference was significantly reducing the loading times but required to specify the schema before inserting any data, meaning that Titan was not able to handle dynamic schema updates transparently. Neo4J, Sqlg, and Sparksee managed loading the data through the Gremlin API without issues, which indicates that they offer a good Gremlin implementation.

TIME. In terms of loading time (Figure 2.3), ArangoDB was the fastest, mainly thanks to the use of native scripts we had to employ to load data in a reasonable time. Neo4J was almost equally fast, proving that a good implemented Gremlin API can achieve as good per-

formance as the native scripts. The loading time of the different size datasets on Sqlg and OrientDB illustrated a high sensitivity to the edge label cardinality. This sensitivity is because both Sqlg and OrientDB create and use different structures for different edge labels. BlazeGraph, on the other hand, updates and balances its B+Tree index structure after every insertion, and this made it up to 3 orders of magnitude slower than the other engines.

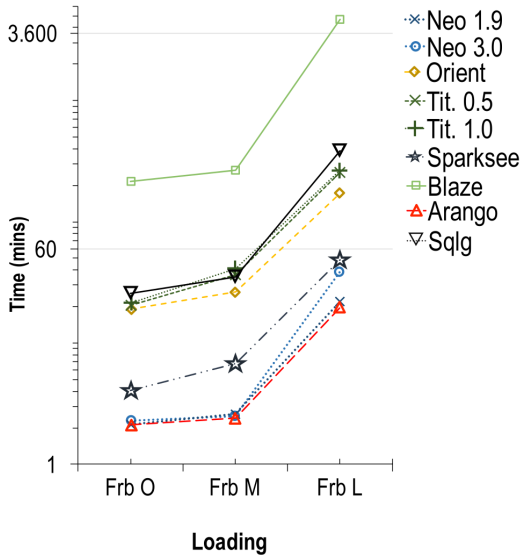


Figure 2.3: Loading time (Set A).

SPACE. We studied the disk space that the datasets occupied in the different systems to identify those with the most effective compression strategies (Figure 2.4). Although disk space may not be a major concern nowadays, it becomes relevant, for instance, in systems with solid-state drives. The results of the datasets *Frb-O*, *Frb-M*, and *Frb-L* show Titan as the one with the best space performance. Its strategy is to compact node identifiers in each adjacency list with a form of delta encoding, a strategy very effective in graphs with nodes of high degrees. For the *ldbc* dataset, instead, where much textual information is shared by many objects, OrientDB and Sparksee achieved the least space consumption because they de-duplicate attribute values. Given that OrientDB creates different files for each distinct edge label, we see that it is the second last in terms of space on the *Frb-S* dataset that contains many different edge labels (~1.8K) for relatively few edges (~300K). Finally, we can see that BlazeGraph requires, on average, three times the size of any other system on all the datasets, and this is because it instantiates a journal file of fixed size and produces a lot of data replication with its different indexes.

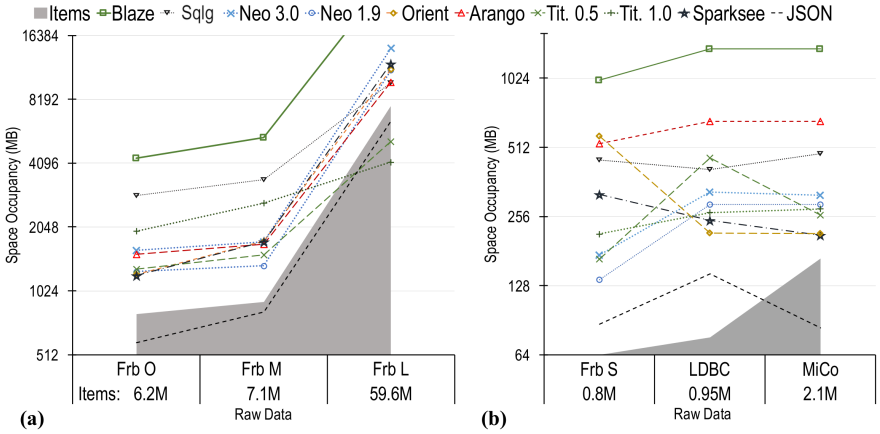


Figure 2.4: Space occupancy (Set A).

SET B

EXECUTION. Many systems were failing or taking days to load the data through Gremlin (using query 1). While ThinkerPop uses GraphSON 3 as the serialization format for the graphs on disk, there is no globally agreed format for serializing property graphs. Many systems assume to load a graph from a set of CSV files, and this is due to their focus on graphs with very simple and regular structure. Instead, the GraphSON format is based on JSON and allows the modeling of very complex and heterogeneous graphs. Each file line serializes one node with all its properties and adjacent edges. This encoding should allow, in theory, to load a graph by reading line-by-line. However, when materializing a node with its incident edges (*i.e.*, all outgoing edges for which the current node represents the source), the system needs to materialize also the other nodes (*i.e.*, the destination nodes) that are referenced only by ID. But, since most systems do not allow direct manipulation of the node and edge ids, the default loader loads the entire graph in memory, materializing all nodes and mapping their IDs in the dataset with the system's internal ids. The default ThinkerPop loader is thus limited by the available main memory and is very slow. To conduct our experiments, we rewrote the loader to work in a streaming fashion: with a first pass over the data, it creates all the nodes with their properties and records their ids into a hashmap ($\mathtt{h}(\mathtt{id}) \rightarrow \mathtt{id}_{\mathtt{sys}}$); with a second pass, it inserts all edges resolving the ids using the map build in the first pass. Nevertheless, only Neo4J and Sqlg managed to complete the loading without any further modifications. For ArangoDB and OrientDB we used instead their native loaders. Moreover, ArangoDB and Titan (v.1.0) both require the schema of the graph to be known a-priory. This requirement implies parsing the dataset, extracting all node labels, edge labels, properties, and tracking all their combinations. *Therefore, contrary, for instance, to the case of*

RDF, there is a large gap in the standardization of dataset serialization and data loading tasks for property graphs.

SPACE. We studied the disk space that the datasets occupied in the different systems to identify the effect of the various data organization strategies (Figure 2.5). For this test, we also added a property, a surrogate key, to all node types and put an index on it. Janus has the most compact data representation. In particular, for *ldbc.10* the uncompressed JSON is twice as large as the representation on disk encoded by Janus, and this takes into account also the index. ArangoDB and Sqlg instead, when enabling indexes, occupy more space than the uncompressed file, with a blow-up of about 45%. Finally, OrientDB and Neo4J require 3 times more space than the uncompressed data, and the index alone accounts for about 50% of such space.

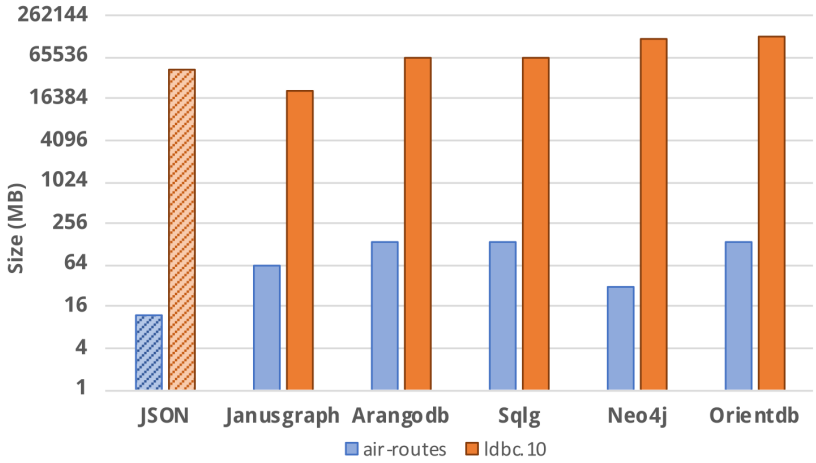


Figure 2.5: Space occupancy (Set B).

2.8.2 *Complex Queries*

For completeness, we first evaluate the graph databases using complex queries of an existing benchmark, the LDBC, applied on their *ldbc* dataset (Figure 2.6). For these we set a timeout of 2 hours. 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. Yet, for ArangoDB this result fails to demonstrate that there are cases (identified below) in which it can actually perform better than others. Titan (v.1.0) was very fast for some queries involving short joins and for some with single-label selections. Yet, *the micro-benchmark analysis* (below) shows that this result does not generalize. This type of performance is due to the specific query and to the help of caching from the Cassandra

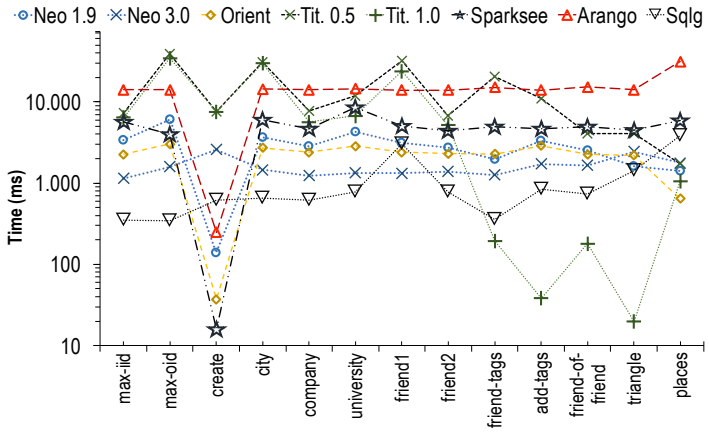


Figure 2.6: Complex Query Performance on *ldbc*.

backend. As we will see below, in the result of our microbenchmark, Titan (v.1.0) is consistently slower than other systems when the graph gets larger and when the query cannot exploit any cache. Sqlg is the fastest in almost half the queries. Hence, we question the reason why in some cases (*e. g.*, the last query) Sqlg is much slower than the competition. Especially, it is puzzling to compare the last and the second to last queries, both performing some sort of traversal. Again, *the micro-benchmark analysis* identifies the characteristics of the best performing operators in Sqlg, which are exploited by those queries that can be translated to a single relational operator or to conditional join queries, with no recursion and short join chains that traverse only few edge labels with limited cardinality. In these queries, the system does not incur expensive joins, and it can take advantage of the relational optimizer and exploit indexes. Those cases in which Sqlg is slower are, instead, queries that traverse many edges and do not filter on a single edge label, and thus generate large intermediate results.

2.8.3 *Micro-benchmark Results*

We now turn to our micro-benchmark queries.

COMPLETION RATE. For online applications, it is crucial to ensure that the queries terminate in a reasonable amount of time. For this, we count the queries that did not complete within 2.5 hours, either in isolation or in batch mode, and illustrate the results in Figure 2.7.

Neo4J, in both versions, is the only system that successfully completed all the tests with all the parameters on all the datasets (omitted in the figure). OrientDB is the second-best, with just a 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 for some batch executions on *Yeast*, and almost on all the 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 breath-first traversals with depth 3 or more reach the timeout on *Frb-O*, *Frb-M* and *Frb-L* in most databases. The filtering of nodes based on their degree (Q.32, Q.33, and Q.34), the search for nodes with at least one incoming edge (Q.35), and the pattern matching based on node labels or edge labels (Q.46, Q.47, Q.49, and Q.50) are proved to be extremely problematic almost for all the databases apart from Neo4J and Titan (v.1.0). In particular, for Sparksee, on all the Freebase subsamples, these queries cause the system to exhaust the entire available RAM and swap space (this has been linked to a known problem in the Gremlin implementation). ArangoDB failed these last queries only on *Frb-M* and *Frb-L*, and OrientDB only on *Frb-L*. These results highlight the benefits of separating the graph structure from the attribute values, allowing native systems to execute fast even queries that require access to large portions of the graph.

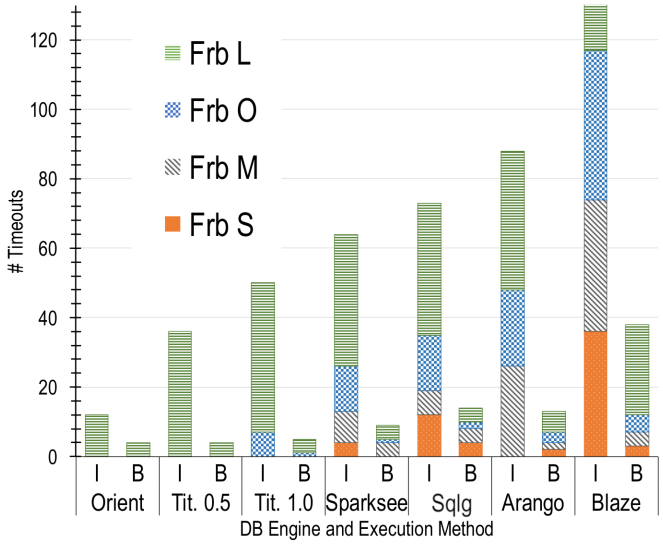


Figure 2.7: Timeouts for Interactive (I) and Batch (B) modes (Set A).

INSERTIONS, UPDATES AND DELETIONS. For operations that add new objects (nodes, edges, or properties), we experienced extremely fast performances for Sparksee, Neo4J (v.1.9), and ArangoDB, with times below 100ms, with Sparksee being generally the fastest (Figure 2.8). Moreover, with the only exception of BlazeGraph, all the databases are almost unaffected by the size of the dataset. We attribute this result to the internal configuration of the data structures adopted where elements are stored in append-only lists, while for ArangoDB these operations are registered in RAM and asynchronously flushed to disk. BlazeGraph, on the other hand, is the slowest with times between 10 seconds and more than a minute as each of these

operations requires multiple index updates. Both versions of Titan are the second slowest systems, with times around 7 seconds for the insertion of nodes, and 3 seconds for the insertion of edges or properties, while for the insertion of a node with all the edges (Q.9) it takes more than 30 seconds. Sparksee, ArangoDB, OrientDB, Sqlg, and Neo4J (v.1.9) complete the insertions in less than a second. OrientDB is among the fastest for insertions of nodes (Q.2) and properties on both nodes and edges (Q.6 and Q.7), but is much slower, showing inconsistent behavior, for edge insertions. 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. We will see below that this depends on some initialization procedures. Sqlg is one of the fastest for insertions of nodes as these operations translate into inserting a tuple into a relational table, while is much slower for all other queries where it has to change the table structure. Similar results are obtained for the update of properties on both nodes and edges (Q.20, and Q.21), and for the deletion of properties on edges (Q.25).

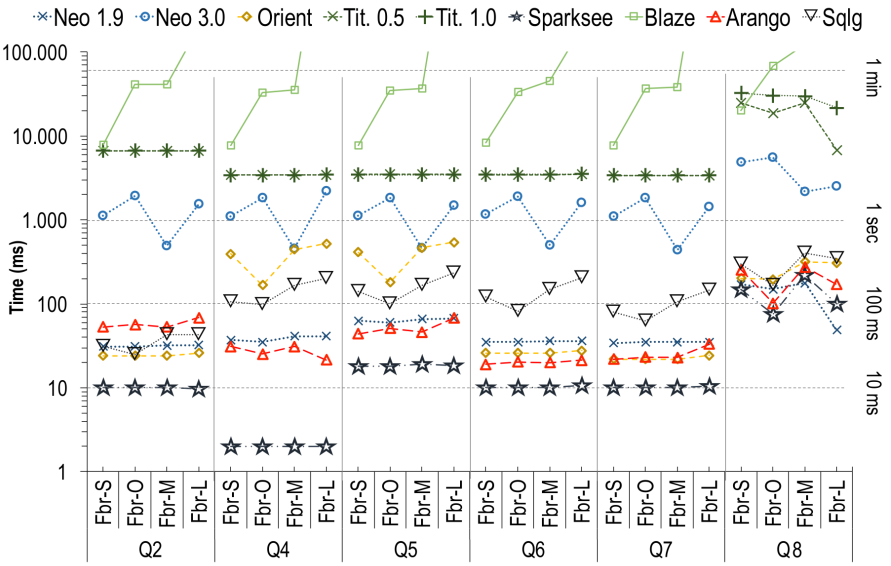


Figure 2.8: Time for insertions.

On the other hand, the performance of node removal (Q.22) for OrientDB, Sqlg, and Sparksee seems highly affected by the structure and size of the graphs (Figure 2.9). While ArangoDB and Neo4J (v.1.9) remain almost constantly below the 100ms threshold, Neo4J (v.3.0) completes all the deletions between 0.5 and 2 seconds, showing that there is some overhead intervening. Finally, for the removal of nodes, edges, and node properties, Titan obtains an improvement of almost one order of magnitude by exploiting the benefits of the data organization in the column stores. Note that ArangoDB is also consistently among the fastest. However, the following factors bias its results: its interac-

tions happen through REST calls, the updates are asynchronous, and it lacks support for transactions. These constitute a bias on results in ArangoDB favor because the time is measured on the client-side, and we have no control over when those operations get materialized on disk.

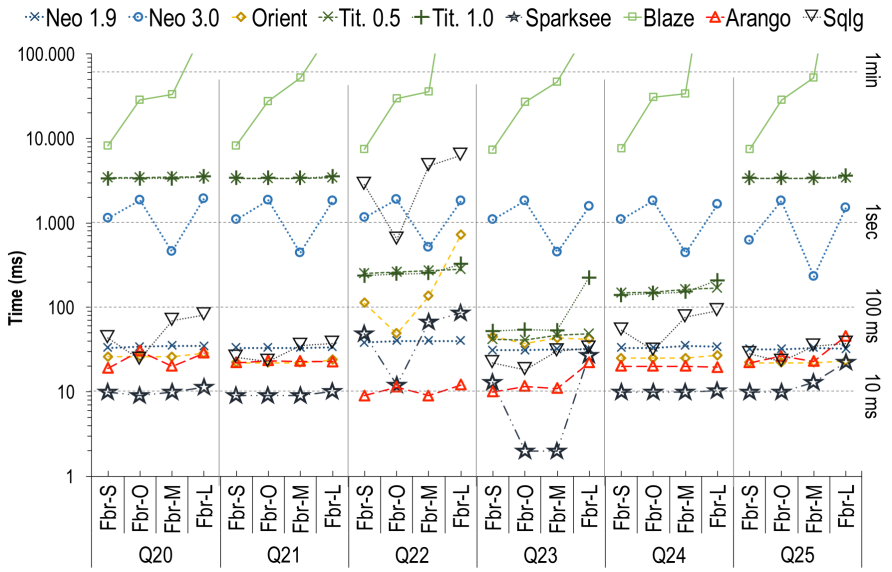


Figure 2.9: Time for updates and deletions.

GENERAL SELECTIONS. With *read* queries, some heterogeneous behaviors show up. The search by ID (Figure 2.10) differs significantly from all the other selection queries (Figure 2.11), and it is in general much faster; this indicates special attention from all vendors on this operation. BlazeGraph is again the slowest. All other systems take between 10ms (Sparksee) to 400ms (Titan) to satisfy both queries.

In counting nodes and edges (Q.9, and Q.10), Sparksee has the best performance followed by Neo4J (v.3.0). For BlazeGraph and ArangoDB, node counting is one of the few queries in this category that complete before timeout. Edge iteration, on the other hand, seems hard for ArangoDB, which rarely completes within 2.5 hours for Freebase and the other medium and large size datasets, as it materializes all edges while counting them.

Computing the set of unique labels (Q.11, and Q.12) does not significantly affect the ranking. Here, the two versions of Neo4J are the fastest databases, while Sparksee is a little slower. Since the previous experiments showed that Sparksee is fast in iterating over the edges, we identified here a sub-optimal implementation of the deduplication step.

The search for nodes and edges (Q.13, Q.14, and Q.15) based on property values perform similar to the search for edges and nodes based on labels (Q.16 and Q.17), for almost all the databases. These

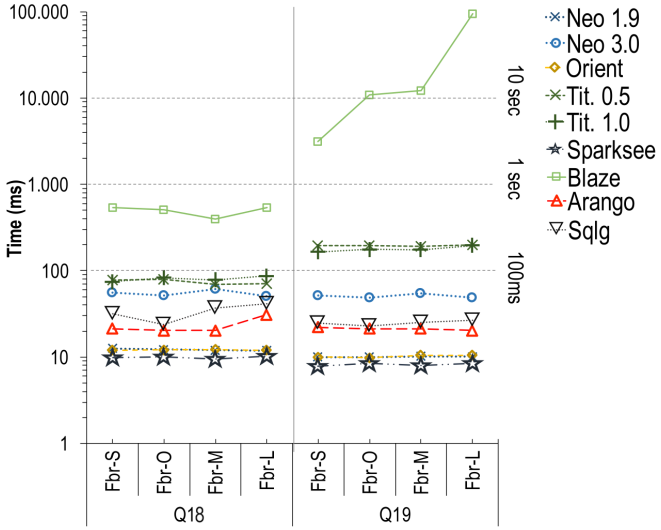


Figure 2.10: Time for searching by id.

3 are some of the few queries where the RDBMS-backed Sqlg works best, with results an order of magnitude faster than the others. Hence, equality search on edge labels has not received special optimizations in the various native systems. This is despite Sparksee and OrientDB have data-structures that should help optimizing this operation.

In general, the above results support the choice of separating structure and data records since it allows to iterate over the entire set of objects without materializing them. They also indicate the importance of indexing the correct properties since, in all the systems, the search task became problematic for large datasets. The RDBMS was less affected in this situation, especially for edge labels due to the storage of the relations in separate tables.

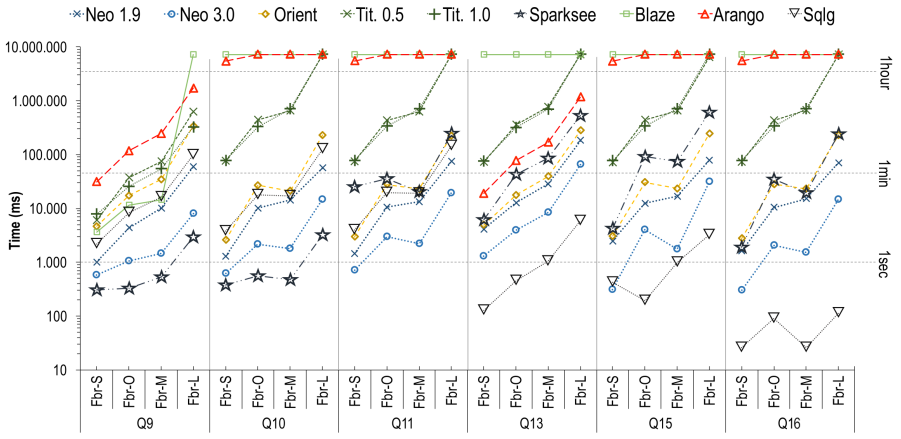


Figure 2.11: Time for general selections.

TRAVERSALS. For traversal queries that access the direct neighborhood of a specific node (Q.26 to Q.31), OrientDB, Neo4J, and ArangoDB are the fastest and are robust to variations in graph size and structure, as shown in Figure 2.12. In contrast, Sparksee seems to be more sensitive, requiring around 150ms on *Frb-L*. The only exception for Sparksee is the visit of the direct neighborhood of a node filtered by edge labels, in which case it is on par with the former systems. BlazeGraph is an order of magnitude slower (~600ms) preceded by Titan (~160ms). We also notice that Sqlg is the slowest engine for these queries unless a filter is required on the label to traverse, in which case Sqlg becomes much faster (explaining the good performance in Figure 2.6).

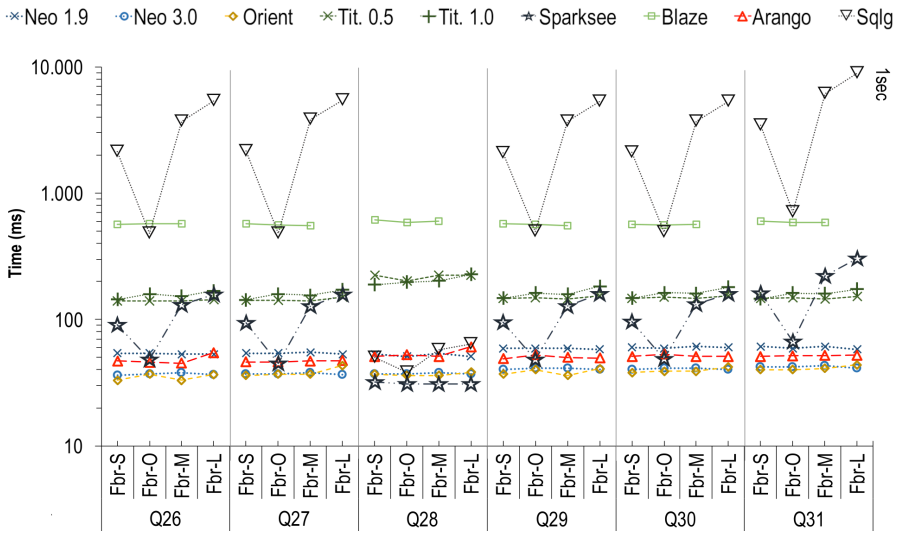


Figure 2.12: Time for traversal operations: local access to node edges.

When comparing the performance of queries from Q.32 to Q.35, which traverse the entire graph filtering the nodes based on the edges around them, as shown in Figure 2.13, Neo4J (v.3.0) presents the best performance, with its older version being the second-fastest. Those two are also the only two engines that complete these queries on all datasets. All the systems tested are 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, identifying a problem in the implementation, as this never happens in any other case. BlazeGraph as well hits the time 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.35, with time comparable to Neo4J (v.1.9). Yet, all systems complete the tasks on *Yeast*, *ldbc* and *MiCo*.

Breadth-first (BFS) (Q.36 and Q.37) and shortest path (SP) search (Q.38 and Q.39) are important operations in graphs. The performance of the unlabeled version of BFS, shown in Figure 2.14, highlights the good scalability of Neo4J at all depths. OrientDB and Titan give the

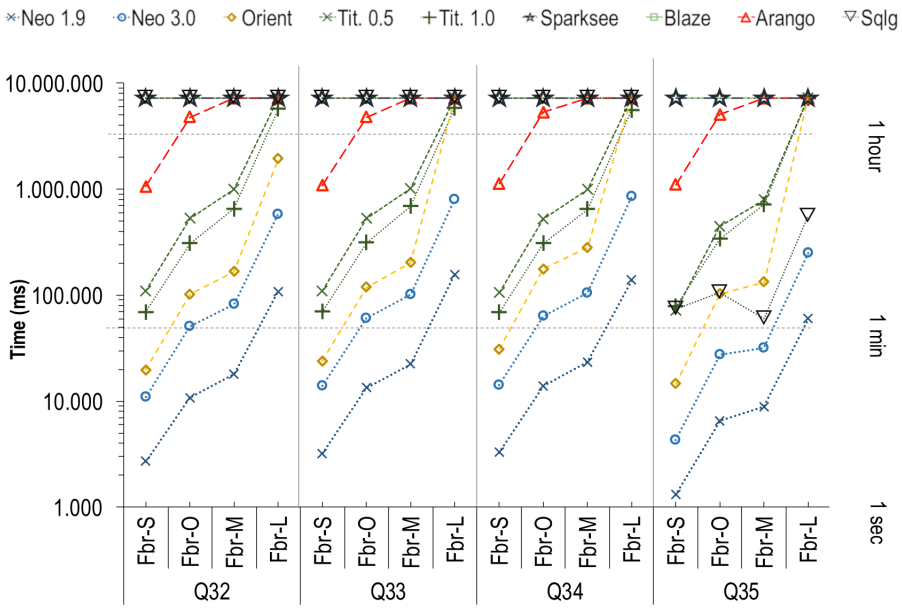


Figure 2.13: Time for filtering on all nodes.

second fastest times for depth 2, with times 50% slower than those of Neo4J.

For depth 3 and higher (Figures 2.14(b,c,d)), 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 sometimes.

For the shortest path with no label constraint (Q.38, Figure 2.15(a)), the performance of the systems was similar to the above, even though BlazeGraph and Sparksee are in this case, very similar, while Sqlg is still the slowest since it accesses all tables for all edges, and performs very large joins.

The label-filtered versions 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 stops the exploration almost immediately, *i.e.*, beyond 1-hop the query returned an empty set, hence the running time was not showing any interesting result. Running the same queries on *ldbc* we still observe (Figure 2.15(b)) that Neo4J is the fastest engine, while Sparksee is the second fastest in par with OrientDB for the breadth-first search. Instead, on the shortest path search filtered on labels, Titan (v.1.0) gets the second place.

Such results support the choice of index-free traversals implemented by the native systems for large and expensive visits on the graph. Yet, the dedicated structural index employed by Titan reaches the second-best performance.

