On Pattern Mining in Graph Data to Support Decision-Making

Von der Fakultät für Mathematik und Informatik
der Universität Leipzig

angenommene

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

Doctor Rerum Naturalium
(Dr. rer. nat.)

im Fachgebiet Informatik

vorgelegt von
Diplom-Ingenieur (FH) André Petermann
geboren am 22. November 1984 in Zwickau

Die Annahme der Dissertation wurde empfohlen von:
1. Prof. Dr. Erhard Rahm, Universität Leipzig
2. Prof. Dr. Esteban Zimányi, Université Libre de Bruxelles

Die Verleihung des akademischen Grades erfolgt mit Bestehen der Verteidigung am 21. Januar 2019 mit dem Gesamtprädikat

Magna cum Laude

Abstract

In recent years graph data models became increasingly important in both research and industry. Their core is a generic data structure of things (vertices) and connections among those things (edges). Rich graph data models such as the property graph model support additional data attached to vertices and edges. Such data ranges from simple labels to named attributes. Thus, graph data models can be used to represent data of different domains. Typically, they are used to represent native network data such as the world wide web, social networks or complex knowledge bases. However, they are also suitable to represent data from domains that are strongly placed in the context of relational databases. An example is data from business information systems. Here, domain objects such as customers, products and sales orders can be represented by vertices and their relationships can be represented by edges.

Graphs have an extraordinary analytical power because relationships can be evaluated without schema knowledge. For example, it is possible to query all vertices that are connected to one another without knowledge about possible relationship types. By contrast, for such queries relationship types are even part of the result. There are also data mining techniques based on this property of graph data models. The present dissertation will study to which extend graph data models and data mining techniques based thereon may help analysts to make better decisions. As general as this problem sounds, so versatile are the studied problems. In particular, they range from transforming relational data into graphs to the extraction of relevant patterns.

Around these subproblems we developed a conceptual framework called BIIIG (Business Intelligence with Integrated Instance Graphs). The term business intelligence is usually associated with data warehouse models which are tailored to the representation of facts, measures and dimensions. With BIIIG, we transfer these concepts to a graph data model. In particular, we propose to represent logical partitions of interrelated data by a collection of many small graphs. In BIIIG's model these graphs contain not only interrelated facts but also subgraphs of dimension values that occur in multiple graphs. Further on, measure values such as business indicators must be evaluated on the graph level. With characteristic subgraph mining, we will propose an example analytical workflow that is capable to extract interesting patterns that cannot be found by existing approaches.

We identified different requirements that must be met to implement BIIIG. Since we evaluate measures in the context of graphs a crucial one is the representation of graphs where not only vertices and edges but also graphs themselves may have attributes. By an overview of recent graph database and graph processing systems we will show that currently no system supports these features. Thus, we developed the so-called Extended Property Graph Model which extends the property graph model by respective graph collections and operators that allow to create, modify or evaluate graph attributes.

Another crucial problem that will be studied is the transformation of business data, which is commonly stored in relational databases, into property graphs. We will present a flexible approach based on metadata evaluation. Our solution provides more features than previous academic approaches and solutions provided by database vendors. Additionally, our approach includes a seamless data integration strategy and a generic algorithm that can turn a large graph of business data into a collection of meaningful subgraphs. Since real business data is not available we used test data of a real business information system and developed a data generator based on business process simulation to evaluate these techniques.

An important primitive to extract patterns from graph collections is frequent subgraph mining. In the past this problem has only been studied for chemical datasets where graphs represent molecules. Since business data shows different characteristics existing algorithms cannot be applied to BIIIG. Thus, we extended an existing algorithm by the missing features. Frequent subgraph mining is our most expensive subproblem in terms of computational complexity. Thus, this dissertation also studies the problem's parallelization in the context of state-of-the-art Big Data technology. In particular, we propose an approach that brings the computation to the data and propose multiple optimizations to decrease runtime. In an extensive experimental evaluation, we will show the scalability of our solution with regard to different parameters.

Many interesting patterns can only be extracted, if dimension values are attached to taxonomies and also generalizations are evaluated. Thus, this dissertation also studies the problem of generalized frequent subgraph mining. We will propose an efficient algorithm with support for multiple dimensional taxonomies. To improve efficiency, we decompose the problem into separate mining steps for frequent subgraphs and frequent vectors that represent vertex dimensions.

With Gradoop, a distributed system for declarative graph analytics, there is already a first system that supports the Extended Property Graph Model. The development of BIIIG was often connected to the development of Gradoop. Thus, some implementations of BIIIG's components, for example our horizontally scalable approach to frequent subgraph mining, were contributed to Gradoop. Thus, just like the framework, they are available under an Open Source licence. Additionally, thread-parallel implementations of all proposed data mining algorithms have been turned into an Open Source Java library. At the end of this dissertation we will report two real-world applications of BIIIG in cooperation with two large-scale enterprises. The application scenarios were targeting the identification of patterns that are characteristic to fraud and security threats. With these evaluations we were able to confirm that our approach works.

The result of this dissertation is BIIIG, a practical conceptual framework for graph-based business intelligence. All components of BIIIG were implemented and the resulting prototypes were evaluated in experiments. Further on, the functionality of BIIIG has been confirmed in real-world applications. By the diversity of these applications' domains we could show that BIIIG is not limited to business intelligence. Its major strength is the extraction of domain knowledge by a generic pattern mining process that requires no schema knowledge as input. In particular, this is useful for an initial discovery but also for continuous monitoring of patterns that occur in one or more interrelated databases. This dissertation will further state open research questions for graph-based business intelligence. These questions will relate to technical problems to further improve performance and to functional extensions to improve usability.

Acknowledgements

There are so many parties whom I would like to thank. Let me start with academia. First of all, I would like to thank Professor Johannes Waldmann for arousing my interest in computing science. Without his lectures I would never have discovered my passion for programming. Further thanks goes to Professor Jörg Bleymehl who brought me to my initial scholarship for cooperative PhD studies between both Leipzig Universities. I would probably never have started this dissertation without this option. My deepest thanks also goes the Professor Robert Müller whose support enabled me to catch the scholarship and who gave me first advice about professional research. Further on, also his lectures played an important role in my life as they woke up my interest in databases that continues unil today. Finally, highest thanks goes to Professor Erhard Rahm who gave me the chance to prove myself in Leipzig's database research group despite my rather untypical CV. Without his guidance and supervision this dissertation would never have been finished.

A second party who I'd like to thank are companies that enabled practical experience that would never have been possible without their cooperation. First, there is Axel Schwanke from Immowelt. I would like to thank him for the option to send a student to Immowelt to evaluate BIIIG on Gradoop. Further thanks goes to Gerald Ulmer from Siemens. He enabled me to evaluate the findings of this dissertation in practice under very good conditions. I would also like to thank the whole STA team for welcoming me warmly despite my short and sporadic presence.

Further thanks goes to my former colleagues of the database research group for interesting discussions about research in general and about the joy and sorrow that we were all facing at certain times of our work. Very special thanks goes to the graph guys which are Martin Junghanns as well as our former student assistants Kevin Goméz, Niklas Teichmann and Stephan Kemper. Without them work would have been quite lonely. In particular, I would like to thank Martin for all the endless discussions about graphs and programming. I am sure without them this dissertation would have had a different outcome. Special thanks also goes to Kevin who did more than just a student job to make experiments running and always kept all these tedious configuration things away from me. Further special thanks goes to Markus Nentwig. I thank him and Martin for all the happy memories about lunch breaks, pub evenings and business trips.

I also want to thank all of my friends for not abandoning me after hearing sentences like *"Sorry, I must focus on a paper right now. I will call you back."*, *"No, not this weekend I have a deadline to catch."* and *"Let's postpone this after I will have finished my dissertation, please."* Very special thanks goes to Dr. Roxana Bujack who is not only a friend but also contributed to this dissertation by giving me some valuable tutoring sessions in mathematics during the first months of my research.

Further highest thanks goes to my parents Egon and Sieglinde Petermann. I am the fist in the family who made it to the dissertation and without them that would never have been possible. Making them proud has always motivated me. I thank my mother in particular for keeping me on track back in school days. Without her commitment and knowledge I would probably have not even made it to university. I also thank my father in particular for his constant support for doing things in my way. Because of him I also know that freedom, the key to personal fulfilment, is not self-evident and that I can consider myself fortunate to grew up in a free country. If education had been linked to political submission even today, I would not have made it this far.

Finally, my greatest thanks goes to Janet Jesemann who intensively supported me over all the years like no other. I thank her not only for the immense effort of proof-reading my papers and this dissertation but also for all the priceless mental support. She has endured an endless number of discussions related to my research, pulled me out of every crisis and stopped me from giving up more than once. So, a part of this dissertation also belongs to her.

About the Author

Andre Petermann studied Multimedia Technology at Leipzig University of Applied Science and graduated as Diplom-Ingenieur (FH) in 2011. In 2009, he also received a bachelor's degree in Information Technology from the University of the West of Scotland as part of an ERASMUS programme. After studies he left univeristy and became a data warehouse developer. When he came into contact with graph databases for the first time he had the idea to utilize them for business intelligence. He though the best way to investigate this topic would be a dissertation and started looking for professors that may support this idea. With Professor Erhard Rahm from Leipzig University and Professor Robert Müller from Leipzig University of Applied Sciences he found two supporters and could successfully apply for a public scholarship [1]. So, in 2013 he went back to university and started research at Leipzig University's database research group. In 2015 he became a research assistant at ScaDS Dresden/Leipzig Competence Center for Scalable Data Services and Solutions[2].

[1]funded within the EU program *Europa fördert Sachsen* of the European Social Fund
[2]funded by the German Federal Ministry of Education and Research (BMBF 01IS14014B)

Dissertation-related Publications

1. Petermann, A., Junghanns, M., Müller, R. and Rahm, E. BIIIG : Enabling Business Intelligence with Integrated Instance Graphs. In Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014 (2014), pp. 4–11.

2. Petermann, A., Junghanns, M., Müller, R. and Rahm, E. FoodBroker - Generating Synthetic Datasets for Graph-Based Business Analytics. In Big Data Benchmarking - 5th International Workshop, WBDB 2014, Potsdam, Germany, August 5-6, 2014, Revised Selected Papers (2014), pp. 145–155.

3. Petermann, A., Junghanns, M., Müller, R. and Rahm, E. Graph-based data integration and business intelligence with BIIIG. PVLDB 7, 13 (2014), 1577–1580.

4. Junghanns, M., Petermann, A., Teichmann, N., Gómez, K. and Rahm, E. Analyzing Extended Property Graphs with Apache Flink. In Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA@SIGMOD 2016, San Francisco, California, USA, July 1, 2016 (2016), pp. 3:1–3:8.

5. Petermann, A. and Junghanns, M. Scalable Business Intelligence with Graph Collections. it - Information Technology 58, 4 (2016), 166–175.

6. Petermann, A., Junghanns, M., Kemper, S., Gómez, K., Teichmann, N., and Rahm, E. Graph Mining for Complex Data Analytics. In IEEE International Conference on Data Mining Workshops, ICDM Workshops 2016, December 12-15, 2016, Barcelona, Spain. (2016), pp. 1316–1319.

7. Junghanns, M., Petermann, A., Neumann, M. and Rahm, E. Management and Analysis of Big Graph Data: Current Systems and Open Challenges. In Handbook of Big Data Technologies. 2017, pp. 457–505.

8. Junghanns, M., Petermann, A. and Rahm, E. Distributed Grouping of Property Graphs with Gradoop. In Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings (2017), pp. 103–122.

9. Junghanns, M., Petermann, A., Teichmann, N. and Rahm, E. The Big Picture: Understanding large-scale graphs using Graph Grouping with Gradoop. In Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings (2017), pp. 629–632.

10. Kemper, S., Petermann, A. and Junghanns, M. Distributed FoodBroker: Skalierbare Generierung graphbasierter Geschäftsprozessdaten. In Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Workshopband (2017), pp. 105–110.

11. Junghanns, M., Kiessling, M., Averbuch, A., Petermann, A. and Rahm, E. Cypher-based Graph Pattern Matching in Gradoop. In Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017 (2017), pp. 3:1–3:8.

12. Petermann, A. Graph Pattern Mining for Business Decision Support. In Proceedings of the VLDB 2017 PhD Workshop co-located with the 43rd International Conference on Very Large Databases (VLDB 2017), Munich, Germany, August 28, 2017. (2017).

13. Petermann, A., Micale, G., Bergami, G., Pulvirenti, A. and Rahm, E. Mining and Ranking of Generalized Multi-Dimensional Frequent Subgraphs. In 2017 Twelfth International Conference on Digital Information Management (ICDIM) (Sept 2017), pp. 236–245.

14. Petermann, A., Junghanns, M. and Rahm, E. DIMSpan - Transactional Frequent Subgraph Mining with Distributed In-Memory Dataflow Systems. In Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (Austin, TX, USA, 2017), BDCAT '17, ACM, pp. 237–246.

15. Bergami, G., Petermann, A. and Montesi, D. THoSP: an Algorithm for Nesting Property Graphs. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018 (2018), pp. 8:1–8:10.

16. Junghanns, M., Kiessling, M., Teichmann, N., Gómez, K., Petermann, A., and Rahm, E. Declarative and distributed graph analytics with Gradoop. PVLDB to appear (2018).

Contents

1 Introduction **10**

1.1 Relational Data vs. Graph Data 10
1.2 Graphs for Business Intelligence 16
1.3 Contributions and Publications 20
1.4 Structure of Dissertation 22

2 Background and Related Work **24**

2.1 Graph Data Structures 24
2.2 Comparison of RDF and Property Graphs 30
2.3 Distributed Dataflow Systems 34
2.4 Graph Processing 36
2.5 Graph Databases 37
2.6 Graph Transformation of Relational Data 42
2.7 Graph-based Data Warehousing 44
2.8 Graph Pattern Mining 46
2.9 Data Generators 53

3 Analytical Framework **55**

3.1 Master and Transaction Data 55
3.2 Business Transaction Graphs 56
3.3 Example Scenario 58
3.4 Measures and Dimensions 59
3.5 Characteristic Subgraph Mining 61
3.6 Requirements 64

4 Extended Property Graph Model **65**

4.1 Motivation 65
4.2 Data Structure and Operators 67
4.3 The Gradoop Framework 70
4.4 FoodBroker Data Generator 72
4.5 Conclusion 79

5 Graph-based Transformation and Integration of Data **80**
5.1 Overview 80
5.2 Metadata Management 84
5.3 Graph Transformation 90
5.4 Data Integration 91
5.5 Business Transaction Graphs 96
5.6 Experimental Evaluation 99
5.7 Conclusion 104

6 Frequent Subgraph Mining in Distributed Graph Collections **105**
6.1 Motivation 105
6.2 Frequent Subgraph Mining in Collections of Directed Multigraphs 107
6.3 Frequent Subgraph Mining with Distributed Dataflow Systems 115
6.4 Comparison to MapReduce-based Approaches 124
6.5 Experimental Evaluation 129
6.6 Conclusion 134

7 Generalized Multidimensional Frequent Subgraph Mining **135**
7.1 Motivation 135
7.2 Problem, Data Model and Terminology 138
7.3 Path Substitution Method 140
7.4 Pattern Decomposition Method 142
7.5 Experimental Evaluation 149
7.6 Conclusion 153

8 Real-World Applications, Conclusion and Outlook **154**
8.1 BIIIG for Real Estate Fraud Detection 154
8.2 BIIIG for Security Threat Analysis 156
8.3 Conclusion 159
8.4 Future Research Directions 162

List of Definitions

2.1 Undirected Simple Graph 27
2.2 Directed Simple Graph 27
2.3 Directed Multigraph 27
2.4 Labeled Directed Multigraph 28
2.5 Property Graph 28
2.6 RDF Triple Graph 29
2.7 RDF N-quad Graph Collection 29
2.8 Hypergraph 29
2.9 Hypervertex Graph 30
3.1 Master Data 55
3.2 Transaction Data 56
3.3 Business Process 56
3.4 Business Case 56
3.5 Business Transaction Graph 58
3.6 Measure 59
3.7 Dimension 59
3.8 Dimension Relationship Pattern 60
4.1 Property Graph Collection 67
4.2 EPGM Operator 67
4.3 Graph Collection Copy 68
4.4 Aggregation 68
4.5 Selection 69
4.6 Property Transformation 69
6.1 Subgraph 108
6.2 Path 108
6.3 Pattern 108
6.4 Isomorphism 108
6.5 Embedding 108
6.6 Support 109
6.7 Frequent Subgraph Mining 109
6.8 Parent-Child Relationship of Graphs 109
6.9 DFS Code 113

6.10 DFS Embedding 113
6.11 DFS Extension 113
6.12 Parent-Child Relationship of DFS Codes 114
6.13 Minimum DFS Code 114
7.1 Taxonomy/Taxonomy Path 139
7.2 Label Generalization 139
7.3 Top-level Label 139
7.4 Bottom-level Label 139
7.5 Multidimensional Graph 140
7.6 Graph Generalization 140
7.7 Top-level Graph 140
7.8 Bottom-level Graph 140
7.9 Generalized Multidimensional Frequent Subgraph Mininig 140
7.10 Global Order of Labels 143
7.11 Taxonomy Path Vector 144
7.12 Vector Generalization 145
7.13 Generalized Frequent Vector Mininig 145

List of Figures

1.1 Schema (Metadata) of a relational database: Every rectangle represents a table definition including the table name (e.g. SalesOrder) and all columns, their data type and primary key position (PK). Further on, each table contains foreign keys including their columns and target table. 11
1.2 Comparison of the same instance data in relational and graph representation. 12
1.3 Comparison of a pattern mining result in graph and relational representation. Both representations contain the same information. 14
1.4 Example graph collection with shared vertices. Dotted lines represent graph boundaries. Solid rectangles overlaying the graphs' boundaries contain graph properties. Bold properties contain measure values and normal font ones dimensional values. 17
1.5 Example Taxonomy of dimension Employee. 18
1.6 Example generalization of a graph pattern. 19

2.1 Graph structures with different edge characteristics. 27
2.2 Graph structures with different attached data formats. 28
2.3 Hypergraphs and hypervertices. 30
2.4 Comparison of an example property graph with mandatory label properties (italic font) and its schema-less RDF representation by standard reification. 32
2.5 Comparison of pattern matching queries. 40

3.1 Example business transaction graphs with aggregated graph measures isClosed and soCount [139] 57
3.2 Example relationship patterns with dimension values as label-properties. 59
3.3 Business transaction graph (id=1) after normalization for pattern mining. 62

4.1 Example graph collection with shared vertices. Dotted lines represent graph boundaries. Solid rectangles overlaying the graphs' boundaries contain graph properties. Bold properties contain measure values and normal font ones dimensional values. 66
4.2 Gradoop architecture [85]. 69

4.3 FoodBroker Schema : The outer rectangles show the boundaries of two systems ERP and CIT. Database tables correspond either to classes or n:m relationship types (*Line). Primary keys are highlighted by italic letters. Relationship types are shown as solid lines. Foreign keys are attached to relationship types. Implicit relationship types in between both databases are represented by dotted lines. For each implicit relationship type, there is a corresponding column with prefix erp. 74

5.1 Overview of the BIIIG framework. 81
5.2 Example unified metadata graph with associated relational database. 83
5.3 Example unified metadata graph with metadata from multiple sources. Vertices represent classes and edges represent relationship types. Gray vertices represent master data classes. Dashed rectangles represent source boundaries and correspond to the ds properties of vertices. . . 92
5.4 Example integrated instance graph with data from multiple sources. Vertices represent data objects and edges represent relationships. Gray vertices represent master data. Dashed rectangles represent source boundaries and correspond to the prefix of sid properties of vertices. The instance graph was generated based on the unified metadata graph shown by Figure 5.3 93
5.5 Example instance graph of Figure 5.4 after vertex fusion. 94
5.6 Example of the relationship integration problem. 96
5.7 Visualization (Gephi [224]) of the integrated instance graph extracted from a dataset that was created by the real-world business information systems ERPNext. 100
5.8 Visualization (Neo4j frontend [232]) of a business transaction graph extracted from a dataset created by the FoodBroker data generator. . 101
5.9 Visualization (Gephi [224]) of a business transaction graph extracted from a dataset created by a real-world business information system. Zoom factor is to low to render labels. 102
5.10 Clipped view on the graph of Figure 5.9 with a higher zoom factor (Gephi [224]). Black vertices show master data and gray vertices show transaction data. The displayed label is the class property of transaction data and a manually chosen business identifier for master data. . 103

6.1 Example illustrations for a collection of labeled graphs, a subgraph (gray background), a frequent pattern lattice and embeddings. The values attached to vertices and edges represent id:label pairs 107
6.2 Pattern lattice search strategies. Bullets represent patterns and lines represent parent-child relationships where parents are shown above children. Red lines are those actually processed by the particular search strategy. 109

6.3 Dataset element representing graph g_3, pattern p_{21} and embedding map $\mu(g_3, p_{21})$ of Figure 6.1. 122
6.4 Illustration of our couterexample showing two graphs g_1, g_2. Each one contains a 3-edge subgraph with automorphisms (black lines) and an extension to a 4-edge subgraph (red lines). Roman numbers are vertex identifiers. 127
6.5 Example graph of Gradoop's predictable transactions graph generator. Colored rectangles represent subgraphs and their guaranteed support. 130
6.6 Scalability for varying input size. 131
6.7 Scalability for varying result size. 132
6.8 Horizontal scalability for varying cluster sizes. 133

7.1 Example multidimensional subgraph and patterns. 136
7.2 Path-substitution method: Taxonomy paths are represented by dedicated vertices and edges (blue lines). 137
7.3 Example taxonomies of Figures 7.1 and 7.2. Bold fonts highlight dummy roots. 138
7.4 Comparison of taxonomy path representations in frequent graph patterns. The gray filled vertex can be assigned to a taxonomy path. Red color indicates infrequent parts of a pattern. 141
7.5 Generalization search lattice for a 2-dimensional example vector set $\mathcal{L} = \{(111, 2111), (111, 2112), (112, 2112)\}$. Common prefixes indicate label generalizations (e.g., $11 <_T 112$). Edges represent vector generalization from bottom to top. 143
7.6 Bottom-up search in the example lattice of Figure 7.5. Edge labels correspond to u_{min} of Algorithm 7.1. Red lines indicate paths that have been traversed unnecessarily at $\phi_{min} = 3$. 144
7.7 Top-down search in the example lattice of Figure 7.5. Edge labels correspond to u_{min} of Algorithm 7.2. Gray lines indicate pruned paths at $\phi_{min} = 3$. 147
7.8 GM-FSM evaluation results for variable minimum support threshold ϕ_{min} and a fixed maximum edge count $k_{max} = 6$. 150
7.9 GM-FSM evaluation results for fixed minimum support threshold $\phi_{min} = 0.2$ and a variable maximum edge count k_{max}. 152

List of Tables

2.1 Glossary of symbols part 1 (greek letters) 25
2.2 Glossary of symbols part 2 (further symbols) 26
2.3 Selected Unary Transformations. I represents the input and O represents the output data set. $A..D$ are distinct data spaces and W is a set of worker threads. 34
2.4 Comparison of Graph database systems. 38
2.5 Citation count of FSM algorithms on Google Scholar (June 2018). . . 50

3.1 Examples transaction data classes, master data classes, measures and dimensions of different business processes. 58

4.1 Operators supported by Gradoop at the time of May 2018. 68
4.2 FoodBroker Configuration Parameters 77
4.3 Measures of FoodBroker datasets for different scale factors (SF) 78

6.1 Methodical comparison of DIMSpan and approaches based on MapReduce. 123
6.2 Cost comparison of DIMSpan and approaches based on MapReduce. . 126
6.3 Embeddings and DFS codes during the pattern growth from 3-edge subgraphs (black lines) to 4-edge subgraphs (red lines) in the graphs of Figure 6.4. 128

List of Algorithms

5.1 Relational Metadata Acquisition (Part I - Classes) 86
5.2 Relational Metadata Acquisition (Part II - Relationship Types) . . 87
5.3 Method *addRelationshipType* 88
5.4 Automated Graph Transformation 89
5.5 Vertex Fusion 95
5.6 Business Transaction Graph Isolation 97
6.1 A priori (BFS) approach to frequent subgraph mining 110
6.2 Pattern growth (DFS) approach to frequent subgraph mining . . . 112
6.3 Level-wise pattern growth (LDFS) approach to FSM 116
6.4 DIMSpan dataflow. 117
6.5 Pattern growth map function 120
7.1 Bottom-up search GFVM 145
7.2 Top-down search GFVM 148

Chapter 1

Introduction

To make good decisions, enterprises have a permanent desire to understand the reasons for success or failure of their business. Nowadays, enterprises use software systems to support this task, so-called *decision support systems (DSS)* [151]. A popular class of these systems, *data-driven DSS*, analyze data that is available in databases to extract information relevant to decision-making. Databases are typically used to store domain objects, for example, a trade company stores customer and sales order data. Additionally, databases store relationships among domain objects, for example, for each order there is a relationship to the ordering customer. Today, the dominating data models of data-driven DSS are the relational data model [37] and its descendants that are tailored to analytical applications, for example, data warehouse models [61]. In fact, DSS data models were not changed in the last two decades.