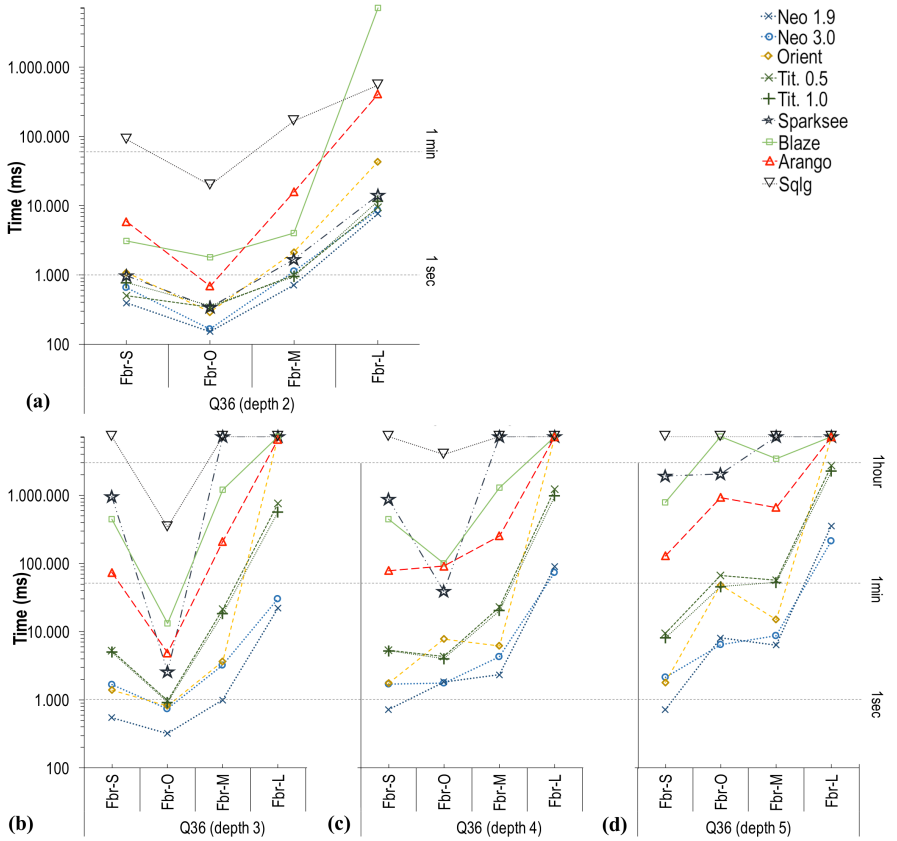


Figure 2.14: Time for breadth-first traversal (a) at depth= 2, and (b,c,d) at depth 3, 4, and 5

EFFECT OF INDEXING. The existing systems provide no support for structural indexes, *i. e.*, user-specified indexes for graph structures. They all have some form of indexes already implemented and hard-wired into the system. The only kind of index that can be controlled by the users is on attributes, and this is what we study. BlazeGraph provides no such capability, so is not considered. ArangoDB showed no difference in running times, so we suspect some defect in the Gremlin implementation. Insertions, updates, and deletions, as expected, become slower since the index structures have to be maintained, but not more than 10% in general, apart from Neo4J (v.3.0) and OrientDB that showed delays of about 30% and 100%, respectively. Despite this increase, Neo4J (v.1.9), Sparksee, and OrientDB remain the fastest systems for *CUD* operations. For search queries on node attributes, *i. e.*, Q.13 (Figure 2.16), the presence of indexes gives 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 witnesses up to a 600x speed up. We also see that Titan (v.1.0) still encounters problems on *Frb-L*. Sparksee and Neo4J (v.3.0) are not able to take advantage of such indexes. It seems that there is space for optimization in this sector.

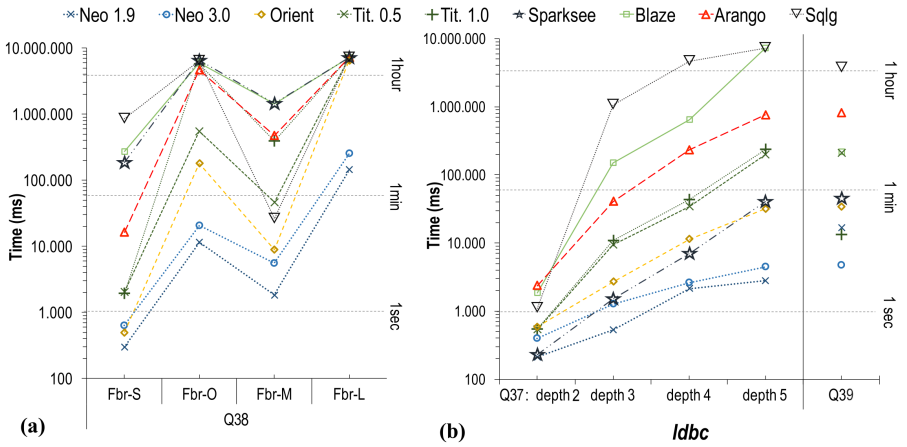


Figure 2.15: Time for (a) SP on Fbr-*, (b) fixed-label BFS & SP on LDBC.

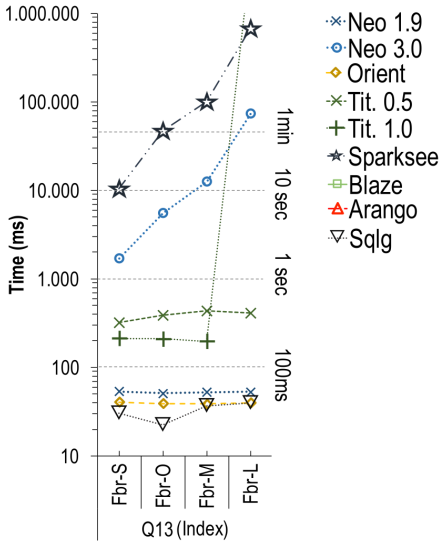


Figure 2.16: Time for indexed property search.

SINGLE *vs* BATCH EXECUTION. We looked at the time differences between single (run in isolation) and batch executions (Figure 2.17(a) and (b)). Tests in batch mode do not create any major changes in how the systems compare to each other. For the retrieval queries, the batch executions of 10 queries take exactly 10 times the time of one iteration, *i. e.*, no benefit is obtained from the batch execution. Instead, for the "CUD" operations, the batch takes less than 10 times the time needed for one iteration, meaning that in single mode most of the running time is due to some initial setup for the operation. For traversal queries, the batch executions only stress the differences between faster and slower databases.

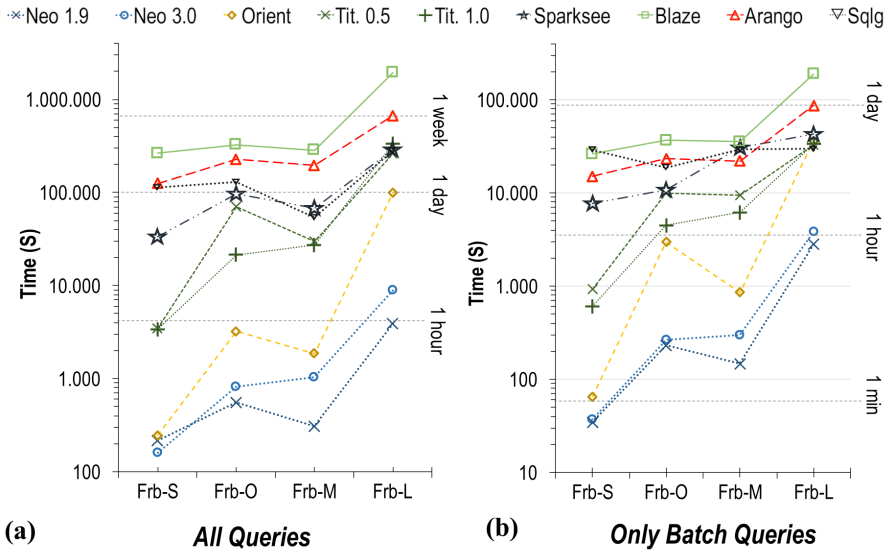


Figure 2.17: Overall time for (a) Single (a) and (b) Batch (Set A).

2.8.4 *Fixed Traversal and Pattern Matching*

As previously discussed, only the systems that support at least TinkerPop 3.4.2 and that well performed in the first set of tests were subjected to the following ones. For this set, we dropped Sparksee and BlazeGraph, for the first does not support the required TP version and the latter is the worst on all accounts. For all remaining systems we just bumped the version as shown in Table 2.2. For this batch of tests, We set the timeout to 3 hours.

FIXED TRAVERSALS. We first compare the performance on shortest path queries (Q.38 and Q.39 in Figure 2.18 and Figure 2.19) to the fixed path traversals (represented by Q.40, Q.42, and Q.44 in Figure 2.20 and Figure 2.21). The main difference between the two group of queries are that in the shortest path queries we are interested in finding at least one and only one shortest path between two nodes, while in the other traversals we are listing all paths of that exact length between the two nodes.

Here we notice that, being the *air-routes* dataset quite small and highly connected, all systems are fast in identifying the shortest path. Yet, when listing all paths, all systems incur in longer response time fast, but while other system see generally a slowdown of 1 order of magnitude (and are actually materializing some thousands paths), Sqlg instead suffers a slowdown of more than 2 orders of magnitude. This is a typical case where the intermediate results are amplified by join fan-out.

In the *ldbc.10* dataset, instead, being much larger but not as highly connected, we notice a different outcome. In the shortest path queries,

Neo4J is again the fastest system, but in this case Sqlg is the second fastest. Here, OrientDB and Titan (v.1.0) seem to suffer more because of the size of the graph. Finally, when it comes to fixed traversals we witness again the dramatic slowdown of Sqlg except that for searching a path between two nodes given a fixed sequence of labels. The fastest system in this dataset is Titan (v.1.0) but only for queries where labels are fixed (Q.42 and Q.44), while Neo4J is the fastest for the unlabeled search. Once more, this proves how Neo4J can effectively handle the widest range of traversals, while other systems are optimized only for a very narrow set of queries.

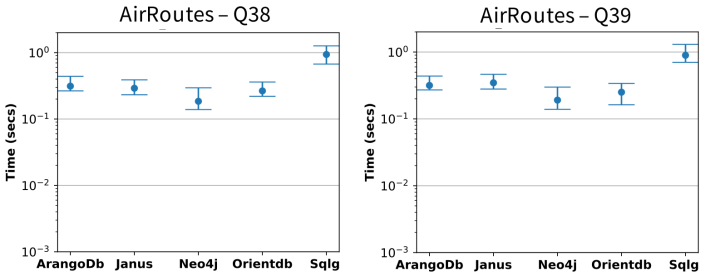


Figure 2.18: Time for SP on *air-routes*.

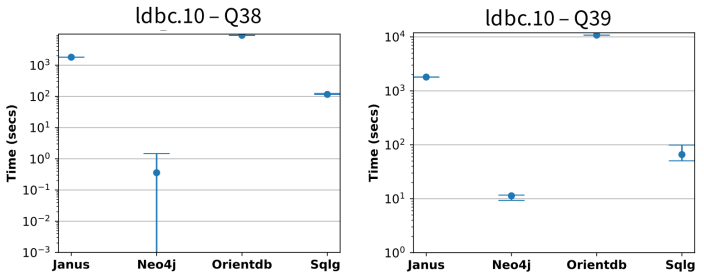


Figure 2.19: Time for SP on *ldbc.10*.

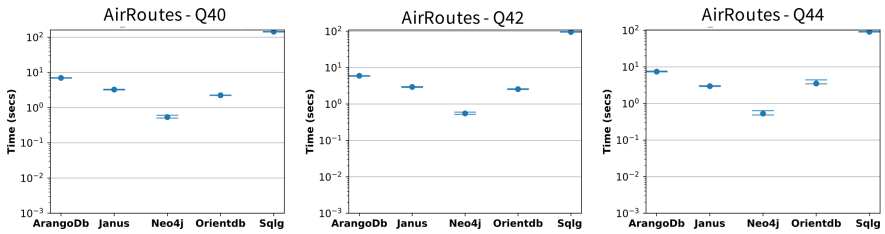


Figure 2.20: Time for fixed traversals on *air-routes*.

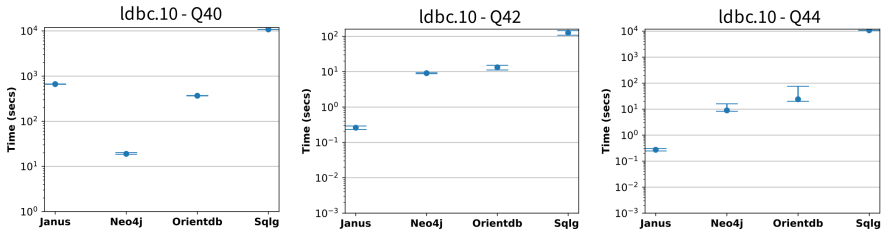


Figure 2.21: Time for fixed traversals on *ldbc.10*.

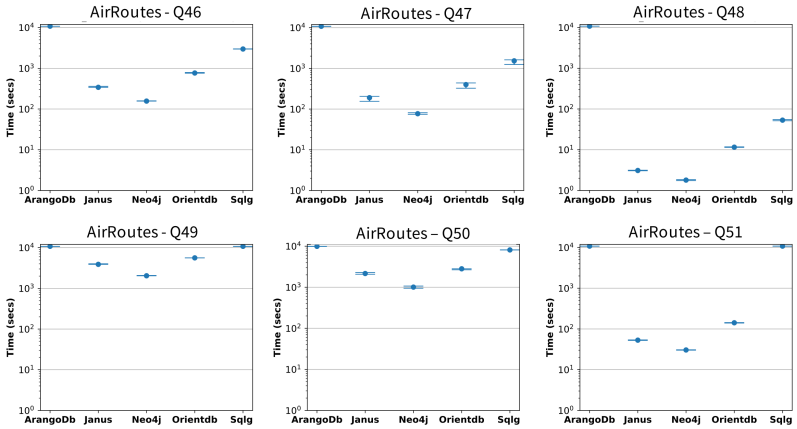


Figure 2.22: Time for pattern matching on *air-routes*.

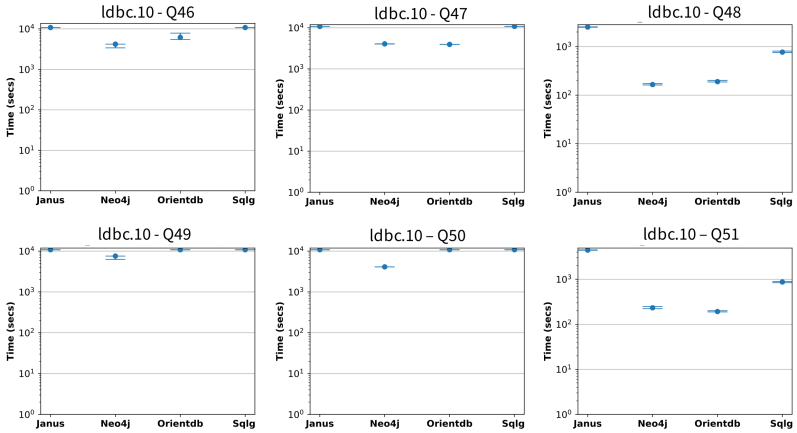


Figure 2.23: Time for pattern matching on *ldbc.10*.

PATTERN MATCHING. Here we test how systems perform when extracting some predefined structures. In particular subgraph shaped as triangle and as squares, the first is usually employed to identify relationships that can form a transitive closure, the second identifies equivalent paths connecting two nodes. In general, we see that these operations are particularly challenging, for some systems (especially ArangoDB) even on the smaller *air-routes* dataset (Figure 2.22). In general, pattern matching based only on edge labels (Q.46 and Q.49) has the worst performances. While, thanks to the much higher selectivity, searching for patterns around a specific node is understandably much faster. Once again, Neo4J presented the best performance, while OrientDB reached second best. Yet, on the larger *ldbc.10* native systems are often up to an order of magnitude faster than hybrid systems (Figure 2.23).

2.8.5 *Progress across Versions*

An important observation regards the difference in performance observed across different versions of the same system, *e.g.*, Neo4J. For Neo4J, in some cases, we see improvements or similar performance of the newer version compared to the old. Yet, we observe the opposite in very fast operations, *i.e.*, "CUD" queries and Search by ID. This is due to the overhead for accessing a wrapper library that was added in the newer version to cope with some licensing issues (*i.e.*, Tinkerpop adopted the Apache license, and the incompatibility in the licenses forced the developer to add a wrapper). Furthermore, other queries that show worsened performance are the global node filtering based on the degree (Q32-35) and some other traversals not filtered by type. We found that in the new version, the storage format and the part for traversal of relationships have been completely rewritten to improve filtered traversals (*i.e.*traversals that are restricted to a single edge type). In particular, relationship chains are now split by type and direction. The disk storage format changed as well. All these changes combined probably adds overhead to queries that access many edges of different types. Titan, on the other hand, demonstrated a slightly improved performance in the newer version. The main difference is that the software became production-ready (from v.0.5 to v.1). It is important to note here that the newer version also supported a newer version of the Gremlin language. The new version offers a cleaner syntax, but the way the operators have been implemented across versions is orthogonal to the language.

2.8.6 *Overall Evaluation and Insights*

Table 2.5 provides a summarization of the observed performance of the different GDBs in our experiments. The tick symbol (✓) means that the system achieved the best or near-to-best performance. The warning symbol (⚠) means that the system performance was towards the low end or indicated execution problems. Through this table, it is possible for a practitioner to identify the best system for a specific workload or scenario. One can see for instance that the native graph databases Neo4J, OrientDB and in part Sparksee, are better candidates for graph traversals operators (**T**). On the other hand, with data of few node and edge types, and a heavy search workload, hybrid systems may be a better fit. Finally, pattern matching queries posed serious challenges to most systems, and Neo4J has been shown to be the only one with consistently good performances.

NEO4J is the system with the shortest execution time when looking at the cumulative time taken by each system to complete the entire set of queries in both single and batch executions (Figure 2.17(a) and

	L	C	R				U	D		T							P	
	Load	Insertions	Graph Statistics	Search by Property\Label		Search by Id	Updates	Delete Node	Other Deletions	Neighbors	Node	Edge-Labels	Degree Filter	BFS	Shortest Path	Fixed Traversal	Pattern Matching	Heterogeneity
ArangoDB		✓	⚠	⚠			✓	✓	✓				⚠	⚠	⚠		⚠	⚠
BlazeGraph	⚠	⚠	⚠	⚠		⚠	⚠	⚠	⚠	⚠		⚠	⚠	⚠	⚠	-	-	-
Neo4J (v.1.9)	✓	✓				✓	✓	✓	✓	✓		✓	✓	✓	✓	-	-	-
Neo4J (v.3.0)	✓		✓							✓		✓	✓	✓	✓	✓	✓	✓
OrientDB		✓				✓		✓	✓	✓		✓		✓				✓
Sparksee	✓	✓	✓			✓	✓		✓						⚠	-	-	-
Titan (v.0.5)	⚠		⚠	⚠												-	-	-
Janus	⚠		⚠	⚠											⚠		✓	✓
Sqlg				✓			✓		✓	⚠		⚠	⚠	⚠	⚠	⚠	⚠	⚠

Table 2.5: Evaluation Summary.
Good: ✓, Bad: ⚠, No data: -.

(b)). It has the best overall stability and support for typical graph operations.

ORIENTDB also obtained relatively fast running times, which are often on par with Neo4J, and in some cases better than one of its two versions. However, it does not perform well when large portions of the graph have to be processed and kept in memory, *e.g.*, computing graph statistics on *Frb-L*.

TITAN results are quite often one order of magnitude slower than the best engine. It is slower in create and update operations but faster in deletions. This is most likely due to the tombstone mechanism that in deletions marks an item as removed instead of actually removing it. Yet, it has shown the best or second to best performances in some traversal and pattern matching operations, although these performances are not consistent across datasets and operations.

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. For instance, it gives a lot of timeouts on degree-based node search queries.

ARANGODB excels only in a few queries. For creations, updates and deletions, it ranks among the best. In the latest version they re-

place mapped-memory files with a persistent key-value store, therefore write performances have been reduced, but many previous limitations in durability have also been overcome. 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. An exception is when searching by ID, which is expected since at the core it is a key-value store, while for traversals, it has a narrow lead over Sparksee, BlazeGraph, and Sqlg demonstrating some 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. However, for queries containing 1-hop traversals restricted to a single edge-label, it performs reasonably well, yet it requires that also the node label is specified; otherwise, it has to union over all node tables. In some complex queries, instead, it takes advantage of the ability to conflate multiple operations in a single query and filter using foreign key indexes for specific edge labels.

BLAZEGRAPH results show in general that the indexes it builds automatically do not help much. Moreover, since every single step is executed against some specific graph API, instead of having the Gremlin query translated into SPARQL and executed as such, its query processing is, in general, less efficient. This graph API implementation does not allow it to exploit any of the optimization implemented by the SPARQL query engine.

SYSTEM SELECTION. All the above observations can serve as a guide in the choice of the right system for the different scenarios. The two main factors that should be considered in each scenario are the characteristics of the dataset and the intended workload, with the latter weighing more. When most of the intended operations are search on node properties, with few traversals, a hybrid system is preferable, *e.g.*, Sqlg. Such systems also allow the re-utilization of the existing technologies in an enterprise and allow the exploitation of robust optimizers and advanced index mechanisms. The choice of hybrid systems is also preferable for data in large enterprises with a low degree of heterogeneity. On the other hand, when the data is highly heterogeneous, *i.e.*, many different edge types, and in the workload is predominant the presence of long traversals, native graph systems appear to be a better choice. This is the case especially when the data comes from a knowledge graph or is the result of the integration of many different heterogeneous sources. Another factor to consider is the dynamicity of the data. If many insert, update, and

delete operators are to be performed, Sparksee, and ArangoDB are the best performing in our study. If, however, the data is going to be relatively static and the majority of operations are going to be search queries, then Neo4J and OrientDB perform better on graph search and Sqlg on content filters. It is important to note that having studied all the well-known systems characterized as GDB, our findings offer a good understanding of the behavior of GDB solutions. There may be, of course, proprietary or special-purpose solutions, which are not characterized as GDBs, yet, they offer some graph data storage and querying functionality. Such a system may show a different behavior but are not part of our focus of the current work.

HYBRID AND NATIVE SYSTEMS. The experiments show that hybrid and native systems perform differently. In general, despite the idea that reusing an existing storage layer looks like a promising solution, typical graph workloads require specialized data organization and query processing to obtain the necessary performances. For a limited set of use cases, the hybrid systems in our study perform equally well as the native. However, for traversal queries, like finding the connectivity between two nodes, BFS visits, and the enumeration of edges, these hybrid systems underperform significantly. Hence, this suggests that the design choices made in native systems, *e.g.*, the separation of the graph structure from other data values, are more effective than the strategies adopted by the hybrid systems in our study.

The benefit of the native GDBs against the hybrid may, however, vary based on the context. In graph analytic pipelines, many tasks need to be performed on the data by different tools. Using a native GDB forces the data to be imported in the GDB for the management and exported for other tasks, diminishing the benefit of the effective management the native GDB offers. However, a hybrid GDB can process the data even while it is residing in external storage. Thus, big analytic pipeline systems, like SAP Hana, may opt for a hybrid GDB solution.

HETEROGENEITY AND KNOWLEDGE GRAPHS. We tested the ability of the various systems in handling a large knowledge graph (*DBpedia*). Handling KGs is notably challenging for the relational model because the nodes feature properties with different types. Typical options are encoding the type explicitly in the attributes or storing everything as strings. Both solutions have notable drawbacks, and they are not currently supported in the Sqlg system implementation. ArangoDB and OrientDB organize nodes and edges in shards or partitions based on their labels. This partitioning strategy limits both systems in the maximum number of node and edge labels supported, which is usually in the order of 2 or 3 thousand. Therefore, only Neo4J

and Janus demonstrated to have an internal data organization able to accommodate such a large number of edge types and attributes. Additionally, systems like Sqlg, ArangoDB, and Janus require costly operations whenever a new attribute, node label, or edge label is introduced in the database, making them ill-equipped in supporting graphs that need to support frequent addition to their schema. Moreover, when handling KGs, typical queries include the so-called Basic Graph Patterns, which require pattern matching capabilities. Our experiments and experiences with the systems highlighted an overall lack of appropriate support for this use case.

QUERY LANGUAGE. Although all the systems we studied support Gremlin, most of them also offer their own custom declarative query language, for which they also offer query planning and optimization for that. Many translate Gremlin queries in a one-to-one fashion to native primitives, but in that way, many Gremlin-side optimizations cannot be implemented. This behavior indicates that for many GDBs, Gremlin is not their first priority. Additional evidence is the struggle of loading the data through Gremlin compared to native API. Yet another shred of evidence is the observed problems with intermediate results exhausting the memory. This optimization, however, is for complex queries, and our microbenchmark approach is based on primitive queries. Hence, this limitation does not affect our findings.

2.9 CONCLUSION

We have performed an extensive experimental evaluation of the state-of-the-art graph databases in ways not tested before. We provided a principled and systematic evaluation methodology based on microbenchmarks. We have materialized it into an evaluation suite, designed with extensibility in mind, containing datasets, queries, and scripts. Our findings have illustrated the advantages microbenchmarks can offer to practitioners, developers, and researchers and how they can help them better understand design choices, performances, and functionalities of the graph database system. As a result, we have presented a number of findings that help understand the trade-offs between native and hybrid graph database systems, their effect on important graph queries like traversals and pattern matching, and also their current capability to handle highly heterogeneous graphs.

AVAILABILITY OF DATA AND MATERIAL: The data used in this work was collected from existing public datasets, references to the datasets exist in Section 2.6. The code and all the datasets used in this work can be accessed online at `https://graphbenchmark.com`

3

COMPUTING & MAINTAINING K–CORE

Over the last decade, the field of distributed processing of large graphs has attracted considerable attention. This field has been highly motivated, not only by the increasing size of graph data, but also by its huge number of applications. Such applications include the analysis of social networks [50, 127], Web graphs [8], as well as spatial networks [113]. k-core decomposition is an important task that has been used to understand large graph data by identifying k-cores, which are a special family of maximally-induced subgraphs. Intuitively, a k-core is obtained by recursively removing all nodes of degree smaller than k, until the degree of all remaining vertices is larger than or equal to k. A node is said to have coreness k if it belongs to the k-core but not to the $(k+1)$-core [21]. The k-core decomposition has been used in several different domains including bioinformatics [18], graph visualization [76] and Internet structure analysis [7].

Several algorithms exist for k-core computation in static graphs, both in centralized and decentralized settings. Yet, modern graphs are growing dramatically and are becoming more and more dynamic, with an ever-increasing rate of node/edge additions or removals. In such environments, there is an urgent need for solutions that not only compute the k-core of large graphs, but are also able to maintain it in an efficient way while the data is constantly changing.

Our work is motivated by two factors. First, the size of the graphs is becoming so large, that makes it difficult to process with off-the-shelf, single machines. Second, and most important, the fact that the majority of the existing large graphs are already stored in a distributed way, either because they cannot be stored on a single machine due to their sheer size, or because they get processed and analyzed with decentralized techniques that require them to be distributed among a collection of machines. For these reasons, we identified the need of methods and techniques that can exploit as much as possible the existing topology of the graph data and perform the k-core decomposition in a cooperative way among the distribution nodes. Our solution is based on the idea of recomputing the coreness only for those nodes of the graph that are affected by the graph updates. The propagation of the effect is done first inside the partition that exists in a single node, and then across partitions by considering the *cut edges*, *i. e.*, edges between nodes of different partitions. To the best of our knowledge, the proposed solution is the first that allows to consider graph streams and incremental changes while computing k-core decomposition in graphs that are already stored in a distributed manner.

CONTRIBUTIONS. More specifically, our contributions are the following:

- We present a distributed and streaming k-core decomposition algorithm for very large graphs that are partitioned and distributed across the nodes of a physically independent network of machines.
- We propose a maintenance strategy that deals with incremental changes on the graph by looking to the nodes that need to be updated in all the partitions and updating the coreness of only those nodes.
- We present an implementation of our algorithms on top of AKKA [148], a framework for building distributed and resilient message-driven applications.
- We experimentally evaluate the performance of the proposed approach on both real and synthetic datasets.

The remainder of this chapter is organized as follows. Section 3.1 presents an overview of the related work and specifically those works that deal with the concept of distributed k-core decomposition. In Section 3.2, we define the problem of distributed k-core decomposition in large dynamic graphs. In Section 3.3, we present our incremental approach for the k-core maintenance in such graphs. In Section 3.4, we describe our experimental evaluation and we discuss our findings.

3.1 RELATED WORK

In this section, we highlight the relevant literature in the field of k-core decomposition. We consider three research areas: (1) centralized algorithms, (2) distributed algorithms, and (3) distributed and streaming algorithms for k-core decomposition and maintenance in dynamic graphs.

3.1.1 *Centralized algorithms*

The first k-core decomposition algorithm was originally proposed by Batagelj and Zavernik (BZ) [21]. The main idea of the algorithm is to recursively delete vertices of degree less than k. It requires random access to the whole graph, which should therefore be kept in the main memory for the sake of performance. Cheng *et al.* [29] have proposed a strategy based on the BZ algorithm to handle graphs that do not fit into main memory. The proposed algorithm requires $O(k_{max})$ scans of the graph, where k_{max} is the largest coreness value of the graph.

3.1.2 *Distributed algorithms*

The problem of distributed k-core decomposition was first studied in [102] and a new algorithm for the computation of the k-coreness of a network was proposed. The proposed approach has been applied to two different computational models, one based on Pregel [92] and one based on a block-centric approach [151]. In the former, one computational unit is associated with one node in the graph, and communication occurs only through direct messages between nodes connected through an edge. In the latter, one host stores many nodes together with their local and remote edges, while communication occurs through messages between hosts.

3.1.3 k-*core decomposition and maintenance in dynamic graphs*

Few works have studied the k-core decomposition problem from large dynamic graphs [4, 29, 67, 81, 101, 125]. Li, Yu, and Mao [81] have presented a k-core maintenance approach in dynamic graphs. They proposed two pruning techniques to remove the nodes whose coreness is definitely unchanged after an update operation over the initial graph. When a dynamic graph is updated, the minimal subgraph for which k-core decomposition might have changed is computed, instead of recomputing everything from scratch. The proposed algorithm keeps track of core number for each vertex and upon an update provides the subgraph for which k-core decomposition needs to be updated. In [125], the authors present an incremental k-core decomposition algorithm for streaming graph data. The main idea of their approach is first to locate a small subgraph that contains the set of vertices whose coreness values have to be updated. Then it processes the located subgraph to incrementally maintain the coreness values of its vertices when a single edge is inserted or removed. In [4], the authors present a distributed incremental algorithm for k-core maintenance in large dynamic graphs. The presented approach uses HBase to store the graph data and hence to exploit the horizontally scaling of its distributed storage. The distributed algorithm constructs a k-core subgraph by progressively removing edges in parallel by remote calls on distributed nodes. It is worthwhile to mention that the approach presented in [4] uses a fixed k value and does not determine all the updated k-cores when dynamic changes are made to the graph.

Most of the above-cited solutions deal with core maintenance of large dynamic graphs. However, these approaches do not consider the case when the graph is too large to be kept in the main memory or when the graph is already distributed across several machines. Only a few works include the core maintenance task in the case of large distributed graphs, which is the addressed issue in this chapter.

3.2 PROBLEM FORMULATION

Given an undirected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, the concept of k–*core decomposition* [21] is condensed in the following two definitions:

Definition 1 (k-core). *A subgraph* $G(C)$ *induced by the set* $C \subseteq V$ *is a* k*-core if and only if* $\forall u \in C : d_{G(C)}(u) \geqslant k$, *and* $G(C)$ *is maximal,* i. e., *for each* $\overline{C} \supset C$, *there exists* $v \in \overline{C}$ *such that* $d_{G(\overline{C})}(v) < k$.