The present dissertation examines to which extend the application of a graph data model can be useful to support decision-making. The remainder of the current chapter offers the reader a basic introduction to graph database models and graph data mining (Section 1.1) and will motivate the research goal (Section 1.2). Further on, it will state the dissertation's contributions including related publications (Section 1.3) and end with an overview of the dissertation's structure (Section 1.4).

1.1 Relational Data vs. Graph Data

The major difference between relational and graph databases is the handling of relationships. In the relational model, all data is stored in tables (relations). Figure 1.2a shows a relational database that stores data of a trade company such as sales orders, customers or products. Basically, there is one table for each class of data objects (e.g., CUSTOMER) and every row describes a single instance. The columns of a table represent attributes (e.g., CUSTOMER.name). All values of a column have the same data type. Further on, there are *keys* to mark attributes to represent identity or relationship information.

SalesOrder		
column	**type**	**PK**
country	varchar	1
id	int	2
created	date	
campaign_id	int	
created_by_id	int	
customer_id	int	
revenue	decimal	
foreign key	**columns**	**target**
campaign	[campaign_id]	Campaign
createdBy	[created_by_id]	Employee
customer	[country, customer_id]	Customer

Campaign		
column	**type**	**PK**
id	int	1
name	varchar	

Employee		
column	**type**	**PK**
id	int	1
name	varchar	
team	varchar	

Customer		
column	**type**	**PK**
country	varchar	1
id	int	2
name	varchar	
contact_id	int	
foreign key	**columns**	**target**
contact	[contact_id]	Employee

contains		
column	**type**	**PK**
country	varchar	
order_id	int	
product_id	int	
quantity	int	
foreign key	**columns**	**target**
order	[country, order_id]	SalesOrder
product	[product_id]	Product

Product		
column	**type**	**PK**
id	int	1
name	varchar	

Figure 1.1: Schema (Metadata) of a relational database: Every rectangle represents a table definition including the table name (e.g. SalesOrder) and all columns, their data type and primary key position (PK). Further on, each table contains foreign keys including their columns and target table.

SalesOrder						
country	**id**	**created**	**campaign_id**	**created_by_id**	**customer_id**	**revenue**
UK	1	2018-01-02	1	2	1	1,000.00
UK	2	2018-01-03	*NULL*	2	1	500.00

Campaign	
id	**name**
1	Eat apples!

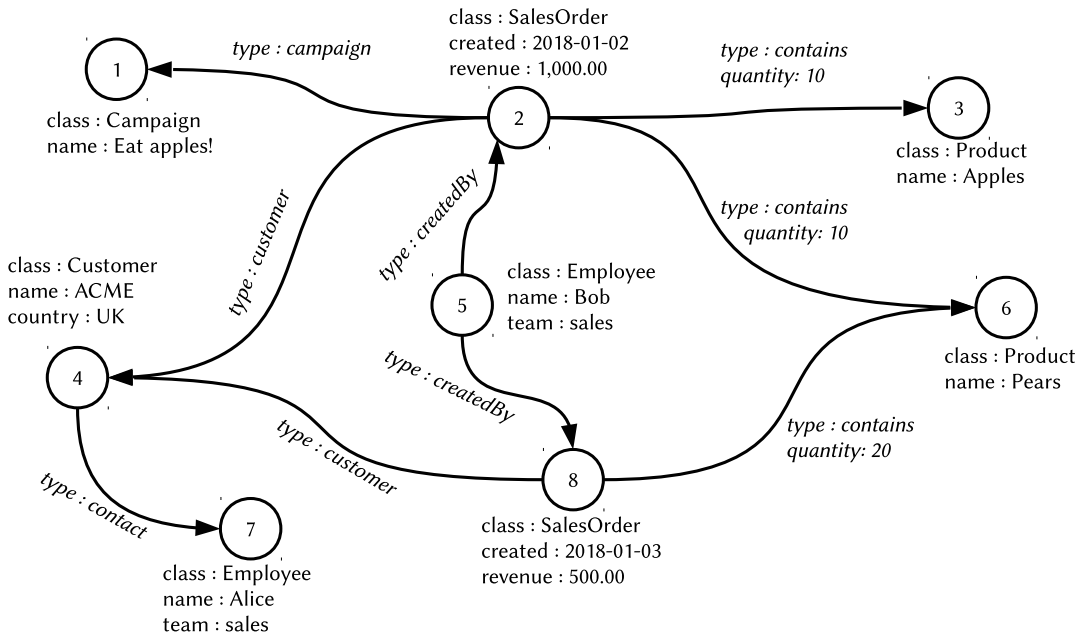
Employee		
id	**name**	**team**
1	Alice	Sales
2	Bob	Sales

Customer			
country	**id**	**name**	**contact_id**
UK	1	ACME	1

contains			
country	**order_id**	**product_id**	**quantity**
UK	1	1	10
UK	1	2	10
UK	2	2	20

Product	
id	**name**
1	Apples
2	Pears

(a) Sample relational database with the schema of Figure 1.1: Every rectangle represents a table with sample data.



(b) Sample graph database with the same data as the relational database of Figure 1.2a. Circles represent vertices with global identifiers and their connections represent edges. Arrowheads indicate an edge's direction. Classes, relationship types and attributes are shown by key-value pairs close to vertices and edges (property graph model). Primary keys of the relational database were not taken over.

Figure 1.2: Comparison of the same instance data in relational and graph representation.

For each relational database, there is a *schema* that describes all tables in terms of columns and keys. Figure 1.1 shows the schema for the database of Figure 1.2a. For each table there may be a *primary key* whose values identify objects. Primary keys may consist of one or more columns and any distinct combination of their values is unique per table, for example, according to Figure 1.2a a SalesOrder is identified by a $\langle \text{country\_id}, \text{id} \rangle$ pair. Further on, there are *foreign keys* whose values reference primary keys, for example, to store a customer reference for each sales order. Our example shows two possible constellations of foreign keys:

For relationship types where one party can be referenced only once, e.g., the employee who created a sales order, foreign keys are part of the referencing row. For example, values of SalesOrder.created_by_id reference values of Employee.id. This constellation is called *one-to-many (1:n)* since one employee may create multiple sales orders but every sales order was created by exactly one employee. Besides this, there is the so-called *many-to-many (m:n)* constellation. Here, both parties may be involved arbitrarily often. For example, the same product may be contained in multiple orders and vice versa. To store instances of this relationship type a dedicated table is required (e.g., contains). Further on, since relationships may also have attributes these tables may have additional columns (e.g, contains.quantity).

In a relational database relationships are implicit, i.e., the schema must be evaluated to decide whether a table represents a class of objects or a relationship type and whether a column stores basic attributes or keys. By contrast, in a graph database, relationships are explicit, so-called *first-level citizen*. Figure 1.2b shows the same data as Figure 1.2a in a graph format. Among different graph models [6], we choose the *property graph model* [157] for our running example due to its wide acceptance in both industry and academia [87]. In this model, domain objects are represented by *vertices* and relationships are represented by *edges*. Attributes may be attached to vertices and edges in the form of key-value pairs called *properties*. Typically, there are no constraints among property keys, property values and possible relationships, i.e., there is no schema.

The major advantage of graph databases over relational ones are additional query capabilities. A basic graph query operation is *traversing*. For example, a neighborhood traversal allows to access vertices that have a common edge with a given start vertex. An equivalent relational solution would require full schema knowledge to create a single query for every possible foreign key. While traversing a graph database leads to homogeneous results, for example, a collection of vertices or subgraphs, the relational pendant leads to one result per queried foreign key. If neighbor vertices are originated from different tables, they cannot be represented in a single result table due to different attributes and primary key spaces. The problem intensifies in the case of multi-hop traversals since every combination of n traversed vertices may have a different structure.

SalesOrder			
pattern_id	**id**	**created_by_id**	**customer_id**
1	1	1	1
2	1	1	1
3	1	*NULL*	*NULL*

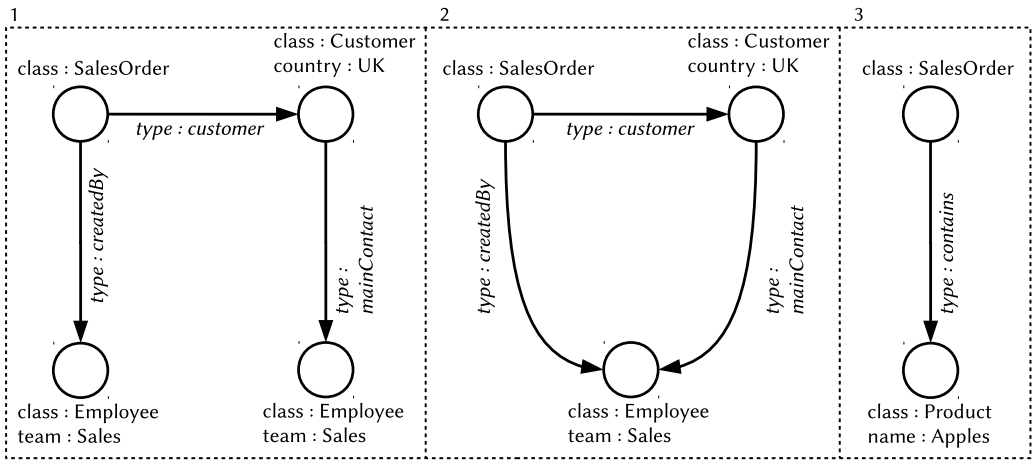
Employee		
pattern_id	**id**	**team**
1	1	Sales
1	2	Sales
2	1	Sales

Customer			
pattern_id	**id**	**country**	**mainContact_id**
1	1	UK	2
2	1	UK	1

contains		
pattern_id	**order_id**	**product_id**
3	1	1

Product		
pattern_id	**id**	**name**
3	1	Apples

(a) Patterns of Figure 1.3b in relational representation. Numbers above patters in Figure 1.3b correspond to pattern_id columns. All primary and foreign keys include pattern_id columns, e.g., SalesOrder.⟨pattern_id, customer_id⟩ references Customer.⟨pattern_id, id⟩.



(b) Collection of three graph patterns. Dotted lines represent pattern boundaries. Patterns 1 and 2 represent sales orders placed by a customer from the UK. In both patterns, the orders were created by employees of the sales team. Further on, in both cases customers have a contact persons from the sales team. However, in pattern 2 the sales order was created by the contact person, too, while the employees were distinct in pattern 1. Pattern 3 shows a sales order that contains apples.

Figure 1.3: Comparison of a pattern mining result in graph and relational representation. Both representations contain the same information.

The traversal operation already clarifies the advantage of explicit relationships and schema-freedom for exploratory queries. Another similar yet powerful operation is *graph pattern matching* [52]. This technique enables a user to identify subgraphs that are equal or similar to a given pattern. The other way around, graph models also allow the extraction of patterns. For example, an analyst may be interested in frequent relationship patterns of lossy sales orders. A solution based on a relational database is facing the same problems as those of traversal queries since relational patterns across tables must be determined by querying all possible schema fragments [46] and represented in multiple tables.

Let's take the example of a not further explained algorithm that returns 3 patterns of interest from a database with the schema of Figure 1.1. Let it be assumed that these patterns are heterogeneous with regard to quantity and structure of relationships. Figure 1.3 shows such patterns in both multi-relational (Figure 1.3a) and graph (Figure 1.3b) representations. The most obvious difference of both representations is the main criterion of data organization. While in Figure 1.3b every pattern is represented by a single graph and all results form a *graph collection*, in Figure 1.3a patterns are fragmented within a single set of tables. These tables are derived from the schema of Figure 1.1. More precisely, these tables cover all columns of all tables that match at least one pattern's data. Additionally, all primary keys have an additional pattern id component (e.g., CUSTOMER.pattern_id).

The obvious advantage of graphs for this scenario is a more intuitive result representation. The graph representation clearly shows the semantic meaning as well as the differences between the three patterns while the relational representation requires further interpretation. Besides the patterns-centric data organization graphs also directly show the difference between cyclic and acyclic relationship patterns (e.g., patterns 1 vs. pattern 2) due to the fact that relationships are first-level citizen. Further on, graph patterns show no absence of possible objects and relationships (e.g., NULL values in SALESORDER.customer_id). Especially for highly heterogeneous results, a relational representation might be confusing due to a dominance of NULL values.

However, these advantages are not solely an issue of result representation. Even data mining algorithms can already take advantage of schema freedom, explicit persistence of relationships and cycle detection, for example, the extraction on frequent patterns [1]. The most famous approach for relational data is frequent itemset mining [2] also known as shopping basket analysis. This approach determines frequent patterns of coexisting attribute values but completely ignores relationship structure. More elaborate approaches such as multi-relational mining [46] consider patterns across multiple tables but provide no mechanism to handle cyclic relationships. However, there are efficient graph-based algorithms that allow the extraction of patterns that include not only attributes but also arbitrary relationship structures [82].

1.2 Graphs for Business Intelligence

The research goal of this dissertation is a framework that enables decision-makers to benefit from the flexibility of graph-based data representation and from the novel capabilities of graph mining algorithms. Therefore, we started the development of *BIIIG (Business Intelligence with Integrated Instance Graphs).* BIIIG is a conceptual framework of a graph-based data-driven DSS [141]. We have chosen *Business intelligence (BI)* as an umbrella term that covers all technologies required by a respective system [33]. BIIIG provides solutions to different problems:

First, BIIIG should support the holistic analysis of data from multiple sources with arbitrary structures, e.g., data from relational databases or from web services that serve formats such as JSON and XML. Thus, BIIIG uses the property graph model which is very suitable for this task. Domain data objects are transformed into vertices, all relationships among them are reflected by edges and all attributes are directly attached to vertices and edges in form of properties. Further on, there is a data integration [107] strategy. For example, it is possible to add edges between vertices that represent the same conceptual entity (sameAs). Such edges can be added either on the instance level (object matching [99]) or on the schema level and either by an expert or by automated approaches [155]. Since these relationships are materialized by edges in a single graph, they can be used to analyze data across sources or to fuse vertices that correspond to each other.

BIIIG shall further allow the evaluation of patterns in a context, for example, patterns that occur frequently together with sales orders that lead to financial loss. Therefore, some established concepts from data warehousing [96] were adopted to the graph model. In a data warehouse measures of facts are evaluated by dimensional values. A *fact* is typically a tuple that represents a business transaction (e.g., a sales order) and whose fields are either measure values (e.g., profit) or foreign keys to dimensions (e.g., customers or products). Thus, it is possible to aggregate measures by dimensional values (e.g. total profit by customer country). BIIIG transfers those concepts to a graph model. To preserve relationship information BIIIG's facts are not tuples but graphs that represent, for example, a well defined neighborhood of sales order vertices. For the application domain of business information systems BIIIG even includes an algorithm to automatically extract such fact graphs called *business transaction graphs (BTGs)* [141].

Figure 1.4 shows an example collection of two BTGs. Both, dimensional values (normal font properties) and measure values (bold font properties) are attached to the graph structure. Common dimensional values of multiple graphs are reflected by shared vertices (e.g. apples). Since there is no schema, not only the actual property values (e.g., name: Apples) but also semantic information are considered as dimensional values (e.g., type: contains and class: Product). Further on, there

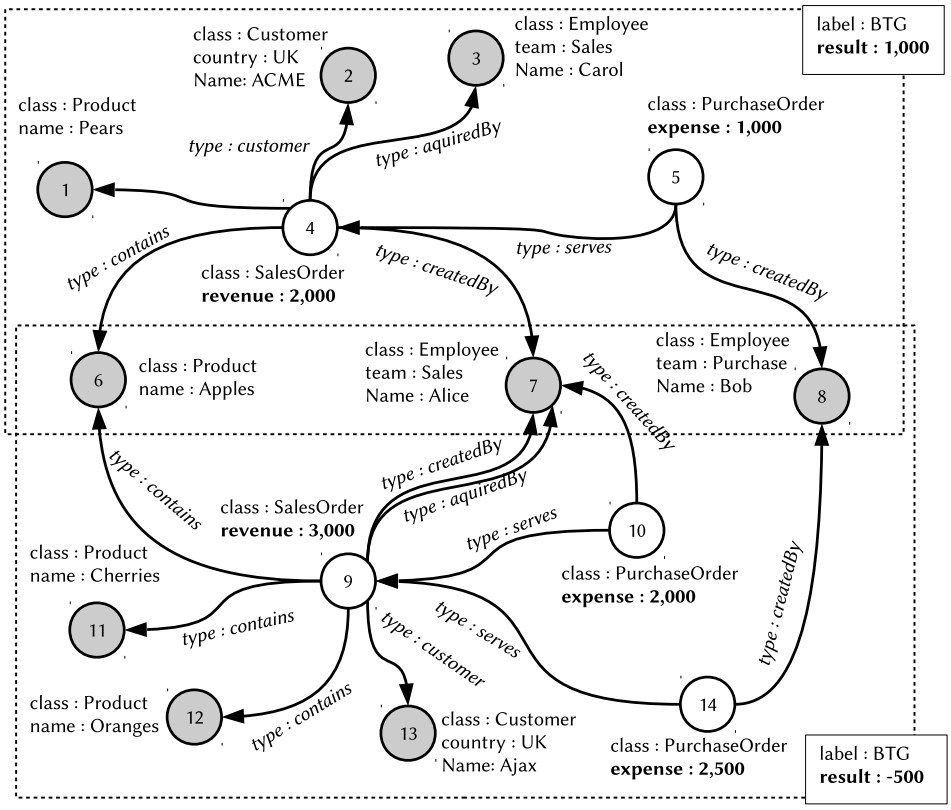


Figure 1.4: Example graph collection with shared vertices. Dotted lines represent graph boundaries. Solid rectangles overlaying the graphs' boundaries contain graph properties. Bold properties contain measure values and normal font ones dimensional values.

are aggregated measures (e.g., result: 1,000) attached to graphs[1]. These so-called *graph measures* allow filtering and categorizing graphs, for example, to select only lossy graphs. In consequence, a suitable graph model must support multiple graphs as first-level-citizen as well as collections of those. Further on, graphs must support properties. The existing property graph model lacks these concepts.

Before BIIIG, there was neither an academic graph database model [9] nor a productive graph database [87] with support for multiple attributed graphs (*graph collections*) and graph attributes. Because of this situation, we developed a new graph database model called *Extendend Property Graph Model (EPGM)*. As the name already implies, it is an extension to the existing property graph model [157]. In particular, its data structure adds support for graph collections and graph properties, so-called *property graph collections*. Beyond meeting the requirements of BIIIG, EPGM includes a set of operators that were majorly developed by Martin Junghanns in the context of the Gradoop framework [89]. Gradoop is a system for declarative graph analytics based on the distributed dataflow system Apache Flink [30].

To extract knowledge from property graph collections the determination of

[1]The measure of our example is calculated by
$\mathsf{BTG.result} \leftarrow \sum(\mathsf{SalesOrderLine.revenue}) - \sum(\mathsf{PurchaseOrder.expense})$.

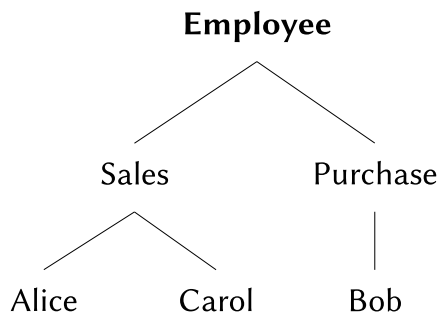


Figure 1.5: Example Taxonomy of dimension Employee.

pattern frequencies is a crucial primitive. With regard to graphs, this process is called *frequent subgraph mining (FSM)* [82]. A number of efficient FSM algorithms were developed in the 2000s [130, 190]. However, these algorithms were only applied to a single use case: chemical compounds. In graphs that represent compounds vertices reflect atoms and edges their bonds. However, chemical compounds and business information data show two fundamental differences: First, relationships in business data are mostly directed, i.e., there is a semantic difference between *A references B* and vice versa. In compounds, bonds have no direction. Second, business data may contain multiple relationships between two distinct entities. For example, the bottom graph of Figure 1.4 contains a pair of parallel edges that is expressing that employee Alice aquired and created a sales order. A graph with support for parallel edges is called *multigraph*. In compounds, any pair of atoms can only be connected by exactly one bond.