Definition 2 (coreness). *A node in* G *is said to have coreness* k ($k_G(u) = k$) *if and only if it belongs to the* k*-core but not the* $(k+1)$*-core.*

$d_G(u)$ and $k_G(u)$ denote the degree and the coreness of u in G, respectively; in what follows, however, G can be dropped when it is clear from the context. The subgraph of G induced by C is defined as $G(C) = (C, E|C)$ where $E|C = \{(u, v) \in E : u \in C \vee v \in C\}$.

A k-core of a graph $G = (V, E)$ can be obtained by recursively removing all the vertices of degree less than k, until all vertices in the remaining graph have degree at least k. While such a centralized solution is simple and works in linear time [21], the situation gets more complicated when the issues of dynamism and scale are considered; even more so when they are considered together. When new edges and nodes are added or removed, this may cause a cascading re-computation of the coreness of the nodes surrounding the newcomers, which can potentially span the entire graph. While recomputing the coreness of the entire graph is always an option, limiting the re-computation to as few nodes as possible is desirable. When the graph is large and cannot be stored or computed using a single machine, its vertex set can be partitioned into p disjoint partitions $\{V_1, \ldots, V_p\}$; in other words, $V = \cup_{i=1}^{p} V_i$ and $V_i \cap V_j = \emptyset$ for each i, j such that $1 \leqslant i, j \leqslant p$ and $i \neq j$. Such partitions induce p subgraphs $G_i = (V_i, E_i)$, where $E_i = E|V_i$. In such subgraphs, an edge (u, v) is called a *frontier edge* of G_i if $u \in V_i$ and $v \in V_j \neq V_i$, *i. e.*the edge links a node in V_i with a node in a different partition. The set of frontier edges of a subgraph G_i is denoted F_i; clearly, $F_i \subseteq E_i$. The set of all frontier edges of a graph G is defined as $V_f = \cup_{i=1}^{p} F_i$ where F_i is the set of frontier edges of a subgraph G_i; clearly, $V_f \subseteq E$.

Given a graph $G(V, E)$, distributed in a number of partitions, and having the k-core decomposition already computed over it, we are interested in finding the coreness after a number of modifications (insertions or deletions) have taken place on the graph, without having to restart the computation from scratch. A by-product of such a maintenance solution is that the k-core can also be computed for the first time by running the k-core computation in each partition independently, and then considering the edges between the partitions as updates, and applying the maintenance approach that updates the coreness of every node into the right value considering the overall

graph. After that, the k-core will be given by the nodes whose coreness is k.

3.3 K-CORE COMPUTATION

We assume that the graph is subdivided into multiple partitions, each of them assigned to a different worker. Inside each partition, a centralized algorithm to compute the coreness is run. At that point, we treat large-scale and dynamism in the same way: whenever a new edge is added to the graph, we first determine the set of candidate nodes (nodes whose coreness needs to be updated); then, we compute the correct values for the coreness of those nodes. The set of nodes to be updated may span multiple partitions, particularly when frontier edges are added. The system overview of our approach is illustrated by Figure 3.1.

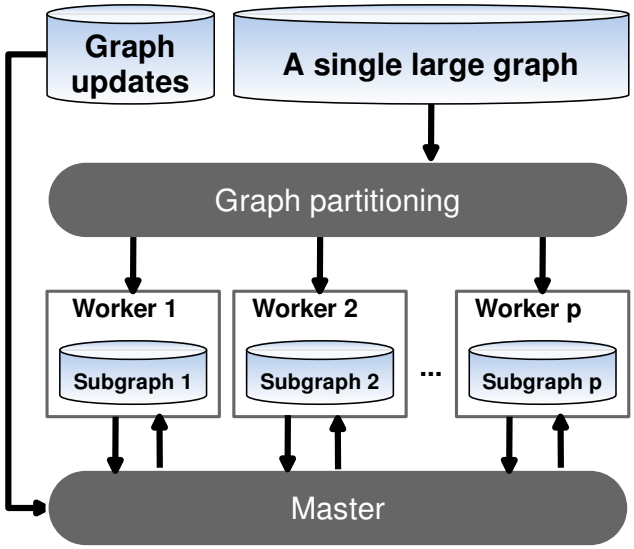


Figure 3.1: System overview.

As illustrated in Figure 3.1, each worker runs a centralized algorithm to compute the coreness of nodes of its associated subgraph. The master worker orchestrates the execution of the update process after considering graph changes. Our approach operates in three computing modes:

- *M2W mode*. In this mode, message exchanges between the master and all workers are allowed. The master uses this mode in order to ask a distant worker to look for candidate nodes. The worker uses this mode in order to send the set of computed candidate nodes to the master.

- *W2W mode*. In this mode, message exchanges between workers are allowed. The workers use this mode in order to propagate the search for candidate nodes to one or more distant workers.

- *Local mode*. In this mode, only local computation is allowed.

Our algorithm exploits Theorem 1, first stated and demonstrated by Li, Yu and Mao [81], that identifies what are the *candidate nodes* that may need to be updated whenever we add an edge:

Theorem 1. *Let* $G = (V, E)$ *be a graph and* (u, v) *be an edge to be inserted in* E*, with* $u, v \in V$*. A node* $w \in V$ *is said to be a* candidate *to be updated based on the following three cases:*

- *If* $k(u) < k(v)$*,* w *is candidate if and only if* w *is* k*-reachable from* u *in the original graph* G *and* $k = k(u)$*;*

- *If* $k(u) > k(v)$*,* w *is candidate if and only if* w *is* k*-reachable from* v *in the original graph* G *and* $k = k(v)$*;*

- *If* $k(u) = k(v)$*,* w *is candidate if and only if* w *is* k*-reachable from either* u *and* v *in the original graph* G *and* $k = k(u)$*.*

A node w is k-*reachable* from u if there exists a path between u and w in the original graph such that all nodes in the path (including u and w) have coreness equal to $k = k(u)$. At this point, we can further prune the number of possible nodes using Theorem 2.

Theorem 2. *Let* $G = (V, E)$ *be a graph and let* C *be the set of candidates nodes after considering the new edge* (u, v)*. Let* $N(w)$ *be the set of neighbors of* w *and let* $X(w)$ *be the number of neighbors of* w *such that* $\forall w' \in N(w)$*,* $k(w') > k(w)$ *or* $w \in C$*. Then,* $\forall w \in C$*,* $X(w) \leqslant k(u)$ *implies that the coreness of* u *is definitely unchanged.*

Proof. After considering an edge to be inserted (u, v), the quantity $X(w)$ consists in the number of neighbors whose coreness values are larger than $k(w)$. We note that if $X(w) \leqslant k(w)$, the node w cannot belong to the $(k(u) + 1)$-core and thus, the coreness of w remains equal to $k(w)$. □

In order to increase the performance of our approach, the partitioning of the input graph needs to be optimized to have balanced partitions and a small number of frontier edges. In the following, we present basic algorithms for the distributed k-core decomposition task. Table 3.1 summarizes the notations used in our algorithms.

Algorithm 1 implements the distributed orchestration mechanism that first computes the coreness in each of the partitions and then adds the frontier edges one by one. This algorithm is run by a master worker under the *M2W* computing mode. Later, any kind of edge can be added, following the same approach.

G	An undirected graph partitioned into p partitions
$k(u)$	Coreness of u
$N(u)$	Neighbors of u
V_f	The set of all frontier edges of G
$\mathsf{kCore}(G_i)$	Executed by worker i and computes the coreness of all nodes of G_i
$\mathsf{partitionID}(u)$	Partition associated to node u
$\mathsf{visited}(u)$	Indicates whether the node u is visited or not while we are looking for reachable nodes.
$p_i.f()$	A remote call to the $f()$ function on partition p_i

Table 3.1: Notation.

Algorithm 1: Distributed k-core decomposition

```
1 foreach j ∈ {1, ..., P} do
2     p_i.kCore(G_i)
3 foreach e = (u, v) ∈ V_f do
4     C ← getCandidates(e)
5     C ← pruneCandidatesInsert(C)
6     foreach u ∈ C do
7         p_u ← partitionID(u)
8         p_u.k(u) ← p_u.k(u) + 1
```

The update process is composed of three steps. The first step consists in activating the *W2W* computing mode and identifying the set of candidate nodes, *i. e.*, the set of nodes that may need to be updated (Algorithm 2). For each frontier edge (u, v), the current coreness of nodes u and v are compared. If the coreness of u (respectively v) is greater than the coreness of v (respectively u), then the set of candidate nodes consists of nodes that are k-reachable from v (respectively u), where $k = k(v)$ (respectively, $k(u)$). If the coreness of u is equal to the coreness of v, then the set of candidate nodes consists of the union of nodes that are k-reachable from u and from v, where $k = k(v) = k(u)$.

The *reachable* (u) function returns the set of nodes that are k-reachable from u, by performing a depth-first visit. The visit can span multiple partitions, meaning that the visit of a frontier edge can lead to the visit of a node in a different partition. The pseudo-code shown below illustrates the behavior of the visit; in the real implementation, the nodes identified as potential candidates are sent back to the master node that orchestrates the execution.

The second step consists in activating the *Local* computing mode of our approach and selecting the set of nodes that need to be updated from the set of candidate nodes. This step is achieved by applying the pruning strategy introduced in Theorem 2.

Algorithm 2: SET getCandidates (EDGE (u, v))

```
1  p_u ← partitionID(u)
2  p_v ← partitionID(v)
3  C ← ∅
4  if k(u) < k(v) then
5      C ← p_u.reachable(u)
6  else if k(u) > k(v) then
7      C ← p_v.reachable(v)
8  else
9      C ← p_u.reachable(u) ∪ p_v.reachable(v)
10 return C
```

Algorithm 3: SET reachable (NODE u)

```
1  SET C ← ∅
2  if visited(u) = false then
3      C ← C ∪ {⟨u, k(u), N(u)⟩}
4      visited(u) ← true
5      p_u ← partitionID(u)
6      foreach v ∈ N(u) do
7          if k(v) = k(u) then
8              p_v ← partitionID(v)
9              if p_u = p_v then
10                 C ← C ∪ reachable(v)
11             else
12                 C ← C ∪ p_v.reachable(v)
13 return C
```

Finally, the third step consists in activating the *M2W* computing mode and updating the coreness values of the set of nodes computed in the second step.

It is important to highlight that the algorithms presented above aim to compute the distributed k-core decomposition in a large partitioned graph. The frontier edges of the original graph are considered one by one after computing the k-cores of the distributed graph partitions separately. The proposed approach deals with frontier edges as edge insertions in a dynamic graph. Consequently, the proposed algorithms can be simply used to deal with edge insertions in large dynamic graphs. Algorithm 5 implements the update process for edge insertion in a large dynamic graph.

The edge deletion task is slightly different from edge insertion. Algorithms 6 implement the update process for edge deletion in a large dynamic graph.

Algorithm 4: pruneCandidatesInsert (SET C)

```
1  changed ← true
2  while changed do
3      changed ← false
4      foreach ⟨u, k(u), N(u)⟩ ∈ C do
5          count ← 0
6          foreach v ∈ N(u) do
7              if ⟨v, k(v), N(v)⟩ ∈ C or k(v) > k(u) then
8                  count ← count + 1
9          if count ⩽ k(u) then
10             changed ← true
11             C ← C − {⟨u, k(u), N(u)⟩}
12 return C
```

Algorithm 5: updateInsertions (SET S)

```
1  foreach e ∈ S do
2      C ← getCandidates(e)
3      C ← pruneCandidatesInsert(C)
4      foreach u ∈ C do
5          p_u ← partitionID(u)
6          p_u.k(u) ← p_u.k(u) + 1
```

Algorithm 6 is run by a master worker under the *M2W* computing mode. It consists of three main steps. The first step consists in activating the *W2W* computing mode and identifying the set of nodes that may need to be updated after the deletion using Algorithm 2. The second step consists in activating the *Local* computing mode and selecting the set of nodes that need to be updated from the set of candidate nodes. The set of nodes with unchanged coreness values is computed by Algorithm 7. We notice that for edge deletions, the set of nodes that need to be updated is slightly different from the set computed by Algorithm 4.

The last step of the edge deletion task consists in activating the *M2W* computing mode and updating the coreness values of the nodes that need to be updated.

3.4 EXPERIMENTS

We have performed an extensive set of experiments to evaluate the effectiveness and efficiency of our approach on a number of different real and synthetic datasets. Additional and more detailed information

Algorithm 6: updateDeletions (SET S)

```
1  foreach e ∈ S do
2      C ← getCandidates(e)
3      C ← C − pruneCandidatesDelete(C)
4      foreach u ∈ C do
5          p_u ← partitionID(u)
6          p_u.k(u) ← p_u.k(u) − 1
```

Algorithm 7: pruneCandidatesDelete (SET C)

```
1  changed ← true
2  while changed do
3      changed ← false
4      foreach ⟨u, k(u), N(u)⟩ ∈ C do
5          count ← 0
6          foreach v ∈ N(u) do
7              if ⟨v, k(v), N(v)⟩ ∈ C or k(v) > k(u) then
8                  count ← count + 1
9          if count < k(u) then
10             changed ← true
11             C ← C − {⟨u, k(u), N(u)⟩}
12 return C
```

about our datasets and our experiments in general can be found in the following link: https://martinbrugnara.it/rp/dkcore.html.

3.4.1 *Experimental data*

Since we are interested in the core of graph data, the characteristic properties of our datasets are the number of nodes, the number of edges, the diameter, the average clustering coefficient, and the maximum coreness. Table 3.2 shows these properties for the datasets we have used in our work. We have used two groups of datasets: (1) real-world datasets, made available by the Stanford Large Network Dataset collection [79] and (2) synthetic datasets, created by a graph generator based on the Nearest Neighbor model [123], that builds undirected graphs with power-law degree distribution with exponent between 1.5 and 1.75, matching that of online social networks.

3.4.2 *Experimental environment*

We have implemented our approach on top of the AKKA framework, a toolkit and runtime for building highly concurrent, distributed, re-

Dataset	Type	♯ Nodes (N)	♯ Edges (M)	⊘	Avg. CC	Max(k)
DS1	Synthetic	10,000	70,622	4	0.3977	33
DS2	Synthetic	20,000	144,741	4	0.3935	38
DS3	Synthetic	50,000	365,883	4	0.3929	42
DS4	Synthetic	100,000	734,416	4	0.3908	46
ego-Facebook	Real	4,039	88,234	8	0.6055	115
email-Enron	Real	36,692	183,831	11	0.4970	43
roadNet-TX	Real	1,379,917	1,921,660	1,054	0.0470	3
roadNet-CA	Real	1,965,206	2,766,607	849	0.0464	3
com-LiveJournal	Real	3,997,962	34,681,189	17	0.2843	296
soc-LiveJournal	Real	4,847,571	68,993,773	16	0.2742	318

Table 3.2: Experiments data.

silient message-driven applications. In order to evaluate the performance of our approach, we used a cluster of 17 `m3.medium` instances on Amazon EC2. Each `m3.medium` instance contained 1 virtual 64-bit CPU, 3.75 GB of main memory, and a 8 GB of local instance storage. We also implemented two existing approaches for k–core decomposition in large dynamic graphs. First, we implemented Li *et al.*'s approach [81] and we run it on a machine equipped with two Intel(R) Xeon(R) E5-2440 CPUs (2.40GHz) and 192 GB of memory. Second, we implemented the HBase-based approach of Aksu *et al.* [4] and we run it on a cluster of 9 `m3.medium` instances on Amazon EC2 (1 master and 8 slaves).

3.4.3 *Experimental protocol*

In order to simulate dynamism in each dataset, we consider two update scenarios. For each scenario, we measure the performance of the system to update the core numbers of all the nodes in the considered graph after insertion/deletion of a constant number of edges.

- In the *inter-partition* scenario, we either delete or insert 1000 random edges connecting two nodes belonging to *different* partitions.
- In the *intra-partition* scenario, we either delete or insert 1000 random edges connecting two nodes belonging to *the same* partition.

We consider three figures of merit to evaluate our approach.

First, we measure the average insertion time (AIT) and the average deletion time (ADT) in the two proposed scenarios. We also compare the results of our algorithm with existing solutions for k–core decomposition in large dynamic graphs, including Li *et al.*'s approach [81] and Aksu *et al.*'s approach [4].

Second, we study data communications and networking. In this context, we measure the amount of exchanged data needed to compute the task of k-core decomposition.

Third, we study the scalability of our approach with respect to the number of machines in our cluster. In this context, we vary the number of worker machines and measure the average insertion/deletion time for each update scenario.

3.4.4 *Experimental results*

SPEEDUP Table 3.3 illustrates the results obtained with both the real and the synthetic datasets. For each dataset, we measure the number of frontier edges and we record the average insertion time (AIT) and the average deletion time (ADT) over the 1000 insertions/deletions for both *inter-partition* and *intra-partition* scenarios. To generate the results of Table 3.3, we randomly partition the graph dataset into 8 partitions.

Dataset	Number of frontier edges	AIT (ms)		ADT (ms)	
		inter	intra	inter	intra
DS1	61,803 (87.51%)	27	6	20	4
DS2	126,720 (87.54%)	39	16	27	9
DS3	320,318 (87.54%)	42	10	32	8
DS4	643,189 (87.57%)	30	10	25	8
ego-Facebook	77,253 (87.55%)	38	15	32	10
email-Enron	161,055 (87.61%)	32	8	28	6
roadNet-TX	1,681,830 (87.51%)	28	9	25	7
roadNet-CA	2,420,674 (87.49%)	30	12	26	10
com-LiveJournal	30,348,426 (87.50%)	256	30	205	27
soc-LiveJournal	59,916,050 (86.84%)	579	27	499	25

Table 3.3: Experiments results.

As shown in Table 3.3, we observe that in the *intra-partition* scenario, the values of the average insertion/deletion time are much smaller than those in the *inter-partition* scenario. This can be explained by the fact that the inserted/deleted edges in the *intra-partition* scenario are internal ones. Consequently, the amount of data to be exchanged between the distributed machines in the case of internal edges is smaller, in most cases, than the amount of exchanged data in the case of edges of the *inter-partition* scenario (*i. e.*, frontier edges). During the k-core maintenance process after insertion/deletion of an internal edge, there is always the chance of not having to visit distributed workers/partitions other than the partition that holds the internal edge.

Figure 3.2 presents a comparison of our approach with both the sequential approach proposed by Li *et al.* and the HBase-based approach proposed by Aksu *et al.* in terms of average insertion/deletion time. For our approach, we used 9 `m3.medium` instances on Amazon EC2 (1 acting as a master and 8 acting as workers). For the HBase-based approach, we used 9 `m3.medium` instances on Amazon EC2 (1 master node and 8 slave nodes).

We notice that Li *et al.*'s approach produces better results in terms of average insertion/deletion time for almost all small datasets. This can be explained by the communication cost of our approach compared to Li *et al.*'s approach, which performs in-memory and centralized computing. For road network and LiveJournal datasets, our approach performs much faster that Li *et al.*'s approach with both *inter-partition* and *intra-partition* scenarios. It is also important to mention here that Li *et al.*'s approach has failed to deal with LiveJournal datasets using one of the `m3.medium` instances used for the evaluation of our approach due to lack of memory.

As shown in Figure 3.2, our approach allows much better results compared to the HBase-based approach for almost all datasets. It is noteworthy to mention that the presented runtime values of the HBase-based approach correspond to the maintenance time of only one fixed k value core ($k = max(k)$ in our experimental study). This means that, for each dataset, the maintenance process of the HBase-based approach needs to be repeated $max(k)$ times in order to achieve the same results as our approach.

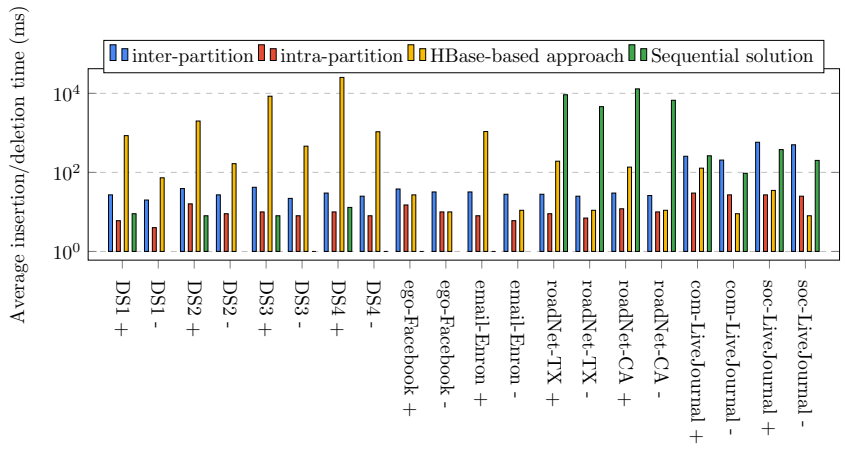


Figure 3.2: Average insertion/deletion time.

DATA COMMUNICATIONS AND NETWORKING In order to study data communications and networking, we begin by examining the amount of exchanged data between the distributed machines.Then, we study the impact of the partitioning method and the number of graph partitions on the amount of exchanged data between the master node and the worker nodes. Figure 3.3 shows the average value of the amount of exchanged data between the master node and the worker nodes. The amount of exchanged data is shown in log-scale. For each dataset, we present the mean value of the exchanged data. As illustrated in Figure 3.3, the amount of data to be exchanged in the

intra-partition scenario is much smaller than the amount of exchanged data in the *inter-partition* scenario.

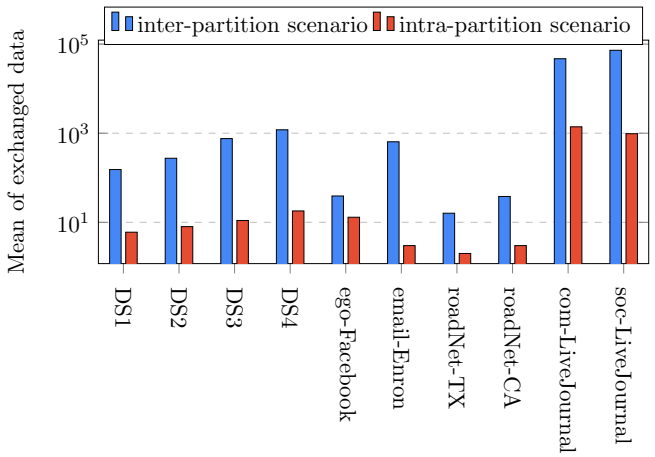


Figure 3.3: Amount of exchanged data.

In order to study the impact of the number of partitions on the amount of exchanged data, we show in Figure 3.4, for each number of partitions, the mean value of the exchanged data and the standard deviation value which corresponds to the error bar. This standard deviation gives a general idea of how the values of the exchanged data are concentrated around the mean value.

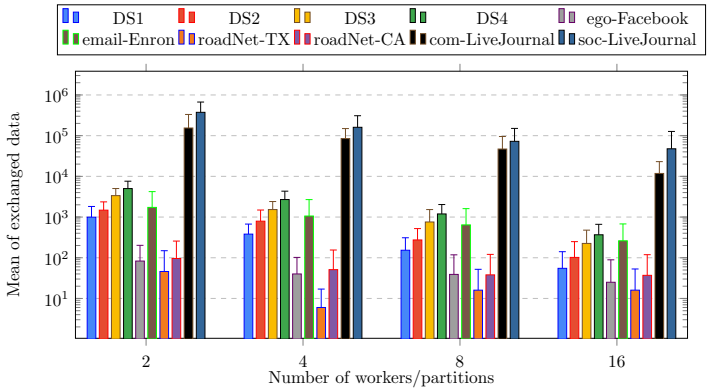


Figure 3.4: Impact of the #partitions on the amount of exchanged data.

We note that the amount of exchanged data is inversely proportional to the number of partitions in almost all datasets (see Figure 3.4).

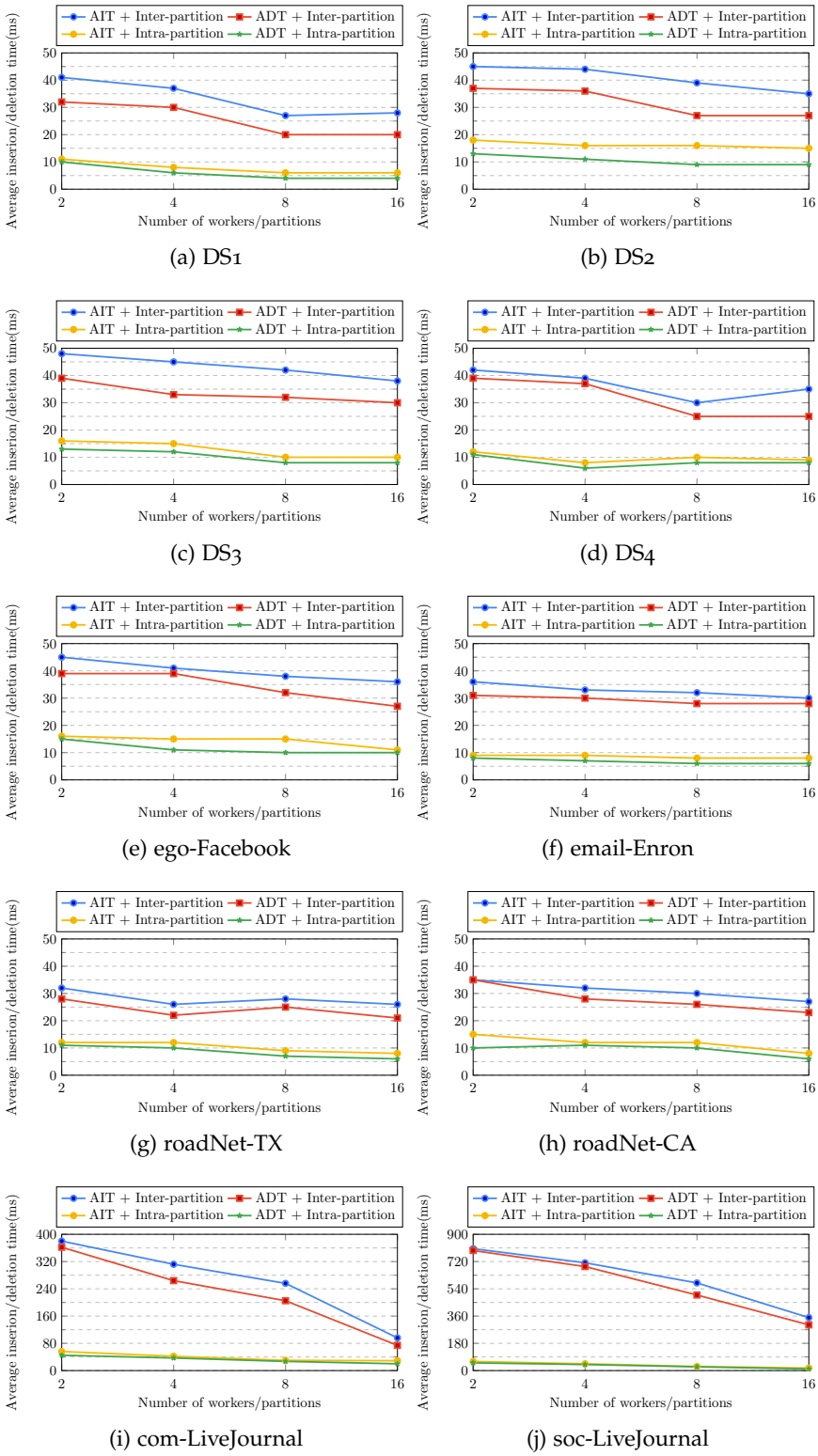
SCALABILITY To study the scalability of our approach and to show the impact of the number of worker machines on the maintenance task runtime in the case of large-scale networks, we measured the

average insertion/deletion time of our approach for each number of worker machines. We present these results in Figure 3.5.

As illustrated in Figure 3.5, our approach scales up as the number of worker machines increases. In fact, our approach's average insertion/deletion time is inversely proportional to the number of such machines.

3.5 CONCLUSIONS

This chapter deals with the problem of distributed k-core decomposition in large dynamic networks. Most of the existing approaches solve the problem of k-core maintenance for graphs that can fit into the main memory of one single machine. They do not consider the cases of already distributed graphs and graphs that can not fit into one single machine. In this chapter, we have introduced an efficient distributed and streaming k-core decomposition approach for large and dynamic networks. Our approach deals with graph changes/updates by selecting only the nodes of a subgraph of the original graph that really need to update their core numbers. We implemented our approach on top of the AKKA framework, a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications. By running experiments on a variety of both real and synthetic datasets, we have shown that the proposed method is interesting in the case of very large graphs with a very satisfactory performance and scalability for large graphs.



(a) DS1 (b) DS2

(c) DS3 (d) DS4

(e) ego-Facebook (f) email-Enron

(g) roadNet-TX (h) roadNet-CA

(i) com-LiveJournal (j) soc-LiveJournal

Figure 3.5: Effect of the number of workers on the average insertion time (AIT) and the average deletion time (ADT).

4

TOPIC RECOMMENDATION: EXPAND YOUR HORIZON

People today have access on the web to a massive amount of information, some more static like general knowledge, some more dynamic, like news and commentary about daily events. Thanks to the accessibility of the web, the growth in popularity of comment sections, reviews sites, and the birth of social networks, the users evolved from pure information consumers for the journals or the TVs to become also producers of information. The information available on the web today varies a lot in quality, correctness, and trustfulness, at least as much it varies in terms of sources. Everybody contributes to it, from governments and reputable institutions with their official and certified documents and data, to people providing their reviews, through bad actors providing fake news [35] and volunteers and professionals fact-checking and democratizing complex topics. The web has become part of everybody's life, and for many, it serves as their primary source of information. While most news and data would be directly available from the sources, many users entrust third parties to provide more concise or simplified interpretations and analyses of the originals. The providers can vary from online journals or corporations to other users. Different authors will produce different opinion pieces [56] influenced by their own opinions and by the expectations of their target audience. Generally, people tend to prefer opinion pieces that reinforce their pre-existing beliefs [99]. They thus get more polarized as they see only a part of the reality and feel support from other users in the same position. While this phenomenon appears in many domains, it is most well documented in forums and social networks [30, 31, 128]. Nevertheless, studies showed it is possible to actively fight polarization and misinformation by providing the user with a comprehensive set of viewpoints [55, 82, 83, 105, 143].