To the best of our knowledge, prior to BIIIG there was no frequent subgraph algorithm that was capable to extract patterns from directed multigraphs. To make a respective algorithm available, we followed a straightforward approach and extended existing techniques. However, the bare extraction of frequent patterns is still not sufficient to answer complex questions relevant to decision support. Thus, BIIIG includes further algorithmic and technical extensions. From the technical point of view, we proposed an efficient parallelization under consideration of the constraints of shared nothing clusters, i.e., following a *bring the computation to the data* approach. With this approach, FSM can also be applied to Big Data scenarios.

An important algorithmic aspect of frequent pattern mining refers to dimensional taxonomies. In a data warehouse [96] based on the multidimensional model, a special variant of the relational model, measures are associated to multiple dimensions. For example, sales can be evaluated by customer and employee information (2 dimensions). For some dimensions, the finest granularity (e.g., an employee with name Alice) can be attached to a *taxonomy* as shown by Figure 1.5. By the use of taxonomies the granularity of single dimensions can be generalized, for exam-

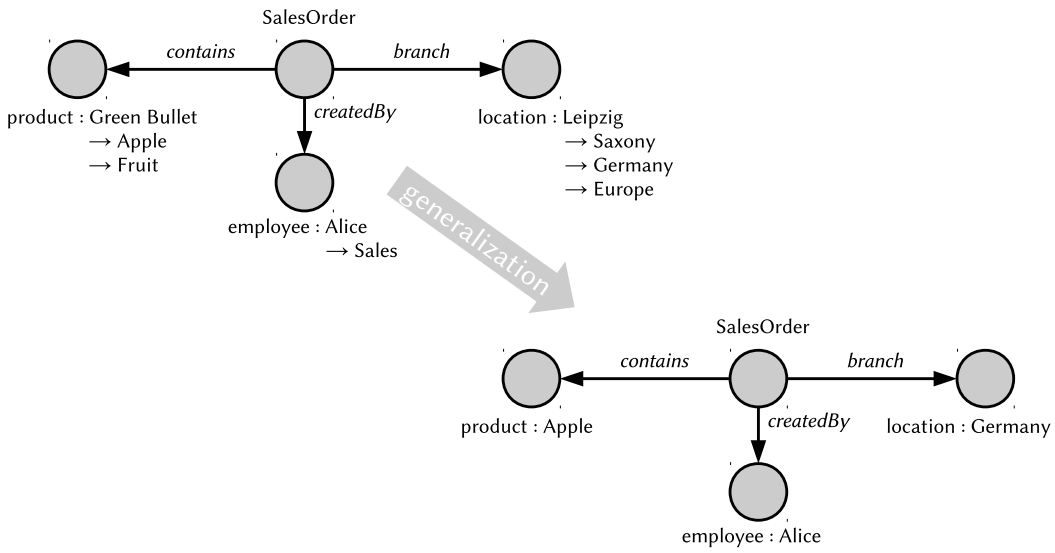


Figure 1.6: Example generalization of a graph pattern.

ple, to calculate the total revenue of all sales people instead of the one of Alice. The generalization of dimensional values has already been applied to frequent itemset mining [67]. For example, the pattern $\{\text{bread}, \text{butter}\}$ could be infrequent while the more general one $\{\text{bakery product}, \text{milk product}\}$ is frequent.

Generalization has also already been applied to graphs (chemical compounds) [76], but without support for directed multigraphs and under the assumption that all vertices belong to the same dimension. However, the data of most applications involves multiple dimensions, for example, to find out that $\{\text{bread}, \text{butter}\}$ is mostly bought in the morning in suburban stores. With BIIIG, we enabled generalized subgraph mining with support for multiple dimensions. Therefore, dedicated vertex properties are attached to taxonomies and generalized during the mining process [145]. For example, the left graph of Figure 1.6 shows a pattern with four dimensions while the right graph shows one example of its generalizations at different levels for each dimension. In particular, it shows that Alice sold apples to Germany.

The remainder of the present dissertation will discuss the contributions of BIIIG in more detail. Although there is no software system that includes all BIIIG components yet, there are prototypes and experimental evaluations for all of its building blocks. Most of the prototypes were turned into Open Source software and made available to the public. Due to the limited availability of real word business data, most evaluation were done using synthetic data. Therefore, a dedicated data generator based on business process simulation [142] was developed. However, parts of BIIIG have been evaluated in real-world scenarios in cooperation with Siemens AG and Immowelt AG.

1.3 Contributions and Publications

This section provides an overview of the contributions made in the context of this dissertation. All of them were peer-reviewed and published in journals or proceedings of conferences and workshops. Additionally, a summary of most contributions was presented on the VLDB 2017 PhD workshop [138].

Comparison of recent graph database systems: Before the development of BIIIG started, both academic and commercial graph database systems were systematically compared with regard to their suitability. In chapter two of *Management and Analysis of Big Graph Data: Current Systems and Open Challenges* [87], published in 2017, we provide an in-depth comparison of their data models and analytical capabilities.

Graph-based Data Transformation and Integration: Although a typical company's business data implicitly describes a graph it is usually stored in business information systems based on relational databases. Further on, companies often use multiple of these systems and there are implicit cross-system relationships. To the best of our knowledge, we proposed the first semi-automated approach to transform data from multiple relational databases into a single graph whose vertices represent domain objects and whose edges represent their mutual relationships.

We further proposed a graph-based approach to data integration. The process is executed after the transformation and consists of two steps. First, correspondence edges among vertices of different graphs are added and, second, clusters of vertices that correspond to each other are fused into a single vertex. The approach was initially proposed on the GDM 2014 workshop and published within the ICDM 2014 workshop proceedings [141]. Further on, an initial prototype was demonstrated on the VLDB 2014 conference [143].

Collections of Property Graphs: In established data mining approaches interrelated input data is mostly represented by tuples of measure values and dimension values. In the context of graphs these values must be attached to the graph structure and aggregated measure values are graph attributes. Since the latter was not supported by any existing model, we proposed the use of *collections of property graphs*. They act as data structure of the novel *Extended Property Graph Model (EPGM)*. The model supports vertices and edges that may appear in different graphs as well as graph properties. Further on, we proposed some operators that benefit from this data structure, for example, graph-based aggregation of measure values. The initial idea was already presented on the GDM 2014 workshop [141]. A more elaborate introduction to the concept of EPGM appeared on the NDA 2016 workshop, co-located with the SIGMOD 2016 conference [89].

FoodBroker Data Generator: Since real business data is not available to the public, we designed a simulation-based generator for data that shows the characteristics of data from business information systems. The structure of FoodBroker graphs is based on a statistical model and contains correlations between single vertices and the surrounding graph structure. Thus, it can be used in experimental evaluations of analytical techniques. FoodBroker was initally| published within the proceedings of the WBDB 2014 workshop [142].

Frequent Pattern Mining for Directed Multigraphs: An primitive operation of graph pattern mining is frequent subgraph mining (FSM). However, existing algorithms provided neither support for directed graphs nor those containing multiple edges between a pair of vertices. With the development of *Directed Multigraph gSpan (DMGSpan)*, a respective extension of the popular gSpan algorithm [196], we proposed an algorithm without these limitations. The extension of gSpan to support directed multigraph was initially published in the context of a Gradoop Demonstration on the ICDM 2016 conference [140].

Distributed Frequent Subgraph Mining: Parts of this work were done in the context of Gradoop, a framework for distributed graph analytics [85]. To make the primitive operation of frequent subgraph mining available to this framework, we developed *Distributed In-Memory gSpan (DIMSpan)*, a variant of DMGSpan tailored to characteristics of shared-nothing clusters and distributed in-memory dataflow systems. Details about DIMSpan were published in the proceedings of the BDCAT 2017 conference [144].

Generalized Frequent Subgraph Mining: Some patterns might not be frequent while their generalizations are. Generalized graph patterns can be mined by attaching vertices to taxonomies. We proposed a novel approach to *Generalized Multidimensional Frequent Subgraph Mining (GM-FSM)*, in particular the first solution to generalized FSM that supports not only directed multigraphs but also multiple dimensional taxonomies. The work was presented on the ICDIM 2017 conference [145].

Characteristic Subgraph Mining (CSM): In scenarios that compare patterns of different categories, e.g., fraud or not, FSM is not sufficient since pattern frequencies may differ by category. Further on, determining all pattern frequencies without frequency pruning is not an option due to the computational complexity of FSM. Thus, we developed an FSM extension to extract patterns that are characteristic for a specific category according to a user-defined interestingness function. The method was already described and published in journal article that appeared in 2016 in *it - information technology* [139].

Open Source Implementations: All data mining algorithms developed during the research that lead to this dissertation are available as Open Source software. Thread-parallel implementations of DMGSpan, GM-FSM and CSM based on both algorithms are available as part of *Directed Multigraph Miner*[2]. DIMSpan and FoodBroker are part of the Gradoop[3] framework. Further on, there is a single-machine version of FoodBroker [4].

1.4 Structure of Dissertation

The remainder of this dissertation consists of the following seven chapters:

Chapter 2 provides the reader with background knowledge relevant to this dissertation. It starts with a formal introduction of all graph structures mentioned and used in the remainder of this work. Afterwards, it will provide a technical introduction into different relevant software systems such as recent graph databases. Finally, there will be a discussion of related work in the fields of relational-to-graph data transformation, graph-based data warehousing, graph pattern mining and graph data generators.

Chapter 3 will introduce our analytical framework, in particular its basic concepts and all relevant terminology such as measures, dimensions and business transaction graphs. It will further present an analytical method that is capable to extract patterns that correlate with business measures, called characteristic subgraph mining. We will use the method together with an example scenario to further motivate BIIIG's application and list the requirements that lead to BIIIG's development.

Chapter 4 will study the Extended Property Graph Model (EPGM) which supports BIIIG's data model requirements such as the representation of business transaction graphs as well as all relevant operations. It will also take a look at FoodBroker, a data generator for data that contains business transaction graphs, and Gradoop, the first system that implements EPGM.

Chapter 5 is dedicated to graph-based transformation and integration of business data, the foundation of the BIIIG approach. After an introduction of its specific data model, all relevant techniques will be discussed in more detail. Finally, the results of an experimental evaluation with data originated from a real business information system will be presented.

[2] `https://github.com/p3et/dmgm`
[3] `https://github.com/dbs-leipzig/gradoop`
[4] `https://github.com/dbs-leipzig/foodbroker`

Chapter 6 studies the core problem of frequent subgraph mining (FSM). After a review of existing methods we will discuss our extensions to support directed multigraphs. Additionally, we will discuss the parallelization of FSM in the context of multi-core CPUs and in particular distributed dataflow frameworks for Big Data processing. Due to their programming paradigms both approaches to parallelization differ fundamentally. Finally, results of an experimental evaluation will show the scalability of our approach.

Chapter 7 studies the problem of generalized frequent subgraph mining. Multiple approaches to solve this problem will be introduced and discussed. The most elaborate algorithm decomposes the problem into frequent subgraph mining of most relevant generalizations and mines all frequent specializations by a less costly approach to vector mining. The results of an experimental evaluation will confirm the effectiveness of this approach.

Chapter 8 concludes the dissertation. First, the results of use case evaluations in cooperation with a large scale enterprise will be presented. This includes a report of practical experiences gained in implementation and application of the proposed algorithms. Finally, after a summary of the previous chapters it will name open research questions based on the outcomes of this dissertation.

Chapter 2

Background and Related Work

This chapter will provide background knowledge that will be useful to understand the remainder of this dissertation. Further on, it will discuss related work.

We will start with an overview of graph data structures (Section 2.1) and, due to to their importance, a detailed comparison of the two most popular graph models (Section 2.2). Nowadays, business intelligence is often related to Big Data technologies. Thus, we will provide an introduction to the programming model of distributed dataflow systems (Section 2.3). Since this dissertation is about graphs, we will also discuss recent graph-related software systems and their analytical capabilities. Due to their fundamental differences, we will do this separately for graph processing systems (Section 2.4) and graph database systems (Section 2.5).

Since BIIIG includes multiple aspects of business intelligence we will also study different fields of related work. The first one is graph-based data transformation and integration (Section 2.6.2). Afterwards, we will study two fields related to data analytics, in particular graph-based OLAP (Section 2.7) and graph pattern mining (Section 2.8). Since we also developed a data generator, we will also cover this field (Section 2.9).

2.1 Graph Data Structures

In this section, all characteristics of graph structures mentioned or used in the remainder of this dissertation will be precisely defined. We will use a common formalism of elements, sets of these and functions. This may appear unfamiliar to some readers as it differs from established formalisms for specific structures. However, the formalism was chosen to ensure comparability and to facilitate understanding of more complex definitions later on in this dissertation. Tables 2.1 and 2.2 provide a glossary of all used symbols.

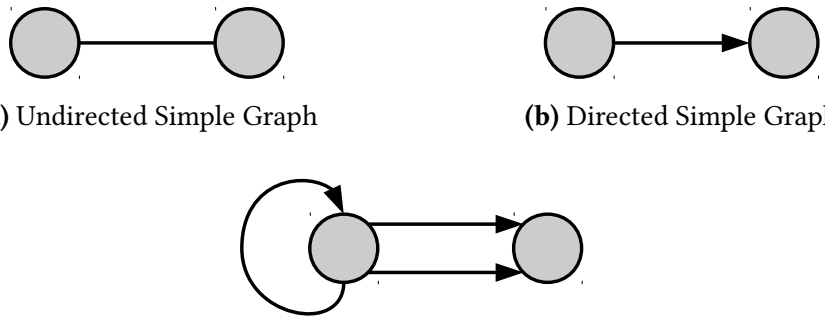
Basically, a graph [44] is a set of identifiers (*vertices*) and connections among them (*edges*). In a simple graph edges have no explicit identifier. Figure 2.1a shows an undirected simple graph.

Symbol	Meaning	Examples
α	undirected adjacency	$\alpha(v_1) = \{v_2, v_3\}$
α_o	outgoing adjacency	$\alpha_o(v_1) = \{v_2, v_3\}$
α_i	incoming adjacency	$\alpha_i(v_1) = \{v_2, v_3\}$
β	bottom level label	$\beta(\mathsf{Employee}) = false$
Γ_f^k	Graph aggregation operator	$C' = \Gamma_{\lvert V_g \rvert}^{\mathsf{vertexCount}}(C)$
γ	graph containment	$\gamma(v) = \{g_1, g_2\}$
ε	empty element (NULL)	$\pi(v, \mathsf{age}) = \varepsilon$
ζ	vertex taxonomy association	$\zeta(v) = T$
η	label parent	$\eta(\mathsf{Employee.Alice}) = \mathsf{Employee}$
ι_v/ι_e	vertex / edge bijection	$\iota_v : \{v_1, v_2\} \leftrightarrow \{v_3, v_4\}$
κ	edge time bijection	$\kappa(2) = e$
Λ	label property key	$\pi(v, \Lambda) = \mathsf{User}$, $\pi(e, \Lambda) = \mathsf{friendOf}$
λ	labeling function	$\lambda(v) = \mathsf{User}$, $\lambda(e) = \mathsf{friendOf}$
μ	embedding map	$\mu(g, p) \mapsto \{m_1, m_2\}$
ν	vertex time bijection	$\nu(2) = v$
Π_f	Property transformation op.	$C' = \Pi_{\forall v \in V.\pi'(v,\mathsf{label})=\pi(x,\mathsf{name})}(C)$
π	property function	$\pi(v, \mathsf{age}) = 42$, $\pi(e, \mathsf{weight}) = 0.7$
ρ	taxonomy path function	$\rho(\mathsf{A.A.B}) = \{\mathsf{A}, \mathsf{A.A}, \mathsf{A.A.B}\}$
Σ_f	Graph selection operator	$C' = \Sigma_{\pi(g,\mathsf{vertexCount}) \geq 3}(C)$
ς	source of an edge	$\varsigma(e) = v$
τ	target of an edge	$\tau(e) = v$
ϕ	absolute support	$\phi(G, p) = 42$
ϕ^{rel}	relative support	$\phi^{rel}(G, p) = 0.7$
ϕ_{min}	minimum support threshold	$\phi_{min} = 42$
ϕ_{min}^{rel}	rel. min. support threshold	$\phi_{min}^{rel} = 0.7$
ψ	hyperedge function	$\psi(e) = \{v_1, v_2, v_3\}$
Ω	EPGM Operator	$C' = \Omega(C)$
ω	top level label	$\omega(\mathsf{Employee}) = true$

Table 2.1: Glossary of symbols part 1 (greek letters)

Symbol	Meaning	Examples
A_b	element of heterogeneous tuple	$g = \langle V_g, E_g \rangle$
C	property graph collection	see Chapter 4
d/D	property value / set of	$D = \{"Alice", 42, true\}$
e/E	edge / set of	$E = \{e_1, e_2, \ldots, e_n\}$
g/G	graph / set (collection) of	$G = \{g_1, g_2, \ldots, g_n\}$
k/K	edge discovery time / set of	$K = \{1, 2, 3, 4\}$
$\ell/L/\mathbb{L}$	label / set of / space of	$L = \{\mathsf{User}, \mathsf{friendOf}\}$
$\vec{\ell}/\mathcal{L}$	vector / set of	see Definition 7.11
$<_T$	label generalization	Employee $<_T$ Employee.Alice
$<_\mathcal{T}$	graph generalization	$g_1 <_\mathcal{T} g_2$
$\vec{<}_\mathcal{T}$	vector generalization	$\vec{\ell_1} \vec{<}_\mathcal{T} \vec{\ell_2}$
m	embedding	see Definition 6.5
p	pattern	$p \simeq s = true$
$\simeq$	isomorphism relationship	$p \simeq s = true$
s	subgraph	$s \sqsubseteq g = true$
$\sqsubseteq$	subgraph relationship	$s \sqsubseteq g = true$
$\overset{\bullet}{\sqsubseteq}$	child of (graph)	$c \overset{\bullet}{\sqsubseteq} p = true$
$T/\mathcal{T}/\mathbb{T}$	taxonomy / set of / space of	see Definition 7.1
t	time	t = 5s
v/V	vertex / set of	$V = \{v_1, v_2, \ldots, v_n\}$
$\overline{v}$	path	$\overline{v} = \langle v_1, v_3, v_4, v_2 \rangle$
x	DFS extension	$x = \langle 0, 1, \mathsf{from}, true, \mathsf{edge}, \mathsf{to} \rangle$
X	DFS Code	$X = \langle x_1, \ldots, x_n \rangle$
u/U	vertex discovery time / set of	$U = \{1, 2, 3, 4\}$
y/Y	property key / set of	$Y = \{\mathsf{age}, \mathsf{weight}, \Lambda\}$

Table 2.2: Glossary of symbols part 2 (further symbols)



(a) Undirected Simple Graph

(b) Directed Simple Graph

(c) Directed Multigraph

Figure 2.1: Graph structures with different edge characteristics.

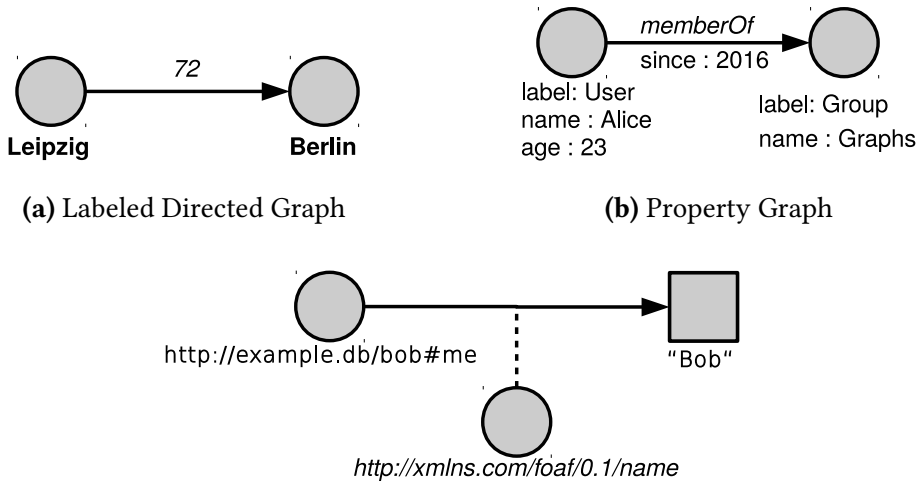
Definition 2.1 (Undirected Simple Graph) An undirected simple graph is defined as a pair $g = \langle V, \alpha \rangle$ of a vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and an adjacency function $\alpha : V \to \mathcal{P}(V)$ that connects a subset of other vertices to every vertex s.t. $v \notin \alpha(v)$ and $v_1, v_2 \in V.\Big(v_1 \in \alpha(v_2) \Leftrightarrow v_2 \in \alpha(v_1)\Big)$. Thus, an edge can implicitly be defined as a 2-element set of vertices $\{v_1, v_2 \mid v_2 \in \alpha(v_1)\}$.

Depending on the semantic meaning of vertices and edges, simple undirected graphs can be used to represent data of different scenarios. For example, for a social network vertices may represent user identifiers and edges may represent their mutual friendships. However, there are also scenarios where an edge has a different meaning depending on its direction. A graph structure that distinguishes between the start point (*source*) and end point (*target*) of an edge is denoted by the attribute *directed* [44]. Figure 2.1b shows a directed simple graph.

Definition 2.2 (Directed Simple Graph) A directed simple graph is defined as a triple $g = \langle V, \alpha_o, \alpha_i \rangle$ which, in contrast to a undirected graph, contains two separate adjacency functions describing outgoing $\alpha_o : V \to \mathcal{P}(V)$ and incoming connections $\alpha_i : V \to \mathcal{P}(V)$ s.t. $v \notin \alpha_o(v); v \notin \alpha_i(v)$ and $v_1, v_2 \in V.\Big(v_1 \in \alpha_o(v_2) \Leftrightarrow v_2 \in \alpha_i(v_1)\Big)$. Thus, an edge can be considered as a pair $\langle v_1, v_2 \mid v_2 \in \alpha_o(v_1) \wedge v_1 \in \iota(v_2) \rangle$.

Besides edge direction, the quantity of edges between a pair of vertices may carry semantics, for example, different roads (edges) may connect the same pair of cities (vertices) in a road network. So-called *multigraphs* [44] support an arbitrary number of edges between any pair of vertices as well as connecting a vertex with itself. Edges of the latter type are called *loop*. Figure 2.1c shows a directed multigraph including a loop and a pair of parallel edges.