Now more than ever, there is a need for novel systems to provide holistic overviews of discussion topics. Moreover, it may be worth mentioning that while this is a *need* for the news sector, it is also an important future for reviewing services, and for the related-articles recommendation problem in general.

We are interested in providing a solution that, given a query article, recommends a set of diverse and related articles. The recommended articles shall thus cover, depending on the text corpora at hand, the different perspectives or the different opinions and sentiments for the subject of the query document and its closely related topics.

We are thus proposing two new approaches. The first, *Orthogonal-topics*, focuses on the relationship of the topics, and it has been designed to generalize well on all datasets. The second, *Sentimented-topics*, focuses on the sentiment expressed by the documents on the different topics, and it has been designed to extract and exploit as much information as possible from text corpora that contain opinionated articles.

To best model our intuition of a set of *diverse* articles, we devised our own diversity-metric, MIN-BW. It models a set of documents as a system of particles with repulsive forces; then, the most diverse set is the one whose system requires less work to balance, *i.e.*, to make it statically stable. We then show how MIN-BW differs from existing diversity-metrics in literature: maximize minimal pairwise points distance (MAX-MIN) and maximize average pairwise points distance (MAX-AVG). After demonstrating how we cannot reuse or adapt existing algorithms, we present FDLS, a fast and scalable time-bounded approximated optimization algorithm to find the maximally diverse set accordingly to MIN-BW.

To validate both our solutions, we perform a user study where we ask the users to compare the quality of the recommendations generated by our two approaches (orthogonal-topics + MIN-BW and sentimented-topics + MIN-BW) against the state-of-the-art recommender from Abbar *et al.* [1] on real-world data. The results showed that both our approaches outperform in all aspects, on average, the solution from Abbar *et al.* [1], especially in diversity and usefulness. They also showed that in the 75% of the instances, the diversity score was 75% or better, demonstrating the power of our MIN-BW model.

Finally, to validate the scalability and precision of FDLS, we perform an extensive set of tests on synthetic data, tracking the running time and diversity score accordingly to MIN-BW. We thus compare them with these from the state-of-the-art approximation algorithms for MAX-MIN and MAX-AVG proposed by Ravi *et al.* [116]. Results show that FDLS scales effectively linearly with the number of points, always terminating under 175ms even for 4096 points, whereas MAX-AVG and MAX-MIN running times explode with just a few points. They require twice the time of FDLS for 256 points, they take more than two seconds for 1024 points, and none of their tests instances with 4096 point could terminate in less than 20s. Moreover, they showed that the quality of FDLS approximations is generally comparable or even better, like in the cases with fewer points, and that our algorithm scales best. The analyses of the precision of the approximations showed little to no difference between the algorithms across both the number of points and the number of dimensions. Our solution performed best on examples with fewer points, but the values still re-enter within standard deviation for all configurations. Our

FDLS algorithm is thus a good approximating for MIN-BW and scales far beyond what the competition can do.

CONTRIBUTIONS. More specifically, our contributions are the following:

- We propose two novel approaches, orthogonal-topics and sentimented-topics, for providing holistic overviews of discussion topics.
- We propose a novel diversity-metric, MIN-BW.
- We provide an efficient approximated algorithm, FDLS, to select the most diverse set accordingly to MIN-BW.
- We experimentally demonstrate the quality of our approaches with a user study on real data.
- We demonstrate the superior scalability of our algorithm with exhaustive tests on synthetic datasets.

The remaining of the chapter is organized as follows. Section 4.1 provides the background needed to define and evaluate our solutions. Section 4.2 present the closest approaches from the literature. Section 4.3 define the recommendation problem, and Section 4.4 introduces our two solutions including orthogonal-topics, sentimented-topics, and the diversity-measure MIN-BW. Section 4.5 describes our evaluation methodology and reports the results. Finally, we conclude with our remarks in Section 4.6.

4.1 BACKGROUND

Section 4.1.1 describes Latent Dirichlet Allocation (LDA), a frequently used technique [62, 72, 80, 144, 146, 152] for text corpora and topic modeling that is also been employed in this work.

The problem of selecting a diverse and representative subset of items has been studied in many different domains [38, 40, 118, 126] in the past. Section 4.1.2 describes the models and solutions exploited in this work and for its evaluation.

4.1.1 *Latent Dirichlet Allocation*

Given a text corpora, Latent Dirichlet Allocation (LDA) [22], generates a probabilistic model which describes the documents in terms of topic distribution probabilities and the topics themselves as word topic distribution probabilities.

Definition 3 (Topics)**.** *The topics produced by LDA are characterized by a distribution over words. In other terms, they are vectors of real numbers, between 0 and 1, where each value corresponds to the probability associated*

with any distinct word in the corpus. Their representation can be lossly compressed by retaining only the most relevant words, i. e.*transforming the vectors in list of tuples* `(word id, prob)`.

Documents themselves are then characterized by a distribution over the topics. In a similar manner to topics&words, documents are vectors of probabilities over topics and can be lossly compressed in the same manner. From now on, we will use the term "documents' topics" to refer *only* to the subset of topics that survive the compression/selection procedure as described above.

4.1.2 *Diversification*

The definition of *diverse results* varies in the literature. Since, in this work, we focus on recommending articles based purely on their text-body, we are only interested in content-based definitions. They interpret diversity as an instance of the discrete p-dispersion problem [40]. As such, the text corpora must be modeled as space with a point for each article and a distance function that measures their diversity, *i. e.*, a *diversity-metric*. The literature provides several methods [1, 49, 80] to perform this mapping. In this work, we use LDA, as discussed in Section 4.4, to construct a geometric space with a metric that also satisfies the triangle inequality.

The discrete p-dispersion problem, also known as k-facility dispersion problem, consists in selecting a subset of p points for which either their minimum pairwise distance, MAX-MIN, or their average distance, MAX-AVG, is maximal. While the problem is known to be NP-Hard, for both definitions, Ravi *et al.* provided the two state-of-the-art approximation algorithms for $\mathbb{R}^d$, with proved optimal performance guaranteed (equal to 2) under the assumption of $P \neq NP$ [116]. Hereafter we present the two problem definitions with the respective approximation algorithms.

Definition 4 (MAX-MIN)**.** *Given a set* $V = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_k\}$*, of* n *points with* $\boldsymbol{p}_k \in \mathbb{R}^d$*, a positive integer* k *smaller that* n*, and a distance function* $d(\boldsymbol{p}_a, \boldsymbol{p}_b) : \mathbb{R}^+$*, find a subset* $P \subseteq V$ *with* $|P| = k$ *for which the minimal pairwise distance among the points is maximal.*

$$\underset{P \in \binom{V}{k}}{\operatorname{argmax}} \min_{a,b \in P} d(a, b) \tag{4.1}$$

where $\binom{V}{k}$ *represents, by abuse of notation, the set of all possible subsets of size* k *of* V, i. e., *all possible combinations of* k *elements from* V.

The greedy algorithm (Algorithm 8) requires $k - 1$ iterations to incrementally build the P set. For the first iteration, it adds the two points with maximal distance. Then, for each subsequent iteration, it

adds the single point from $S \setminus P$ which minimal distance from any point in P is maximal. The complexity thus is

$$O\left(|V|^2/2 + (k-2)\cdot(k-3)\cdot|V|\right),$$

where the first term maps to the first iteration and the second to all subsequent ones.

Algorithm 8: Greedy MAX-MIN

```
Data: V, k, d(),
Result: P
// Find most distant pair of points.
d_min, a, b ← ∞, 0, 1
for i ∈ [0, |V − 1|] do
    for j ∈ [i + 1, |V|] do
        if d(V_i, V_j) < dmin then
            dmin, a, b ← d(V_i, V_j), V_i, V_j
P ← {a, b}
// Greedy search reamaing k − 2
for _ ∈ [0, k − 2] do
    d_max, d_p ← 0, ∅
    for p ∈ V \ P do
        // Min distance from anything in P
        d_min ← ∞
        for p ∈ P do
            if d(p, p) < d_min then
                d_min ← d(p, p)
        // p has maximal min-dist "from" P
        if d_min > d_max then
            d_max, d_p ← d_min, p
    P ← P ∪ d_p
return P
```

Definition 5 (MAX-AVG). *Given a set $V = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_k\}$, of n points with $\boldsymbol{p}_k \in \mathbb{R}^d$, a positive integer k smaller that n, and a distance function $d(\boldsymbol{p}_a, \boldsymbol{p}_b) : \mathbb{R}^+$, find a subset $P \in V$ with $|P| = k$ for which the average pairwise distance among the points is maximal.*

$$\underset{P \in \binom{V}{k}}{\operatorname{argmax}} \ \underset{a,b \in P}{\operatorname{avg}}\ d(a,b). \tag{4.2}$$

The greedy algorithm for MAX-AVG (Algorithm 9) is closely related to that for MAX-MIN, albeit that in the main loop, it searches for the point that has the maximal average distance from the points already in P, instead of the minimal.

Algorithm 9: Greedy MAX-AVG

Data: V, k, $d()$,
Result: P
// ... Preamble same as Algorithm 8 ...
// Greedy search reamaing $k-2$
for $\_ \in [0, k-2]$ **do**
  $d_{max}, d_p \leftarrow 0, \emptyset$
  for $\boldsymbol{p} \in V \setminus P$ **do**
    // Avg distance "from" P
    $d_{sum} \leftarrow 0$
    for $p \in P$ **do**
      $d_{sum} \leftarrow d_{sum} + d(\boldsymbol{p}, p)$
    // $\boldsymbol{p}$ has maximal avg-dist "from" P
    if $d_{sum} > d_{max}$ **then**
      $d_{max}, d_p \leftarrow d_{sum}, \boldsymbol{p}$
  $P \leftarrow P \cup d_p$
return P

From now on, since the subject will be clear from the context, we will use the same terms MAX-MIN and MAX-AVG to refer to both the optimization problems and to the greedy algorithms.

4.1.3 *Unit-Hyper-Sphere*

In many solutions, like in the ones we propose in this work, the relevance or relatedness score is a function of the two documents' distance, `score(q, doc) -> f(||doc - q||)`. A relevance/relatedness filter reduces the entire space to the interior of a sphere. Moreover, whenever the maximal minimum distance in an optimal dispersion solution is greater than the radius of the hyper-sphere, all points lay on the surface of the sphere itself. This condition presents itself for small values of k and dense spaces.

Furthermore, in our solution, like in some others, we will also limit the minimum distance to avoid almost-identical documents, effectively limiting the search space to a hyper-corona. This filtering strategy makes even more likely the arising of the condition just described since the distance shall now be just greater than the difference in radii of the hyper-corona.

How should then k points be distributed on a unit (hyper) sphere $\mathbb{S}^{d-1}$ embedded in the Euclidean space $\mathbb{R}^d$, such that they are optimally spaced?
There currently exist two different answers to the question. The first answer yields the Tammes [132] problem, the second the Thomson [137]

problem. They were originally formulated for $S^2 \subset \mathbb{R}^3$, but they can be easily stated for higher dimensions. In the Tammes problem, the minimum distance between pair of points is maximized. This is a special case of the MAX-MIN optimization problem where the points are constrained on the surface of the unit hyper-sphere. In the Thomson problem, the energy of a central field of forces with origin at the center of the sphere is minimized. It has been formulated originally for the electrostatic field (electrons around the kernel of an atom in the plum pudding model), but the specific constants can be here neglected.

The unit sphere $S^{d-1} \subset \mathbb{R}^d$ is defined as

$$S^{d-1} = \{\boldsymbol{x} \in \mathbb{R}^d \mid ||\boldsymbol{x}|| = 1\},$$

where $\|\cdot\|$ is the Euclidean norm of $\mathbb{R}^d$. The angle α between two unit vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is obtained from the standard scalar product via $\boldsymbol{x} \cdot \boldsymbol{y} = \cos(\alpha)$. The Euclidean distance can be written as

$$||\boldsymbol{x} - \boldsymbol{y}|| = \sqrt{2 - 2\boldsymbol{x} \cdot \boldsymbol{y}} = \sqrt{2 - 2\cos(\alpha)}$$

The geodetic distance between two points on the sphere (spherical distance) is equivalent to the angle α since the sphere has a radius of 1. It is related to the Euclidean distance by $||\boldsymbol{x} - \boldsymbol{y}|| = 2\sin(\alpha/2)$.

Definition 6 (Tammes)**.** *Given a positive integer* k*, find* k *points* $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k\}$ *with* $\boldsymbol{p}_j \in S^{d-1} \subset \mathbb{R}^d$ *such that they maximize their respective distance* s*:*

$$\begin{aligned} &\max \quad s \\ &\textit{subject to: } ||\boldsymbol{p}_i - \boldsymbol{p}_j|| \geqslant s, \qquad i \neq j, \\ &\boldsymbol{p}_i \in S^{d-1} \quad i, j = 1, \ldots, k. \end{aligned}$$

A second answer considers the optimal positioning as the minimum energy displacement of particles around a central point. The Thomson formulation considered the Coulomb law for electrostatic potential. The particles were electrons, and the central point was the kernel of the atom. Here, an abstract central force is considered: its potential is characterized by the reciprocal interaction of two particles that satisfies the classic relation

$$U(\boldsymbol{p}_i, \boldsymbol{p}_j) = \frac{1}{||\boldsymbol{p}_i - \boldsymbol{p}_j||}.$$

The global potential energy of all particles is then defined as the sum of all pairwise interactions:

$$U = \sum_{i<j}^{n} \frac{1}{||\boldsymbol{p}_i - \boldsymbol{p}_j||}. \tag{4.3}$$

Definition 7 (Thomson). *Given a positive integer* k*, find* k *points* $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k\}$ *with* $\boldsymbol{p}_j \in S^{d-1} \subset \mathbb{R}^d$ *such that they minimize their respective potential energy:*

$$\min U = \sum_{i<j}^{k} \frac{1}{\|\boldsymbol{p}_i - \boldsymbol{p}_j\|}$$
$$\text{subject to: } \boldsymbol{p}_i \in S^{d-1}, \quad i, j = 1, \ldots, n,$$

where U *is the total potential defined in* (4.3).

Classic and intuitive solutions to both problems, in the planar case they are the regular n-polygons; in the case $S^2 \subset \mathbb{R}^3$, common solutions are the triangle for $n = 3$ and the tetrahedron for $n = 4$.

However, *the solution of the two problems is not always the same.* The first counterexample for $d = 3$ is given for $n = 10$, where the solution of the Tammes problem is $d_{Ta} = 1.091426290$ and the associated solid, which does not have a specific name and is composed by different polygons, has an irregular distribution of points and normalized potential energy of $E_{Ta} = 32.796048964$. On the other hand, the Thomson solution has a minimum distance between points of $d_{Th} = 1.074534852$ and potential energy of $E_{Th} = 32.716949460$. The associated solid is more regular and composed only by triangles, a gyroelongated square bipyramid, that is a square antiprism inserted between the basis of an octahedron. The differences in the solution are the regularity that brings a configuration of lower energy in the Thomson problem (99.75%) at the price of more close points (98.45%) with respect to the Tammes solution.

Examples of optimal solutions for $d = 3$ is given for $n = 10$:

Optimal solution accordingly to Tammes.

$$\begin{pmatrix} 0.000000000 & 0.000000000 & 1.000000000 \\ 0.914584731 & 0.000000000 & 0.404394325 \\ 0.101026934 & 0.908987782 & 0.404394325 \\ -0.841397575 & 0.358490376 & 0.404394325 \\ 0.263354009 & -0.875848101 & 0.404394325 \\ -0.744490792 & -0.666330505 & 0.041678755 \\ 0.654028153 & 0.585365077 & -0.479160623 \\ -0.426378376 & 0.729938829 & -0.534219792 \\ 0.678373208 & -0.504399648 & -0.534219792 \\ -0.329471593 & -0.294882053 & -0.896935363 \end{pmatrix}$$

Optimal solution accordingly to Thomson.

$$\begin{pmatrix}
0.26295766989233 & -0.77367763277684 & 0.57643402431295 \\
0.99173360691940 & 0.01088493241960 & -0.12785136351584 \\
0.18386542849509 & 0.97655761918020 & 0.11193176772221 \\
-0.73873395788922 & -0.65620949982719 & 0.15382207838239 \\
-0.79466061788263 & 0.58139371195556 & -0.17463062184202 \\
-0.52855026444359 & -0.23998820230990 & -0.81427285396768 \\
0.52855026833562 & 0.23998819628140 & 0.81427285321810 \\
0.29191085307296 & 0.45332951632285 & -0.84218786709897 \\
-0.54491050501616 & 0.19199505935279 & 0.81621715168646 \\
0.34783751836567 & -0.78427370036534 & -0.51373516886861
\end{pmatrix}$$

Despite the straightforward definitions, both problems are extremely difficult to solve; since they are non-convex and NP-hard. In fact, they are often used as benchmarks for testing clusters of computers and optimization software. In practice, only for a few cases and $d = 3$, the global optimum has been found for both problems. Moreover, the solution is not found via standard optimization techniques like gradient descent (because of the presence of many local optima) but with algebraic techniques based on graphs. These problems are further complicated by the fact that the solution of the case (d, k) does not help in any way for the solution of $(d, k+1)$, even if sometimes the configuration (d, k) is obtained from $(d, k+1)$ by removing a point.

4.2 RELATED

The problem of recommending a diverse set of topics or articles can be decomposed in finding a good model for textual data and selecting a *diverse* sub-set of elements.

The modelling of text corpora is generally performed by LDA [1, 22, 62, 72, 80, 144, 146, 152], which we already introduced in Section 4.1.1, or by Term frequency-inverse document frequency (TF-IDF) [63, 131], or by a mix of the two [72]

The spectra of existing approaches for recommending a diverse sub-set of elements is broad; it ranges from simple clustering algorithms to deep neural networks. Hereafter we report relevant examples of systems that can work with just the article bodies as input.

Drosou and Pitoura proposed Disc Diversity [41, 42] for result diversification. Their approach takes as input a set V and a distance metric. It looks for a subset P of V such that: 1) all the items in V lay within distance r from at least one of the element in P; 2) all items in P are at least at distance r between them. The set P is called a `r-Dissimilar-and-Covering diverse subset` (`r-DisC`). The first set found to match these criteria is returned as the final result.

Liu and Jagadish [88] investigated the performance of the Random Selection, Density biased sampling, k-medoids, and top-k sorting algorithms in solving the "Many-Answers" problem and identified the k-medoids as the best solution.

Abbar *et al.* [1] proposed a novel approach to recommending diverse and related articles. The documents are modeled as sets of relevant topics (these with high probability), and the Jaccard distance is used as the similarity metric. Related documents are thus clustered via Locality Sensitive Hashing (LSH). The algorithm takes as input a query document and a value k. As a first step, it finds the LSH bucket that the document would map to. Then it identifies the subset of k documents from such bucket that maximize the minimum inter-distance (MAX-MIN). The approach uses the greedy algorithm from Ravi *et al.* [116], as described in Section 4.1.2, to solve the MAX-MIN optimization problem.

4.3 PROBLEM STATEMENT

Given a query article, q, and a positive integer k, recommend k articles related but not quasi-identical to q, while maximizing their diversity.

The recommended articles shall thus cover, depending on the corpora at hand, the different perspectives or the different opinions and sentiments for the subject of the query document and its closely related topics.

4.4 SOLUTION

Herein we propose two novel approaches to the problem of recommending diverse related articles. The first, *Orthogonal-topics*, focuses on the relationship of the topics, and it has been designed to generalize well on all datasets. The second, *Sentimented-topics*, focuses on the sentiment expressed by the documents on the different topics, and it has been designed to extract and exploit as much information as possible from text corpora that contain opinionated articles.

4.4.1 *Approach 1: Orthogonal-topics*

Given a text corpora, first learn an LDA model (Section 4.1.1). Let us then consider a vector space $\mathbb{R}^t$, where t is the number of topics of the LDA model and where each document is represented by a vector whose values correspond to the documents' topics probabilities.

The query operation thus consists in:

1. Projecting the query-article, q, into this space.
2. Filtering for relevant articles, $r_{int} \leqslant ||doc - q|| \leqslant r_{ext}$. They cannot be too similar, so they have to be at a minimum distance of r_{int} and, at the same time, they must be related and thus close with a maximum distance of r_{ext}.

3. Selecting k articles as diverse as possible.

After carefully considering MAX-MIN/ Tammes, MAX-AVG, and Thomson, we decided to propose a new model Min Balancing Work (MIN-BW) described in Section 4.4.3.

Choosing the correct number of topics, t, for the LDA model is hard. The parameter, t, can either be provided by a domain expert or automatically learned by training multiple models varying t, α, and η, and looking for the combination that yields a model with the best coherence score.

Selecting meaningful values for r_{int} and r_{ext}, can be challenging for a user and can be easily lead to *empty answers* or very unrelated recommendations. We propose to infer their values from the distribution of the point themselves and a single optional parameter $r, r \in [0, inf)$ as follows. Let d_μ be average of the distance of each point from its closes neighbor. Let d_σ be standard deviation of the distance of each point from its closes neighbor.

$$\begin{aligned} u &= (d_\mu + 2 \cdot d_\sigma) \cdot k \\ r_{int} &= \max(0, r - 2) \cdot u \\ r_{ext} &= r \cdot u. \end{aligned}$$

This While this algorithm cannot guarantee to select boundaries encompassing at least k points, it usually does, and it does it fast since both u can be pre-computed at space creation time. Hover, whenever this remote scenario realizes itself, we relax the constraints to include the $2 \cdot k$ points with distance from q closer to the center of the corona, $(r_{int} + r_{ext})/2$.

4.4.2 *Approach 2: Sentimented-topics*

The *Sentimented-topics* approach focuses on recommending articles that expressed different feelings about the subject covered by the query articled, q. We thus introduce Sentimented-topics to model the sentiment expressed by a document on a particular topic.

Definition 8 (Sentimented-topics)**.** *Sentimented-topics are topics (Section 3) enhanced with a sentiment score, a real value between* -100 *and* 100*. Given a document, the sentiment score for each of its topics is defined as the average sentiment score of all the sentences it appears in. Sentimented-topics are thus represented as a list of triples* `(topic-id, perc, sent-score)`*.*

Given a text corpora, first learn an LDA model (Section 4.1.1). Let us then consider a vector space, the *word-space*, with w dimensions, where w is the number of the different words used to define the topics in the LDA model. Each topic is represented by a vector whose values correspond to the topics' word probabilities.

The query operation thus consists in:

1. Selecting the ρ most relevant topics for the document-query, q. To this end, project q into the *word-space* by computing the linear combination of its topics' vectors using as weights their probability values. Select then the ρ closest topics to q accordingly to the Euclidean distance.

2. Filtering for relevant articles. Let us consider a vector space, the sentimented-topics-*space*, and with ρ dimensions. The space contains only documents featuring at least one of the selected topics. Each document is represented by the sentiment score of its Sentimented-topics (missing topics default to a neutral score of 0). From this space, select documents that are not too close nor too far away from the origin, *i. e.*$r_{int} \leqslant \|\text{doc\_sent}\| \leqslant r_{ext}$.

3. Selecting k articles as diverse as possible. Again, we propose to use MIN-BW (Section 4.4.3).

This approach shares most of the challenges of Orthogonal-topics and its solutions, but it also introduces a new variable ρ. The selection of this parameter is far from trivial as it may seem at first glance since the distribution of documents over topics cannot be assumed to be uniform. Thus, depending on the projection of q, ρ must be dynamically derived. A ρ value chosen too small would produce empty (or too sparse) sentimented-topics-*space*, and, on the other side, one too large would produce spaces that are too large, thus diminishing the focus of the query and thus the relevance of the recommendations. We propose to start with a small value, *e. g.*, $\rho = 3$, and then increment it by multiplying it by 2 until the size of the generated sentimented-topics-*space* is at least $2 \cdot k$. Note that ρ can be derived with one single pass over the documents LDA data using only $\hat{}\left(\log_2 \lceil t/3 \rceil\right)$ memory, without the need actually to build the spaces. Just keep one counter and a selected-topics bitmap (assuming topics have been assigned a serial ID). Scan the data and increment the counter of each bucket for which at least one of the document's topics is present.

4.4.3 *Min Balancing Work*

Given a set, $V = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_k, \boldsymbol{p}_k\}$, of n points in an Euclidean space, and a positive integer k.

Let us model each possible subset of k points of V as a system of particles with unitary mass that repel each other with force proportional to the inverse square of their distance. Find the subset that required the minimal amount of additional work to balance it, *i. e.*, to make it statically stable.

Definition 9 (MIN-BW). *Let* P, $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_k\}$, *be a set of* k *points with* $\boldsymbol{p}_k \in V$, *and* $f_P(\boldsymbol{p})$ *the* net-force *acting on* p, $p \in P$, *as the sum of the repelling action exerted by the other particles in the system,* P.

$$f_P(\boldsymbol{p}) = \sum_{p' \in (P - \{p\})} \frac{p' - p}{\|p' - p\|^3}.$$

The additional balancing work required to make the system statically stable is directly proportional *to the sum of the module of the net-forces acting on each particle. The optimization problem can thus be formally defined as finding the configuration of particles P such that the sum total of the modulo of the net-forces acting on each particle in P is minimal,* i. e.*:*

$$\underset{P \in \binom{V}{k}}{\operatorname{argmin}} \sum_{\boldsymbol{p} \in P} |f_P(\boldsymbol{p})| \tag{4.4}$$

where $\binom{V}{k}$ *represents, by abuse of notation, the set of all possible subsets of size* k *of* V*,* i. e.*, all possible combinations of* k *elements from* V*.*

Differences with the existing formulations. As stated, we expect articles to be diverse and to cover different points of view, sentiments, or orthogonal topics from the query document, q.

The methods k-medoid, k-mean, and similar clustering approaches would follow the distribution density of the points; we instead desire an overview of all existing prospective. Nevertheless, neighborhood density can be provided to the user as a meta-information but shall not exclude the perspective itself.

General MAX-MIN, MAX-AVG, and alike DiscC [41, 42] do not consider query centrality; on the other hand, our approach reinforces this behavior by optimizing for the inverse square of the distance. Thomson method is the most similar problem, but, as required by its application, it works only with the module of the pairwise forces instead of relying on the net-force as MIN-BW does. As such, Thomson strongly prefers solutions that only use *outer* points, even if that means having a pair of them much closer than what they could have been if one of the inner points was to be chosen. This behavior can be observed in Figure 4.1 which shows the two optimal solutions for the points in Table 4.1. The solutions produced by the two approaches have similar global potential energy (PE), 5.1695 *vs* 5.1707, but very different MIN-BW score 9.3109 *vs* 7.0768.

	1	2	3	4	5
x	-0.6000	-0.2365	-0.8484	-0.4061	0.8749
y	-0.0400	-0.8692	0.3534	0.6755	0.0315

Table 4.1: Space used in Figure 4.1 to show Thomson *vs* MIN-BW different behavior.

Exiting algorithms and proofs cannot be reused. MAX-MIN (Tammes), MAX-AVG, and MIN-BW optimization problems may seem similar, but they produce different results.

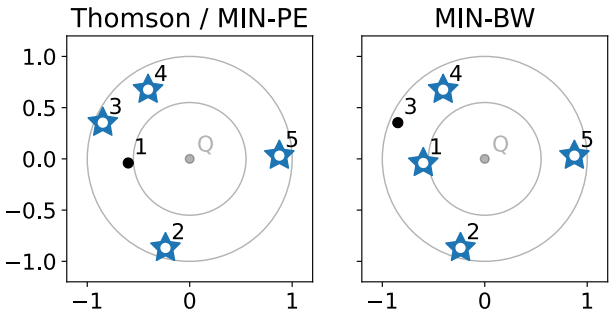


Figure 4.1: Thomson solution prefers *outer* points sacrificing distance and coverage. Selected points are indicated by ⋆.

Table 4.2 shows a space with 2 dimensions and 5 points $(\mathcal{X}, \mathcal{Y})$. The last three columns of the table mark whether the point belongs to the optimal solution for a given problem definition.

	1	2	3	4	5
MAX-MIN	✔		✔		✔
MAX-AVG	✔	✔			✔
MIN-BW	✔	✔		✔	
$\mathcal{X}$	0.3310	0.8034	0.4720	-0.9095	-0.9877
$\mathcal{Y}$	0.8962	-0.5951	-0.7533	-0.4014	-0.1373

Table 4.2: Example of a space for which each optimization problem yields a different solutions.

Figure 4.2 displays the space just described with a 2d chart.

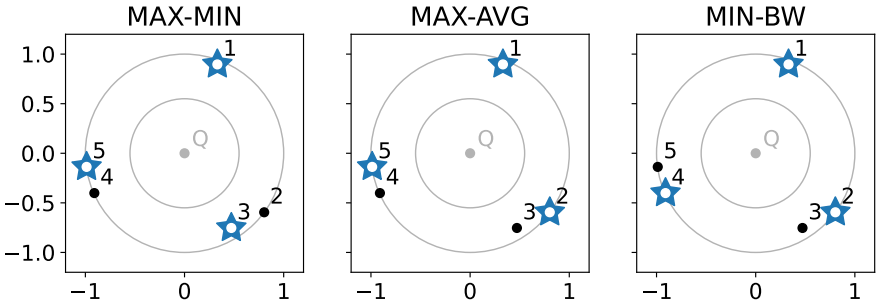


Figure 4.2: The three different optimization problems have different optimal results. Selected points are indicated by ⋆.

Table 4.3 shows the three optimal solutions statistics, highlighting how a solution, optimal for one method, is not necessarily optimal for the other.

We propose a new approximation algorithm: FDLS.

	MAX-MIN	AVG dist.	B. Work
MAX-MIN	**1.584354**	1.638435	1.938652
MAX-AVG	1.564332	**1.696150**	1.827654
MIN-BW	1.564332	1.694437	**1.826499**

Table 4.3: Statistics for the optimal solutions, in bold, computed accordingly to the three different optimization problems; they all differ. Recall, the first two maximize whereas the last minimizes.