Definition 2.3 (Directed Multigraph) A directed multigraph is defined as a quadruple $g = \langle V, E, \varsigma, \tau \rangle$ of vertex identifiers $V = \{v_1, v_2, \ldots, v_n\}$, edge identifiers $E = \{e_1, e_2, \ldots, e_m\}$ as well as two functions mapping a source vertex $\varsigma : E \to V$ and a target vertex $\tau : E \to V$ to every edge.



(a) Labeled Directed Graph

(b) Property Graph

(c) RDF Triple Graph

Figure 2.2: Graph structures with different attached data formats.

A further way of adding semantics to a graph is attaching data values to vertices and edges. The simplest form of data values are so-called *labels*. Figure 2.1b shows a labeled graph where vertex labels represent cities and edge labels express a weight (minimal train connection time in minutes).

Definition 2.4 (Labeled Directed Multigraph) A labeled directed multigraph is defined as a sextuple $g = \langle V, E, \varsigma, \tau, L, \lambda \rangle$ containing, in addition to Definition 2.3, a set of labels $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$ and a function $\lambda : (V \cup E) \rightarrow L$ assigning a label to every vertex and edge.

Labels can be used to distinguish types, e.g., users and groups in a social network, or to add specific data values, e.g., edge weights. However, especially when graph structures are used as part of database models [9] simple labels are not sufficient. Thus, two popular extensions of labeled directed multigraphs arose in this context. The first so-called *property graph model* [157] supports attaching an arbitrary number of key-value pairs to the graph structure. Figure 2.2b shows a property graph whose vertices and edges show different properties.

Definition 2.5 (Property Graph) A property graph is defined as a septuple $g = \langle V, E, \varsigma, \tau, Y, D, \pi \rangle$ containing, in addition to Definition 2.3, a set of property keys $Y = \{\Lambda, y_0, y_1, \ldots, y_n\}$, a set of data values $D = \{\varepsilon, d_0, d_1, \ldots, d_n\}$ and a function $\pi : (V \cup E) \times Y \rightarrow D$ that maps a data value to every combination of vertex or edge and property key. D contains the empty value ε to express the nonexistence of a data value for a given combination. Further on, K may contain a dedicated label symbol Λ where $\forall x \in (V \cup E).\pi(x, \Lambda) \neq \varepsilon$ s.t. the property graph is implicitly labeled by a mandatory label property.

Besides property graphs, there is a second graph model that gained wide acceptance and application within the research community. The *Resource Description*

Framework (RDF) [238] allows the description of graphs by so-called *International Resource Identifiers (IRI)* as vertices and statements about these as edges. Figure 2.2c shows an example RDF graph. Note, the dotted line does not represent an edge but shall express that every edge label is actually a vertex (IRI).

Definition 2.6 (RDF Triple Graph) A RDF triple graph is defined as a sextuple $g = \langle V, D, E, \varsigma, \lambda_o, \lambda_p \rangle$ of a set of IRIs $V = \{v_1, v_2, \ldots, v_n\}$, a set of literals $D = \{d_1, d_2, \ldots, d_m\}$ and a set of triples $E = \{e_1, e_2, \ldots, e_p\}$. Each triple has an assigned subject $\varsigma : E \rightarrow (V \cup E)$, object $\lambda_o : E \rightarrow V$ and predicate $\lambda_p : E \rightarrow (V \cup D)$.

With regard to the structural properties introduced in Definitions 2.3 to 2.5, RDF triple graphs are edge-labeled directed multigraphs with two kinds of vertices - IRIs and literals. Edge sources correspond to subjects and edge targets to objects. However, the model additionally supports edges whose sources are edges. While literals correspond to data values of property graphs, IRIs represent not only vertices but also edge labels. Thus, there is no distinction between these two concepts on the structural level. Properties are expressed by edges with a literal predicate. To support collections of RDF triple graphs, there is a extension called N-quads [239]:

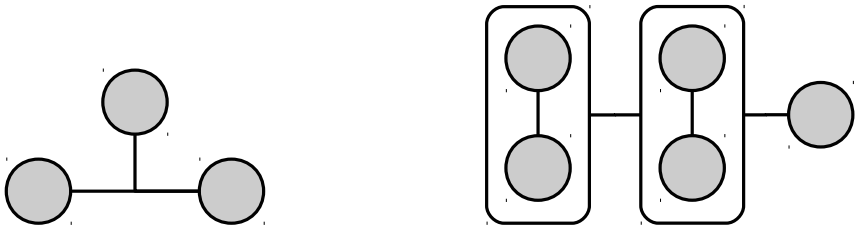
Definition 2.7 (RDF N-quad Graph Collection) A RDF N-quad graph collection is defined as a septuple $g = \langle V, D, E, \varsigma, \lambda_o, \lambda_p, \gamma \rangle$ containing, in addition to Definition 2.6, an additional graph containment function $\gamma : E \rightarrow V$ mapping every edge to a single IRI. Because of this fourth function, edges E are called quads instead of triples.

In comparison to property graph collections according to Definition 4.1, RDF collections have no explicit mapping of vertices to graphs and edges are assigned to exactly one graph. Graphs are identified by IRIs and, thus, there is also support for graph properties. Since the same IRI may be subject or object in edges of different graphs, it is possible to represent graphs with overlapping vertices. However, since properties are edges they are only valid in the context of a specific graph.

For the sake of completeness there are two further graph structures worth mentioning although neither relevant for the remainder of this dissertation nor widely applied. First, there are graphs supporting n-ary edges [78]:

Definition 2.8 (Hypergraph) A hypergraph is defined as a triple $g = \langle V, E, \psi \rangle$ of vertex identifiers V and edge identifiers E analogously to a multigraph (Definition 2.3) but with a specific function $\psi : E \rightarrow \mathcal{P}(V)$ that maps an arbitrary number of vertices to every edge.

Figure 2.3a shows a hypergraph with a 3-ary edge. Second, there are graphs whose vertices may nest graphs [150]:



(a) Hypergraph **(b)** Graph with Hypervertices

Figure 2.3: Hypergraphs and hypervertices.

Definition 2.9 (Hypervertex Graph) A hypervertex graph is any graph according to Definitions 2.1 to 2.5 whose vertices may be graphs themselves. Vertices will be called *hypervertices* if they are non-empty graphs and *simple vertices* otherwise.

Figure 2.3b shows an undirected simple graph with two hypervertices.

2.2 Comparison of RDF and Property Graphs

There are two graph data models that gained extraordinary interest by both industry and academia: Resource Description Framework (RDF) and the Property Graph Model (PGM). During the research that lead to this dissertation we were often asked why we favored the latter. To answer this question, this section will provide an in-depth comparison of both model with regard to their role as a database model.

2.2.1 Resource Description Framework

In its core, RDF is an machine-readable data exchange format consisting of $\langle \text{subject}, \text{predicate}, \text{object} \rangle$ triples. Considering subjects and objects as vertices and triples as edges, a dataset consisting of such triples forms a labeled directed multigraph (Definition 2.6). Labels are either internationalized resource identifiers (IRIs), literals like numbers and strings or so-called *blank nodes* (blank vertices). The latter is used to reflect vertices that are not representing an actual resource. In the following, we will use the format `:Identifier` to represent IRIs. There are domain constraints depending on the triple position. Subjects are either IRIs or blank nodes, predicates must be IRIs and objects may be IRIs, literals or blank nodes. In contrast to other graph models, RDF also allows edges between edges and vertices, which can be used to add schema information to the graph. For example, the type of an edge `:alice,:knows,:bob` can be further qualified by another edge `:knows,:isA,:Relationship`. A schema describing an RDF database is a further RDF graph of metadata and is often referred to as *ontology* [222]. RDF is most popular in the context of the semantic web where its major strengths are standardization, the availability of web knowledge bases to flexibly enrich user databases and the resulting reasoning capabilities over linked RDF data [186].

2.2.2 Property Graph Model

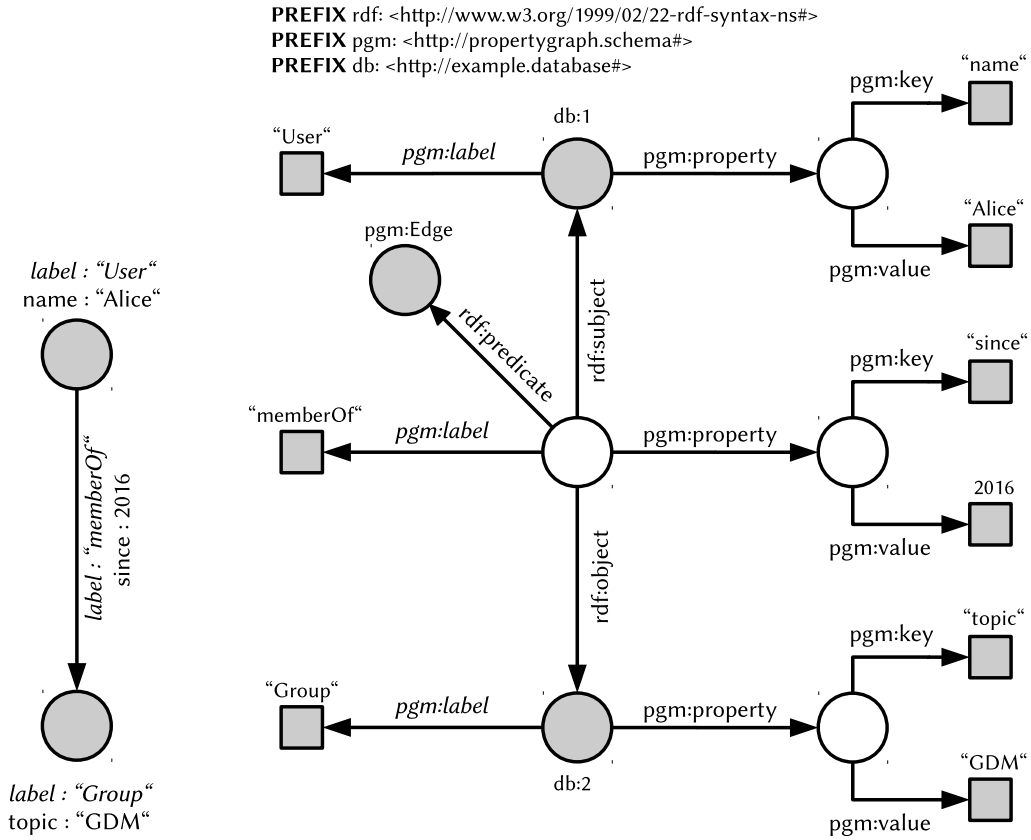
While RDF is well discussed in research literature, PGM and the de facto standard Apache TinkerPop [242] found by far lower interest. However, many commercial graph database products use TinkerPop and the popularity ranking of May 2018 from DB-Engines[1] shows a nearly twenty times higher score for the most popular PGM database (Neo4j, 40.58 [218]) than for the most popular dedicated RDF store (Jena, 2.32 [219]). As DB-Engines, among other criteria [220], evaluates the presence of databases in the web and social networks, the score indicates a high relevance of PGM for productive graph data management. In Section 2.5, we will present a comparison of current graph database systems. With one exception, all of the considered PGM databases support TinkerPop. The TinkerPop property graph model describes a directed edge-labeled multigraph with properties for vertices and edges. Some systems also require vertices to be labeled, i.e., to have a dedicated label property. This is used for different purposes. For example, Sparksee uses labels strictly to represent vertex and edges types and requires a fixed schema for all of its instances. Other systems such as ArangoDB manage schema-less graphs, i.e., labels may indicate types but may be coupled with arbitrary properties at the same time. In most graph databases a schema is optional.

The best comparison of both models can be done by the discussion of representing RDF using PGM and vice versa. At first glance, PGM subsumes RDF. However, to evaluate edge labels in the context of external ontologies, a distinct model feature of RDF is its capability to use the same IRI as predicate and as subject or object, i.e., the support for edges that connect edges and vertices. The PGM is not designed to cover such application scenarios. The other way around, i.e.,to represent a property graph using RDF, we identified three major problems that will be discussed next.

2.2.3 Different Roles of Edges

In PGM, every edge represents a logical relationship. By contrast, RDF edges (triples) have two roles: First, they are used to represent actual resource relationships (e.g., `db:alice, schema:knows, db:bob`). Second, they are used for technical associations such as vertex properties (e.g., `db:alice, schema:name, "Alice"`). In consequence, every PGM vertex is expressed by n triples (one for each of n properties). In consequence, the interpretation of RDF will always require schema knowledge to process the graph data correctly, i.e., to decide if an edge is a logical relationship or something else. To speedup traversals of actual relationships some dedicated RDF databases hold properties separated from vertices. However, by this approach the retrieval of vertices with all of their properties requires a potentially expensive join operation. We consider the necessity for this trade-off as the first disadvantage of RDF.

[1] `https://db-engines.com/en/ranking/graph+dbms`



(a) Property Graph. **(b)** Schema-less RDF representation. Gray circles represent vertices, white circles blank nodes and gray squares literals.

Figure 2.4: Comparison of an example property graph with mandatory label properties (italic font) and its schema-less RDF representation by standard reification.

2.2.4 Edge Properties

Edges in RDF (triples) must have exactly one IRI label (predicate). However, this IRI is not an edge identifier but a reference to its semantic type. Adding attributes to an edge requires *reification*, i.e., representing a single logical relationship by multiple triples. In the standard way [240], edges are represented by blank nodes. For example, the logical relationship `db:alice,schema:knows,db:bob` can be represented by a blank node `_:bn` and separate edges to express subject, object and predicate (e.g., `_:bn,rdf:subject,db:alice`). Properties are expressed analogously to vertices (e.g. `_:bn,schema:since,2016`). In consequence, every PGM edge is expressed by $3 + m$ triples, where m is the number of properties. Two graph databases of Section 2.5 store property graphs using RDF. However, both are using alternative, non-standard ways of reification. Stardog is using n-quads [239] for PGM edge reification. N-quads extended triples by a fourth position that stores an IRI to identify a graph (Definition 2.7). Applied to reification every edge is represented by a graph. This one and other approaches to reification are discussed in [40]. Addtionally, Blazegraph follows a further, non-standard approach to reification and implements custom RDF extensions [70].

We consider the necessity for reification as the second disadvantage of RDF. In particular, the standard way adds many technical edges which is not only increasing data volume but also leads to an larger number of edges that need to be processed at the traversal of logical relationships. Further on, the problem seems to be neglected in popular RDF literature and despite the high standardization efforts of RDF there seems to be no widely accepted approach to represent edge properties.

2.2.5 Schema-less Data

Property graphs can be schema-less, i.e., property keys are not defined by a schema. PGM data is often referred to as *self-descriptive* as metadata about type labels and properties is retrievable directly from vertices and edges. Some database systems of Section 2.5 support or even require vertex and/or edge labels which could be interpreted as a partial schema. However, we will consider such data as schema-less, too, as long as there are no constrains between labels and property keys. Figure 2.4 shows a small property graph with a partial schema (vertex and edge labels) next to an equivalent schema-less RDF triple representation where type labels and property keys are literals. In PGM, type labels as well as property keys and values are logically embedded in vertices and edges. Native PGM implementations also use a respective storage implementation.

In contrast, an equivalent RDF representation requires reification not only for edges but also for properties. While edge reification is required to support edge properties, property reification is originated by the restriction of subjects to be IRIs, i.e., a string literal may not be used as an edge label. For the same reason, even a mandatory edge label require an additional triple. In the shown standard-compliant solution, every vertex is represented by $1+3n$ (1 label and n properties) and every edge by $4+3m$ (subject, object, predicate, label and m properties) triples. Without a partial schema even vertex and edge labels would require 3 triples each. Since we target a schema-less approach of data transformation (Chapter 5), we consider the massive overhead of technical edges that is required to represent schema-less data by RDF as a third disadvantage.

2.2.6 Summary

RDF is capable to represent a property graph but not vice versa, i.e., RDF has greater expressive power. Further on, RDF shows significant advantages for the integration of web knowledge bases and other semantic applications since there is a standardized data format. However, expressing (in particular schema-less) property graphs using RDF leads to a rather voluminous representation and might cause a poorer traversal performance as many technical edges must be evaluated in addition to the logical ones. Thus, for all applications that neither need RDF's

Transf.	Signature	Constraints
single element transformations		
Filter	$I, O \subseteq A$	$O \subseteq I$
Map	$I \subseteq A, O \subseteq B$	$\lvert I\rvert = \lvert O\rvert$
Flatmap	$I \subseteq A, O \subseteq B$	-
MRMap	$I \subseteq A \times B; O \subseteq C \times D$	-
element group transformations		
Reduce	$I, O \subseteq A \times B$	$\lvert I\rvert \geq \lvert O\rvert \wedge \lvert O\rvert \leq \lvert A\rvert$
Combine	$I, O \subseteq A \times B$	$\lvert I\rvert \geq \lvert O\rvert \wedge \lvert O\rvert \leq \lvert A \times W\rvert$

Table 2.3: Selected Unary Transformations. I represents the input and O represents the output data set. $A..D$ are distinct data spaces and W is a set of worker threads.

expressiveness nor web data exchange but require edge properties, support for schema-less data and efficient traversal operations, the property graph model is favorable. Since the latter criteria are not random but apply to BIIIG, we decided to prefer PGM over RDF.

2.3 Distributed Dataflow Systems

Nowdays, business intelligence is often connected to the term *Big Data*. The latter again is frequently associated with *distributed dataflow systems* such as MapReduce [42], Apache Flink [30] or Apache Spark [202]. These systems follow the *bring the computation to the data* paradigm which has two implications: First, data is partitioned across a cluster of individual computers that are only connected by a local area network but have no shared memory. Second, complex programs must be split into local processing and communication steps, for example, to count words in a distributed collection of text files there must be one program to count words per machine and a second one to exchange and aggregate the local results.

In comparison to the development of distributed programs from scratch the usage of distributed dataflow systems may decrease development time. These systems provide a functional programming abstraction and handle all technical aspects of parallelization. Since we use this programming model to implement distributed graph mining (Section 6.3) and it will be mentioned at the discussion of related work this section provides a brief introduction to it.

The fundamental programming abstractions are datasets and transformations among them. A *dataset* is an immutable set of data objects partitioned over a cluster of computers. A *transformation* is an operation that is executed on the elements of one or two input datasets. The output of a transformation is a new dataset. MapReduce includes only two transformations (MRMap and Reduce) while Flink

and Spark show a wider range. Transformations can be executed concurrently on $W = \{w_0, w_1, .., w_n\}$ available *worker threads*. Every thread executes the transformation on an associated partition of a dataset. There is no shared memory among threads and all required data must be exchanged over the network by specific transformations.

Depending on the number of input datasets we distinguish *unary* and *binary* transformations. Table 2.3 shows examples of unary transformations. We further divide them into transformations that process *single elements* and those that process *groups of elements*. All of the shown functions require the user to provide a *transformation function* which will be executed for each element or group. A simple transformation is *filter*, were the function is a predicate and only those elements for that it evaluates to true will be added to the output. Another simple transformation is *map*, where the function describes how to derive exactly one output element from an input element. *Flatmap* is similar to map but allows an arbitrary number of output elements. MapReduce provides only one single-element transformation (denoted by *MRMap* in Table 2.3) which is a variant of flatmap that requires input and output elements to be key-value pairs.

The most important element group transformation is *reduce*. Here, input as well as output are key-value pairs. For each execution all elements sharing the same key are group. The transformation function describes the generation of a single output pair with the same key from each of these goups. Since input pairs with the same key may be located in different partitions they need to be *shuffled* among threads which is typically causing network traffic among physical machines. If the function is associative (e.g. summation), an additional combine transformation can be used to reduce this traffic. *Combine* is equivalent to reduce but skips shuffling, i.e., in the worst case one output pair is generated for each key and thread. Afterwards, these partial aggregation results can be passed to a reduce transformation.

As map and filter can also be expressed using MRMap, MapReduce and the new generation of *distributed in-memory dataflow systems* (DIMS) like Flink and Spark have the same expressive power in terms of unary transformations and the additional operations could be seen as convenience features. However, in the case of successive or iterative MRMap-reduce phases intermediate results need to be read from disk at the beginning and written to disk at the end of each phase. Thus, MapReduce is not well suited to solve iterative problems and problem-specific distributed computing models arose, for example, to process very large graphs (Section 2.4). In contrast, MapReduce and DIMSs are general purpose platforms and not dedicated to a specific problem. However, DIMSs support more complex programs including iterations, binary transformations (e.g., set operators like *union* and *join*) and are able to hold datasets in main memory during the whole program execution.

2.4 Graph Processing

As already mentioned in the previous section, there is a class of distributed data processing systems dedicated to graph data called *graph processing systems (GPS)*. These systems fundamentally differ from *graph database systems (GDBS)*. While GDBS such as Neo4j [232] primarily focus on OLTP and query workload, GPS, such as Google Pregel [117] are dedicated to graph analytics. Although we will not use GPS in the remainder of this dissertation they are worth discussing in the context of graph analytics. Thus, in this section, we will have a closer look on graph processing system. An overview of GDBS will follow in the next section.

In the following we will summarize our recent overview of [87]. Graph processing is especially applied to problems where a whole graph needs to be processed iteratively. Examples for such algorithms are *pagerank* [133], *triangle counting* or *connected components* [44]. Since respective graphs are ofter very voluminous (e.g., the web, social networks) graph processing systems are typically based on shared nothing clusters. Their programing models are dedicated to graphs and differ from general models of distributed dataflow systems. However, even these systems may provide graph processing capabilities such as Apache Spark GraphX [64] or Apache Flink Gelly [223]. However, according to a recent comparative study [181] their performance cannot compete to dedicated systems.

Although programming models of different graph processing systems differ in detail there is a general architecture of a distributed graph processing framework. It consists of a *master node* for coordination and a set of *worker nodes* for actual distributed processing. The input graph is partitioned among all worker nodes. Depending on the actual algorithm partitioning may impact the overall performance [181]. Most common is the *vertex-centric model*. Here, worker nodes store vertices together with vertex data values as well as all outgoing edges including their data values. Further on, also vertex identifiers (ids) of incoming edges are stored attached to vertices. Some approaches such as Giraph++ [176] ensure that copies of all vertices that are adjacent to vertices of a partition will be stored on the same partition.