4.4.4 *FDLS*

The idea behind our algorithm is to use the optimal-disposition (OD) for k points on a sphere of d dimensions and center q to aid the search for the real solution. Clearly, it must be an approximated algorithm since just a few of its steps would be too costly to be computed for a real-time application. These will then be appropriately approximated.

The intuition is to rotate the OD around q such that $\min d, k-1$ points of OD *maps* to real points. Mapping here is loosely defined as "coincide with" or "is close to". The selection of such points completely determines the ideal location of the remaining points. Thus, the best real solution that includes the selected points and those that best map to the remaining free ideal points. Since there is no guarantee that the best solution contains any of the points just selected, different rotations must be tested to find the absolute best one. Technically, for each fixed point, all rotations that map to another point must be considered.

This ideal algorithm would work well with a low dimensionality space, like $d{=}2$ or $d{=}3$, where optimal-dispositions are known or can be computed in a reasonable time; but does not scale well. First, we already know that even restricted k-dispersion problems such as Tammes and Thomson are NP-Hard, and even if there was an efficient (approximate) solution, we are not always in the position where d is known a-priori, *e. g.*sentimented-topics. Second, with greater values of d the number of rotations to be tested would approach the number of solutions, $O\left(\binom{|V|}{k-1}\right)$ vs $\hat{}$ $\left(\binom{|V|}{k}\right)$, making this approach inferior to a naïve one. This is even more true when considering the mapping cost, which is also doomed to increase for the hyper-sphere case.

While the deterministic solution above is not suitable for our application, the underlying idea is worth exploring.

FDLS

stands for Force Driven Local Search optimization algorihtm, and it is an iterative time-bounded approach based on the very same intuition that solves the scalability problem by carefully approximating each step (Algorithm 10).

The algorithm starts by computing an ideal point disposition. Instead of pre-computing optimal-dispositions, it generates several (default 300) uniform samples from a unit $(d-1)$-sphere using the method proposed by Harman *et al.* [58, 141], and selects the one that has minimal MIN-BW, the `BS` (Lines 5-9).

It will then rescale ($BS \cdot (r_{ext} + r_{int})/2$), recenter ($BS + q$), and map each of the points in `BS` to nearest points in the space, `P` (Line 10).

Finally, it tries to locally optimize `P` by repeatedly replacing the point for which the module of the sum of its net-force, p_m, and the inverse of the drift of the system barycenter (Line 14) from `q` (Line 15) is maximal. The point is thus swapped in the solution with one of its *neighbors*, p_n, preferring the one that is closest in direction to the net-force (Lines 19-21). The *neighborhood* of a node consists in the $(k+1)$—Nearest-Neighbor, since this is sufficient to guarantee that all points can be eventually reached. The map, M, of the `(k+1)-NN`s can either be pre-computed or derived at runtime. The score for each neighbor point is thus computed as the cosine similarity between the net-force and $p_m - p_n$.

The optimization step terminates when either the solution is stable (Lines 16-17), *i.e.*, the maximum net-force is less than a given threshold, or when all local perturbations have been tested, or when the sequence of optimizations leads to a never-ending loop (think 3 points on a circle of 5). This second termination condition is implemented by tracking the sequence of the latest swaps (Lines 12,24), by default 30, and then by verifying the existence of a repeating pattern in the history (Lines 25,26).

The whole procedure, from computing the ideal point disposition to locally optimizing the solution, is repeated until the time-budget, t_b, is consumed (Lines 1,27,31). The algorithm eventually returns the best solution across all the runs.

4.5 EVALUATION

To validate both our solutions, we performed a user study where we asked the users to compare the quality of the recommendations generated by our two approaches (orthogonal-topics + MIN-BW & sentimented-topics + MIN-BW) against the state-of-the-art recommender from Abbar *et al.* [1] on real-world data. In the study, the users are not aware of the origin of the recommendations.

To validate the scalability and precision of FDLS, we performed an extensive set of tests on synthetic data, tracking the running time and MIN-BW value. We thus compared them with these from the state-of-the-art approximation algorithms for MAX-MIN and MAX-AVG proposed by Ravi *et al.* [116].

Algorithm 10: FDLS

Data: V, k, q, M, t_b: `time budget.`
Result: P

$t_s \leftarrow$ `now()`
$w_{min}, w_p \leftarrow \infty, \emptyset$
repeat
  $S_{min}, S_b \leftarrow \infty, \emptyset$
  for $\_ \in 300$ **do**
    $S \leftarrow$ select k points uniformly from S^{d-1}
    $S_w \leftarrow \sum_{\boldsymbol{p} \in S} |f_P(\boldsymbol{p})|$; `// Track best disp.`
    if $S_w < S_{min}$ **then**
      $S_{min}, S_b \leftarrow S_w, S$
  $P \leftarrow$ map S_b to points from V
  $X \leftarrow \emptyset$; `// Clean "cannot replace" set.`
  $H \leftarrow []$; `// Clean History.`
  repeat
    `// Next to replace?` $\boldsymbol{p}$ `with max net-force.`
    $f_q \leftarrow q - \sum_{\boldsymbol{p} \in P} \boldsymbol{p}$; `// q pull`
    $\boldsymbol{p}_m \leftarrow \operatorname{argmax}_{\boldsymbol{p} \in P \setminus X} f_P(\boldsymbol{p}) + f_q$
    if $f_P(\boldsymbol{p}_m) < \epsilon$ **then**
      break ; `// Stable solution!`
    `// Find neighbor to swap with.`
    $\boldsymbol{p}_n \leftarrow \operatorname{argmax}_{\boldsymbol{p} \in M[\boldsymbol{p}_m] \setminus P} \frac{\boldsymbol{p} \cdot \boldsymbol{p}_m}{\|\boldsymbol{p}\| \|\boldsymbol{p}_m\|}$
    if $\boldsymbol{p}_n = \emptyset$ **then**
      `//` $\nexists$ `valid neighbor.`
      $X \leftarrow X \cup \{\boldsymbol{p}_n\}$
      continue ; `// Next swap candidate.`
    $P \leftarrow P \cup \{\boldsymbol{p}_n\} \setminus \{\boldsymbol{p}_m\}$
    $X \leftarrow \emptyset$
    `// Prevents swap loops.`
    H`.append(`(vp_m, vp_n)`)`
    if *HistoryLoop(*H*)* **then**
      break ; `// Cannot improve.`
    `// Done if time is up or no node can be replaced.`
  until $(now() - t_s) \geqslant t_b \vee X = P$
  $w_{p'} \leftarrow \sum_{\boldsymbol{p} \in P} |f_P(\boldsymbol{p})|$; `// Best solution?`
  if $w_{p'} < w_{min}$ **then**
    $w_{min}, w_p \leftarrow w_{p'}, P$
until $(now() - t_s) \geqslant t_b$
return P

4.5.1 *Quality*

In the user study, we asked the users to evaluate the quality of the recommendations from our two approaches and the results produced by the solution proposed by Abbar *et al.* [1].

We used PetScan [100] to create a dataset[1] of Wikipedia pages that belong to one or more of the following categories: Client-server database management systems, Database management systems, Relational model, Database theory, Types of databases, Relational database management systems. Specifically, we used the MediaWiki APIs to fetch only the pure content, as plain-text, of both the full page and the abstract [98].

To streamline the user study experience, we decided to pre-compute a batch of examples. For this, we implemented both our approaches and Abbar's one in Python 3.9; the code is available, alongside the data, on the project page [95]. Abbar's original implementation used `opencalais.com` to extract the topics from the text, but the tool was not available to us, so we used the same LDA model from our approaches and transformed the vectors into sets by applying a threshold on the probability (0.01). Moreover, since we were not concerned with running time, we replaced LSH with a distance filter based on Jaccard. Note that this does not degrade the quality of the results. On the contrary, it should improve them as the filter precisely computes what LSH only approximates.

Then, we trained an LDA model on the *full pages* dataset following the procedure described in Section 4.4.1 to autonomously derive the number of topics, t, and the other parameters. With the LDA model ready, we executed the pre-processing steps pre-computing the spaces for orthogonal-topics, sentimented-topics, and Abbar's Jaccard-based model with their distance matrixes.

We randomly selected 30 pages from the dataset and used them to generate as many queries for five recommendations, $k = 5$. For orthogonal-topics, r_{int} and r_{ext} were personalized for each query in such a way that the corona contained 50 points and another 50 were contained inside the inner boundary. For Abbar's, we replicated what LSH would have done by setting r_{int} to 0 and r_{ext} such that 50 elements were included. For sentimented-topics, we provided no parameters and let the algorithm adapt by itself.

Finally, we created a website where we could send the user to evaluate the recommendations of the three approaches. At each visit and page refresh, the website would load a webpage with a different query.

The webpage (Figure 4.3) was structured in three sections. The first one showed the title, abstract, and link to the source for the query page. The second one was a table-form, and the third one listed the three sets of recommendations (one per approach) adjacent to one another. Each recommended page was presented in the same manner as the query page. The layout of the page, *i.e.*, the order of the three sets and the order within the sets themselves, changed at each visit. In this way, we make it impossible for a user to map a specific approach

1 https://petscan.wmflabs.org/?psid=21221976

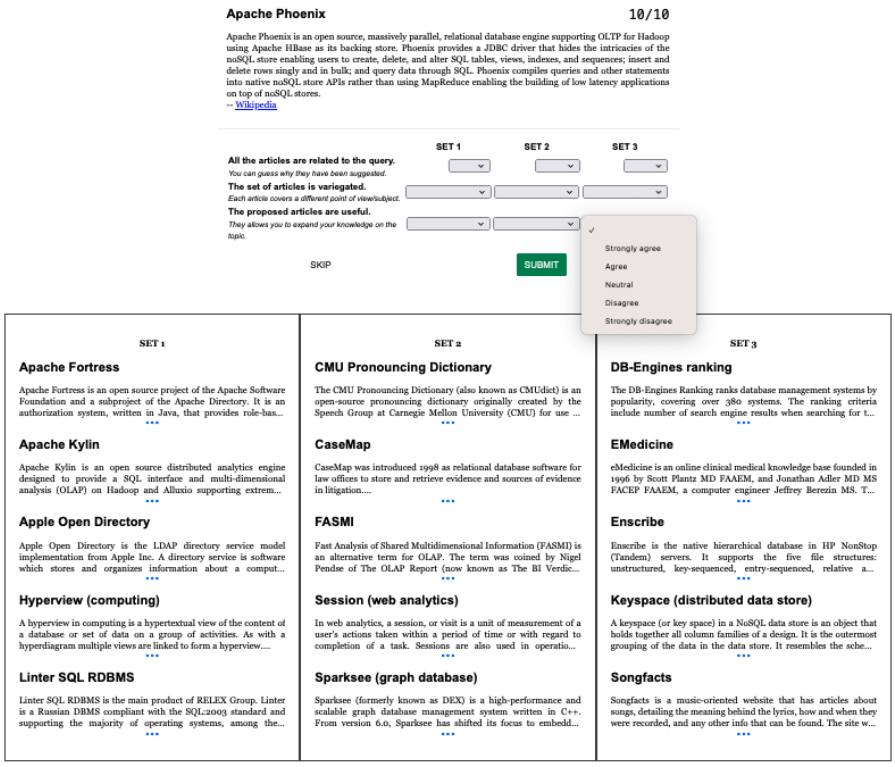


Figure 4.3: User-study interface.

to its results. Moreover, we used session-based tracking to ensure no user was presented the same query twice.

The table-form was used to evaluate how much *diverse*, *related*, and *useful* were the recommendations. In the form, we asked the users how much they agreed, for each set, on the following statements. To collect their answers, we used Likert-type scales [84], reported herein with the statements.

[Diverse] All the articles are related to the query.
You can guess why they have been suggested.
Scale: 100%, 75%, 50%, 25%, 0%.

[Related] The set of articles is variegated.
Each article covers a different point of view/subject.
Scale: Strongly agree, Agree, Neutral, Disagree, Strongly disagree.

[Useful] The proposed articles are useful.
They allow you to expand your knowledge on the topic.
Scale: Strongly agree, Agree, Neutral, Disagree, Strongly disagree.

We received a total of 109 submissions. To analyze the responses, we assigned to each possible answer an integer number, starting from 1 for the worst up to 5 for the best.

Table 4.4 shows the average score for all statements and for all approaches.

	Orthogonal-topics	Sentimented-topics	Abbar *et al.* [1]
Diverse	**3.9174**	**3.9357**	3.7798
Related	**2.8440**	2.4311	2.4220
Useful	**3.0458**	2.7064	2.5412

Table 4.4: Mean score for each approach.

The results showed that both our approaches outperform in all aspects, on average, the solution from Abbar *et al.* [1], especially in diversity & usefulness. They also show that in the 75% of the instances, the diversity score was 4 or better (75%, 100%), showing the power of our MIN-BW model – the other solutions use MAX-MIN and do so only in the 60% of cases. We can attribute the difference in usefulness and relevance between our two approaches to the double space modeling of Sentimented-topics. The technical pages we extracted from Wikipedia tend to be unopinionated.

4.5.2 *Scalability & Performance*

To validate the scalability and precision of FDLS, we performed an extensive set of tests on synthetic data, tracking the running time and MIN-BW value. We thus compared them with these from the state-of-the-art approximation algorithms for MAX-MIN and MAX-AVG proposed by Ravi *et al.* [116].

For this test, we implemented the three algorithms with Python 3.9 as a single process/thread solution alongside a space-generator. This last component takes in input two integers and one float, respectively the number of dimensions d, the number of points pts, and the inner boundary of the corona r_{int}. It generates a $d-$space with pts points randomly distributed in a corona centered in 0 with radii r_{int} and 1.

As discussed in Section 4.4.4, our algorithm is time-bounded. It is designed to generate good approximations as fast as possible and return the best one over multiple iterations. The quality of the result is thus expected to improve (statistically) with the number of iterations. We then decided to test two different configurations: 1) FDLS, uses the default configuration with a time-budget of 1s (as used in our Quality tests), 2) FDLS-0s, is configured to be as fast as possible by performing only a single iteration, *i.e.*, it has a time-budget of 0s. They are used together to verify: the minimal running-time of the algorithm, if the algorithm can reliably meet its deadline, and to increase in approximation quality with the variation of the time budget.

The whole experiment was comprised of 6480 tests where the MAX-AVG, MAX-MIN, FDLS, and FDLS-0s were executed with 5 different

values of k, $k = \{5, 10, 25, 50\}$, against 450 difference spaces generated from the following settings, 3 for each valid combination:

- Number of points, pts: $\{4^2, 4^3, 4^4, 4^5, 4^6\}$.
- Number of dimensions, d: $\{4, 5, 7, 9, 11, 13, 15, 32, 64, 128\}$.
- Corona's r_{int}: $\{0.5, 0.75, 1.0\}$.

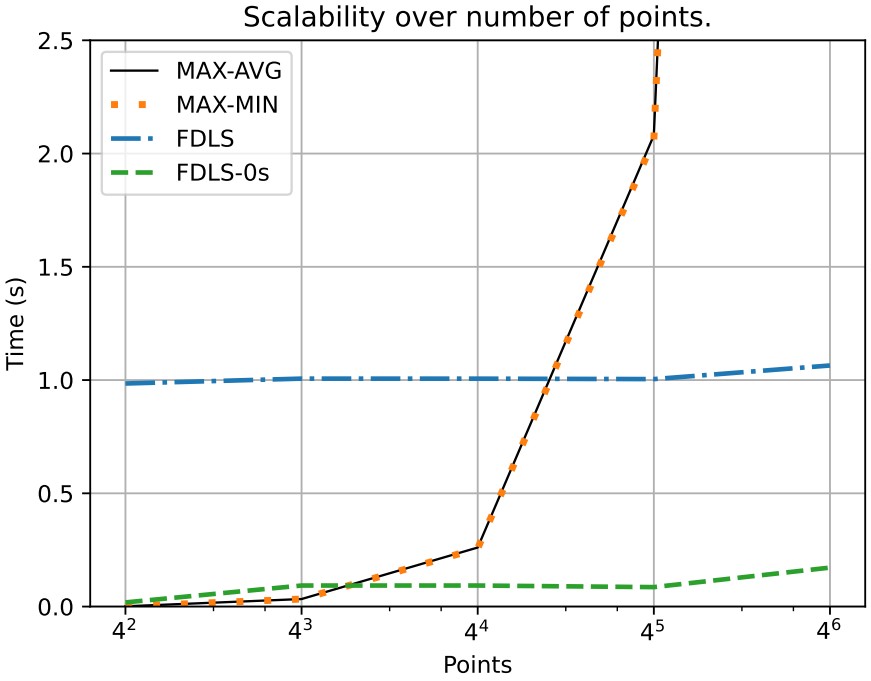


Figure 4.4: Average running-time, for different values of k and number of dimensions, over the number of points. FDLS targeted 1s, where FDLS-0s targeted 0s (*i. e.*, 1 iteration). MAX-MIN and MAX-AVG, exceeded the 20s timeout for all tests with 4^6 points.

SCALABILITY Figure 4.4 shows the results of the scalability test in terms of seconds over the number of points for MAX-AVG, MAX-MIN, and two versions of our algorithm averaged over the different values of k, r_{int}, and the number of dimensions. All tests were executed on a Linux machine running kernel version Debian-5.2.9 on an AMD Ryzen 5 2600 with 32GB of RAM, with a hard timeout of 20s. Reported timings do not include pre-computation of the distance-matrix and neighbor-map, for we are interested in the real-case scenario, and in such case, they would reasonably be pre-computed.

MAX-AVG and MAX-MIN running time are, as expected, identical. We can observe how they are the fastest up to 64 points, then their execution time starts diverging. Already with 256 points, they required twice the time of FDLS-0s, with 1024 they are far away from interactive performance requiring more than 2s, and by 4096 points,

none of their instances terminated under 20s independently from the number of dimensions and values of k. This behavior is not surprising, since the complexity analysis from Section 4.1.2 shows that $O\left(V^2/2 + (k-2)\cdot(k-3)\cdot|V|\right)$, where V is set of all the points in the space.

Interestingly, it appears that for very small values of $|V|$, the cost of FDLS-0s is dominated by the cost of the local optimization step, and its running time exceeds the one from MAX-AVG and MAX-MIN. Nevertheless, FDLS-0s, running time is *always* under 175ms.

FDLS-0s scales linearly, with a very shallow angle, with respect to the number of points. This is because the bootstrap step depends linearly on the number of points and k, *i. e.*, 300 iterations times the $O(k)$ cost for sampling step and the mapping step, which is $\hat{}(k\cdot|V|)$. While the local optimization step could be, in theory, combinatorial with respect to k and $|V|$, in practice, it behaves pretty well, requiring a small number of inner iterations over the neighbors-map, making it linear with respect to k.

We can observe how FDLS consistently approaches the target-running time of 1 second only exceedingly ever so slight in tests with most points by 64ms. This can be attributed to each iteration taking longer, as seen by FDLS-0s data, and the algorithm having fewer opportunities to check for the termination condition. This corner case can easily be managed by either employing, on platforms that support it, an external timer triggering an exception[2], or by tracking the average iteration time and modifying the check on Lines 27,31 to verify whether there is enough time for yet another one.

PERFORMANCE To evaluate the performance, we compared the quality of the results generated by the four approaches in terms of Balancing-work as defined in Section 4.4.3.

They showed that the quality of FDLS approximations are generally comparable (or even better in the cases with fewer points) and that our algorithm scales best. The analyses of the precision of the approximations showed little to no difference between the algorithms across both the number of points and the number of dimensions. Moreover, FDLS was, on average, slightly better than FDLS-0s, as expected, but the quality of the latter itself was already comparable or better than MAX-MIN and MAX-AVG.

The results demonstrated that our FDLS algorithm is thus a good approximating for MIN-BW and scales far beyond what the competition can do.

2 For example using https://docs.python.org/3/library/signal.html

4.6 CONCLUSIONS AND FUTURE WORK

In this work, we proposed two novel solutions, orthogonal-topics and sentimented-topics, in the form of recommender systems to the problem of suggesting related but diverse article sets.

We leveraged on the state-of-the-art text modeling technique, such as LDA, and sentiment analysis to map documents into hyper-spaces and thus treat the problem geometrically.

In orthogonal-topics (Section 4.4.1), the document's topics probability distributions are used values for vectors representing the documents in a space where each axis maps to a topic. The query then consists in finding k documents that are at the correct distance from the query document and that are as *diverse* as possible among them.

In sentimented-topics (Section 4.4.2), instead, we first identify the topics that are closely related to the document's topics. This is achieved by projecting a query document into the space of the topics by computing the linear combination of the documents' topics using the document's topic distribution probability values as weights, and then selecting the closest ones (topics are just distributions of probabilities over words). We then build a new space, with an axis for each of the selected topics, and represent the documents are by their topic-sentiment score (Section 8) for each axis. The query process then terminates, similarly to orthogonal-topics, by finding k documents that are at the correct distance from the query document and that are as *diverse* as possible among them.

To model *diversity* among the recommended documents, we proposed a new diversity-metric based on the modeling of their point representation as repulsive particles, and requiring the minimization of the work required to balance the system representing the solution (Section 4.4.3). We showed how this definition differs and compares to the existing literature, and how existing solutions for similar problems cannot be directly applied (Section 4.1.2).

We thus proposed a new approximation algorithm, FDLS, for solving the MIN-BW optimizing problem (Section 4.4.4).

We demonstrate the superior quality of our solutions through a user study that compared the end-to-end results produced by our two approaches (orthogonal-topics + MIN-BW & sentimented-topics + MIN-BW). against the state-of-the-art recommender from Abbar *et al.* [1] (Section 4.5.1).

Finally, we demonstrate the superior scalability and performance of FDLS, by comparing it to the state-of-the-art approximation for MAX-MIN and MAX-AVG proposed by Ravi *et al.* [116].

5

ON DESCRIBING THE CONTENTS OF A DATASET

Companies and organizations alike have started to realize the value of Big Data. They aggressively collect data from their daily operations with the intention of later analyzing it and turning it into valuable insights that can drive their business into offering new innovative services. Unfortunately, this collection of data is often coming without a clear plan in mind, and once the data has been collected, data owners remain wondering how to use the data and what kind of analysis to run on them. Before answering that question, it is fundamental to have a good understanding of the data that has been collected. This is typically done through data exploration. In data Exploration, the users start by looking at parts of the data and making their way to more specific or different parts until they find what is really of interest in it.

We claim that there is a need for a more systematic data exploration approach. To achieve this, a user must first obtain an overview of the contents of the data. We aim at providing such an overview through a set of descriptions for different parts of the data. In particular, we deal with the problem of generating an informative set of descriptions for the available data. For description, we adopt views, since they provide a formal and consistent way one can use to refer to parts of the data. Furthermore, given their formalism, they are easy to manage and reason about.

On the question of what description constitutes an informative set, we consider different but complementary factors. The first is to generate a set of descriptions that are about the whole dataset, meaning that no part of the data remains that is not covered by some description in the set, or if such part exists, it is as small as possible. The second factor has to do with redundancy. We would like to avoid descriptions of the same part of the data because that would be redundant, and if this is not possible, this should also be kept to a minimum.

There have been tools and techniques that have been proposed with the purpose of helping the user understand the data. Some are based on interactive visualization. By showing facets of the data allow the user to dig deeper into dimensions that may be of interest [70, 124]. Others summarize the data through some clustering or sampling techniques, providing certain quality guarantees [3]. Further approaches have focused on finding relationships and similarities among datasets in data-lakes [36]. Other techniques describe the result of a query by highlighting the differences to the rest of the data [147], or by de-

scribing specific structures in the data, like patterns, through some compact textual representation [66]. Despite the fact that all these works have the right aim, they do not take into consideration relationships across the descriptions they generate, neither consider how all these descriptions collectively relate to the whole dataset. With this in mind, we study the problem of generating an informative set of descriptions.

We see the task as an optimization problem. We take the different factors into consideration, prioritize them, and investigate combinations that give the best values to metrics that quantify these factors. We start by formally defining the notion of a data description and the intuition behind the concept of *goodness* for such a description (Section 5.2). We propose three solutions for selecting the best descriptions, namely, the Naïve, which considers all the different descriptions and evaluates them, the Vertical, which exploits a smart exploration strategy and heavy pruning, and the Adaptive, that builds on the Vertical and brings in an auto-tuning capability for its parameters (Section 5.3). We follow-up with an evaluation of their scalability and their performance on datasets with different characteristics (Section 5.4).

CONTRIBUTIONS. More specifically, our contributions are the following:

- We propose a new formal definition for a dataset description.
- We propose a new formal *goodness* metric for dataset descriptions.
- We provide three algorithms to generate dataset descriptions accordingly to such metric.
- We extensively test our algorithms on a multitude of real and synthetic datasets.

5.1 MOTIVATING EXAMPLE

Consider the case of a government analyst that was provided with a dataset with job opening records collected by various offices. A fraction of the dataset is illustrated in Figure 5.1. The dataset spans thousands of records, but for the purpose of illustration, we can assume for the moment that it contains only those present in the figure. The analyst does not have some specific task in mind but is only wondering if there are any interesting facts recorded in the data which would be worth investigating further. Furthermore, she would like to have an overview of the overall situation the dataset describes in order to present the findings to her superiors.

#	JpID	City	Employer	Job
0	6489	Seattle	Amazon	Systems Eng.
1	2598	Toronto	Amazon	Software Dev.
2	1561	San Francisco	Discord	iOS Dev.
3	2609	San Francisco	VMware	Software Dev.
4	9426	Newark	Amazon	Software Dev.
5	1828	Toronto	Amazon	Software Dev.
6	6027	San Jose	Adobe Systems	Software Dev.
7	8238	Seattle	Tableau	Software Dev.
8	7555	Seattle	Amazon	Data Eng.
9	3090	San Francisco	Clustrix	Software Dev.
10	8678	Seattle	Amazon	Data Eng.
11	7587	Palo Alto	Microsoft	Software Dev.
12	7325	Seattle	Google	Software Dev.
13	4096	San Francisco	Twitter	Android Dev.
14	7614	Seattle	Amazon	Data Eng.

Figure 5.1: A Job Openings dataset.

She can notice a number of interesting facts in the data, either by looking at it on a record-by-record basis or by posing specific queries. For instance, she can notice that there are four job openings in San Francisco, three openings at Amazon as Data Engineer, or that Microsoft is looking for a Software Developer in Paolo Alto. Some additional facts can be found in the list provided in Figure 5.2.

The analyst is wondering which of these statements of facts constitute useful information to include in the report describing the records of the dataset. Considering only the statement that there are openings in San Francisco (Statement 1), it is useful but is only about a small portion of the data. The statement about the positions for software developers (Statement 5) may be a stronger statement since it is supported by more records in the dataset. Yet, this is not the only information provided. There are additional facts supported by other records. Thus, selecting more statements instead of only one would provide a better overview of the dataset. For instance, considering statements 2, 3, and 5 is preferable to considering only 2, since they provide information about a larger portion of the records dataset. Under this reasoning, the set of statements $\{2, 3, 5, 6, 7\}$ will be an even better choice since all records in the table are described by at least one statement (we can informally say that there is at least one statement that *"covers"* the record).

The set of statements $\{1, 5, 8\}$ has a similar property to the one of $\{2, 3, 5, 6, 7\}$, *i. e.*, they cover every tuple in the dataset. Yet, between the two, the former seems preferable since the same set of records in the dataset is described with a *smaller number of statements*. The former set of statements is preferable for one additional reason. Across all its statements, the *number of attributes* mentioned is *less*. Thus, the

1.	There are four openings in San Francisco.
2.	There is a job post with ID 4096.
3.	There are three openings at Amazon as Data Engineer.
4.	Amazon has four open positions in Seattle.
5.	There are nine positions available for Software Developers.
6.	There is only one position for Systems Engineers.
7.	There is only one position for iOS Developers.
8.	Amazon has seven open positions.
9.	Microsoft is looking for a Software Developer in Paolo Alto.
10.	There is only one position in Newark.

Figure 5.2: Statements about the Job Openings dataset records.

analyst can better grasp the information they provide. Overall, these two examples indicate that the more compact the description is, the more desirable.

Another set of statements in which again all the records of the dataset are covered is $\{1, 4, 5\}$. Compared to the one with $\{1, 5, 8\}$, it can be noted that they both have, apart from the same "coverage" of the dataset records, the same number of statements, and also they involve the same number of attributes. There is, however, one additional argument that makes the former set preferable. Notice that among the records covered by statement 5, there is record #3, which is also included in statement 1, since it has "San Francisco" for the city. This phenomenon may be seen as some form of redundancy. If the statements are seen as groups of records, it is preferable to have groups with minimum overlap, *i.e.*, avoiding having records "covered" by more than one statement. Statements set $\{1, 4, 5\}$ describes only two (#3 and #4) multiple times, whereas for $\{1, 5, 8\}$ they are five (#1, #3. #4, #5, and #9). Under this observation, the analyst chooses the set of statements $\{1, 4, 5\}$.

5.2 PROBLEM STATEMENT

Let $\mathcal{A}$ be an infinite set of *attribute names*, and $\mathcal{V}$ be an infinite set of *atomic values*. A **tuple** is a finite sequence $\langle a_1{:}v_1, a_2{:}v_2, \ldots, a_k{:}v_k\rangle$ where $a{\in}\mathcal{A}$ and $v{\in}\mathcal{V}$. The sequence of attributes $\langle a_1, a_2, \ldots, a_k\rangle$, is the **schema** of the tuple. A **relation** is a finite set of tuples, all with the same schema. The schema of a relation is the schema of its tuples. A dataset is a set of relations. We assume next that our dataset consists of one relation only, but whatever we present applies naturally to many relations. We use views to refer to portions of a relation.

Definition 10 (View)**.** *A view is a conjunction of equality predicates of the form* $a{=}v$ *where* $a{\in}\mathcal{A}$ *and* $v{\in}\mathcal{V}$*. The* ***length of a view*** W*, denoted as* $|W|$ *is the number of predicates it contains. The view is* ***well-defined*** *over a relation* R *if and only if, for every predicate* $a{=}v$ *in the view, the attribute* a *is in the schema of* R*.*

A view is like a window on a relation. It selects some of its tuples, in particular, those that satisfy all the predicates in the view. The set of selected tuples is referred to as *coverage*.

Definition 11 (coverage)**.** *The coverage of a view* W *in a relation* R*, denoted as* $\overline{W}$ *is the set of tuples in* R *that satisfy all the predicates of the view,* i. e.*,* $\overline{W}=\{\langle a_1{:}v_1, a_2{:}v_2, \ldots, a_k{:}v_k\rangle \mid \langle a_1{:}v_1, a_2{:}v_2, \ldots, a_k{:}v_k\rangle \in R \wedge (\forall \langle a{=}v\rangle \in W, \exists i \in [1..k] \text{ such that } a{=}a_i \wedge v{=}v_i)\}$*.*

Intuitively, a view is a summarization of the tuples it covers. Since our goal is to provide a description for the content of a dataset, the idea is to use views for that purpose. Views will constitute the fundamental blocks of descriptions and correspond to what was referred to as statements in the previous sections. We consider a description of a dataset to be a collection of views.

Definition 12 (description)**.** *A description is a set of views. The length of a description* D*, denoted as* $|D|$*, is the number of views it contains.*

Different views can be defined on a given relation R. A challenging task is deciding which subset of these views form a good description of the dataset. One of the factors that can be used for this is the portion of the dataset that the description is about. The larger that portion, the better the description. To quantify this, the notion of coverage can be extended to the level of the dataset.

Definition 13 (coverage)**.** *The coverage of a description* D *on a relation* R*, denoted by abuse of notation as* $\overline{D}$*, is the union of the coverages of the views in* D*,* i. e.*,*

$$\overline{D} = \bigcup_{W \in D} \overline{W}.$$

It is always possible to find a description, the coverage of which contains all the records in a relation. This is done, for instance, by considering a description with all the possible views that can be defined on the relation. In situations like this, there will be records in the relation that belong to the coverage of more than one view. Such descriptions are not so desired since the same piece of information (*i. e.*, the tuple) is described in many different ways (*i. e.*, the views). To quantify this, we introduce the notion of *overlap*.

Definition 14 (overlap)**.** *The overlap of a description* D *on a relation* R*, denoted as* $\widetilde{D}$*, is the times above 1 a tuple is covered by a view in the description,* i. e.*,*

$$TC_t^D = \{W | W \in D \wedge t \in \overline{W}\}$$
$$\widetilde{D} = \sum_{t \in R} \max(\, |TC_t^D| - 1, 0).$$

to denote the set of views in D *that cover the tuple* t*.*

Last but not least, it is desired for the description to be compact representations of the tuples in their coverage so that they can be better understood. This compactness is quantified by the intricacy.

Definition 15 (intricacy)**.** *The intricacy of a view, W, denoted as, $\widehat{W}$, is the number of equality conditions it contains. The intricacy of a description,* D*, is the sum of the intricacies of the views it contains,* i. e.*,*

$$\widehat{D} = \sum_{W \in D} \widehat{W}.$$

Given a relation R, let $\mathcal{W}_R$ denote the set of views that can be defined over it. There are $2^{|\mathcal{W}_R|}$ possible descriptions using $\mathcal{W}_R$, *i. e.*, those in its powerset. The goal is to select the one that best describes the data in the relation. For this, the first condition would be that all the tuples are covered. At the same time, we would like a description as compact as possible, meaning that the length of the description should be minimum. Let D^{CL} be such description, *i. e.*,

$$D^{CL} = \underset{D \in \mathcal{D} \,\wedge\, \overline{D} = R}{\operatorname{argmin}} \; (|D|)$$

There may be more than one description with this property. In that case, the one preferable is the one that achieves the minimum overlap. Let that be denoted as D^{CLO}, *i. e.*,

$$D^{CLO} = \underset{D \in D^{CL}}{\operatorname{argmin}} (\widetilde{D}).$$

Last but not least, there can still be more than one description with the above properties. Since we need to create descriptions that are as compact as possible and easier to understand by the user, we select the description that is less intricate. Those descriptions, denoted as D^{CLOI}, are the

$$D^{CLOI} = \underset{D \in D^{CLO}}{\operatorname{argmin}} (\widehat{D}).$$

5.3 IDENTIFYING THE BEST DESCRIPTION

To identify the best description, we consider three different solutions. The first is an exhaustive space search. The second is a multi-step approach that exploits smart exploration and heavy pruning, and the third incorporates parameter auto-tuning to achieve the best performance.

5.3.1 *Naïve approach*

The straightforward approach, used as the baseline, first generates the set of all possible descriptions and then compares them to identify the

one that best satisfies the set criteria. This approach, although time-consuming, is guaranteed to find the best solution. The algorithm that implements this idea is shown in Algorithm 11.

As a first step, it constructs the domain of each attribute, *i. e.*, the set of values that the attribute's column contains (Line 1). Then, list of all possible views the relation is constructed by concatenating (`flatMap`) the result of computing the cartesian product (`product`) of all possible attribute combinations, *i. e.*, by computing the powerset of their domains (Line 2). By construction, such views will not have the same attribute repeated two times. Then, the set of all valid descriptions for the relation is derived from the views list by enumerating all its possible subsets (Line 3).

The process to find find the best description starts by selecting as *best*, D_b, one of them (Line 4), and then it visits all the others in order (Line 5). In the visit, the description is compared against the one currently selected in terms of *Coverage, Overlap, and Intricacy* (in this order), the best of the two becomes the new selected one (Lines 9-15).

We assume that the members of the powerset are monotonically increasing in size — as most implementations guarantee. This assumption allows us to skip the check on the *length* and to introduce an early termination condition (Line 6): if the currently selected solution covers the whole relation and the next candidate is longer, then none of the remaining candidates can be better in terms of length, so the iteration over the remaining descriptions can be interrupted.

Algorithm 11: Naïve algorithm

Data: R: relation
Result: D_b: description for R.

```
1  dom ← map(attr → set(attr), R^T)
2  views ← flatMap(attrs → product(...attrs), Powerset(dom))
3  descs ← Powerset(views)
4  D_b ← descs[0]
5  for D ∈ descs[1 :] do
6      if D̄_b = R ∧ |D| > |D_b| then
7          break ;                                  // Early exit
8      if D̄ ≠ D̄_b then
9          D_b ← D̄ > D̄_b : D, D_b
10     else if D̃ ≠ D̃_b then
11         D_b ← D̃ < D̃_b : D, D_b
12     else
13         D_b ← D̂ < D̂_b : D, D_b
14 return D_b
```

This naïve implementation stresses the computational complexity of the problem that is exponential. Let the number of well-defined views, n_W, be

$$\begin{aligned} n_W &= -1 + \prod_{a \in A} |\{v, v \in R_a\}| + 1 \\ &= O\left((|R|+1)^{|A|}\right). \end{aligned}$$

The plus 1 accounts for the "do not filter by this attribute" case, and the -1 for the "do not filter by anything" one. The number of all possible descriptions is 2^{n_W}. Given C_{cmp} be the constant cost for the comparison of any two solution, by the mean of the `if-elseif` chain, the computational complexity of the solution is $O\left(C_{cmp} \times 2^{n_W}\right)$.

DOMAIN PRUNING The performance can be improved by excluding some extreme cases that provide no useful insights. One such example is the case in which an attribute has the same value in all the records or has a different value in each different record. Excluding the predicates from such cases will significantly reduce the number of possible views and, consequently, the number of descriptions. To achieve this, we consider the entropy of an attribute and the support P_S of an attribute value. Given an entropy threshold H_ϵ, we can prune all the attributes with entropy less than H_ϵ or more than $1-H_\epsilon$. The former will exclude attributes that in almost all the tuples the same value, while the second those that are almost like having distinct values in every tuple. Furthermore, given support threshold P_S, we can exclude predicates with support lower than that.

Thresholds set to 0 will induce the system to consider all views and descriptions. Thresholds set to any other values will reduce search space. In the latter case, however, the coverage we can achieve may not be the whole relation, and we use the term maximal coverage to indicate the maximum set of tuples that can be covered. The size of the maximal coverage, denoted as C_{max} can be computed as the union of the coverages of the views with only one predicate ($W{=}\langle a : v\rangle{=}\langle p\rangle$), *i. e.*,

$$C_{max} = \left|\cup_{p \in dom} \overline{p}\right|.$$

In the case of Algorithm 11, applying this form of pruning is achieved by modifying line 1. As an indication of the benefit that can bring, in the example of Figure 5.1, setting the values of H_ϵ and P_s to 0.01 and 2, respectively, allows only 6 predicates to pass the pruning, from the 35 that were originally (that were leading to 6720 views generating 10E2022 descriptions).

5.3.2 *Vertical approach*

A more systematic approach to the description discovery considers the different requirements in three incremental steps. *Step 0*, domain

pruning; same as in the case of the naïve approach. *Step 1*, the expansion phase; the algorithm starts with short descriptions and keeps expanding them with views until they reach the coverage required. By construction, the mined descriptions also have minimal length. *Step 2*, the refinement phase; the identified descriptions are refined by specializing their views (*i. e.*, adding predicates to them) to minimize the overlap. This phase terminates when a description has no overlap or no further refinements without sacrificing coverage are possible. This approach is entirely deterministic and always produces an optimal description. Given the same domain pruning configuration, it produces either the same or an equally good description as the naïve approach.

STEP 1: EXPANSION

In the expansion phase, the algorithm looks for descriptions with maximal coverage and minimal length, ignoring the overlap they may have.

Lemma 3. *Let $\mathcal{W}^n$ denote the set of views with n predicates,* i. e.*,*

$$\mathcal{W}^n = \{W | W \in \mathcal{W}_R \wedge |W| = n\}.$$

There cannot be a description covering the whole relation that is shorter (uses fewer views) then the shortest one among those covering the whole relation and composed only of views from $\mathcal{W}^1$.

Proof. Assume there exists a description, D_1, that covers the whole relation and is shorter than the shortest one among those composed only of $\mathcal{W}^1$ views. Let description D_2 be built as follows: for each view w in D_1 create a view that contains only one of the predicates of w. By construction, D_2 is built only of $\mathcal{W}^1$ views, has the same length as D_1, and its coverage is a superset of the coverage of D_1. Thus, D_1 cannot be not shorter than the shortest. □

Given the above lemma, to maximize coverage, the algorithm can restrict its focus to the descriptions that are composed only of views from $\mathcal{W}^1$. The process aims at maximizing the probability of finding a suitable description early and it also minimizes the overall number of candidates that have to be evaluated with a number of pruning techniques.

Let $\mathcal{D}^n$ be all the descriptions of length n. Let $\overleftarrow{\mathcal{W}}^1$ be the list of all $\mathcal{W}^1$ ordered decreasingly by coverage. $\mathcal{D}^0$ contains only one description, the description with no views. Then $\mathcal{D}^n$ is generated as follow, for each description, D, in $\mathcal{D}^{n-1}$, for each view, W, in $\overleftarrow{\mathcal{W}}^1$ that comes after all the views in D, create a new description as $D \cup \{W\}$. We will refer to the D part as the *root*, and to the W as the *expansion*.

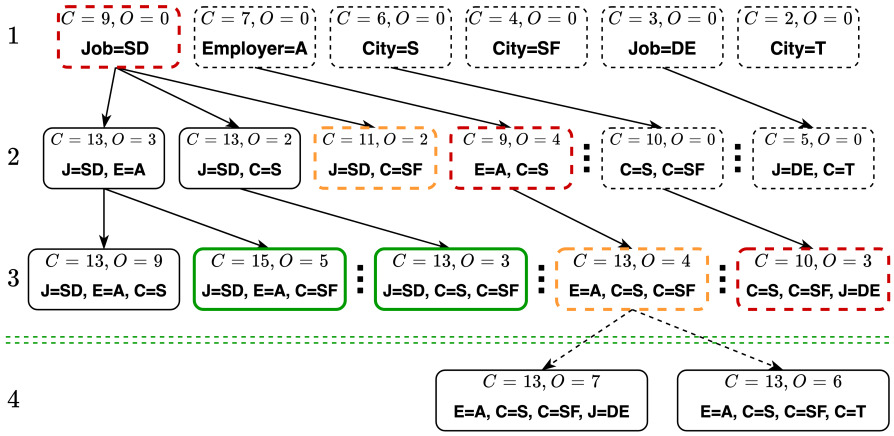


Figure 5.3: Expansion highlights for running example (Fig 5.1).

Figure 5.3, shows a subset of the generated descriptions in the order that they are explored (from left to right and from top to bottom). Nodes represent descriptions, and arrows represent the root-expanded relationship. Each node shows coverage (C), overlap (O), and its $\mathcal{W}^1$ views. The nth *level* contains descriptions from $\mathcal{D}^n$. By exploiting this ordering, several pruning techniques can be applied.

COVERAGE ESTIMATION BASED PRUNING. The size of the coverage of a description, D, can be estimated from the coverage of its *root*, $\overline{R(D)}$, and the coverage of the *expansion*, $\overline{E(D)}$. More precisely, the following interval can be identified $[\overline{D}_\perp, \overline{D}_\top]$. The lower bound corresponds to the case where the coverage of the root and the coverage of the expansion are one subset of the other; the upper bound corresponds to the case where the two sets are disjoint. The lower-bound is then defined as $\overline{D}_\perp = max(|\overline{R(D)}|, |\overline{E(D)}|)$, and the upper-bound as $\overline{D}_\top = |\overline{R(D)}| + |\overline{E(D)}|$. The coverage of the expansion is always known, as it is equal to the coverage of the $\mathcal{W}^1$ views, which is equal to the predicate support (number of tuples that contain that value for that attribute) of the single predicate in that view. The coverage of the root instead may not be. When this is the case, its upper bound is $\sum_{W \in R(D)} |\overline{W}|$. Moreover, a description, D, can be safely ignored whenever $\overline{D}_\top$ is less than the dataset's maximal coverage, C_{max}.

(1) **Root pruning.** When $\overline{D}_\top$ is less than C_{max}, not only can D be ignored, but so can all the remaining expansions for the *root* of D. For they have support smaller than the expansion used in D, thus neither they can reach the coverage threshold.

In Figure 5.3, the descriptions with $D_\cap$ being less than C_{max} have a dashed border and those for on which this pruning is applied have an orange border. Consider the description `((Job=Software Dev.), (City=San Francisco))`, The $|\overline{R(D_1)}|$ is 9 and the $|\overline{E(D_1)}|$ is 4. The successive expansions would replace `(City=San Francisco)`, with `(Job=Data Eng.)`

and (City=Toronto), which respectively have coverage of 3 and 2, *i. e.*cannot cover more than D_1.

(2) **Level pruning.** This pruning is based on the descriptions that are sub-sequences of $\overleftarrow{\mathcal{W}}^1$. If in such description, D_2, the sum of the coverages of the views is less than the required coverage, $\sum_{W \in D_2} |\overline{D_2}| < C_{max}$, then not only it can be safely ignored, but so can all the remaining description from $\mathcal{W}^{|D|}$, *i. e.*, in the same level.

Since $\overleftarrow{\mathcal{W}}^1$ is ordered decreasingly with respect to the coverage, the sum of the coverage of a sub-sequence description of length $(n+1)$, $[W_a, W_{a+n}]$, will be always greater or equal than $[W_{a+1}, W_{a+n+1}]$ as $\overline{W_a} \geqslant \overline{W_{a+n+1}}$. The same reasoning can be applied to all the descriptions that are absolutely rooted in W_a, *i. e.*, that has as most covering view W_a. To differ from the sub-sequence, they must have at least one different view, W_x, since W_a must belong to each of them, and views $W_b, \ldots, W_{a+n}$ belong to sub-sequence, x must be greater than $a+n$. Thus, W_x would have less or equal coverage of whichever view it is replacing. The new descriptions must then have less or equal coverage.

It is worth noting that this pruning function cannot be applied in a tree pruning manner, as it usually happens in a lattice. For, as coverage keeps increasing with each view addition, descriptions that were pruned may expand to some that shall actually be considered. As an example, ((Employ=Amazon), (City=Seattle)) is discarded by level pruning on level 2 – nodes with red border in Figure 5.3 – then, its first expansion is pruned on level 3 by root pruning, but it would expand, on level 4, into two descriptions with $\overline{D}_\top$ being 16 and 15, respectively.

EARLY TERMINATION. The exploration process can terminate earlier in two situations:

1) If the *Perfect Description* is found: A perfect description is a description that has the required coverage and no overlap. It can be returned immediately as a result of the whole computation. For, any other description that might fulfill the coverage threshold would have the same length. Moreover, if it had any overlap, it would require refining, and as such, its intricacy would be greater than the current one; if it did not, they would be equally optimal, and the algorithm returns only one description.

2) Once a description that satisfies the minimum coverage requirement is found, $|\overline{D}| \geqslant C_{max}$, only these remaining from the same length, $\mathcal{D}^{|D|}$, shall be considered. It is not possible to interrupt the exploration immediately as there might be another description that satisfies the constraint that may lead to a better solution after the refinement process, *i. e.*, less overlap. Thus all the description candidates, DC, of $\mathcal{D}^{|d|}$ – nodes with green border in Figure 5.3 – must be

identified and then provided as input to the refinement step (STEP 2).

EFFICIENT IMPLEMENTATION. Algorithm 12 illustrates an efficient implementation of STEP 1.

Algorithm 12: Expansion

Data: R, $\overleftarrow{\overline{w}^1}$, C_{max}.
Result: DC, O_{min}: $\widetilde{cbest}$, $cbest$: current best desc.

```
1  DC, O_min, cbest ← [], None, None
2  for l ∈ [1, |w̄¹||] do
3      del cache[l − 2]
4      if |DC| > 0 then
5          break ;                          // Early ret: Min length
6      lvl: for R ∈ combinations(|w̄¹|, l − 1) do
7          R̄_⊤ ← cache[l − 1][to_bitset(R)] or ∑_{W∈R} W̄
8          del cache[l − 1][to_bitset(R)]
9          for E ∈ [R[−1] + 1, |w̄¹|) do
10             D ← R|(E)
11             if R[0] + l − 1 = E ∧ ∑_{W∈D} |W̄| < C_max then
12                 break lvl ;                  // Level pruning
13             if R̄_⊤ + Ē < C_max then
14                 cache[l][to_bitset(D)] ← R̄_⊤ + Ē
15                 break ;                      // Root pruning
               // Evaluate description.
16             D̄, D̃, R̄_⊤ ← query(R, D)
17             cache[l][to_bitset(D)] ← D̄
18             if D̄ < C_max then
19                 continue
20             if D̃ = 0 then
21                 return (), 0, D ;            // Early ret: Perfect
               // Save candidate for refinement.
22             DC ← DC|(D)
23             if ¬O_min ∨ D̃ < O_min then
24                 cbest, O_min ← D, D̃
25 return DC, O_min, best
```

[*Description enumeration*] It is possible to implement a generator that yields description accordingly to the aforementioned search strategy in constant time and $O(|\overleftarrow{\overline{w}^1}|)$ main memory. Moreover, many library implementations of the `combinatorial` function, like `combinations`

from the *itertools* python package[1], already behaves in this manner and can thus be exploited.

To speed up the pruning functions, the generator uses the indexes of the views in $\overleftarrow{\mathcal{W}}^1$ instead of the views themselves or pointers. In the pseudo-code, the notation for coverage and overlap is abused since the mapping of a list of indexes to a list of items is trivial. Furthermore, the generator must be decomposed and integrated into the algorithm to allow finer control over the pruning points. As such, the enumeration is implemented with three nested loops, one that selects $\mathcal{W}^n$ (Line 2), one that enumerates all the roots in the class (Line 6), one that generates all the expansions for a given root (Line 8).

[*Pruning*] The first pruning function to be applied is the *level pruning* function. The "is sub-sequence of $\overleftarrow{\mathcal{W}}^1$" precondition is verified by checking the difference between the index of the first view of the root and that of the expansions, which should be equal to the description length minus one – array and lists indexes start at 0. When the precondition is met, the maximal coverage upper bound is derived from the coverage of the views, which was already computed for the domain pruning step (Line 11). The two inner loops are thus interrupted if the C_{max} threshold is not met (Line 12). The *root pruning* function follows next. The $\overline{D}_\top$ is estimated by summing $\overline{R}_\top$ and $\overline{E}$, If it is smaller than C_{max}, the inner loop is interrupted (Lines 13,15).

[*Caching*] Computing $\overline{R}_\top$ may not be necessary, since $\overline{R}$ may have been already computed when exploring the previous $\mathcal{W}$ class as a description instance, if it was not pruned. Thus, every time a coverage is estimated (Line 14) or computed (Line 17), the value is cached. To avoid exhausting the main memory, the Record TTL (time to live) and the record key encoding must be carefully designed. Cached records can be safely deleted in two spots: 1) After $\overline{R}_\top$ has been estimated (Line 8). For, roots are evaluated only once. 2) When the exploration moves to $\mathcal{D}^n$, then the description cached from $\mathcal{D}^{n-2}$ will not be used anymore, as the estimation relays only on $\mathcal{D}^{n-1}$ (Line 3). The cache shall then be organized hierarchically, like by using HashMaps inside a HashMap, *i. e.*`{length: {desc: coverage}}`.

In order to have a consistent and compact record key, the description can be mapped to `bitsets`. It is sufficient to flip the bits corresponding to the index of the views that belongs to the description.

[*Query evaluation & Early Termination*] . The evaluation of a description consists, at its minimum, in computing coverage and overlap. The function proposed in the pseudo-code also tracks the coverage of the root itself; this value is then used to update the root coverage upper bound estimation, $\overline{R}_\top$.

Depending on the backing store (*e. g.*, bi-dimensional in-memory array, a relational database, or columnar store), computing the actual

1 https://docs.python.org/3/library/itertools.html

coverage and the amount of overlap can be much more costly than just verifying all tuples are covered. In this case, it is possible to optimize in the following manner. If it is the first expansion, compute all the values; otherwise, check if the description covers the whole relation; if it does, then and only then compute the overlap to know whether it is a perfect description or not.

If the description happens to be perfect, the exploration terminates immediately and returns the description as a result (Line 21). Otherwise, the description is added to the list of the description candidates, DC, for refinement (Line 22), and the statistics for the current best description are updated, if necessary (Lines 23,24).

Adding a description to DC also enables the guard for the early termination for the minimal length constraint (Line 4).

STEP 2: REFINEMENT

In this second step, the candidates are refined to reduce their amount of overlap, which has so far been ignored. To reduce the amount of overlap in a description, one or more views must be replaced by others that cover only a subset of the respective tuples. In other words, a view shall be replaced by another that features a superset of its predicates. This operation is called view *specialization*. This operation will cause an increase in *intricacy*. A smart strategy should then be employed to consider the possible refinements in order with respect to the increase in intricacy that they imply.

The proposed strategy: *a*) efficiently enumerate the possible refinements in such order. *b*) ensures that each refinement is considered at most one single time, even if they could be generated by different description candidates. For example, both the candidates identified in our running example, green border in Figure 5.3, would otherwise lead to consider multiple times refinement like `((Job=Software Dev.), (City=Seattle, Employer=Amazon), (City=San Francisco))` or `((Job=Software Dev.), (City=Seattle, Job=Data Eng.), (City=San Francisco))`. *c*) minimizes, by pruning, the number of views specializations to consider.

Given a description, generate a set of "specialization templates", such that each specializes only one view by adding one single predicate (Property *a*).

To avoid duplicates (Property *b*), track the index of the view that has been last specialized to generate the description, *i. e.*`(D, i`$_1$`)`; limit the generation of refinements only to the views with an index equal or greater than the last specialized view; if no view has been specialized, yet use 0. Without loss of generalization, consider the following description candidate template `{-: ((`$\mathcal{W}^1$`, `$\mathcal{W}^1$`, `$\mathcal{W}^1$`), -)}`, with no parent id (*specialization path*), intricacy 3, and never specialized before. There are 3 ways it can be refined to increase the intricacy progressively

{parent id: ((template), last specialized)}
{-: (($\mathcal{W}^2$, $\mathcal{W}^1$, $\mathcal{W}^1$), 0)}
{-: (($\mathcal{W}^1$, $\mathcal{W}^2$, $\mathcal{W}^1$), 1)}
{-: (($\mathcal{W}^1$, $\mathcal{W}^1$, $\mathcal{W}^2$), 2)}

These in turn, yield:

{-,0: (($\mathcal{W}^3$, $\mathcal{W}^1$, $\mathcal{W}^1$), 0)}
{-,0: (($\mathcal{W}^2$, $\mathcal{W}^2$, $\mathcal{W}^1$), 1)}
{-,0: (($\mathcal{W}^2$, $\mathcal{W}^1$, $\mathcal{W}^2$), 2)}
{-,1: (($\mathcal{W}^1$, $\mathcal{W}^3$, $\mathcal{W}^1$), 1)}
{-,1: (($\mathcal{W}^1$, $\mathcal{W}^2$, $\mathcal{W}^2$), 2)}
{-,2: (($\mathcal{W}^1$, $\mathcal{W}^1$, $\mathcal{W}^3$), 2)}

For each specialization template, generate a new refined description, specializing the prescribed view by adding a new predicate. Once again, to avoid considering the same option multiple times, utilize only predicates that are defined on attributes that have an id, position in the relation, greater than any of the ones in use in the view. In this way the refinement (`(Job=Software Dev.), (City=Seattle, Employer=Amazon), (City=San Francisco)`), which happens to be also the solution, is only generated once from the second candidate identified in Step 1 Figure 5.3, (`(Job=Software Dev.), (City=Seattle), (City=San Francisco)`).

The refinements can be evaluated as soon as they are generated. The evaluation can have three outcomes: 1) the coverage has fallen under the C_{max} threshold; this is not anymore a valid solution, and neither can be any of its specializations; it is discarded (Property *c*). 2) the amount of overlap is now zero, *i. e.*the description is now *perfect*; the execution terminates by returning this description as a result. 3) the overlap is still greater than zero; the statistics for the current best description are updated, if necessary, and the description is enqueued to DC to be further refined.

Algorithm 13 shows how the refinement process can be implemented efficiently. The algorithm uses DC as a queue and keeps iterating over it (Line 1). An inner loop is used to cycle over the id of the next view that must be refined, *i. e.*from the last modified i_{-1} to the last one in the description (Line 2). Another inner loop iterates over all the suitable predicates that can be added to the view under specialization.

The `extend_view` function is responsible for enumerating these predicates. Two possible implementations of this function are devised: the first uses the domain of the attributes to derive the predicates; the second instead queries the relation to see with which other predicates the tuples selected by this view appear. The first option is faster but potentially generates many views with no coverage, and thus invalid descriptions. The second one is more costly but generates only

non-empty views. Which one is better strongly depends on the domain cardinality/distribution and query time. In the evaluation process (Section 5.4) the former was used.

Then, a new description is created by merging the unmodified views with the specialized view, and it is thus evaluated (Lines 4-6). The checks for the coverage requirement and perfect solution follows immediately after. If none of the checks is triggered, then the description is enqueued in DC for further refinement, and the statistics for the current best description are updated, if necessary (Lines 7-13).

Algorithm 13: Refinement

Data: R, dom, DC: [(desc, o)], O_{min}, cbest, C_{max}.
Result: The best description for R.

```
1  for (D, i_{-1}) ∈ DC do
2      for i ∈ [i_{-1}, |D|) do
3          for p ∈ extend_view(D[i], dom) do
4              W_e ← d[i] + [p] ;                          // Extend view
5              D_e ← D[:i] + W_e + D[i + 1:] ;              // Extended D
6              D_e‾, D_e~ ← query(R, D_e, C_max)
7              if D_e‾ < C_max then
8                  continue ;                  // Not a solution anymore
9              if D_e~ = 0 then
10                 return D_e ;                          // Perfect solution
11             DC.append((D_e, i)) ;                     // Refining more
12             if o < O_min then
13                 cbest, O_min ← D_e, D_e~ ;              // Best so far
14 return cbest
```

5.3.3 *Adaptive*

The Adaptive algorithm is designed for scalability and intended for scenarios requiring minimal supervision. It builds on the Vertical. By dynamically adjusting the parameters for the domain pruning, H_ϵ and P_s, it adapts to the data at hand and to the user's desired maximal work effort, *max_effort*. The latter parameter allows for a trade-off between coverage and time. The results generated by this approach may differ from these yielded by the naïve and Vertical solutions.

Figure 5.4 provides an overview of how the algorithm works. It starts by pruning the dataset domain with some initial H_ϵ and P_s. By default, these are set respectively to 0 and to 2% of the number the tuples, but they can be override instance by instance if desired. In case the resulting domain is empty, *i.e.*no description could be mined, the

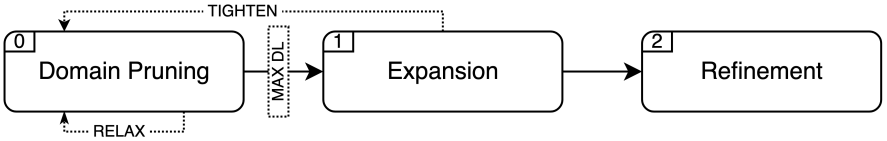


Figure 5.4: Adaptive approach overview.

value of the two parameters are reduced accordingly to the `relaxing` function, and the domain pruning step is re-executed.

Then, the algorithm estimates the maximal length, `max_dl`, of any description it may find within the boundaries of the requested `max_effor`.

Next follows the expansion step. If the expansion continues up to $\mathcal{W}^{\texttt{max\_dl}}$, it means that no description can be found with the current settings. Thus, the domain size is reduced, possibly causing a reduction in C_{max}, by increasing the domain pruning parameters accordingly to the `tightening` function and going back to step 0.

To avoid infinite loops the two functions must cooperate to properly restrict the parameters window in what is a search for a suitable configuration. Furthermore, in the unlikely case where no further parameter tuning is possible, and no description reaches the coverage threshold before `max_dl`, the threshold is lowered to the maximal coverage seen in $\mathcal{W}^{\texttt{max\_dl}}$. In other words, the descriptions with maximal coverage in the class will be considered description candidates and provided as input to Step 2, the refinement.

The refinement step is also modified to consider refinements with intricacy only up to `max_int`; which, by default, is set equal to two times `max_dl`, but also this one is user-customizable.

RELAXING & TIGHTENING The two functions act primarily on P_s. The value is modified as $P_s \mp P_s * av * 2^s$, where av is the Adaptive Value (by default: 0.25) and s the number of subsequent times the same operation has been invoked. To ensure that the search window always reduces, each time the two functions alternate `(T, R, T)` or `(R, T, R)`, the Adaptive Value is halved.