The first popular vertex-centric programming model was the *Think Like a Vertex* approach that has been pioneered by Google Pregel in 2010 [117] and adopted or extended by other frameworks [63, 94, 103, 165, 172, 225]. In this model vertices send a message to their neighbors iteratively. To write a program in a Pregel-like model, a so called *vertex compute function* has to be implemented. This function consists of three steps: Read all incoming messages, update the internal vertex state (i.e., its data value) and send information (i.e., messages) to its neighbors. Each vertex only has a local view of itself and its immediate neighbors. Any other information about the graph that is necessary for computation has to be sent along the edges.

Many extensions have been made to the vertex-centric programming model. Powergraph [63] introduced the *Gather Apply Scatter (GAS)* model. Here, the user has to provide three functions. First, there is a *gather function* to aggregate messages that address the same vertex on the sending worker nodes. Second, there is an *apply function* that has the incoming messages as input and updates the vertex state. Third, there is a *scatter function* that creates outgoing messages based on the vertex state. The systems Signal/Collect [172] and Chaos [158] introduced the *Scatter Gather Model.* This model requires the user to provide an edge and a vertex function. The vertex function has all incoming messages as input and can modify the vertex value. An alternative to vertex-centric approaches is the *Think like a Graph* introduced by Giraph++ [176]. Here, a compute function has acess to all vertices of a whole worker node (partition).

Most graph processing systems work bulk synchronous parallel (BSP), i.e., the execution of program logic is bound to iterations. However, some algorithms [97, 127] converge faster or can only be implemented [154] with an asynchronous execution model. Thus, systems such as GraphLab [113] and GraphChi [103] support asynchronous execution.

2.5 Graph Databases

For the management of graph data there are dedicatedt *graph database systems* or simply *graph databases.* Graph database systems are based on a *graph data model* representing data by graph structures and providing graph-based operators such as neighborhood traversal and pattern matching [6]. Their recent applications are manifold, for example, collaborative software development [11], business data management [159] and knowledge management [208] in biological [115] or medical applications [110]. Research on graph database models started in the nineteen-seventies, reached its peak popularity in the early nineties but lost attention in the two-thousands [9]. Then, there was a comeback of graph data models as part of the NoSQL movement [31] with several commercial systems [6]. However, these new-generation graph data models arose with only few connections to early rather theoretical work on graph database models.

In this section, we will compare recent graph database systems as well as their analytical capabilities and their potential usage for graph-based business intelligence. Table 2.4 provides an overview of recent graph database systems including supported data models, their application scope and the used storage approaches. The selection claims no completeness but shows representatives from current research projects and commercial systems with diverse characteristics.

	Data Model			Scope		Storage		
	RDF/ SPARQL	PGM/ TinkerPop	Generic	OLTP/ Queries	Analytics	Approach	Replication	Partitioning
Apache Jena TDB [230]	✓/✓			✓		native		
AllegroGraph [213]	✓/✓			✓		native	✓	
MarkLogic [231]	✓/✓			✓		native	✓	✓
Ontotext GraphDB [226]	✓/✓			✓		native	✓	
Oracle Spatial and Graph [234]	✓/✓			✓		native	✓	
Virtuoso [50]	✓/✓			✓		relational	✓	✓
TripleBit [201]	✓/✓			✓		native		
Blazegraph [216]	✓/✓	✓/✓		✓	✓	native RDF	✓	✓
IBM System G [29, 191]	✓/✓	✓/✓	✓	✓	✓	native PGM or wide column store	✓	✓
Stardog [241]	✓/✓	✓/✓		✓	◦	native RDF	✓	
SAP Active Info. Store [159]		✓/-		✓		relational		
ArangoDB [215]		✓/✓		✓		document store	✓	✓
InfiniteGraph [228]		✓/✓		✓		native	✓	✓
Neo4j [125]		✓/✓		✓		native	✓	
Oracle Big Data [235]		✓/✓			✓	key value store	✓	✓
OrientDB [236]		✓/✓		✓		document store	✓	✓
Sparksee [118]		✓/✓		✓		native	✓	
SQLGraph [173]		✓/✓		✓		relational		
Titan [243]		✓/✓		✓	◦	wide column store or key value store	✓	✓
HypergraphDB [78]			✓	✓		native		

◦: via plugin mechanism

Table 2.4: Comparison of Graph database systems.

2.5.1 Data Models

The majority of the considered systems support one or both of two data models, in particular the property graph model (PGM) and the resource description framework (RDF). While RDF is standardized, for PGM there is only the industry-driven de facto standard Apache TinkerPop (Section 2.2). A few systems use generic graph models. We use the term *generic* to denote graph data models supporting arbitrary user-defined data structures (ranging from simple scalar values or tuples to nested documents) attached to vertices and edges. Such generic graph models are also used by most *graph processing systems* (Section 2.4). The support for arbitrary data attached to vertices and edges is a distinctive feature of generic graph models and can be seen as a strength and a weakness at the same time. On the one hand, generic models give maximum flexibility and allow users to imitate other graph models like RDF or PGM. On the other hand, such systems cannot provide built-in operators related to vertex or edge data as the existence of certain features like type labels or attributes are not part of the database model.

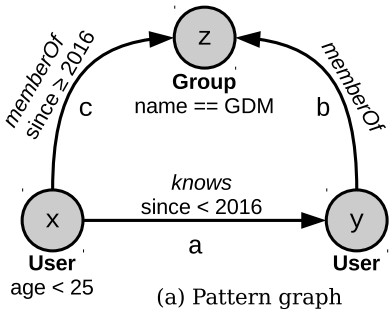
2.5.2 Application Scope

Most graph databases focus on OLTP workload, i.e., CRUD operations (create, read, update, delete) for vertices and edges as well as transaction and query processing. Queries are typically focused on small portions of the graph, for example, to find all friends and interests of a certain user. Some of the considered graph databases already show built-in support for graph analytics, i.e., the execution of graph algorithms that may involve processing the whole graph, for example to calculate the pagerank of vertices [117]. These systems try to include the typical functionality of graph processing systems by different strategies. IBM System G and Oracle Big Data provide built-in algorithms for graph analytics, for example, pagerank, connected components or k-neighborhood [29]. The only system that is capable to run custom graph processing algorithms within the database is Blazegraph by its gather-apply-scatter (Section 2.4) API . Additionally, the current version of TinkerPop includes the virtual integration of graph processing systems in graph databases, i.e., from the user perspective graph processing is part of the database system but data is actually moved to an external system. However, indicated by a circle in the analytics column in Table 2.4, we could identify only two systems currently implementing this functionality.

2.5.3 Storage Techniques

The majority of the considered graph databases is using a so-called *native* storage approach, i.e., the storage is tailored to characteristics of graph database models, for example, to enable efficient edge traversal. A typical technique of graph-

Figure 2.5: Comparison of pattern matching queries.



(a) Pattern graph

```
MATCH  (x:User)-[a:knows]->(y:User),
       (x)-[b:memberOf]->(z:Group),
       (y)-[c:memberOf]->(z)
WHERE  x.age < 25 AND z.name = 'GDM' AND
       a.since < 2016 AND c.since >= 2016
RETURN y.name, x.name
```

(b) Cypher

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example.schema#>
SELECT ?y_name ?x_name
{
  ?x rdf:type :User;  :name ?x_name; :age ?x_age. FILTER(?x_age < 25).
  ?y rdf:type :User;  :name ?y_name.
  ?z rdf:type :Group; :name 'GDM'.
  ?a rdf:subject ?x; rdf:predicate :knows;    rdf:object ?y; :since ?a_since. FILTER(?a_since < 2016).
  ?b rdf:subject ?y; rdf:predicate :memberOf; rdf:object ?z .
  ?c rdf:subject ?x; rdf:predicate :memberOf; rdf:object ?z; :since ?c_since. FILTER(?c_since >=2016).
}
```

(c) SPARQL