Each function has a corner case when invoked on $P_s = 0$. Relaxing will halve H_ϵ, while Tightening will set P_s to the default value.

The algorithm stops making adjustments once the result of halving produces a delta smaller than a user-provided *sensitivity threshold*. Furthermore, to limit the effects of extreme tuple distributions, a `max_retries` constraint is introduced (by default: 5). It limits how many times the tightening function can be invoked.

MAXIMAL DESCRIPTION LENGTH The maximal description length is computed by exploiting two estimators: the *maximal coverage probability* and the *level mining cost*.

The *maximal coverage probability* for description with length q, $mcov(q)$, is the probability for such a description of covering the whole re-

lation. The coverage probability of description can be computed as the union of the probabilities that each of its views has to cover the whole relation. Assuming that the probabilities of the different views are independent; the $mcov(q)$ corresponds to the probability of the description build with the first q views from $\overleftarrow{\mathcal{W}}^1$, $mcov(q) = \bigcup_{i\in[0,q)} P(\overleftarrow{\mathcal{W}}^1{}_i)$. The coverage probability of a single view, W, corresponds to its relative coverage, $P(W) = |\overline{W}|/|R|$. By applying De Morgan's law, $mcov(q)$ can be efficiently computed as $1 - \prod_{i=0}^{q} 1 - P(\overleftarrow{\mathcal{W}}^1{}_i)$.

The *level mining cost* can be derived by the number of solutions that will be evaluated and the size of the relation. While it is not possible to quantify exactly how many views will be evaluated, due to the pruning functions, it is possible to compute a lower and upper bound. The lower bound correspond to the number of roots that will not be pruned by level pruning, and the upper bound by the number of views these expand into.

The number of evaluated roots can be computed as the total number of roots minus the pruned one. To find the first pruned description, it is sufficient to identify the first instance, ι, of a sliding window of size q on $\overleftarrow{\mathcal{W}}^1$ for which the sum of the coverage of its views is less than C_{max}. With that, the bounds can be computed as follows:

$$roots = \binom{|\overleftarrow{\mathcal{W}}^1|}{q-1} - \binom{|\overleftarrow{\mathcal{W}}^1|-\iota}{q-1} \qquad views = \binom{|\overleftarrow{\mathcal{W}}^1|}{q} - \binom{|\overleftarrow{\mathcal{W}}^1|-\iota}{q}$$

$$\text{with} \quad \iota = \operatorname{argmin}_{i\in[|\overleftarrow{\mathcal{W}}^1|-q]} \max\left(\sum_{j=0}^{q} \overleftarrow{\mathcal{W}}^1{}_j - C_{max},\ -1\right)$$

The maximal description length, max_dl is computed as follows. Try increasing the description length, q, starting from 2 up to the number of $\mathcal{W}^1$ views. Check whether the amount of work is reasonable for the expected coverage, *i. e.*if $mcov(q) > \texttt{sigmoid}(q/max\_effort - max\_effort)$. If it is not, invoke the *tightening* function and go back to the domain pruning (Step 0). Check whether the increase in coverage, Δ_{mcov}, is worth the extra work with $\Delta_{mcov} * |R| * (1 - \texttt{sigmoid}(q/max\_effort - max\_effort)) < log\_2(views(q))$, and if it is not then stop and $max\_dl = q-1$.

It is worth mentioning that the chosen method is domain agnostic. If required, this can be optimized for different domains and settings.

SOLUTION INTEGRATION Minimal changes are required to the vertical approach.

In the expansion step (Algorithm 12), the outer loop condition should be changed to iterate up to max_dl (Line 2) – if DC is empty, the *tightening* function will be invoked by the main loop.

The algorithm should take another input parameter, `last execution`, that indicates whether this is the last time the expansion step will be invoked (*i. e.*no further domain adjustments are possible). When this parameter is True, then the two pruning functions shall operate using

a local variable `partial max cover` that records the maximal coverage observed so far (Lines 11,13).

The same intuition applies for DC and `cbest`. The code can be moved from Lines 22-24 up to Line 17, The check shall be modified to first compare the coverage.

Finally, the function should be modified to return also the `partial max coverage`, such that the refinement process can use it in place of C_{max}

The only other modification to the refinement process (Algorithm 13) consists of breaking the outer loop once the intricacy reaches `max_int` through an if statement on Line 2.

TIME CONSTRAINED EXECUTION Thanks to the modifications mentioned above, it is also possible to execute the algorithm in Vertical and Adaptive mode in a time-constrained manner. It is sufficient to always execute the exploration step with the `last execution` parameter set, and whenever the time is up, read the content of the `cbest` variable.

5.4 EVALUATION

We performed an extensive set of tests to assess the effectiveness and scalability of the proposed approaches. We used several datasets that can be grouped as: **[R]** real-world data; **[V]** variegated synthetic data, aimed at studying the behavior of domain pruning and the three algorithms on different kinds of data (heterogeneity); **[S]** large scale synthetic datasets intended to evaluate the scalability of the techniques.

The *real-world* set **[R]** features 10 relational tables from open data repositories used also in similar works [115]. The characteristics of these tables are shown in Figure 5.5.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Cols	9	6	9	7	9	5	5	7	5	7
Rows	1078	921	9102	307	6705	9537	2410	859	813	1201
$\|\mathcal{W}^1\|$	960	6000	21670	640	10390	57410	10920	1940	2060	2470

Figure 5.5: **[R]** Real-World Datasets.

The synthetic datasets have been produced with an ad-hoc generator that we developed to support fine-tuning of the different data characteristics. The generator supports offers a number of parameters: the number of repeated value patterns (`npatterns`), the level of overlap among these patterns in terms of coverage (`max depth`), the percentage of overlap of patterns within the same tuple (`max overlap`), the number of `columns` and the number of `rows`. This data generator is made available online on our project webpage [109].

The first set of synthetic datasets, **[V]**, contains 2220 different tables created from 740 different configurations:

- `columns` from 3 to 11 - odd only;
- `max depth` from 1 to 5;
- `max overlap` from 0.2 to 1 with increments of 0.2;
- `npatterns` from 2 to 24 - even only.
- `rows` fixed to 100,

The scalability set, **[S]**, consists of 60 datasets that have been generated with the 20 most challenging characteristics and wth size up to 1.000.000 tuples:

- `columns` fixed to 6;
- `max depth` fixed to 5;
- `max overlap` fixed to 0.6;
- `npatterns` from 6 to 18 with increments of 4;
- `rows` 10^n for n from 2 to 6.

Figure 5.6 shows the distribution of $|\mathcal{W}^1|$ for all the synthetic datasets.

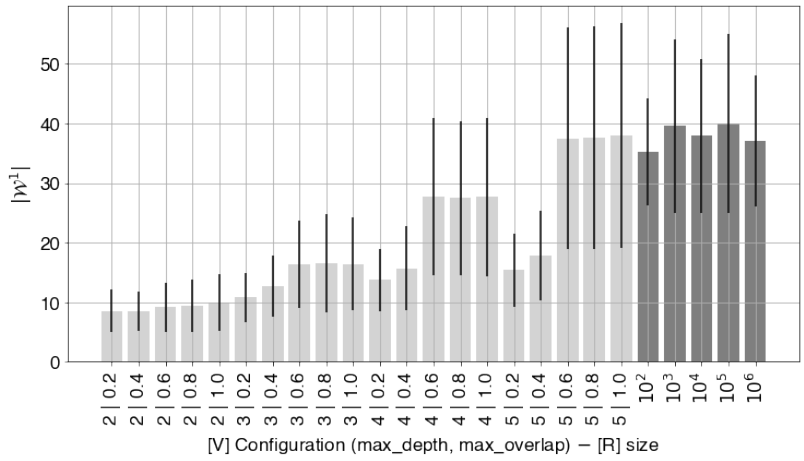


Figure 5.6: Number of $\mathcal{W}^1$ in synthetic datasets.

To perform the tests, we developed a single-threaded python prototype for the proposed approaches[2]. The tests were carried out with GNU Parallel [133] on a cluster of machines, each equipped with pair of Intel Xeon E5-2420 and DDR3 1333MT/s RAM running Linux 4.15.0-91 and Python 3.6.9. The tests on **[V]** have been constrained to a maximum running time of 15 minutes, with the GNU `timeout` utility, while these on **[S]** and **[R]** were instead limited to 45 minutes. Each test was assigned a core and 20GB of main memory using Linux control groups.

2 Source code & datasets at martinbrugnara.it/rp/describing_data.html.

5.4.1 *Results*

EFFECTS OF DOMAIN PRUNING Table 5.1 and Table 5.2 show the effects of P_s and H_ϵ on **[V]**. The figure shows the average of the remaining percentage of patterns after pruning with different coefficients. While the means provide a general overview, they alone do not describe the whole picture for the higher pruning values, as with these, some datasets are left with no predicates at all. The ones with a non-empty domain will then have many more views than expected. The table thus shows how many of these last datasets exist for each configuration. To avoid skewed results, the remaining of the evaluation relies only on experiments with a non-empty domain.

Furthermore, these statistics show the complexity of choosing proper pruning parameters and provide even more value to the Adaptive approach, which always provides a solution by autonomously tuning the parameters.

h_ϵ-p_s	0	0.02	0.12	0.22	0.32	0.42	0.52
0	20.11 ± 14.62	20.11 ± 14.62	5.39 ± 2.78	1.87 ± 1.78	0.85 ± 1.19	0.45 ± 0.83	1.00 ± 1.00
0.05	17.99 ± 14.61	17.99 ± 14.61	5.36 ± 2.80	1.84 ± 1.77	0.82 ± 1.17	0.42 ± 0.80	0.58 ± 0.76
0.15	10.97 ± 11.32	10.97 ± 11.32	4.14 ± 2.83	1.80 ± 1.75	0.81 ± 1.16	0.41 ± 0.79	0.04 ± 0.24
0.25	4.67 ± 6.20	4.67 ± 6.20	2.37 ± 2.50	1.26 ± 1.47	0.75 ± 1.10	0.38 ± 0.74	0.01 ± 0.10
0.35	1.62 ± 2.80	1.62 ± 2.80	1.06 ± 1.68	0.71 ± 1.13	0.47 ± 0.82	0.32 ± 0.67	0.00 ± 0.07
0.45	0.40 ± 1.14	0.40 ± 1.14	0.29 ± 0.79	0.22 ± 0.60	0.15 ± 0.42	0.10 ± 0.33	0.00 ± 0.06

Table 5.1: Domain Pruning on **[V]**: % of residual $|\mathcal{W}^1|$.

h_ϵ-p_s	0	0.02	0.12	0.22	0.32	0.42	0.52
0	2220	2220	2172	1505	953	623	5
0.05	2216	2216	2167	1490	934	595	5
0.15	2065	2065	2037	1484	925	582	4
0.25	1501	1501	1490	1251	898	557	4
0.35	892	892	892	854	681	503	4
0.45	323	323	323	320	284	195	3

Table 5.2: Domain Pruning on **[V]**: Dataset with a non-empty description.

INFLUENCE OF STRUCTURES & PATTERNS Figure 5.7 shows the number of failed experiments on **[V]** by reporting the percentage of the two failure mode, Timeout (TO) and Out Of Memory (OOM), for each combination of (`max depth` and `max overlap`). It shows that the Naïve approach suffered the most timeouts, that it has issues dealing with even the simplest datasets, and that sometimes it goes out of memory (OOM) on the more complex ones. On the contrary, the Vertical approach never failed on the memory constraint but experienced a limited amount of timeouts on the more challenging instances. Remarkably, the Adaptive approach never experienced OOM

and, thanks to its adaptive pruning strategy, never timed out either. The observed behavior is also consistent with the running time shown

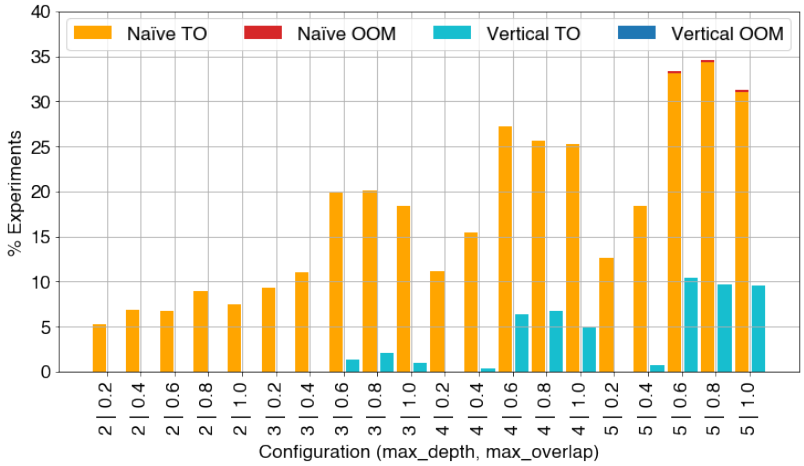


Figure 5.7: Execution failures % in **[V]**. *Adaptive never failed*.

in Figure 5.8. The means have been computed by accounting only the time equivalent to the timeout (15 minutes) for failed experiments, thus heavily favoring the slower solutions. The matching trend of the three series confirms that the combination of `max depth` and `max overlap` is a good indicator of the complexity of the dataset mining problem. The Vertical approach is generally faster by orders of magnitudes than the naïve, and the Adaptive is, in turn, again faster by orders of magnitudes. The Vertical approach appears to get slower faster with the growth of `max overlap` than the Adaptive. This behavior can be attributed to the reduction in pruning opportunities, especially for the level pruning function. The Adaptive approach suffers less as it can adjust `max_dl` and use its adaptive pruning strategy to get to a description faster.

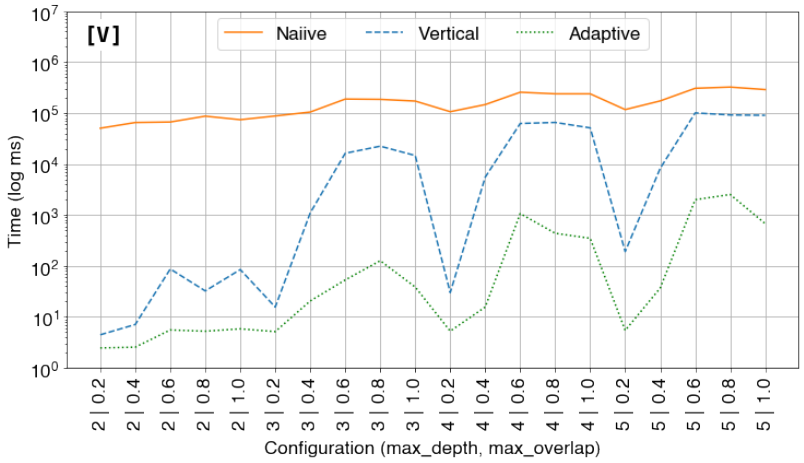


Figure 5.8: Mean running time of the 3 approaches on **[V]**.

SCALABILITY Figure 5.9 shows the mean running time on **[S]**. Timeouts and out-of-memory errors are counted as the timeout time: 45 minutes. It shows a direct logarithmic relationship between the size and the running time for the Vertical and the Adaptive solutions (the chart is log/log). Showing, as expected, that the complexity of the problem is not directly correlated with the size but instead with the number of predicates and their structures. However, the impact of the size on the scan time is not negligible. The naïve approach constantly times out with the two smaller P_s values.

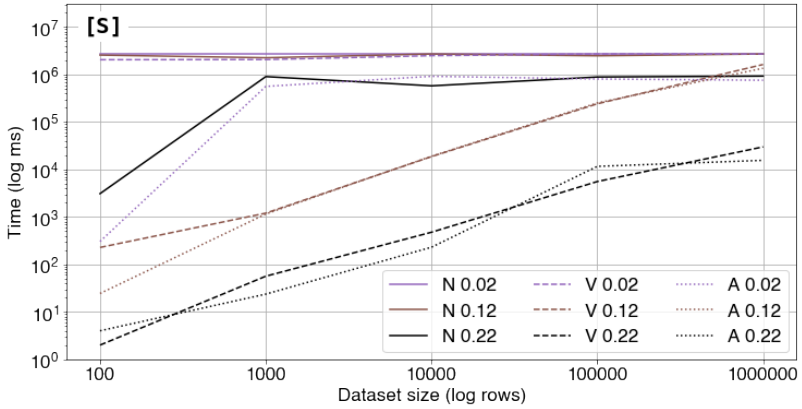


Figure 5.9: Mean running time of the 3 approaches on **[S]**.

REAL-WORLD DATASET Figure 5.10 shows the mean running time on real-world datasets **[R]**. Vertical always outperforms the Naïve by at least three orders of magnitude. This figure also highlights the flexibility of the Adaptive solutions; it is generally as fast or faster than Vertical, except for T4. This dataset is so simple and small that the effort needed to adapt is more than the time to mine the solution: the running time increases from ~3ms to ~9ms. A small increment compared to the reductions in the more complex tables like T2 where it traded 5% coverage for two orders of magnitudes in running time.

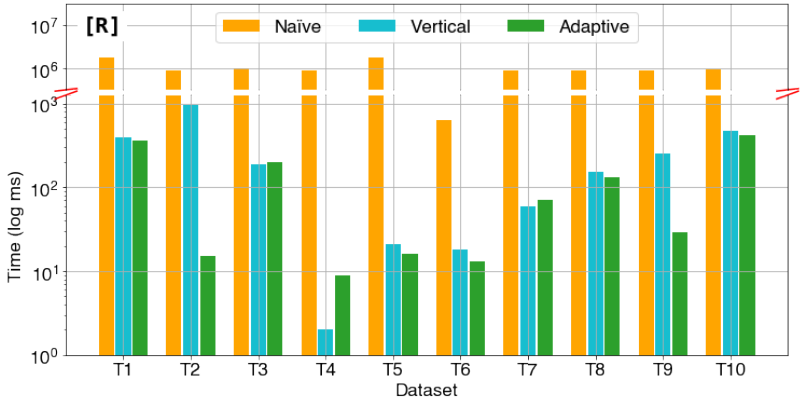


Figure 5.10: Mean running time of the 3 approaches on **[R]**.

ADAPTIVE EFFECTIVENESS The Adaptive approach trades coverage for running time. As discussed, all experiments run with the `max_effort` parameter set to 2.5. Figure 5.11 shows the ratio (in percentage) of the coverage of the Adaptive solution over the Vertical one for the same set of initial conditions. The grey series represents the result for the **[V]**; it shows that there never is a reduction in mean coverage for more than 20%, and that it increases, as expected, with the complexity of the datasets. The black series represents the result for **[S]** and shows a behavior similar to the former. Here, the smaller coverage values are to be expected since the tests were carried out only with the most challenging configurations. The brown series represent the results for **[R]**. Each data point corresponds to a dataset. The results can be divided into two groups, where the coverage exceeds 95% and where it does not. The first contains challenging datasets with many rows, columns, and interleaving patterns, while the second contains the simpler ones. The second group's existence suggests that the adaptive algorithm is effective also on real-world datasets and that the current cost model works best, as by design, with moderate/complex datasets. Nevertheless, if the application domain was to be rich in simple datasets, the model could be easily adapted, as discussed in Section 5.3.3.

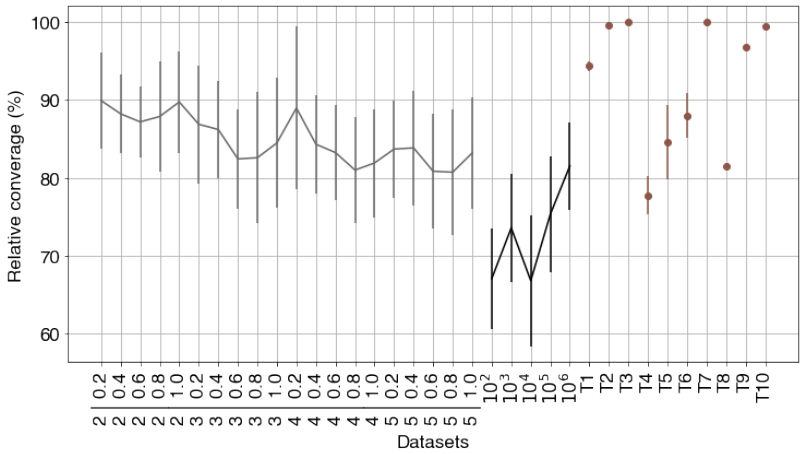


Figure 5.11: Adaptive coverage *vs* Vertical (`max_effort`=2.5). Left **[V]**, center **[S]**, right **[R]**

5.5 RELATED WORK

Describing datasets is a topic that has been widely studied in the past. Most of the approaches provide only a partial description of the data. Scorpion [147], for instance, is a system that, given a user-provided set of outliers from the results of an aggregation query, identifies the tuples that mainly influence the abnormal behavior. The system takes into consideration only formulas that are single con-

junctions of **in-range** and **in-set** predicates, each for one attribute. Some approaches specialize in describing what is mostly influencing a given metric. MRI [32], for example, is tailored for the AVG function. Other works are domain-specific, like PerfXplain [71] that focuses on MapReduce [34] job logs. Other approaches identify and describe a subset of tuples that highly differentiate from the others, under the assumption that those are what actually characterizes the dataset. For instance, in [66] a generalization of **frequent pattern** mining is used to find Fascicles with high support instead of patterns. Some works capture the possible hints that a dataset may contain with respect to some goal. A recent work like this [48] finds groups of tuples that all share the same value for a subset of attributes as well as the same value for the goal column. In user group analytics, researchers look for descriptions that describe the general population. In one of these works [108], a multi-objective optimization problem is defined across dimensions like coverage, set diversity, and optionally on the diameter (defined on the distribution of a single scalar column). Multidimensional patterns are used in OLAP cubes to both define and describe regions, in [75] similar dimensions are used as part of MDL/GMDL to decide how to slice the cube. Generalizing the frequent pattern approach, conditional functional dependencies (CDF) / pattern tableaux have been used to describe datasets and their semantics [51, 52]. Thanks to their expressive power, they have also been considered in organizing the data in relational databases [33] Descriptions often are the result of some clustering process. CLIQUE [3] finds dense clusters in subspaces of maximum dimensionality and then describes the clusters with DNF expression optimized to ease the comprehension for the user, and a similar work [145] uses also clustering to perform query result summarization. Another approach [73] proposes a clustering technique that allows the user to provide as input the desired in-cluster distribution. There also exists mixed approaches; for example, Paganelli *et al.* [111] proposed an interactive system to describe a dataset by the description of its partitions. The partitions can either be defined by the user, as (`GROUP BY attr1, attr2, ...`), or be automatically derived by a clustering algorithm such as the k-means. The system does not use partitions' definition clauses as labels; instead, it generates new boolean formulas, d-formuals, optimized for two user parameters: *coverage* and *diversity*. A d-formula consists of a set of conjunctions of *in-set* predicates. The coverage parameter expresses the preference for highlighting outliers over frequent patterns. The diversity parameter indicates the preferred number of different attributes to use. The final description then consists in the conjunction of one d-formula per partition; The formulas are selected for their diversity and the discrimination factor of their attributes. Data descriptions have also been studied in the semantic web community under the umbrella of "dataset description" [5, 24, 91]. In that context,

the description is *superimposed* information provided often by the user, describing specific meta-data properties and not the content in terms of values.

5.6 CONCLUSIONS

We have studied the problem of generating meaningful, concise, and informative descriptions for a dataset. We have modeled the descriptions as sets of views over the datasets, where the views are defined as filtering clauses. To judge the quality of a description, we considered four factors: the length, the coverage on the dataset, the overlap, and the intricacy. We have thus presented three algorithms that generate such descriptions given these four optimization objectives. We have then evaluated the algorithms by performing extensive experimental tests on synthetic and real-world datasets. Results show the effectiveness of our pruning strategies in Vertical and the increased scalability of Adaptive.

6

CONCLUSIONS

The world is producing data at an unprecedented rate measured petabytes per minute or in hundreds of zettabytes per year [59, 139]. The growth in size is only matched by its growth in complexity and variety. The share amount of data available makes it impossible for a single person to examine or know it all. There is thus a pressing need for: scalable and efficient storage and processing capabilities for the new ever more presents kinds of data, such as graphs; unbiased solutions to simplify access to the internet content, such as search engine and recommender systems, capable of minimizing the polarization effects of most existing solutions; techniques to expedite the exploration and understanding of all the datasets created from this data deluge to enable targeted processing and thus maximize the value of the data. This thesis contributed to *Understanding and Managing Complex Datasets* in the following ways.

In Chapter 2 we tackled the problem of graph data storage. We have performed an extensive experimental evaluation of the state-of-the-art graph databases in ways not tested before. We provided a principled and systematic evaluation methodology based on microbenchmarks. We have materialized it into an evaluation suite, designed with extensibility in mind and containing datasets, queries, and scripts. Our findings have illustrated the advantages microbenchmarks can offer to practitioners, developers, and researchers and how they can help them understand better design choices, performances, and functionalities of the graph database system. As a result, we have presented a number of findings that help understand the trade-offs between native and hybrid graph database systems, their effect on important graph queries like traversals and pattern matching, and their current capability to handle highly heterogeneous graphs.

In Chapter 3, we tackled the problem of processing existing large graphs that are already stored in a distributed way. We developed a technique that can exploit as much as possible the existing topology of the graph data and perform the k-core decomposition in a cooperative way among the distribution nodes. We have introduced an efficient distributed and streaming k-core decomposition approach for large and dynamic networks. Our approach deals with graph changes/updates by selecting only the nodes of a subgraph of the original graph that really need to update their core numbers. We implemented our approach on top of AKKA framework, a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications. By running experiments on a variety of both real

and synthetic datasets, we have shown that the proposed method is interesting in the case of very large graphs with a very satisfactory performance and scalability for large graphs.

In Chapter 4, we addressed the polarization problem in content recommendation systems. We proposed two novel solutions, orthogonal-topics and sentimented-topics, in the form of recommender systems to the problem of suggesting related but diverse article sets. *Orthogonal-topics* focuses on the relationship of the topics, and it has been designed to generalize well on all datasets. *Sentimented-topics*, focuses on the sentiment expressed by the documents on the different topics, and it has been designed to extract and exploit as much information as possible from text corpora that contain opinionated articles. Moreover, to model *diversity* among the recommended documents, we proposed a new diversity-metric based on the modeling of their point representation as repulsive particles and requiring the minimization of the work required to balance the system representing the solution, MIN-BW. We thus proposed a new approximation algorithm, FDLS, for solving the MIN-BW optimizing problem.We demonstrated the superior quality of our solutions through a user study that compared the end-to-end results produced by our two approaches (orthogonal-topics + MIN-BW & sentimented-topics + MIN-BW). against the state-of-the-art recommender from Abbar *et al.* [1].We demonstrated the superior scalability and performance of FDLS, by comparing it to the state-of-the-art approximation for MAX-MIN and MAX-AVG.

In Chapter 5, we streamline the exploration of data lakes and data warehouses. In particular, we have studied the problem of generating meaningful, concise, and informative descriptions for a dataset. We have modeled the descriptions as sets of views over the datasets, where the views are defined as filtering clauses. To judge the quality of a description, we considered four factors: the length, the coverage on the dataset, the overlap, and the intricacy. We have thus presented three algorithms that generate such descriptions given these four optimization objectives. We have then evaluated the algorithms by performing extensive experimental tests on synthetic and real-world datasets. Results show the effectiveness of our pruning strategies in Vertical and the increased scalability of Adaptive.

AVAILABILITY OF DATA AND MATERIAL

Source code for all the software developed for this thesis is publicly available online alongside all the used datasets.

- Graph benchmarking:
  https://graphbenchmark.com
- Distributed k-core decomposition:
  https://martinbrugnara.it/rp/dkcore.html
- Article Recommendation:
  https://martinbrugnara.it/rp/tsearch.html
- Dataset Description Generator:
  https://martinbrugnara.it/rp/describing_data.html

BIBLIOGRAPHY

[1] Sofiane Abbar, Sihem Amer-Yahia, Piotr Indyk, and Sepideh Mahabadi. "Real-time recommendation of diverse related articles." In: *Proceedings of the 22nd international conference on World Wide Web*. 2013, pp. 1–12.

[2] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. "A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data." In: *PVLDB* 10.13 (2017), pp. 2049–2060. ISSN: 2150-8097. DOI: 10.14778/3151106.3151109. URL: https://doi.org/10.14778/3151106.3151109.

[3] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications." In: *SIGMOD 1998*, pp. 94–105.

[4] H. Aksu, M. Canim, Yuan-Chi Chang, I. Korpeoglu, and O. Ulusoy. "Distributed k -Core View Materialization and Maintenance for Large Dynamic Graphs." In: *IEEE Trans. Knowl. Data Eng.* 26.10 (2014), pp. 2439–2452. ISSN: 1041-4347. DOI: 10.1109/TKDE.2013.2297918.

[5] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. "Describing Linked Datasets with the VoID Vocabulary." In: *W3c recommendation* (2011).

[6] B. Alexe, W. C. Tan, and Y. Velegrakis. "STBenchmark: towards a benchmark for mapping systems." In: *PVLDB* 1.1 (2008), pp. 230–244.

[7] J. Ignacio Alvarez-Hamelin, Mariano G. Beiró, and Jorge Rodolfo Busch. "Understanding Edge Connectivity in the Internet through Core Decomposition." In: *Internet Mathematics* 7.1 (2011), pp. 45–66. DOI: 10.1080/15427951.2011.560786. URL: http://dx.doi.org/10.1080/15427951.2011.560786.

[8] José Ignacio Alvarez-Hamelin, Alain Barrat, Alessandro Vespignani, and et al. "K-core decomposition of Internet graphs: hierarchies, self-similarity and measurement biases." In: *Networks and Heterogeneous Media* 3.2 (2008), p. 371.

[9] Renzo Angles. "A Comparison of Current Graph Database Models." In: *ICDEW*. 2012, pp. 171–177.

[10] 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." In: *SIGMOD Rec.* 43.1 (May 2014), pp. 27–31. ISSN: 0163-5808. DOI: 10.1145/2627692.2627697. URL: http://doi.acm.org/10.1145/2627692.2627697.

[11] Renzo Angles and Claudio Gutierrez. "Survey of Graph Database Models." In: *ACM Comput. Surv.* 40.1 (Feb. 2008), 1:1–1:39. ISSN: 0360-0300. DOI: 10.1145/1322432.1322433. URL: http://doi.acm.org/10.1145/1322432.1322433.

[12] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. "Benchmarking Database Systems for Social Network Applications." In: *GRADES*. New York, New York: ACM, 2013, 15:1–15:7. ISBN: 978-1-4503-2188-4. DOI: 10.1145/2484425.2484440. URL: http://doi.acm.org/10.1145/2484425.2484440.

[13] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. "RDF and Property Graphs Interoperability: Status and Issues." In: *Alberto Mendelzon Workshop on Foundations of Data Management*. 2019.

[14] *Apache Tinkerpop*. http://tinkerpop.apache.org/.

[15] Apache Tinkerpop. *GraphSON data format*. http://tinkerpop.apache.org/docs/current/reference/#graphson-io-format. 2020.

[16] *ArangoDB*. https://www.arangodb.com/.

[17] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. "Distributed k-core decomposition and maintenance in large dynamic graphs." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*. 2016, pp. 161–168. DOI: 10.1145/2933267.2933299. URL: https://doi.org/10.1145/2933267.2933299.

[18] GaryD Bader and ChristopherWV Hogue. "An automated method for finding molecular complexes in large protein interaction networks." English. In: *BMC Bioinformatics* 4.1, 2 (2003). DOI: 10.1186/1471-2105-4-2. URL: http://dx.doi.org/10.1186/1471-2105-4-2.

[19] 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*. Seoul, Korea: ACM, 2014, pp. 95–98. ISBN: 978-1-4503-2745-9. DOI: 10.1145/2567948.2577016. URL: http://doi.acm.org/10.1145/2567948.2577016.

[20] Batagelj, Vladimir and Mrvar, Andrej. *Yeast, Pajek dataset*. 2006.

[21] Vladimir Batagelj and Matjaž Zaveršnik. "Fast algorithms for determining (generalized) core groups in social networks." English. In: *Advances in Data Analysis and Classification* 5.2 (2011), pp. 129–145. ISSN: 1862-5347. DOI: 10.1007/s11634-010-0079-y.

[22] David M Blei, Andrew Y Ng, and Michael I Jordan. "Latent dirichlet allocation." In: *the Journal of machine Learning research* 3 (2003), pp. 993–1022.

[23] Haran Boral and David J. Dewitt. "A Methodology for Database System Performance Evaluation." In: *Proceedings of the International Conference on Management of Data*. 1984, pp. 176–185.

[24] Martin Brümmer, Ciro Baron, Ivan Ermilov, Markus Freudenberg, Dimitris Kontokostas, and Sebastian Hellmann. "DataID: Towards Semantically Rich Metadata for Complex Datasets." In: *SEM '14*. 2014, 84–91.

[25] 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." In: *Nucleic acids research* 31.9 (2003), pp. 2443–2450.

[26] Ed Bullmore and Olaf Sporns. "Complex brain networks: graph theoretical analysis of structural and functional systems." In: *Nature Reviews Neuroscience* 10.3 (2009), p. 186.

[27] 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*. Melbourne, VIC, Australia: ACM, 2015, 7:1–7:6. ISBN: 978-1-4503-3611-6. DOI: 10.1145/2764947.2764954. URL: http://doi.acm.org/10.1145/2764947.2764954.

[28] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Ozsu. "Efficient Core Decomposition in Massive Networks." In: *ICDE*. 2011, pp. 51–62. ISBN: 978-1-4244-8959-6. DOI: 10.1109/ICDE.2011.5767911. URL: http://dx.doi.org/10.1109/ICDE.2011.5767911.

[29] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Ozsu. "Efficient Core Decomposition in Massive Networks." In: *Proc. of the 27th Int. Conf. on Data Engineering*. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 51–62. ISBN: 978-1-4244-8959-6. DOI: 10.1109/ICDE.2011.5767911. URL: http://dx.doi.org/10.1109/ICDE.2011.5767911.

[30] Matteo Cinelli, Emanuele Brugnoli, Ana Lucia Schmidt, Fabiana Zollo, Walter Quattrociocchi, and Antonio Scala. "Selective exposure shapes the Facebook news diet." In: *PloS one* 15.3 (2020), e0229129.

[31] Matteo Cinelli, Gianmarco De Francisci Morales, Alessandro Galeazzi, Walter Quattrociocchi, and Michele Starnini. "The echo chamber effect on social media." In: *Proceedings of the National Academy of Sciences* 118.9 (2021).

[32] Mahashweta Das, Sihem Amer-Yahia, Gautam Das, and Cong Yu. "MRI: Meaningful Interpretations of Collaborative Ratings." In: *PVLDB* 4.11 (2011), pp. 1063–1074.

[33] P. De Bra and J. Paredaens. "Conditional dependencies for horizontal decompositions." In: *Automata, Languages and Programming*. 1983, pp. 67–82.

[34] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Commun. ACM* 51.1 (2008), pp. 107–113.

[35] Michela Del Vicario, Alessandro Bessi, Fabiana Zollo, Fabio Petroni, Antonio Scala, Guido Caldarelli, H Eugene Stanley, and Walter Quattrociocchi. "The spreading of misinformation online." In: *Proceedings of the National Academy of Sciences* 113.3 (2016), pp. 554–559.

[36] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. "The Data Civilizer System." In: *CIDR 2017*.

[37] 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. Jiuzhaigou Valley, China: Springer-Verlag, 2010, pp. 37–48. ISBN: 3-642-16719-5, 978-3-642-16719-5. URL: http://dl.acm.org/citation.cfm?id=1927585.1927590.

[38] Zhicheng Dou, Sha Hu, Kun Chen, Ruihua Song, and Ji-Rong Wen. "Multi-dimensional search result diversification." In: *Proceedings of the fourth ACM international conference on Web search and data mining*. 2011, pp. 475–484.

[39] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. "Clustering Large Graphs via the Singular Value Decomposition." In: *Mach. Learn.* 56.1-3 (June 2004), pp. 9–33. ISSN: 0885-6125. DOI: 10.1023/B:MACH.0000033113.59016.96. URL: http://dx.doi.org/10.1023/B:MACH.0000033113.59016.96.

[40] Marina Drosou and Evaggelia Pitoura. "Search result diversification." In: *ACM SIGMOD Record* 39.1 (2010), pp. 41–47.

[41] Marina Drosou and Evaggelia Pitoura. "DisC diversity: result diversification based on dissimilarity and coverage." In: *Proc. VLDB Endow.* 6.1 (2012), pp. 13–24. DOI: 10.14778/2428536.2428538. URL: https://doi.org/10.14778/2428536.2428538.

[42] Marina Drosou and Evaggelia Pitoura. "Multiple Radii DisC Diversity: Result Diversification Based on Dissimilarity and Coverage." In: *ACM Trans. Database Syst.* 40.1 (2015). ISSN: 0362-5915. DOI: 10.1145/2699499. URL: https://doi.org/10.1145/2699499.

[43] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R Narasayya, and Manoj Syamala. "Columnstore and B+ tree-Are Hybrid Physical Designs Important?" In: *SIGMOD*. 2018, pp. 177–190.

[44] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. "GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph." In: *PVLDB* 7.7 (2014), pp. 517–528. ISSN: 2150-8097. DOI: 10.14778/2732286.2732289. URL: http://dx.doi.org/10.14778/2732286.2732289.

[45] 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*. Melbourne, Victoria, Australia, 2015, pp. 619–630. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742786. URL: http://doi.acm.org/10.1145/2723372.2742786.

[46] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. "The Case Against Specialized Graph Analytics Engines." In: *CIDR*. 2015.

[47] Alan Fekete, Shirley N Goldrei, and Jorge Pérez Asenjo. "Quantifying isolation anomalies." In: *PVLDB* 2.1 (2009), pp. 467–478.

[48] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. "Interpretable and Informative Explanations of Outcomes." In: *PVLDB* 8.1 (2014), pp. 61–72.

[49] Giorgos Giannopoulos, Marios Koniaris, Ingmar Weber, Alejandro Jaimes, and Timos Sellis. "Algorithms and criteria for diversification of news article comments." In: *Journal of Intelligent Information Systems* 44.1 (2015), pp. 1–47.

[50] C. Giatsidis, D.M. Thilikos, and M. Vazirgiannis. "Evaluating Cooperation in Communities with the k-Core Structure." In: *Proc. of the Int. Conf. on Advances in Social Networks Analysis and Mining*. ASONAM'11. July 2011, pp. 87–93. DOI: 10.1109/ASONAM.2011.65.

[51] Lukasz Golab, Howard Karloff, Flip Korn, and Divesh Srivastava. "Data Auditor: Exploring Data Quality and Semantics Using Pattern Tableaux." In: *PVLDB* 3.1–2 (2010), 1641–1644.

[52] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. "On Generating Near-Optimal Tableaux for Conditional Functional Dependencies." In: *PVLDB* 1.1 (2008), 376–390.

[53] Google. *Freebase Data Dumps*. https://developers.google.com/freebase/data. 2015.

[54] Oshini Goonetilleke, Saket Sathe, Timos Sellis, and Xiuzhen Zhang. "Microblogging Queries on Graph Databases: An Introspection." In: *GRADES*. Melbourne, VIC, Australia: ACM, 2015, 5:1–5:6. ISBN: 978-1-4503-3611-6. DOI: 10.1145/2764947.2764952. URL: http://doi.acm.org/10.1145/2764947.2764952.

[55] Eduardo Graells-Garrido, Mounia Lalmas, and Daniele Quercia. "People of opposing views can share common interests." In: *Proceedings of the 23rd International Conference on World Wide Web*. 2014, pp. 281–282.

[56] Felix Hamborg, Karsten Donnay, and Bela Gipp. "Automated identification of media bias in news articles: an interdisciplinary literature review." In: *International Journal on Digital Libraries* 20.4 (2019), pp. 391–415.

[57] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Ozsu, Xingfang Wang, and Tianqi Jin. "An Experimental Comparison of Pregel-like Graph Processing Systems." In: *PVLDB* 7.12 (2014), pp. 1047–1058. ISSN: 2150-8097.

[58] Radoslav Harman and Vladimír Lacko. "On decompositional algorithms for uniform sampling from n-spheres and n-balls." In: *Journal of Multivariate Analysis* 101.10 (2010), pp. 2297–2304.

[59] Arne Holst. "Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025." In: *Statista, June* (2021).

[60] 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. 2013, pp. 195–204.

[61] 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. Genoa, Italy: ACM, 2013, pp. 195–204. ISBN: 978-1-4503-1599-9. DOI: 10.1145/2457317.2457351. URL: http://doi.acm.org/10.1145/2457317.2457351.

[62] Yanrong Huang, Rui Wang, Bin Huang, Bo Wei, Shu Li Zheng, and Min Chen. "Sentiment Classification of Crowdsourcing Participants' Reviews Text Based on LDA Topic Model." In: *IEEE Access* 9 (2021), pp. 108131–108143. DOI: 10.1109/ACCESS.2021.3101565. URL: https://doi.org/10.1109/ACCESS.2021.3101565.

[63] Wouter IJntema, Frank Goossen, Flavius Frasincar, and Frederik Hogenboom. "Ontology-based news recommendation." In: *Proceedings of the 2010 EDBT/ICDT Workshops*. 2010, pp. 1–6.

[64] E. Ioannou, N. Rassadko, and Y. Velegrakis. "On Generating Benchmark Data for Entity Matching." In: *J. Data Semantics* 2.1 (2013), pp. 37–56. DOI: 10.1007/s13740-012-0015-8. URL: http://dx.doi.org/10.1007/s13740-012-0015-8.

[65] E. Ioanou and Y. Velegrakis. "EMBench++: Data for a Thorough Benchmarking of Matching-Related Methods." In: *Semantic Web Journal* 9 (2018).

[66] H. V. Jagadish, J. Madar, and Raymond T. Ng. "Semantic Compression and Pattern Extraction with Fascicles." In: *VLDB 1999*, pp. 186–198.

[67] Paul Jakma, Marcin Orczyk, Colin S. Perkins, and Marwan Fayed. "Distributed k-core Decomposition of Dynamic Graphs." In: *Proc. of the 2012 ACM CoNEXT Student Workshop*. Nice, France: ACM, 2012. ISBN: 978-1-4503-1779-5. DOI: 10.1145/2413247.2413272. URL: http://doi.acm.org/10.1145/2413247.2413272.

[68] Salim Jouili and Valentin Vansteenberghe. "An Empirical Comparison of Graph Databases." In: *Proceedings of the 2013 International Conference on Social Computing*. SOCIALCOM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 708–715. ISBN: 978-0-7695-5137-1. DOI: 10.1109/SocialCom.2013.106. URL: http://dx.doi.org/10.1109/SocialCom.2013.106.

[69] Stephen Kaisler, Frank Armour, J. Alberto Espinosa, and William Money. "Big Data: Issues and Challenges Moving Forward." In: *2013 46th Hawaii International Conference on System Sciences*. 2013, pp. 995–1004. DOI: 10.1109/HICSS.2013.645.

[70] Daniel A Keim. "Information visualization and visual data mining." In: *IEEE TVCG* 1 (2002), pp. 1–8.

[71] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. "PerfXplain: Debugging MapReduce Job Performance." In: *PVLDB* 5.7 (2012), pp. 598–609.

[72] Sang-Woon Kim and Joon-Min Gil. "Research paper classification systems based on TF-IDF and LDA schemes." In: *Hum. centric Comput. Inf. Sci.* 9 (2019), p. 30. DOI: 10.1186/s13673-019-0192-7. URL: https://doi.org/10.1186/s13673-019-0192-7.

[73] Sofia Kleisarchaki. "Difference Analysis in Big Data: Exploration, Explanation, Evolution." PhD thesis. University of Grenoble Alps & University of Crete, 2016.

[74] Vojtěch Kolomičenko, Martin Svoboda, and Irena Holubová Mlýnková. "Experimental Comparison of Graph Databases." In: *IIWAS*. Vienna, Austria, 2013, 115:115–115:124. ISBN: 978-1-4503-2113-6. DOI: 10.1145/2539150.2539155. URL: http://doi.acm.org/10.1145/2539150.2539155.

[75] Laks VS Lakshmanan, Raymond T Ng, Christine Xing Wang, Xiaodong Zhou, and Theodore J Johnson. "The generalized MDL approach for summarization." In: *VLDB*. 2002, pp. 766–777.

[76] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J.J. van Wijk, J.-D. Fekete, and D.W. Fellner. "Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges." In: *Computer Graphics Forum* 30.6 (2011), pp. 1719–1749. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2011.01898.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2011.01898.x.

[77] D. Laney. "3D Data Management: Controlling Data Volume, Velocity and Variety." In: *META Group Research Note, vol. 6* (2001).

[78] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. "DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia." In: *Semantic web* (2015), pp. 167–195.

[79] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[80] Liang Li, Zhongmin Zhang, and Shengli Wu. "LDA-Based Resource Selection for Results Diversification in Federated Search." In: *Web Information Systems and Applications - 15th International Conference, WISA 2018, Taiyuan, China, September 14-15, 2018, Proceedings*. Ed. by Xiaofeng Meng, Ruixuan Li, Kanliang Wang, Baoning Niu, Xin Wang, and Gansen Zhao. Vol. 11242. Lecture Notes in Computer Science. Springer, 2018, pp. 147–156. DOI: 10.1007/978-3-030-02934-0_14. URL: https://doi.org/10.1007/978-3-030-02934-0_14.

[81] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. "Efficient Core Maintenance in Large Dynamic Graphs." In: *IEEE Trans. Knowl. Data Eng.* 26.10 (2014), pp. 2453–2465. DOI: 10.1109/TKDE.2013.158. URL: http://doi.ieeecomputersociety.org/10.1109/TKDE.2013.158.

[82] Q Vera Liao and Wai-Tat Fu. "Can you hear me now? Mitigating the echo chamber effect by source position indicators." In: *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*. 2014, pp. 184–196.

[83] Q Vera Liao and Wai-Tat Fu. "Expert voices in echo chambers: effects of source expertise indicators on exposure to diverse opinions." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2014, pp. 2745–2754.

[84] Rensis Likert. "A technique for the measurement of attitudes." In: *Archives of psychology* (1932).

[85] Matteo Lissandrini. *Freebase ExQ Data Dump*. https://disi.unitn.it/~lissandrini/notes/freebase-data-dump.html. 2017.

[86] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. *An Evaluation Methodology and Experimental Comparison of Graph Databases*. Tech. rep. Available at https://graphbenchmark.com. University of Trento, Apr. 2017. URL: \url{https://disi.unitn.it/~lissandrini/pdf/lissandrini-techreport-gdb.pdf}.

[87] Matteo Lissandrini, Davide Mottin, Themis Palpanas, Dimitra Papadimitriou, and Yannis Velegrakis. "Unleashing the Power of Information Graphs." In: *SIGMOD Rec.* 43.4 (Feb. 2015), pp. 21–26. ISSN: 0163-5808. DOI: 10.1145/2737817.2737822. URL: http://doi.acm.org/10.1145/2737817.2737822.

[88] Bin Liu and H. V. Jagadish. "Using Trees to Depict a Forest." In: *Proc. VLDB Endow.* 2.1 (2009), 133–144. ISSN: 2150-8097. DOI: 10.14778/1687627.1687643. URL: https://doi.org/10.14778/1687627.1687643.

[89] 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." In: *PVLDB* 5.8 (2012), pp. 716–727. ISSN: 2150-8097. DOI: 10.14778/2212351.2212354. URL: http://dx.doi.org/10.14778/2212351.2212354.

[90] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. "Large-scale Distributed Graph Computing Systems: An Experimental Evaluation." In: *PVLDB* 8.3 (2014), pp. 281–292. ISSN: 2150-8097. DOI: 10.14778/2735508.2735517. URL: http://dx.doi.org/10.14778/2735508.2735517.

[91] Fadi Maali, John Erickson, and Phil Archer. "Data catalog vocabulary (DCAT)." In: *W3c recommendation* 16 (2014).

[92] 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: *Proc. of the ACM Int. Conf. on Management of Data*. SIGMOD'10. ACM, 2010, pp. 135–146.

[93] 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*. 2010, pp. 135–146. DOI: 10.1145/1807167.1807184. URL: http://doi.acm.org/10.1145/1807167.1807184.

[94] 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)*. USENIX Association. 2017, pp. 631–643.

[95] Martin Brugnara. *TSearch - project page*. https://martinbrugnara.it/rp/tsearch.html.

[96] 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. Prague, Czech Republic: ACM, 2012, pp. 110–119. ISBN: 978-1-4503-1234-9. DOI: 10.1145/2351476.2351489. URL: http://doi.acm.org/10.1145/2351476.2351489.

[97] 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: USENIX Association, 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry.

[98] MediaWiki. *Extension:TextExtracts — MediaWiki*, 2021. URL: \url{https://www.mediawiki.org/w/index.php?title=Extension:TextExtracts&oldid=4940004}.

[99] Debra Burns Melican and Travis L Dixon. "News on the net: Credibility, selective exposure, and racial prejudice." In: *Communication Research* 35.2 (2008), pp. 151–168.

[100] Meta. *PetScan/en — Meta, discussion about Wikimedia projects*. 2021. URL: \url{https://meta.wikimedia.org/w/index.php?title=PetScan/en&oldid=21883648}.

[101] D. Miorandi and F. De Pellegrini. "K-shell decomposition for dynamic complex networks." In: *Proc. of the 8th Int. Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks)*. WiOpt'10. May 2010, pp. 488–496.

[102] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. "Distributed k-Core Decomposition." In: *IEEE Trans. Parallel Distrib. Syst.* 24.2 (2013), pp. 288–300. ISSN: 1045-9219. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.124.

[103] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. "DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data." In: *The Semantic Web – ISWC 2011*. 2011, pp. 454–469.

[104] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. "Exemplar Queries: A New Way of Searching." In: *The VLDB Journal* 25.6 (Dec. 2016), pp. 741–765. ISSN: 1066-8888. DOI: 10.1007/s00778-016-0429-2. URL: https://doi.org/10.1007/s00778-016-0429-2.

[105] Sean Munson, Stephanie Lee, and Paul Resnick. "Encouraging reading of diverse political viewpoints with a browser widget." In: *Proceedings of The International AAAI Conference on Web and Social Media*. Vol. 7. 1. 2013, pp. 419–428.

[106] *Neo4j*. http://neo4j.com.

[107] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. "Industry-scale knowledge graphs: lessons and challenges." In: *Queue* (2019), pp. 48–75.

[108] Behrooz Omidvar-Tehrani, Sihem Amer-Yahia, and Ria Mae Borromeo. "User group analytics: hypothesis generation and exploratory analysis of user data." In: *VLDB J.* 28.2 (2019), pp. 243–266.

[109] *On describing the contents of a dataset - project page.* https://martinbrugnara.it/rp/describing_datasets.html.

[110] *OrientDB*. http://orientdb.com/orientdb/.

[111] Matteo Paganelli, Paolo Sottovia, Antonio Maccioni, Matteo Interlandi, and Francesco Guerra. "Understanding Data in the Blink of an Eye." In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*. Ed. by Wenwu Zhu, Dacheng Tao, Xueqi Cheng, Peng Cui, Elke A. Rundensteiner, David Carmel, Qi He, and Jeffrey Xu Yu. ACM, 2019, pp. 2885–2888. DOI: 10.1145/3357384.3357849. URL: https://doi.org/10.1145/3357384.3357849.

[112] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* TR 1999-66. Stanford InfoLab. URL: http://ilpubs.stanford.edu:8090/422/.

[113] Roberto Patuelli, Aura Reggiani, Peter Nijkamp, and Franz-Josef Bade. "The evolution of the commuting network in Germany: Spatial and connectivity patterns." In: *Journal of Transport and Land Use* 2.3 (2010). ISSN: 1938-7849. URL: https://www.jtlu.org/index.php/jtlu/article/view/23.

[114] E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF. W3C Recommendation 15 January 2008*. 2008.

[115] Abdulhakim Ali Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. "Pattern Functional Dependencies for Data Cleaning." In: *PVLDB* 13.5 (2020), pp. 684–697.

[116] Sekharipuram S Ravi, Daniel J Rosenkrantz, and Giri Kumar Tayi. "Facility dispersion problems: Heuristics and special cases." In: *Workshop on Algorithms and Data Structures*. Springer. 1991, pp. 355–366.

[117] Marko A. Rodriguez. "The Gremlin Graph Traversal Machine and Language (Invited Talk)." In: *DBPL*. Pittsburgh, PA, USA, 2015, pp. 1–10. ISBN: 978-1-4503-3902-5. DOI: 10.1145/2815072.2815073. URL: http://doi.acm.org/10.1145/2815072.2815073.

[118] LT Rodrygo, Craig Macdonald, and Iadh Ounis. "Search result diversification." In: *Foundations and Trends in Information Retrieval* 9.1 (2015), pp. 1–90.

[119] Rami Rosen. "Resource management: Linux kernel namespaces and cgroups." In: (2013).

[120] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. "Nobody Ever Got Fired for Using Hadoop on a Cluster." In: *HotCDP*. Bern, Switzerland, 2012, 2:1–2:5. ISBN: 978-1-4503-1162-5. DOI: 10.1145/2169090.2169092. URL: http://doi.acm.org/10.1145/2169090.2169092.

[121] *SYSTAP LLC., Blazegraph*. https://www.blazegraph.com/.

[122] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing." In: *PVLDB* 11.4 (2017), pp. 420–431.

[123] Alessandra Sala, Lili Cao, Christo Wilson, Robert Zablit, Haitao Zheng, and Ben Y. Zhao. "Measurement-calibrated Graph Models for Social Network Experiments." In: *Proc. of the 19th Int. Conf. on World Wide Web*. WWW'10. Raleigh, North Carolina, USA: ACM, 2010, pp. 861–870. ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772778. URL: http://doi.acm.org/10.1145/1772690.1772778.

[124] Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. "Discovery-driven exploration of OLAP data cubes." In: *EDBT 1998*, pp. 168–182.

[125] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. "Streaming Algorithms for K-core Decomposition." In: *PVLDB* 6.6 (Apr. 2013), pp. 433–444. DOI: 10.14778/2536336.2536344. URL: http://dx.doi.org/10.14778/2536336.2536344.

[126] Sheikh Muhammad Sarwar, Raghavendra Addanki, Ali Montazeralghaem, Soumyabrata Pal, and James Allan. "Search Result Diversification with Guarantee of Topic Proportionality." In: *ICTIR '20: The 2020 ACM SIGIR International Conference on the Theory of Information Retrieval, Virtual Event, Norway, September 14-17, 2020*. Ed. by Krisztian Balog, Vinay Setty, Christina Lioma, Yiqun Liu, Min Zhang, and Klaus Berberich. ACM, 2020, pp. 53–60. DOI: 10.1145/3409256.3409839. URL: https://doi.org/10.1145/3409256.3409839.

[127] Stephen B. Seidman. "Network structure and minimum degree." In: *Social Networks* 5.3 (1983), pp. 269 –287. ISSN: 0378-8733. DOI: http://dx.doi.org/10.1016/0378-8733(83)90028-X. URL: http://www.sciencedirect.com/science/article/pii/037887338390028X.

[128] Anne Shelley. "Book review of Eli Pariser's *The filter bubble: What the Internet is hiding from you*." In: *First Monday* 17.6 (2012).

[129] *Sparsity Technologies, Sparksee*. http://www.sparsity-technologies.com/.

[130] 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*. ACM. 2007, pp. 697–706.

[131] Mir Saman Tajbakhsh and Jamshid Bagherzadeh. "Microblogging hash tag recommendation system based on semantic TF-IDF: Twitter use case." In: *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. IEEE. 2016, pp. 252–257.

[132] P. M. L. Tammes. "On the origin of number and arrangement of the places of exit on the surface of pollen-grains." PhD thesis. Groningen: J.H. De Bussy, 1930.

[133] Ole Tange. *Gnu parallel 2018*. 2018. ISBN: 9781387509881. DOI: https://doi.org/10.5281/zenodo.1146014.

[134] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. "YAGO 4: A Reason-able Knowledge Base." In: *ESWC 2020*. 2020, pp. 583–596.

[135] Robert Tarjan. "Depth first search and linear graph algorithms." In: *Siam Journal On Computing* 1.2 (1972).

[136] *Thinkaurelius, Titan*. http://titan.thinkaurelius.com/.

[137] JJ Thomson. "London, Edinburgh Dublin Philos. Mag." In: *J. Sci* 7.39 (1904), pp. 237–265.

[138] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. "Contention-aware lock scheduling for transactional databases." In: *PVLDB* 11.5 (2018), pp. 648–662.

[139] Mircea Răducu Trifu, Mihaela Laura Ivan, et al. "Big Data: present and future." In: *Database Systems Journal* 5.1 (2014), pp. 32–41.

[140] Koji Ueno and Toyotaro Suzumura. "Highly Scalable Graph Search for the Graph500 Benchmark." In: *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '12. Delft, The Netherlands: ACM, 2012, pp. 149–160. ISBN: 978-1-4503-0805-2. DOI: 10.1145/2287076.2287104. URL: http://doi.acm.org/10.1145/2287076.2287104.

[141] Aaron R Voelker, Jan Gosmann, and Terrence C Stewart. "Efficiently sampling vectors and coordinates from the n-sphere and n-ball." In: *Centre for Theoretical Neuroscience-Technical Report* (2017).

[142] Denny Vrandečić and Markus Krötzsch. "Wikidata: a free collaborative knowledgebase." In: *Communications of the ACM* 57.10 (2014), pp. 78–85.

[143] VG Vinod Vydiswaran, ChengXiang Zhai, Dan Roth, and Peter Pirolli. "Overcoming bias to learn about controversial topics." In: *Journal of the Association for Information Science and Technology* 66.8 (2015), pp. 1655–1672.

[144] Arisa Watanabe and Basabi Chakraborty. "Time-series Analysis of Newspaper Articles for Automatic Event Detection using LDA." In: *4th IEEE International Conference on Knowledge Innovation and Invention, ICKII 2021, Taichung, Taiwan, July 23-25, 2021*. Ed. by Teen-Hang Meen. IEEE, 2021, pp. 166–169. DOI: 10.1109/ICKII51822.2021.9574704. URL: https://doi.org/10.1109/ICKII51822.2021.9574704.

[145] Yuhao Wen, Xiaodan Zhu, Sudeepa Roy, and Jun Yang. "Interactive Summarization and Exploration of Top Aggregate Query Answers." In: *CoRR* (2018).

[146] Di Wu, Rui Xin Yang, and Chao Shen. "Sentiment word co-occurrence and knowledge pair feature extraction based LDA short text clustering algorithm." In: *J. Intell. Inf. Syst.* 56.1 (2021), pp. 1–23. DOI: 10.1007/s10844-020-00597-7. URL: https://doi.org/10.1007/s10844-020-00597-7.

[147] Eugene Wu and Samuel Madden. "Scorpion: Explaining Away Outliers in Aggregate Queries." In: *PVLDB* 6.8 (2013), pp. 553–564.

[148] Derek Wyatt. *Akka Concurrency*. USA: Artima Inc., 2013. ISBN: 0981531660, 9780981531663.

[149] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. "RDF Data Storage and Query Processing Schemes: A Survey." In: *ACM Comput. Surv.* (2018), 84:1–84:36.

[150] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, and James Cheng. "Big Graph Analytics Systems." In: *SIGMOD*. San Francisco, California, USA, 2016, pp. 2241–2243. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2912566. URL: http://doi.acm.org/10.1145/2882903.2912566.

[151] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. "Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs." In: *PVLDB* 7.14 (2014), pp. 1981–1992.

[152] Nakyeong Yang, Jeongje Jo, Myeong Jun Jeon, Wooju Kim, and Juyoung Kang. "Semantic and explainable research-related recommendation system based on semi-supervised methodology using BERT and LDA models." In: *Expert Syst. Appl.* 190 (2022), p. 116209. DOI: 10.1016/j.eswa.2021.116209. URL: https://doi.org/10.1016/j.eswa.2021.116209.

[153] 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. Santa Clara, California: ACM, 2017, pp. 40–51. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3128606. URL: http://doi.acm.org/10.1145/3127479.3128606.