```
g.V().match(
        __.as('x').has(label, 'User').and().has('age', lt(25)),
        __.as('y').has(label, 'User'),
        __.as('z').has(label, 'Group').and().has('name', 'GDM'),
        __.as('x').outE('knows').has('since', lt(2016)).inV().as('y'),
        __.as('y').out('memberOf').as('z'),
        __.as('x').outE('memberOf').has('since', gte(2016)).inV().as('z')
).select('y', 'x').by('name')
```

(d) Gremlin

optimized storage are adjacency lists, i.e., storing edges redundantly attached to their connected vertices [29]. By contrast, some systems implement the graph database on top of alternative data models such as relational or document stores. IBM System G and Titan are offering multiple storage options. The used storage approach is generally no hint for database performance [173]. Most systems can utilize computing clusters by replicating the entire database on each node to improve read performance. About half of the considered systems also have some support for partitioned graph storage and distributed query processing. Systems with non-native storage typically inherite data partitioning from the underlying storage technique but provide no graph-specific partitioning strategy. For example, OrientDB treats vertices as typed documents and implements partitioning by type-wise sharding.

2.5.4 Query Language Support

In [6], Angles named four operators specific to *graph database query languages*: adjacency, reachability, pattern matching and summarization queries. *Adjacency*

queries are used to determine the neighborhood of a vertex while *reachability* queries identify if and how two vertices are connected. Reachability queries are also used to find all vertices reachable from a start vertex within a certain number of traversal steps or via vertices and edges meeting given traversal constraints. *Pattern matching* retrieves subgraphs (embeddings) isomorphic to a given pattern graph [58]. Pattern matching is an important operator for data analytics as it requires no specific start point but can be applied to the whole graph. Figure 2.5a shows an example pattern graph representing an analytical question about social network data. Finally, *aggregation* is used to derive aggregated, scalar values from graph structures (Section 4.2). In contrast to Angles, we use the term aggregation instead of *summarization*, as the latter is also used to denote structural summaries of graphs (Section 2.7). Such summarization queries are not supported by any of the considered systems.

Most of the recent graph database systems either support SPARQL for RDF or TinkerPop Gremlin for PGM. Both query languages support adjacency, reachability, pattern matching and aggregation queries. Fig. 2.5c and 2.5d show example pattern matching queries equivalent to the pattern graph of Fig. 2.5a expressed in SPARQL and Gremlin. The results are pairs of `Users` who are members of the same `Group` named `GDM`. Further on, one `User` should be younger than 25, a member since 2016 and already have known the other user before 2016. The query was chosen to highlight syntactical differences and involves predicates related to labels and properties of vertices and edges. To support edge predicates, the SPARQL query relates to edge properties expressed by standard reification (Section 2.2.4). While such complex graph patterns in SPARQL are expressed by a composition of triple patterns and literal predicates (`FILTER`), the Gremlin equivalent is a composition of traversal chains, similar to the syntax of object-oriented programming languages.

Besides SPARQL and Gremlin, there are also some vendor-specific query languages or vendor-specific SQL extensions. However, most of these languages miss pattern matching. A notable exception is Neo4j Cypher[217]. In Cypher, pattern graphs are described by ASCII characters where predicates that are related to vertices and edges are separated within a `WHERE` clause. Currently, Cypher is exclusively available for Neo4j but there are recent activities to turn it into an open industry standard similar to Gremlin [119]. A common limitation of SPARQL, Gremlin and Cypher is the representation of pattern matching query results in the form of tables or single graphs (SPARQL CONSTRUCT). In consequence, it is not possible to evaluate the embeddings in more detail, e.g., by visual comparison, and to execute any further graph operations on query results. A recently proposed solution to this problem is representing the result of pattern matching queries by a graph collection [84].

2.5.5 Summary

There are several databases that support our favoured data model PGM (Section 2.2). However, the application scope of graph databases are primarily data management applications and there are only few analytical capabilities. As the most crucial lack, none of the studied graph databases supports graph collections in a way we require them to implement our analytical approach (Chapter 3).

2.6 Graph Transformation of Relational Data

Domain objects of business information systems as well as their relationships are usually stored in relational databases. To make them available for graph-based analytics, they must be turned into graphs beforehand. Since this dissertation includes a contribution to the field of *relational to graph transformation* (Chapter 5), this section is dedicated to related work about this problem.

2.6.1 Domain Objects and Relational Databases

Relational databases [37] are often no natural representation of domain objects. Typically, there is an *object-relational mapping (ORM)* [5] to describe how main memory objects are persisted in relational databases. A popular example is the Hibernate framework [132]. Since objects of programming languages can be arbitrarily complex, for example by nesting or inheritance, there are different patterns of ORM [56]. To give a simple example, inheritance can be implemented by separate tables for superclasses and their descendants or by a single table where one column is used to assign each row to its class and some columns are only used by specific subclasses.

The conversion of relational data into other non-relational database models has been studied before the first graph-based approaches: In [54] Fong studied the problem for object-oriented databases, i.e., the reverse problem of ORM. He proposed a 2-step solution that creates a schema-translation from the relational to the object-oriented schema before actual data conversion. He already addressed the problem of inheritance (*isA*-relationships) but neglected different implementation patterns [56]. Later on, in [55] the same author proposed a similar approach for relational to XML conversion, which is less related to graphs since the central problems are very specific to XML, for example, file writing. In [116] Maatuk et al. discussed the transformation from relational to object-relational databases which is, in the case of business information systems, rather the reversal of ORM than a conversion into an alien model.

2.6.2 Relational to Graph Transformation

There was only few work on relational to graph transformation before 2012, i.e., there is a temporal connection to the rise of the second generation of graph databases [6]. But there are also exceptions: First, in the context of RDF there was the development of languages that map relational schemas to ontologies [19, 166] which form the base for automated instance transformation. However, as already discussed in Section 2.2, RDF leads to a very fragmented graph representation even in the presence of a schema. For example, at least one dedicated technical edge (triple) will be required to represent a single attribute on the instance level. Recent work on the reversion of this process even shows that existing RDF data can be analyzed faster, if a relational schema is extracted and data is imported into a relational database [147].

To the best of our knowledge, the first approach of relational to graph transformation that was not based on RDF was proposed by Soussi [168]. Her tool DB2SNA aims to enable social network analyses on relational databases. It is based on the hypervertex model [150], i.e., one of the first generation of graph database models [9]. DB2SNA uses the same model to represent a database schema and to store instance data. Complex object structures such as inheritance are not resolved and will remain in the final graph, i.e., one logical data object might be represented by more than one vertex and respective edges. Probably due to the generally low popularity of the hypervertex model the approach found not much attention.

The neglection of ORM patterns together with the neglection of multi-column keys can also be found at the early approaches that convert relational databases into property graphs [41, 134]. With R2G [41] Devirgilio et al. proposed the first solution to this problem. R2G is using different graph models to represent a databases schema and instance data. In the schema graph, just like RDF, attributes are represented by own vertices while instance attributes are represented by properties. The approach is treating m:n relationship tables in a special way but represents them by vertices in the final graph. R2G has no support for edge properties but requires a label for every edge. Our approach to relational to graph transformation [141] that will be studied in Chapter 5 was published in the same proceedings as 3NF Equivalent Graph (3EG) by Park et al. [134]. The development of 3EG was driven by a healthcare application. Just as R2G the approach provides no support for edge properties. In contrast to R2G it works directly on the relational schema and omits a dedicated schema model.

Later, the approach proposed by Lee et al. [105] added functional features as well as distributed processing based on MapReduce. Further on, it includes the support for edge properties, an editable schema mapping and an URI concept, i.e., global identifiers, to support the integration of multiple sources. However, the approach still assumes an exact correspondence of tables and classes and turns m:n

relationships into vertices instead of edges. Schema mappings (XML documents) must be added manually in advance.

Besides scientific work also graph database vendors propose techniques to import relational databases into their products. Neo4j requires the creation of CSV files for each class of objects by user-defined SQL statements [233]. The database's query language Cypher supports to import these files and to turn them into vertices. Cypher is also used to create edges by joining previously indexed key properties. The system GraphBase provides a technique called RapidGrapher [237] to import a relational database into a graph database that is just like [105] based on a user-defined XML schema mapping. Unfortunately, no further details are available to the public.

2.6.3 Further Related Techniques

There are further studies that are not exactly about relational to graph transformation but still related enough to be worth mentioning: First, SAP and its research project Active Information Store [159] integrates a property graph database model into their relational database HANA to support graph-based queries. However, at the time of this dissertation neither detailed information about this project is available nor an announcement for productivity was made. Beyond business applications, Vasilyeva et al. studied the import of Linked Open Data [128] in RDF format into the Active Information Store [180].

With GraphiQL [83] Jindal and Madden brought the graph model to the relational database in a different way. In particular, they proposed a graph query language [7] that is translated on the fly into SQL statements. However, GraphiQL requires a very specific relational schema and cannot be applied to databases that were made for a different purpose. This is different for GraphGen, an approach to user-defined graph extraction proposed by Xirogiannopoulos and Deshpande [193]. Their domain specific language requires the user to describe vertices and edges. The approach targets only structural analyses and graph processing applications. The resulting graphs are rather homogeneous and not suitable to represent complex domain models. Finally, there is GraphBuilder [80], an ETL Tool based on MapReduce that supports the user-defined construction of graphs. However, the tool is very generic and includes no specific techniques to deal with relational databases.

2.7 Graph-based Data Warehousing

This dissertation is about business intelligence and graph data models. Probably the most popular area that connects both worlds is graph-based data warehousing. In reference to classic data warehousing [96] the terms *graph OLAP* and *graph cube*

are often used as synonyms. Altough we make no contribution to this field this section will give a brief introduction into it:

Graph-based data warehousing aims to create summaries of an input graph. We consider a *summary (graph)* to be a graph whose vertices represent groups of vertices of an input graph and whose edges represent groups of input edges that connect members of grouped vertices. We consider the term *graph-based data warehousing* as an umbrella term for all approaches that generate user-defined and user-predictable summaries of a graph, i.e., there is a query language or a similar way to declare a summary with regards to the graph's data such as labels or attributes. Typically, these summaries contain classical data warehouse elements such as aggregated measures, dimensions and their hierarchies.

The existence of a technique to declare summaries is the major criterion to differentiate graph-based data warehousing from *graph summarization* which is a data mining technique that automatically creates summaries based on topological conditions such as a maximum vertex count of the summary graph [177]. A recent survey about this field [112] is provided by Liu et al. However, some approaches related to graph summarization are also interesting for business intelligence applications: For example, in [98] Koop et al. use graph summarization to visualize similar patterns in multiple graphs and in [207] Zhang et al. introduce the CANAL algorithm that enables grouping by numerical measures by automatically balanced quantization.

In [34] Chen et al. proposed GraphOLAP, the first approach to graph-based data warehousing, and introduced *roll-up/drill-down* and *slice/dice* operations by overlaying and filtering graphs. Further on, GraphOLAP provides support for hierarchical dimension values. InfoNet OLAP [153], an approach proposed by Qu et al., extends GraphOLAP by partial roll-up/drill-down, i.e., parts of a summary can be expanded or collapsed on demand. Further on Yin et al. developed HM-Graph OLAP [199] which extends the concepts of GraphOLAP by the support for heretogenous networks (vertex and edge types) and two further operations called *rotate* and *stretch*.

With GraphCube [209] Zhao et al. extended concepts of GraphOLAP by the definition of crossboid queries which enable analyses at different summary levels. Denis et al. developed a distributed variant of GraphCube [43] based on Apache Spark. A further distributed variant is Pagrol [187] by Wang et al. However, Pagrol further adds support for edge dimensions and allows the multigraph property for summaries, i.e., there may be more than one edge group between a pair of summarized vertices. Finally, Ghrab et al. extended the concepts of GraphOLAP and Pagrol by presenting GRAD [60], the first approach to graph-based data warehousing that supports truly heterogeneous property graphs. In the context of Gradoop we also proposed a distributed approach to declarative grouping of property graphs [90, 88].

In multiple publications [21, 22, 100] Bleco and Kotidis proposed operators and technical optimizations for aggregation queries on huge collections of small graphs. A further approach related but not similar to graph-based data warehousing is the Core-Facets Model [104] proposed by Dung et al. Here, not summaries but homogeneous views on heterogeneous graphs are the point of interest. Besides the technical solutions to the graph-based data warehousing problem that have been discussed before there are also Rudolph et al. who proposed summarization templates to declare summaries in [160] as well as notions of terms like measure, fact and dimension in the context of graph-based data warehousing in [161].

2.8 Graph Pattern Mining

Some contributions of this dissertation are related to graph pattern mining, in particular to frequent subgraph mining which is the problem of frequent pattern mining in graph structures. Since many of the approaches to frequent subgraph mining have their origin in different data structures, this section will first provide a brief overview of the history of frequent pattern mining in general (Section 2.8.1). Afterwards, we will study related work about frequent subgraph mining in more detail (Section 2.8.2). One of our contributions is an approach to generalized pattern mining. Thus, we will provide a dedicated discussion of this topic with regard to different data structures (Section 2.8.3). Finally, since our major analytical approach is related to the problem of process mining we will give a brief introduction to this area to highlight the differences to BIIIG (Section 2.8.4).

2.8.1 Frequent Pattern Mining

Frequent pattern mining is a *data mining* [68] problem that aims to extract a set of patterns that occur with a minimum frequency (*support*) in some kind of input data. Mostly, the term is related to *frequent itemsets*. The first important work about this problem [2] was presented by Agrawal et al. together with association rule mining. In the economic context the problem is also known as *shopping basket analysis*. Here, the input data structure is a set of itemsets (*transactions*) that, for example, may represent shopping baskets such $\{\text{bread}, \text{butter}, \text{coffee}\}$. The aims of association rule mining are, first, to identify frequent itemsets, for example, that $\{\text{bread}, \text{butter}\}$ is contained in (*supported by*) 20% of transactions, and, second, to detect rules like coffee is typically bought together with bread and butter. Example applications of these rules are predictions or recommendations.

A recent book by Aggarwal and Han [1] provides an overview of frequent pattern mining. The authors state that frequent pattern mining can be considered with regard to four dimensions: First of all, there are different algorithmic approaches to determine pattern frequencies. Second, in the era of Big Data, scalability is an

important issue. Third, there are different variations of the problem, for example, instead of minimum support a different criterion can be used to extract patterns of interest. Fourth, frequent pattern mining can be applied to different data types. Frequent pattern mining has been extensively studied in the last two decades an nowadays forms a whole research area with many specializations with regard to the stated dimensions. We will focus our discussion on work that is most related to this dissertation. Section 2.8.2 is dedicated to the most relevant data types of graphs and will study the other three dimensions in their context. Afterwards, in Section 2.8.3 we will have a closer look on the problem variation of generalized pattern mining, again, with regard to the other dimensions. Before, this section aims to give an introduction to the area and will present important examples for different variations as well as general concepts that are independent from data type and variation.

The most important classification of algorithmic approaches distinguishes between *a priori* and *pattern growth* algorithms. A priori algorithms were presented first [1, 3]. These algorithms, first, generate candidate patterns and, second, count their support by *pattern matching*, i.e., for each transaction that contains a pattern the support is increased by one. They exploit the *anti-monotonic* property of the problem, i.e., they exploit that larger patterns such as $\{\mathsf{bread}, \mathsf{butter}, \mathsf{coffee}\}$ can only be frequent if all of their sub-patterns $\{\mathsf{bread}, \mathsf{butter}\}$, $\{\mathsf{bread}, \mathsf{coffee}\}$ and $\{\mathsf{butter}, \mathsf{coffee}\}$ are frequent, too. By mutual containment patterns form a lattice. We will use the term *parent* for a pattern that is contained and the term *child* to denote a pattern that contains another pattern. An a priori algorithm must scan the database once for each generated candidate. In [25] Brin et al. proposed an improved support counting method that generates children as soon as their parents are known to be frequent.

However, the biggest bottleneck of a priori algorithms is the lattice-based search itself. With FP-Growth [69] Han et al. proposed the first approach that belongs to the second class of *pattern growth* algorithms. The basic concept has been adopted to other data structures such as sequences [137], trees [204] and graphs [196]. In contrast to a priori algorithms, these algorithms report canonical forms of actually supported patterns. By the use of a lexicographical order among patterns these algorithms ensure that every lattice node can be visited by a tree search. Independent from the actual data structure, this leads to a performance gain of pattern growth approaches over a priori algorithms.

An important variation of the frequent pattern mining problem is *constrained pattern mining*. Here, patterns must not only be frequent but also satisfy other filter criteria. In [136] Pei and Han provide an overview of this variation. Constraints may be related to a pattern's items, its size, its contained sub-patterns and aggregated scalar values such as an average label length. If only a certain kind of constraint is required its characteristics can be used to improve the algorithm's

efficiency. An important example is *closed pattern mining* [135, 197, 205]. A pattern will be considered *closed*, if all of its children are less frequent, i.e., a further specialization will decrease support. An even more restrictive problem is *maximal frequent pattern mining* [27, 65, 175]. A pattern will be considered *maximal*, if it has no frequent child, i.e., due to its size it is more discriminative than all of its parents but still frequent. A pattern must be closed to be maximal [27]. A further variation is *significant pattern mining* [108, 156, 188, 194] where the objective is a threshold that refers not to a pattern's support but to a significance measure such as p-values.

2.8.2 Frequent Subgraph Mining

Frequent subgraph mining (FSM) is the problem of frequent pattern mining in graph data, in particular vertex- and edge-labeled graphs. In this context patterns are graphs that are isomorphic to subgraphs of the input data with regard to equal vertex and edge labels. A recent book chapter by Cheng et al. [35] provides an introduction to this topic. Further on, a survey [82] by Jiang et al. provides an extensive overview of mining of frequent subgraphs and frequent subtrees.

In contrast to frequent itemsets, FSM has a fifth dimension to categorize algorithms: Depending on their input data algorithms belong either to the graph transaction setting or to the single graph setting. The *graph transaction setting* is the pendant of frequent itemset mining where transactions are graphs instead of itemsets. By contrast, in *single graph setting* there is only a single, mostly large, input graph and the frequency threshold relates to the number of pattern occurrences instead of the number of supporting transactions. In the following, we will discuss the other general dimensions of frequent pattern mining for both settings separately. Since this dissertation contributes to the graph transaction setting it will be discussed in more depth than the single graph setting.

Algorithms: Just like frequent itemset mining transactional FSM algorithms can also be categorized into a priori and pattern growth algorithms with basically the same characteristics. All frequent subgraphs (graph patterns) form a lattice based on parent-child relationships. The latter are again based on containment, i.e., a child contains its parent. However, most frequent subgraph algorithms further require patterns to be connected, i.e., there must be a path between all pairs of vertices. Because of the different search strategies in the lattice a priori algorithms are often refered to as *breath first search (BFS)* algorithms and pattern growth algorithms are also known as *depth first search (DFS)* algorithms. A priori (BFS) algorithms [77, 101] first generate candidates and count support by subgraph isomorphism testing while pattern growth (DFS) algorithms represent actual subgraphs by canonical labels. In Section 6.2.3 we will provide a generic comparison of both strategies.

The first transaction FSM algorithms followed an a priori approach. First, there was AGM [77] by Inokouchi et al. To avoid the extraction of duplicates and to ensure an efficient candidate generation AGM is using canonical adjacency matrices to represent graph patterns. Thus, in contrast to most other algorithms, AGM also extracts non-connected graph patterns. About the same time and independent from AGM Kuramochi et al. proposed FSG [101], another a priori approach. In contrast to AGM, FSG uses a rather expensive join operation in the candidate generation step. General disadvantages of these a priori algorithms are a high memory consumption as all patterns of the same size must be hold in main memory before children can be generated [82], the expensive candidate generation process itself [35] and that support counting must be done by subgraph isomorphism testing which is also known as the graph pattern matching problem [58]. Further on, it is possible that many generated candidates might not even appear.

Thus, all newer approaches use a pattern growth strategy. These approaches extend parent patterns by single edges and apply constraints and checks to avoid generating the same child in different ways. Independent from each other, the first two approaches were MoFa [24] by Borgelt and Berthold and gSpan [196] by Yan and Han. MoFa uses embeddings, i.e., mappings between transactions and patterns as its main data structure. During the mining process these embeddings will be extended and a canonical representation is used to count pattern support. There are some label-based constraints to avoid generating the same pattern in multiple ways. However, MoFa puts out duplicates that must be eliminated by isomorphism testing among the resulting patterns. By contrast, gSpan uses canonical labels that represent patterns (DFS codes) as its main data structure and holds only occurrence lists in main memory. Thus, embeddings must be recovered by subgraph isomorphism testing before every extension. However, gSpan is using more elaborate growth constraints and directly generates canonical forms. gSpan even allows to verify a pattern to be no duplicate without comparison to other patterns. This verification happens directly during the mining process [195].

FFSM [74], a further pattern growth approach proposed by Huan et al. uses canonical adjacency matrices to represent patterns and grows patterns by joining search tree nodes. Like MoFa, it keeps embeddings in main memory and like gSpan, duplicates can be detected by verification. A further embedding-centric approach is Gaston [129]. It applies different types of canonical labels for paths, trees and cyclic graphs. Based on this Gaston efficiently mines paths and trees before cycles are closed. Thus, duplicate verification is only required for a subset of patterns. However, the latter is done by isomorphism testing among the results.

In [190] Wörlein et al. provide a comparison of MoFa, gSpan, FFSM and Gaston with regard to their performance. They draw the conclusion that embedding-centric approaches speed up support counting but are only beneficial for large patterns and cause a significantly higher memory usage. Further on, the time to

Algorithm	Year of publication	Citation count
gSpan [196]	2002	2238
FSG [101]	2001	1338
AGM [77]	2000	1222
FFSM [74]	2003	705
MoFa [24]	2002	497
Gaston [129]	2005	157

Table 2.5: Citation count of FSM algorithms on Google Scholar (June 2018).

generate canonical representations is more important and isomorphism testing should be avoided. In [130] Nijssen and Kok, the developers of Gaston, compared Gaston, FFSM, gSpan, FSG and two tree miners. First, They have experimentally shown that pattern growth approaches are clearly preferable over a priori ones. Further on, they confirmed the impact of computing canonical forms and found gSpan's DFS codes to be preferable. They also confirmed the trade-off between runtime and memory usage by deciding for or against holding embeddings in main memory.

Scalability: Besides efficiency also scalability has always been an issue of frequent subgraph mining and there is active research on this topic until today. Many works that claim to have improved scalability of frequent subgraph mining actually relaxed the problem definition. Thus, we consider them as variants and not as scalability improvements. With regard to the standard variant of frequent subgraph mining, it is our impression that since the 2000s no more new efficient algorithms have been proposed and that gSpan became the most popular base algorithm for scalability extensions. A comparison of citation counts (Table 2.5) adds a measure to this impression. Most work about FSM scalability uses parallelization to speed up computation. A notable exception is ADI-Mine [184]. Here, Wang et al. extended gSpan by an adjacency index to provide scalability for scenarios where the graph database cannot be held in main memory.

The most straight forward parallelization is the one on multiple threads of a single machine. In [120] Meinl et al. presented respective versions of gSpan and MoFa. They were able to gain very good speedups for both algorithms. In [183] Vo et al. proposed another parallel version of gSpan. They also gained good speedups but reported that main memory consumption is increasing according to the number of threads. Further successful approaches to parallelize gSpan were proposed by Kessl et al. for GPUs [93] and by Stratikopoulos et al. for FPGAs [171]. Further on, there were several horizontally scalable approaches based on MapReduce [10, 18, 72, 109, 114]. Due to the close relationship of these approaches to this dissertation they will be discussed in further detail in Section 6.4.

Variants: First of all, there are variants of graph transaction FSM that are already known from frequent itemset mining: In particular, there are approaches to constrained [211], closed [197] and maximal [75, 175] mining. However, there are also variants specific to graphs such as the extraction of frequently correlated subgraphs pairs [91] by Ke et al. Of particular interest for business intelligence applications are approaches to significant subgraph mining [156, 164, 194] whose extraction criterion is a significance measure instead of a support threshold. Significance measures such as p-values [71, 123] are also used to rank graph mining results. A variant that is closely related to this dissertation is generalized frequent subgraph mining [76] and will be discussed in the next section.

Single Graph Setting: Frequent subgraph mining in single graph setting is a much different problem than graph transaction FSM and not much related to this dissertation. Nevertheless it is worth a brief introduction. Before early approaches to exact FSM in single graphs appeared Cook and Holder proposed SUBDUE [38], an approach to compression based on approximate frequent subgraph mining. The most crucial problem of early approaches to the standard variant [26, 28, 102, 179] was the definition of a suitable and scalable frequency threshold (*support measure*). While earlier approaches store patterns and embeddings, GRAMI [48] by Elseidy et al. uses a novel embedding-free approach with multiple optimizations that, according to the authors' evaluation, could outperform the existing approach by orders of magnitude. Further on, GRAMI supports variants of approximate and constrained frequent subgraph mining.

Just like the transaction setting, recent work is mostly focusing on scalability. For example, Zou and Holder use sampling to overcome memory limitations for very large graphs [212]. In recent years also many horizontally scalable approaches have been proposed. In contrast to the transaction setting these are not limited to approaches based on MapReduce [124, 167]. Graph processing (Section 2.4) has never been applied to the transaction setting because the data model is unsuitable. However, this is not the case for a single large graph. Thus, there are multiple approaches to single graph FSM based on graph processing systems and their programming models [174, 185, 210]. Additionally, in [152] Qiao et al. propose an approach based on the distributed in-memory dataflow system Apache Spark.

2.8.3 Generalized and Multidimensional Pattern Mining

Independent from the actual data type all frequent pattern mining approaches evaluate labels. For example, itemsets could be seen as sets of labels and a labeled graph represents a certain topology of labels. In many domains these labels can be associated to taxonomies which are also referred to as *isA hierarchies*. For example, in

a shopping basket scenario wholegrain bread and soft white bread could be linked to a taxonomy that shows that both have a common *generalization* of bread. In the problem of *generalized frequent pattern mining* all labels are assigned to taxonomies and frequent pattern candidates are not only constructed from the labels themselves but also from their generalizations.

Most work on generalized frequent pattern mining was made for frequent itemsets and respective association rules. For example, $\{\text{bread}, \text{butter}\}$ is a generalization of $\{\text{wholegrain bread}, \text{Irish butter}\}$. The first approach to generalized frequent itemset mining [169] was proposed by Srikant and Agrawal. In [67] Han and Fu extended the approach by the use of level-dependent minimum support thresholds. In [198] Yen and Chen used a graph that represents relationships among patterns to improve the mining process. These first solutions were all based on an a priori approach. In more recent work by Eavis and Zheng [47] as well as Jayanthi et al. [81] presented two pattern growth solutions to the problem to speed up computation.

There are some studies where generalized frequent pattern mining was extended to different data types. In [170] Srikant and Agrawal proposed a generalized approach to frequent sequence mining. Inokuchi even presented an approach to generalized frequent subgraph mining [76]. His solution is based on an a priori algorithm based on undirected simple graphs and assumes, just like itemsets, that all vertices belong to the same taxonomy. To the best of our knowledge no approach to scalable generalized frequent pattern mining has been proposed yet.

The problem of *multidimensional frequent subgraph mining* did not find not much attraction yet. However, in our opinion it is very relevant for business intelligence applications since business data typically represents not only a single class of things. We will consider data to be *multidimensional* if labels represent different classes, for example, products and locations. In [148] Pinto et al. proposed an approach to multidimensional sequence mining. However, in this approach not the sequence labels but the sequence itself is attributed by multiple dimensions. A further different notion of multidimensional mining [200] was studied by Yu and Chen. Here, the term is used because they mine nested sequences similar to multidimensional arrays. However, the approach to multi-relational mining [46] by Džeroski is closer to our notion since different tables can be used to represent different dimensions. To the best of our knowledge, the only approach [149] that combines generalized and multidimensional mining was proposed by Plantevit et al. Here, the main data structure are sequences of multidimensional vectors.

2.8.4 Process Mining

In [178] van der Aalst provides an introduction to this problem and states that *"process mining aims to discover, monitor and improve real processes by extracting knowledge from event logs"*. BIIIG can be used for *process mining* but is not limited to this single purpose. We will even use a related application, where graphs represent business process executions, as our running example.

However, typical approaches to process mining use different representations than graphs, for example, artifact choreographies [51] as proposed by Fahland et al. There are also some further relationships. For example, Nooijen et al. studied the automated extraction of interrelated data objects from ERP systems [131]. Further on, tailored graph abstractions have also been applied to process mining. For example, Beheshti et al. [16] and Fazzinga et al. [53] proposed extensions to graph query languages to analyze process data. In [15] Beheshti et al. even proposed a RDF-based approach to OLAP on process data.

2.9 Data Generators

One of our contributions is a data generator (Section 4.4). Thus, we will discuss related work about graph data generators. First of all, there are data generators for OLAP benchmarks such as TPC [244] and APB-1 [214]. APB-1 and all TPC benchmarks, except TPC-DS, are focused on a single class of transaction data and, thus, not suitable to analyze relationships. The data generator of BigBench [59] by Ghazal et al. extends the TPC-DS data model by related transaction data from web logs and reviews.

Typical generators for graph data generate graphs by given metrics but without semantic meaning. Respective approaches exist for single graphs, for example, GTgraph [12] by Bader and Madduri or the data generator of the Graph500 benchmark [126], and for collections of graph transactions such as GraphGen [36] by Cheng et al. Further on, some graph data generators were proposed in the context of benchmarking graph database systems. In [182] Vicknair et al. used a generator for directed acyclic graphs to compare relational and graph databases. In [45] Dominguez-Sal et al. benchmark different GDBMS and use the R-MAT algorithm [32] to generate synthetic graphs with characteristics of real networks.

There are also graph data generators that focus on domain data. The Berlin SPARQL benchmark [20] by Bizer and Schultz includes a data generator for products and related information. In [73] Holzschuher and Peinl also compare a relational and a graph database system and focus on the evaluation of graph query languages. The generated datasets resemble the structure and content of online social networks. In [66] Gupta proposes a data generator for heterogeneous multigraphs with labeled vertices and edges that represent meaningful data from a drug discovery scenario.

The generators discussed so far generate relationships more or less randomly. With S3G2 [146] Pham et al. proposed the first approach that leads to plausible structural correlations between domain objects and their relationship structure. In particular, S3G2 uses dictionaries and correlation rules to generate social networks with real-world characteristics in terms of relationship and property semantics. Further on, S3G2 is based on MapReduce and, thus, horizontally scalable. The Linked Data Benchmark Council (LDBC) [23] is an independent organization with members from academia and industry that maintains multiple graph benchmarks and respective data generators. LDBC's social network benchmark [49] is based on S3G2. Further on, there is a semantic publishing benchmark [8] whose RDF data generator includes real linked open data. The most recent one is a benchmark for graph processing systems [79] which combines real data, S3G2 and the data generator of Graph500 to generate highly scalable datasets.

Recently, Bagan et al. proposed gMark [13], a query benchmark that includes a generalized graph data generator for domain data. Here, the user can not only define schema and size of the resulting graph but also structural constraints related to different types of vertices and edges.

Chapter 3

Analytical Framework

This chapter will introduce the analytical foundations of our framework *BIIIG (Business Intelligence with Integrated Instance Graphs).* BIIIG aims to enable analytical workflows that rely on the graph abstraction of business data, for example, to extract meaningful relationship patterns that correlate with a given *business indicator.* Example business indicators are *financial result* and the existence of *fraud.* BIIIG covers all steps that are required to extract relationship patterns represented by graphs from source data in relational databases. Details about BIIIG's components will follow in the subsequent chapters. This chapter focuses on the introduction to its complete terminology as well as on giving an overview of its analytical idea.

First, we will discuss the distinction between master and transaction data (Section 3.1). Afterwards, we will introduce business transaction graphs (Section 3.2), a key concept of BIIIG's analytical idea, and provide an example scenario that will be referenced throughout this paper (Section 3.3). Business transaction graphs are a graph-based alternative to data warehouse models. Thus, we will further discuss the terms measure and dimension (Section 3.4) as well as pattern mining (Section 3.5) in the context of these graphs. Finally, we will state requirements that need to be met to implement the BIIIG approach (Section 3.6).

3.1 Master and Transaction Data

The data recorded in a company's business information systems can be divided into two superclasses [229]: First, there is *master data* about internal and external entities such as employees, products and customers. Second, there is *transaction data* about business actions such as sales orders, invoices and phone call logs. This distinction plays an important role for the BIIIG approach.

Definition 3.1 (Master Data) *"Master data is held by an organization to describe the entities that are both independent and fundamental for that organization, and referenced in order to perform its transactions."* (text quote from [229])

Definition 3.2 (Transaction Data) Transaction data represents business transactions. A business transaction is the completion of a business action or a course of action. [229]

In a data warehouse, there are two types of tables that more or less correspond to this classification. In particular, there are *dimension tables* to represent master data and *fact tables* to represent transaction data [96]. Fact tables include predefined foreign keys to dimension tables. In BIIIG's graph abstraction master data objects are represented by *master data vertices* and transaction data objects are represented by *transaction vertices*. Further on, relationships among all types of objects are reflected by edges. More details about our graph-based representation of business data will follow in Chapter 5.

3.2 Business Transaction Graphs

Business data, and so their graph representation, can be logically partitioned in different ways, for example, geographically to represent different countries or organizationally to model access privileges. BIIIG's main analytical data structure is business transaction graphs. These graphs reflect a process-centric data partitioning. Some economical background knowledge will be required to understand their basic concept:

Definition 3.3 (Business Process) *"A business process consists of a set of activities that are performed in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations."* (text quote from [189])

Our example business process is trading goods. Here, the business goal is selling goods with a maximum profit. Related activities are actions to engage customers to buy things (e.g., sales campaigns, phone calls, customer visits) as well as all required steps that ensure that the customer is receiving its goods (e.g., order processing, purchasing, shipping) and the profit is booked in the ledger (e.g., invoicing, payment, accounting). Instances of a business process are typically referred to as case [178].

Definition 3.4 (Business Case) A case of a business process is an actual execution of its activities. A case leads to a measurable increment of a business goal.

With regard to our example process of trading goods, each case will lead to an increment of the company's financial result either by profit, for example, if a deal was made, or by loss, if a quotation was not confirmed by the customer although financial efforts were already made. For each case there are different

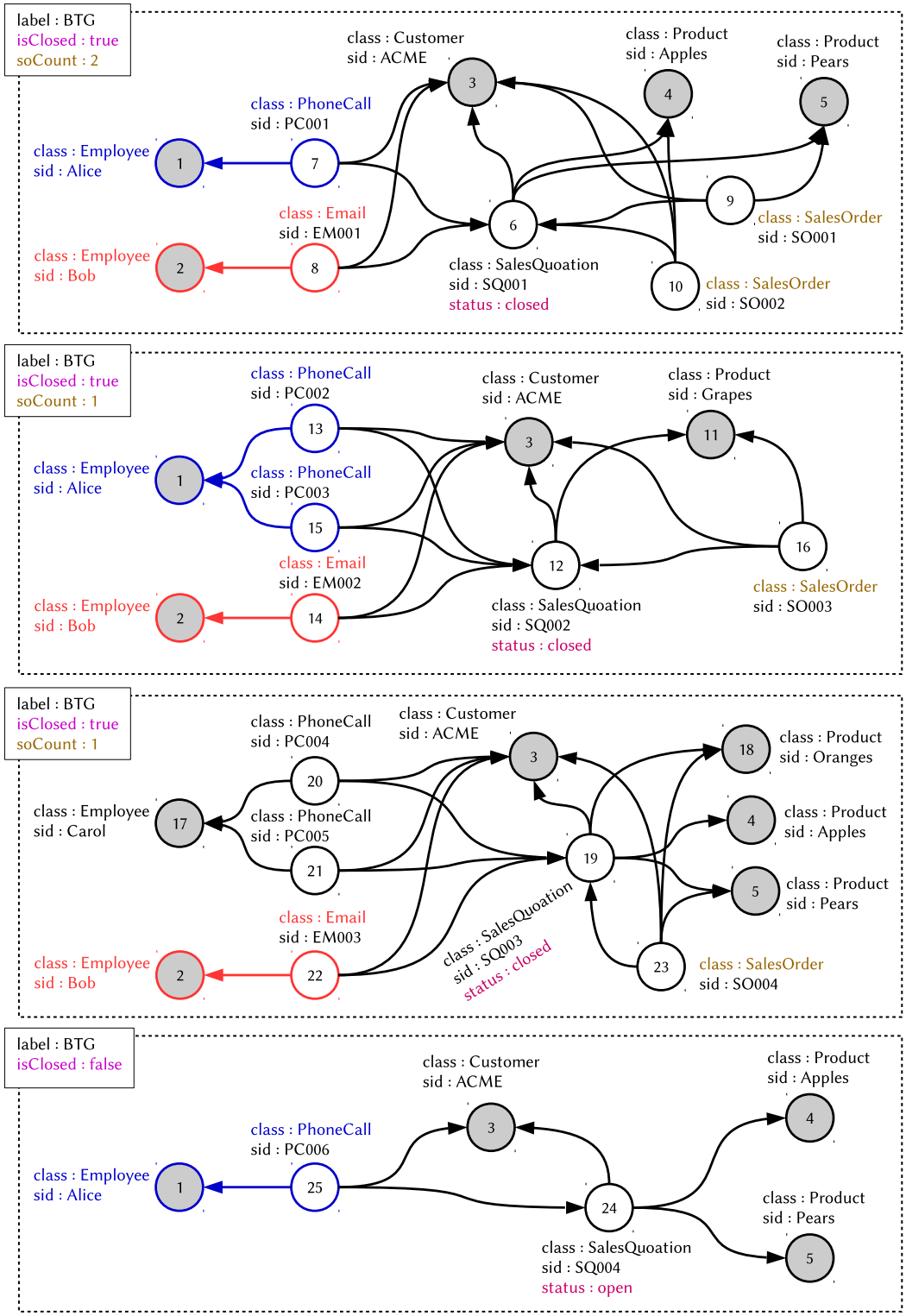


Figure 3.1: Example business transaction graphs with aggregated graph measures isClosed and soCount [139] .

Process	Transaction Data	Master Data	Measure	Dimension
Trading goods	Phone call, Sales order	Customer, Product	Financial result	Order year
Car assembly	Assembly step, Car (instance)	Worker, Car model	Assembly time	On time
Medical treatment	Examination, Prescription	Doctor, Drug	Recovery rate	Age group

Table 3.1: Examples transaction data classes, master data classes, measures and dimensions of different business processes.

classes of transaction data to represent activities and actions. For example, if the *quotation* is confirmed a *sales order* will be created and if a *payment* with a specific amount is received to balance the value of the ordered goods they will be shipped and a *packing slip* will be created. Further on, every case includes certain master data such as employees, customers and products. Table 3.1 shows typical classes of master and transaction data for different business processes. However, even relationships are part of case-related data. For example, goods were shipped by a certain logistics company (involved master data) because they were ordered and payed (*causal connections*). In a trade company, the business goal is reached by a potentially infinite number of cases. With business transaction graphs we propose a representation for all data that is directly related to a single case:

Definition 3.5 (Business Transaction Graph) A business transaction graph reflects all data that was created for exactly one case of a business process. This data consists of vertices to reflect transaction data objects and edges to reflect their relationships. To guarantee consistency, business transaction graphs may also contain vertices (e.g., master data) that were created for more than one case but only if they are incident to an edge that was created for the case that is represented by the graph.

In Section 5.5, we will discuss an algorithm to extract business transaction graphs automatically from graph-integrated business data.

3.3 Example Scenario

Our example scenario is about a fictive company that is trading food and an analyst who wants to identify reasons for won and lost quotations in sales cases. Figure 3.1 shows example business transaction graphs of this company. White vertices are created for exactly one case (transaction data) and gray vertices may be part of multiple business transaction graphs (master data), for example, vertex 1 which represents an employee with name Alice. Each graph represents a sales case. The

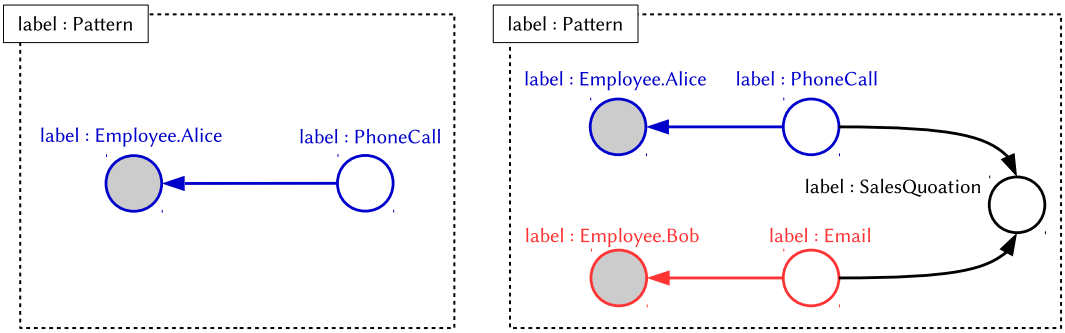


Figure 3.2: Example relationship patterns with dimension values as label-properties.

analytical strategy is to evaluate closed cases to give recommendations for sales activities in open cases. A case will be considered *closed*, if there are no open quotations, i.e., no vertices with class=SalesQuotation and status=open.

The first three graphs of Figure 3.1 represent closed cases (graph property isClosed=true) and the fourth graph is an open one (isClosed=false). Further on, a closed case will be considered *won*, if at least one sales order was created (graph property soCount>0). Otherwise, a closed case will be *lost*. Consequently, the first two graphs of Figure 3.1 represent won cases and the third graph a lost one. The fourth graph cannot be considered any of both because it is still open. The values of the numeric graph properties are persisted results of applied aggregate functions. More details about our data model that supports graph collections, graph aggregation and graph properties will be provided in Chapter 4.

Reasons for won and lost cases should be represented by meaningful patterns like *'a phone call made by Alice'*. However, patterns should not be limited to such simple statements, but should reflect also compositions like *'Alice made a phone call and Bob sent an email regarding the same quotation'*. Figure 3.2 shows respective graph representations of both patterns. Additionally, patterns must not be trivial. For example, if a pattern occurs frequently in won cases and also in lost cases, it cannot be considered to be characteristic for either of the outcome categories. Details about our analytical approach to gain the targeted result will follow in Section 3.5.

3.4 Measures and Dimensions

In the context of BIIIG, we use the following notions of measure and dimension:

Definition 3.6 (Measure) A *measure* is a quantifiable (numeric) property. A *measure value* is the actual quantity of a measure. The value range of a measure is infinite.

Definition 3.7 (Dimension) A *dimension* is a descriptive property with a finite range of at least two discrete *dimension values*. There may exist a taxonomy among the values of a dimension.

An example measure is the *revenue* of a sales process and an example dimension is the *city* of a customer's address. In a data warehouse, dimensions are the non-key attributes of dimension tables such as a table of customer data. Additionally, these attributes may be attached to taxonomies, for example, to group customers by region and country. In this case parent values such as customers' countries will be rather stored as redundant attributes instead of a foreign key to a dedicated table. By contrast, the attributes of a fact table represent either measures or foreign keys to dimension tables, for example, the revenue of a sales order and a reference to the ordering customer. Thus, it is possible to evaluate aggregated measures by dimension values, for example, the sum of all revenue values by customer country. Fitting data into a schema of fact and dimension tables typically includes a loss of relationship information. This is especially the case since transaction data is stored in fact tables which are not designed to have foreign keys to other fact tables. Thus, in particular relationships among transaction data cannot be evaluated. By contrast, BIIIG explicitly aims to evaluate measures in the context of not only dimension values but also their relationship patterns:

Definition 3.8 (Dimension Relationship Pattern) A *dimension relationship pattern*, in the remainder of this dissertation simply referred to as *relationship pattern*, is a graph whose vertices and edges have attached dimension values to reflect the relationship structure among them.

Figure 3.2 shows two example relationship patterns. To extract these, dimension values must be directly attached to the graph structure already in the input data. In Figure 3.1, we highlighted two example patterns. The pattern in blue color represents *a phone call made by Alice* and the one in red color *an email sent by Bob*. Comparing both Figures, it can be seen that the same information is encoded differently in terms of property usage. Details on that will follow in the next section. With regard to the input data of Figure 3.1 typical dimensions that provide valuable information are vertex classes, edge types or master data entity names. We use properties to store dimension (key) and dimension values (value), for example, name=Alice and class=SalesQuotation. The latter shows that BIIIG considers metadata (classes and relationship types) just as a special dimension. While classes and types of transaction data reflect actions and causal connections, master data names provide context information, for example, which product was sold or which employee was processing the order. In analogy to dimensions, measure values are represented by properties of vertices and edges, too. Typically, only transaction vertices have measure properties. Both dimensions and measures even exist on the level of business transaction graphs. The graphs of Figure 3.1 have a dimension isClosed and a measure soCount. Thus, it is possible to filter graphs. For example, the last graph has no soCount property as the aggregate function was only applied to those filtered by isClosed=true.

3.5 Characteristic Subgraph Mining

To identify correlations between the occurrence of relationship patterns and business indicators we propose a method called *characteristic subgraph mining*. It extracts relationship patterns that are characteristic for different values of a given case dimension (*categories*) and consists of the following four steps:

1. Categorize business transaction graphs based on a measure or a dimension.
2. Normalize input properties to emulate a labeled graph.
3. Determine pattern frequencies for all categories.
4. Select patterns that are characteristic for a category.

In the following, we will describe these steps in more detail.

3.5.1 Input Categorization

Our problem can be defined as follows: Let $G = \{g_1, \ldots, g_n\}$ be a collection of business transaction graphs and let $C = \{c_1, .., c_m\}$ be a set of categories with a mapping $\zeta_G : G \rightarrow C$ that associates input graphs to categories then we want to identify a collection of graph patterns $P = \{g_1, .., g_k\}$ and a mapping $\zeta_P : P \rightarrow C$ that is expressing for which category a pattern is characteristic. While P and ζ_P are result of an algorithm that will be introduced in Section 3.5.3, ζ_G is specific to the particular analysis and, thus, must be defined in a preprocessing phase.

For example, the business transaction graphs of Figure 3.1 can be associated to a category set $C = \{\text{won}, \text{lost}\}$ based on the aggregated measure soCount. In particular, all graphs whose value is greater than zero will be considered won while the remaining ones will be considered lost. This example shows, that categorization is a complex process that requires multiple steps itself. In particular, we must aggregate soCount for each graph and derive a new property to represent the *category* of each graphs. Further on, since we are only interested in closed cases, we must aggregate the closed property before and apply our actual aggregation only to the remaining ones. In Chapter 4 we will present a data model that is capable of expressing such complex processing of graph data.

3.5.2 Input Normalization for Pattern Extraction

To benefit from the graph representation we must apply a generic algorithm that extracts patterns without schema knowledge. Respective graph algorithms such as those of frequent subgraph mining assume a single label attached to every vertex and edge [82], i.e., labeled graphs but not property graphs. Since the property graph model subsumes labeled graphs this model glitch can be solved by an additional normalization step. In this step, properties of vertices and edges will be

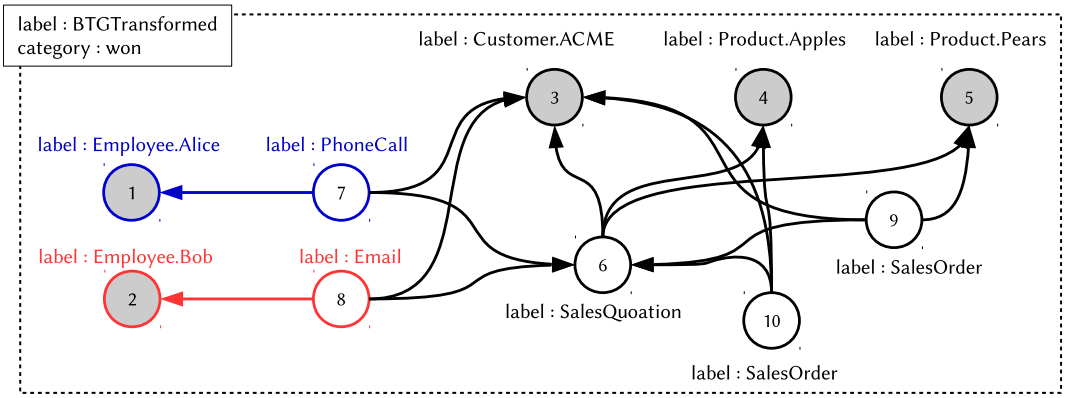


Figure 3.3: Business transaction graph (id=1) after normalization for pattern mining.

evaluated to derive a single property with the reserved key label for every vertex and edge. Although this transformation could be another user input, we also propose a generic approach based on the distinction of master and transaction data:

After BIIIG's approach to data transformation and integration (Chapter 5) every vertex has a set of reserved properties. First, one of them associates every vertex to either master or transaction data, represented by gray and white vertices in Figure 3.1. Second, there is a class property to add a semantic type and, third, there is a *sid* property to store a domain-specific identifier. For master data, the latter is typically a meaningful business identifier or there is at least an exact mapping to real-world entities such as customers or products. Thus, master data identifiers must be included in patterns because, for example, it would not be point of interest if products were sold but which ones.

By contrast, transaction data identifiers are only technical, mostly automatically incremented values. Further on, according to Definition 3.5 they may not occur in more than one business transaction and, thus, cannot contribute to interesting patterns. By contrast, the class property adds much semantic meaning since it is expressing which action a vertex represents. The same applies to the reserved *type* property of edges which are not shown by Figure 3.1 to keep the illustration clear. However, it is obvious that different types of relationships add further semantic meaning to patterns.

In consequence, our normalization process is fairly simple: For all master data take its source identifier (sid) as label, for all transaction data use the value of the class property and for all edges take their relationship type. Figure 3.3 shows the first graph of Figure 3.1 after this *transformation* process. In this step, even the properties of the graphs were changed. In particular, categorization was persisted in a category property. We further see, that also some properties such as the status of SalesQuotations were dropped since they will add no value to the subsequent pattern mining process. A formal introduction of the *property transformation operator* that is required to implement our normalization step will follow in Section 4.2.

3.5.3 Frequency Determination

The determination of graph pattern frequencies corresponds to the problem of frequent subgraph mining (Section 2.8.2). So, a naive approach would be to determine frequencies of all patterns in all category subsets and pass the result over to the pattern selection step (Section 3.5.4). However, since frequent subgraph mining is a NP-complete problem this would be a very inefficient approach. Thus, regular frequent subgraph mining algorithms require a *minimum support threshold* to express that only pattern frequencies of patterns that occur more frequently than this threshold (e.g. 50%) will be extracted. We adopted this idea to characteristic subgraph mining and require a *relative minimum support per category* $0 \leq \phi_{min}^{rel} \leq 1$ which needs to be specified by the user. Based thereon we extract a set of candidate patterns P' and the *category support* $\phi_{cat}^{rel} : P' \times C \rightarrow \mathbb{Q}^+$ only for those patterns that will be at least frequent for one category s.t. $\forall p \in P'.\exists c \in C.\phi_{cat}^{rel}(p, c) \geq \phi_{min}^{rel}$. Characteristic subgraph mining can be simply implemented by adding an alternative pruning criterion based ϕ_{min}^{rel} and ϕ_{cat}^{rel} to an arbitrary frequent subgraph mining algorithm. We provide an Open Source implementation[1] of this approach.

3.5.4 Pattern Selection

Our problem differs from frequent subgraph mining as we require patterns not just to be frequent but to be characteristic for a category as well. For example, the blue pattern in Figure 3.1 is *interesting* as it occurs in all of the won cases but not in the lost one. By contrast, the red pattern occurs in all graphs of both categories and, thus, is considered *trivial*. To distinguish interesting from trivial patterns we propose the use of an *interestingness measure* comparing the frequency of a subgraph in different categories. Let P' be a set of candidate patterns and ϕ_{cat}^{rel} their category support then a simple interestingness measure could be the ratio of category support and average support in all categories. Based on this measure ζ_P can be determined by assigning the category with the maximum value to the pattern. Finally, a *minimum interestingness threshold* should be used to filter trivial patterns and determine the final set of characteristic patterns $P \subseteq P'$. We only sketched this process since interestingness measures will always be domain specific, i.e., there is no general approach. Further on, this dissertation is about the benefits of graph models but once ϕ_{cat}^{rel} is determined, all further logic to select results will be plain statistics. We will report our practical experience in pattern selection in Sections 8.1 and 8.2.

[1] `https://github.com/p3et/dmgm`

3.6 Requirements

In this chapter, we introduced fundamentals of the BIIIG approach as well as characteristic subgraph mining, an analytical method that is utilizing graph pattern mining for business intelligence. However, BIIIG is not about a single static data processing flow but a general idea of analyzing business data by the use of the graph abstraction. In the past, this application domain was strongly associated to relational and multidimensional models. BIIIG shall be flexible, i.e., it must allow users to mix their domain-specific analytical targets with general graph operations. Characteristic subgraph mining is meant to demonstrate a complex analytical workflow that covers different aspects of graph-based data analytics and motivates their application. Based on characteristic subgraph mining, we can state the following requirements that need to be met to implement BIIIG:

1. To represent business transaction graphs, we need a data model, that supports collections of directed multigraphs, graph properties and elaborate operators based thereon (Chapter 4).

2. There must be an approach that is capable to turn data that is hold in business information systems into business transaction graphs (Chapter 5).

3. Since extraction and counting of graph patterns are computationally expensive, we need a respective graph mining algorithm that is not only efficient and scalable but also support all features of our data model (Chapter 6).

4. To be competitive to data warehouse models, we also need a way to deal with taxonomies of dimension values and support patterns of values from muiltiple taxonomy levels (Chapter 7).

Chapter 4

Extended Property Graph Model

In this chapter, we will discuss the *Extended Property Graph Model (EPGM)*. After a motivation of its development (Section 4.1) we will provide a formal introduction (Section 4.2). We will further present Gradoop, the first framework implementing EPGM (Section 4.3) and FoodBroker, an elaborate data generator that generates EPGM data for evaluation and testing purposes (Section 4.4). Finally, we will provide a brief conclusion (Section 4.5).

4.1 Motivation

Graph data models enable flexible and powerful evaluations of domain objects and their relationships. Graph analytics and graph mining have become popular among researchers of different domains, for example, to analyze social networks, the world wide web or biological networks. An important group of analytical algorithms are those evaluating a graph's topology such as centrality measures [57]. These algorithms derive single measures to describe a graph on the whole, for example, to characterize its structure. Further on, there are many popular algorithms that focus on the evaluation of single vertices withing a graph's structure. For example, the page rank algorithm [133] is executed to identify highly linked vertices such as popular websites or influencers in social networks.

BIIIG targets the analysis of business data that can be abstracted as a graph. In a respective graph vertices represent heterogeneous domain objects such as customers, products or sales orders and edges represent relationships among those objects. Just like typical business intelligence based on data warehouses [62] neither the evaluation of whole databases nor the evaluation of single records are points of interest. On the contrary, to support decision-making we must systematically evaluate the coexistence of records. This is crucial since business measures such as financial profit and their influence factors such as products, sales people and marketing activities are stored in different records but the extraction of knowledge requires to analyze them in connection.

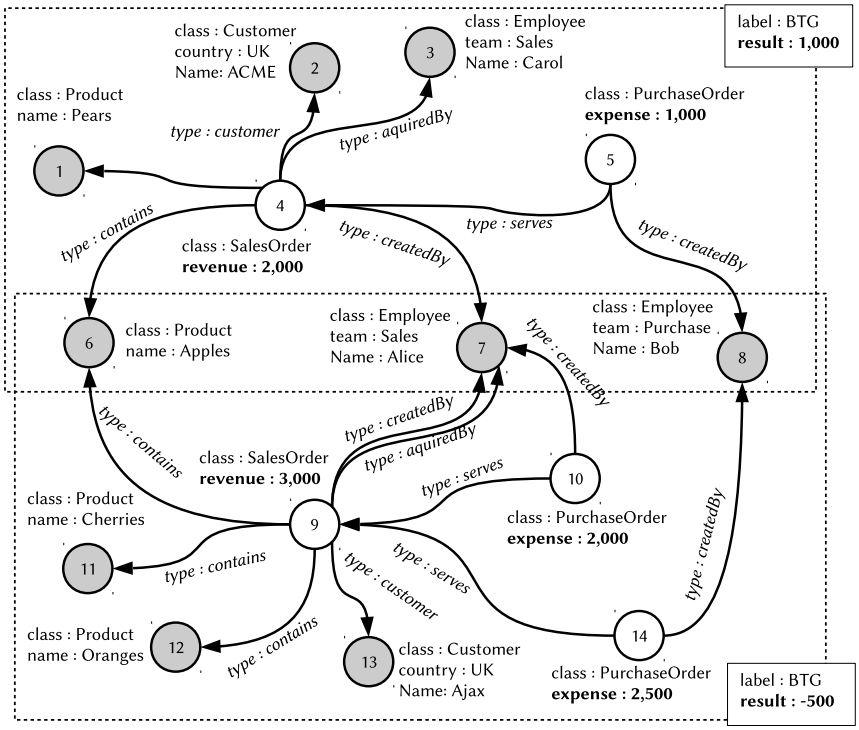


Figure 4.1: Example graph collection with shared vertices. Dotted lines represent graph boundaries. Solid rectangles overlaying the graphs' boundaries contain graph properties. Bold properties contain measure values and normal font ones dimensional values.

In a data warehouse, there is a clear structure of fact and dimension tables [96]. Fact tables hold base data about business measures and dimension tables provide attributes about potentially influencing entities. In this way facts can be evaluated in the context of one or more dimensions. In a graph there is no corresponding concept due to the lack of a schema. By the observation of typical analytical questions we found logical subgraphs as an appropriate way to persist meaningful partitions which contain not only measure relevant data and potential influence factors but also, in addition to data warehouses, their full relationship structure. An example of logical partitioning are business transaction graphs (Section 3.2). Figure 4.1 shows an example graph with two logical business transaction graphs.

By the abstraction of logical partitions it is possible to perform comparative analytics among partitions. For example, we can calculate measures on the graph level to filter graphs by these measures (Section 4.2). Further on, we can detect common graph elements among partitions such as common master data vertices (e.g., vertices 6, 7, 8 in Figure 4.1) as well as complex patterns that occur frequently in large collections of such partitions (Chapter 6). Finally, common substructures of partitions can be analyzed in the context of business indicators (Chapter 3). However, this abstraction is not only valuable for the business domain. For example, country boundaries of railway networks and communities in social networks [14] can be considered as logical partitions of a single large graph.

Despite manifold applications of logical graph partitions, current database systems lack their support (Section 2.5). Thus, we started the development of a novel data model. This so-called *Extended Property Graph Model (EPGM)* includes logical subgraphs as first-level citizen, supports graph properties and enables powerful operators based on both concepts.

4.2 Data Structure and Operators

The *Extended Property Graph Model (EPGM)* extends the property graph model [157] by structural features and operators [89]. With regard to its data structure EPGM is based on *property graph collections*. In comparison to the standard property graph model, property graph collections include, next to vertices and edges, *logical graphs* as a third class of first-level citizen. Vertices and edges may be part of an arbitrary number of logical graphs, i.e., logical graphs may *overlap*. Just like vertices and edges, logical graphs have an identifier and may have arbitrary properties.

Definition 4.1 (Property Graph Collection) A property graph collection is defined as a nonuple $C = \langle G, V, E, \gamma, \varsigma, \tau, Y, D, \pi \rangle$. It includes three sets of identifiers for logical graphs $G = \{g_0, g_1, \ldots, g_k\}$, vertices $V = \{v_0, v_1, \ldots, v_n\}$ and edges $E = \{e_0, e_1, \ldots, e_m\}$. Further on, there is a graph containment function $\gamma : (V \cup E) \rightarrow (\mathcal{P}(G) \setminus \emptyset)$ that maps each vertex and edge to a graph and two functions that associate a source vertex $\varsigma : E \rightarrow V$ and a target vertex $\tau : E \rightarrow V$ to every edge. To preserve consistency, vertices must also be mapped to all graphs of their incident edges s.t. $\forall e \in E. \forall g \in \gamma(e). \Big(g \in \gamma(\varsigma(e)) \wedge g \in \gamma(\tau(e))\Big)$. Additionally, not only vertices and edges but also graphs support properties. Properties are reflected by a set of property keys $Y = \{y_0, y_1, \ldots, y_n\}$, a set of data values $D = \{\varepsilon, d_0, d_1, \ldots, d_n\}$ and a property mapping whose domain includes also graphs, i.e., $\pi : (G \cup V \cup E) \times Y \rightarrow D$.

Besides this data structure, EPGM consists of a set of operators based thereon. At the time of this dissertation, these operators were under continuous development within the Gradoop framework (Section 4.3). Table 4.1 presents a current list of operators. These are categorized into unary and binary operators based on their domain. While the input of an unary operator is a single property graph collection, the input of a binary operator are two of them.

Definition 4.2 (EPGM Operator) In the Extended Property Graph Model an *unary operator* is defined as a function $\Omega : \mathcal{C} \rightarrow \mathcal{C}$ whose domain and co-domain are the space of property graph collections $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$. An operator will be considered *binary* if its domain is the cross of property graph collections
$\Omega : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$.

Name	Description
Unary Operators	
Aggregation	Add aggregated values to graphs (Definition 4.4).
Selection	Determine a subset of graphs (Definition 4.5).
Transformation	Change properties (Definition 4.6).
Subgraph	Determine subgraphs by vertex and edge predicates.
Pattern Matching	Determine all subgraphs that match a given pattern [84].
Grouping	Create structurally condensed graphs [88].
Binary Operators	
Union	Union of graph ids.
Intersection	Intersection of graph ids.
Difference	Difference of graph ids.
Combination	Union of vertices and edges.
Overlap	Intersection of vertices and edges.
Exclusion	Difference of vertices and edges .
Equality	Test for equality.

Table 4.1: Operators supported by Gradoop at the time of May 2018.

In the initial publication of EPGM [89] there is a further distinction between graph and graph collection operators. In this dissertation, we consider "single graphs" to be a special case of graph collections where $|G| = 1$, i.e., those that contain only a single graph. We further consider property graph collections, in the following simply denoted by *graph collections*, to be *immutable*, i.e., there is no update of graph collections s.t. every operator execution $\Omega(C) \mapsto C'$ puts out a copy C' of input collection C.

Definition 4.3 (Graph Collection Copy) Let C, C' be two graph collections then $C' = \langle G', V', E', \gamma', \varsigma', \tau', Y', D', \pi' \rangle$ will be copy of $C = \langle G, V, E, \gamma, \varsigma, \tau, Y, D, \pi \rangle$ if $V = V' \wedge E = E' \wedge \varsigma = \varsigma' \wedge \tau = \tau' \wedge Y = Y' \wedge D = D' \wedge \pi = \pi'$.

Only a subset of EPGM operators is required to implement BIIIG. In particular, these are three operators. First, there is *aggregation* to calculate business measures on the graph level. The operator takes advantage of the concept of graph properties to persist its result. Second, the *selection* operator allows to filter graph collections by a predicate based on graph properties. Third, the *transformation* operator allows to create, delete or modify properties. In the following operator definitions, we will describe them in terms of their output's difference to a copy:

Definition 4.4 (Aggregation) Aggregation is an operator $\Gamma_f^k(C) \mapsto C'$ with two parameters: k is a property key and $f : G \to D$ is an *aggregate function* that maps graphs to the set of property values. C' is a copy of C, except that the result of f is persisted for every graph s.t. $\forall g \in G'.\pi(g, y) = f(g)$.

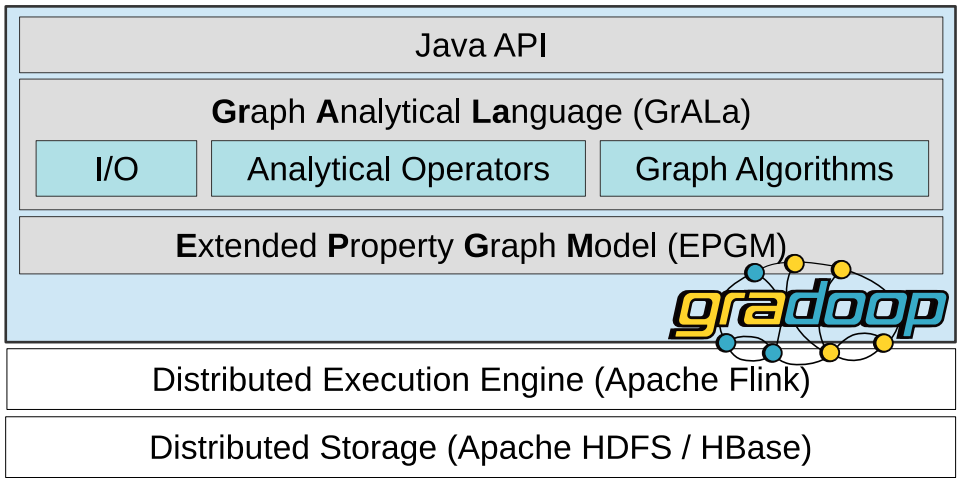


Figure 4.2: GRADOOP architecture [85].

Specific aggregate functions may access all data related to the input graph such as its vertices, edges and properties. A simple example of an aggregate function is the vertex count of a graph, formally $f(g) \mapsto |V_g|; \forall v \in V_g.g \in \gamma(v)$. If applied, every graph of the result collection will have a property reflecting its vertex count. In BIIIG, the aggregation operator is used to calculate business measures (Section 3.4).

Definition 4.5 (Selection) Selection is an operator $\Sigma_p(C) \mapsto C'$ that is parameterized by a predicate $p : G \rightarrow \{true, false\}$. C' is a copy of C which contains exactly those graphs satisfying the predicate
s.t. $G' \subseteq G \wedge \forall g \in G.(p(g) = false \vee g \in G')$.

The graph selection operator is similar to the relational selection operator [37] but selects graphs from a collection instead of rows from a table. For example, suppose a previous aggregation ensures the availability of a vertexCount property then all graphs with more than three vertices can be selected by the predicate $p(g) \mapsto \pi(g, \text{vertexCount}) > 3$. In the context of BIIIG, the operator is, for example, used to select graphs by business measures.

Definition 4.6 (Property Transformation) Property transformation is an operator $\Pi_f(C) \mapsto C'$ with a *transformation function* $f : (G' \cup V' \cup E') \rightarrow \mathcal{P}(K' \times D')$ as parameter. The operator preserves graph memberships of vertices and edges as well as source and target of all edges s.t. $V = V' \wedge E = E' \wedge \varsigma = \varsigma' \wedge \tau = \tau'$. Function f completely defines properties for all elements s.t.
$\Big(\forall \langle x, y\rangle \in (G' \cup V' \cup E') \times Y'\Big).\Big(\pi'(x, y) = \varepsilon \vee \exists \langle y, \pi'(x, y)\rangle \in f(x)\Big)$.

Property transformation allows a complete change of properties for all elements of a graph collection. For BIIIG, this is in particular necessary to implement the input normalization of characteristic subgraph mining (Section 3.5).

4.3 The Gradoop Framework

Although great research efforts have been made to develop efficient graph mining algorithms, most implementations are stand-alone research prototypes. Thus, combining graph operators and graph mining algorithms to complex analyses like characteristic subgraph mining requires the combination of different tools. This quickly becomes a difficult task as such tools may differ with regard to the underlying platform, graph models, availability (e.g., source code, binaries, on request only) or in- and output formats. In consequence, answering a single analytical question requires excessive development effort to set up a toolchain whose execution can be inefficient and prone to failures. For non-graph data (e.g., relational or multidimensional) database systems of large vendors and big data processing platforms [121] already provide toolkits with support for seamless multi-step analyses. However, there in no graph pendant so far.

To support such multi-step graph analytics in a single system, we started developing Gradoop [89]. The system shall enable flexible evaluation and modification of graph data by declarative analytical programs composed from different graph operators. To reach this goal, Gradoop is the first system that implements EPGM. In particular, Gradoop supports the declaration of complex analytical programs composed from EPGM operators. Further on, these programs are designed to be executed on computing clusters without shared memory (*shared-nothing clusters*). The development of Gradoop was motivated by missing functionality of graph processing systems (Section 2.4) and the insufficient scalability of graph databases in Big Data scenarios (Section 2.5). At the time of this dissertation, the system was under active development and new operators were added continuously. The source code[1] of Gradoop is available online under an Open Source license.

4.3.1 Concept

Gradoop is designed to be implemented on top of a big data processing platform such as Apache Spark [202] or Apache Flink [30]. The fundamental concepts of these platforms are distributed collections of data objects (*datasets*) and *transformations* of these. In these systems application logic is expressed by *dataflow programs*. These are represented by a directed acyclic graph (DAG) where vertices represent datasets and edges represent transformations. In comparison to MapReduce [42], these frameworks offer a wider range of operators as well as the possibility to hold data in distributed main memory between single processing steps. There are also graph processing libraries based on such systems (e.g., Apache Spark GraphX [192], Apache Flink Gelly [30]). However, these libraries neither include a rich data model such as EPGM nor operators based thereon.

[1] `www.gradoop.com`

Because of this lack, using these systems to answer complex analytical questions that involve multiple graph operators is still causing notable programming effort. Further on, there is no abstraction of graph collections. By analogy to general distributed dataflow programs, Gradoop's analytical programs also have the form of DAGs but vertices represent either graphs or graph collections and edges represent either built-in operators or custom algorithms, e.g., those for graph mining. Additionally, Gradoop supports different data sources and sinks (e.g., files, HBase). Programs are declared using a domain specific language GrALa (**Gr**aph **A**nalytical **La**nguage). Behind the scenes, operators and algorithms are mapped to datasets and transformations of the underlying big data processing platform and, thus, are horizontally scalable by default.

4.3.2 Implementation

After an initial prototype [86] based on MapReduce and Apache Giraph [225], the current version [85] is implemented on top of the distributed dataflow framework Apache Flink [30], the successor of the former research project Stratosphere [4]. Figure 4.2 shows the architecture of Gradoop. Data is persisted either directly in the distributed file system (Apache HDFS) or in the distributed wide column store Apache HBase [95]. Besides direct HBase storage, Gradoop supports multiple in- and output file formats, for example, CSV, JSON and DOT[2]. It also provides interfaces to conveniently implement additional sources and sinks for arbitrary databases or further file formats.

Data representation and operator execution are tailored to Flink. To express and run Gradoop programs there is a Java API that reflects GrALa. Operators are implemented using Apache Flink's dataset transformations. In the default representation, a property graph collection is represented by three distributed datasets for graphs (without vertices and edges, so-called *graph heads*), vertices and edges. All elements of these datasets include a globally unique indentifier, a mandatory and reserved label property as well as further arbitrary properties. The dedicated label properties are, for example, used to improve query performance by indexing vertices of the same semantic class [217]. For large collections of rather small graphs, it is also possible to keep data in a single dataset of graphs which include all (potentially redundant) vertices and edges. Depending on the executed algorithm, different mappings of property graph collections to Flink's datasets will influence their runtime. However, different implementations will be required if an operator should be executed on different model mappings.

[2] `http://www.graphviz.org/doc/info/lang.html`

4.3.3 Analytical Programs

The domain specific language GrALa (**Gr**aph **A**nalytical **La**nguage) is used to express EPGM programs. GrALa includes the structural concepts of single graphs and collections of these. Every GrALa program has one or more data sources and one data sink. Further on there is a distinction between general-purpose operators that are part of our data model (e.g., the union of two graph collections) and specific algorithms (e.g., for graph pattern mining). To support custom graph mining algorithms, Gradoop offers the generic *call* operator and fitting Java interfaces whose implementations can be included in analytical programs. The provided interfaces cover all algorithms with one or two graphs or graph collections as input and one graph or graph collection as output in arbitrary constellation. For example, a graph partitioning algorithm takes a single graph as input and results into a collection of partitions. To be compatible with EPGM, all algorithm implementations must be capable to handle multigraphs. Further on, it must be considered that EPGM edges are always directed. However, to support undirected graph data (e.g., chemical compounds) algorithm implementations may provide a parameter to optionally ignore edge directions.

4.4 FoodBroker Data Generator

Data of business information systems is basically unavailable. The main reason might be data protection concerns of companies. Further on, business data, in partiular large amounts of it, carry a certain value and will not be provided to public research. Thus, falling back on synthetic data is the only option. If there is a suitable data generator, synthetic data will even have the advantage of scalability. However, typical data generators (Section 2.9) are designed for performance evaluations and their data is not complex enough to evaluate elaborate analytics as targeted by BIIIG. For this reason, we developed FoodBroker, a data generator that is using business process simulation to create datasets with characteristics similar to data from business information systems. Although FoodBroker was initially intended to create relational test data [142] there is also a recent Gradoop implementation by Kemper [92] that directly creates EPGM data.

4.4.1 Requirements

Business information systems support the operations of a company and store data related to its business processes. In particular, there are different domain objects such as employees, products or sales orders as well as mutual relationships. Business information systems typically store objects in relational databases. Companies often use multiple systems for different purposes. For the evaluation of BI-

IIG's data integration component (Chapter 5) FoodBroker can be used to create relational data that emulates multiple systems. The main analytical goal of BIIIG is the detection of correlations among domain objects, their relationships and process outcomes. For example, the output of a trade process is its financial result. Here, unfriendly sales people could have a negative influence while fast logistics companies may have a positive impact. Thus, data generated by FoodBroker must reflect this kind of influences. The precise requirements are as follows:

1. The generated data must contain *heterogenous domain objects*, i.e., they must belong to different classes representing both master or transaction data.

2. There must be *heterogenous relationships* among domain, i.e., different relationship types. Relationships further have to involve master and transaction data in any combination.

3. The data must represent *business process cases*.

4. The data must contain transaction data with *measure properties*, master data with *dimension properties* and significant *correlations* between dimension properties and measure values, i.e., there shall be master data objects that will always have a positive or negative impact on the process outcome.

5. The data must emulate origins from *multiple interrelated systems* to reflect jointly used business information systems.

6. To support benchmarking datasets must have *scalable size*.

To the best of our knowledge, before FoodBroker there was no data generator that met all of the stated requirements.

4.4.2 Simulation

The FoodBroker simulation reflects the core business of a fictive company that is trading food between producers (vendors) and retailers (customers). The company only offers a brokerage service and has no warehouse. Process-related data is recorded in an enterprise resource planning (ERP) system and a customer issue tracking (CIT) system. Before the actual simulation is started, FoodBroker generates master data in a customizable manner. Transaction data is created during the simulation of two interrelated subprocesses. One process is for food brokerage and one is for complaint handling. All data recorded during complaint handling belongs to an initial case of food brokerage, i.e., there are no cases of only complaint handling.

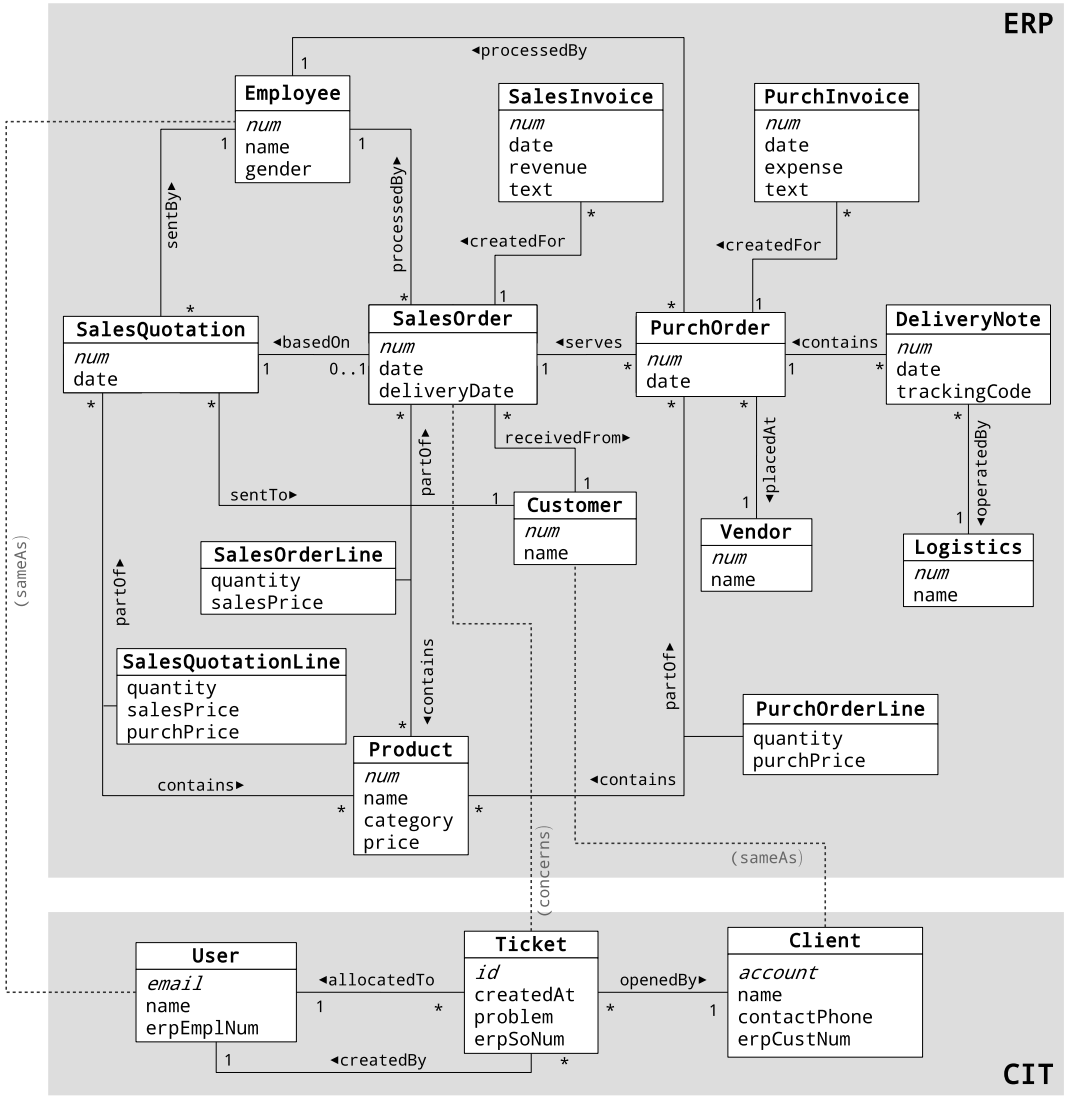


Figure 4.3: FoodBroker Schema : The outer rectangles show the boundaries of two systems ERP and CIT. Database tables correspond either to classes or n:m relationship types (*Line). Primary keys are highlighted by italic letters. Relationship types are shown as solid lines. Foreign keys are attached to relationship types. Implicit relationship types in between both databases are represented by dotted lines. For each implicit relationship type, there is a corresponding column with prefix erp.

Schema: The schema for the ERP and CIT systems is shown by figure 4.3. The ERP system stores master data objects of the classes Employee, Product, Customer, Vendor and Logistics. Products are categorized into different product categories such as *fruits*, *vegetables* and *nuts*. In the CIT system the instances of master class User refer to employees in the ERP system while master data class Client refers to ERP class Customer. For each master data object we provide a quality attribute with one of the values *good*, *normal* or *bad*. We use these attributes at the simulation of business processes to achieve correlations between certain master data objects and the outcome of process executions. For example, if multiple *good* master data objects are jointly involved, the probabililty of a positive process outcome (e.g., profit) will be increased.

The ERP system further records transaction data related to trading and refunds. In more detail, these are the classes SalesQuotation, SalesOrder, PurchOrder, DeliveryNote, SalesInvoice and PurchInvoice. The *Line classes SalesQuotationLine, SalesOrderLine and PurchOrderLine represent n:m relationship types between the respective transaction classes and Product. The CIT system has only a single transaction class Ticket to represent customer complaints. All transaction classes have relationship types to other transaction and master data classes.

Food Brokerage: A food brokerage starts with a SalesQuotation sent by a random Employee to a random Customer. A quotation has SalesQuotationLines that refer to random products. Each SalesQuotationLine provides a salesPrice that is determined by adding a sales margin to the product's purchPrice. A SalesQuotation can be either confirmed or rejected. To simulate the interaction of employee and customer, the probability of confirmation as well as the sales margin will be significantly higher if a *good* Employee and a *good* Customer are involved and correspondingly lower for *normal* or *bad* master data objects. A confirmed quotation results in a SalesOrder and a set of SalesOrderLines. The SalesOrder includes a reference to the underlying SalesQuotation as well as a deliveryDate. To reflect partial confirmations, there may be fewer SalesOrderLines than SalesQuotationLines. While the Customer is the same as the one of the SalesQuotation, a new Employee will process the SalesOrder.

For each SalesOrder, one ore more PurchOrders, each with one or more PurchOrderLines, will be placed at random Vendors. A random Employee (purchaser) is associated per PurchOrder. Actual purchase prices are subject to variations. To simulate the interaction of purchaser and vendor, a *good* Employee and a *good* Vendor will lead to a lower purchPrice in comparison to *normal* or *bad* ones. Furtheron, a *good* Employee will place PurchOrders faster. After a PurchOrder is processed by the Vendor, the company will receive information about the DeliveryNote, in particular date of delivery, operating Logistics company and operator-specific trackingCode. The delivery time is influenced by the quality of both Ven-

dor and Logistics company such that *good* business partners will lead to faster delivery than *bad* or *normal* ones. Finally, one SalesInvoice per SalesOrder will be sent to the Customer and one PurchInvoice per PurchOrder will be received from the corresponding Vendor. All transaction data objects created within cases of food brokerage refer to their predecessor (except SalesQuotation) and provide a date property with a value greater than or equal to the one of the predecessor.

Complaint Handling: For every customer complaint an instance of class Ticket that refers to the corresponding SalesOrder object will be created. For the first complaint per Customer, additionally a Client instance will be created. The employee that handles the complaint is recorded in class User by analogy to Client. There are two problems which may cause complaints: *late delivery* and *bad product quality*. Late delivery complaints will occur, if the date of a DeliveryNote is later than the agreed deliveryDate of the SalesOrder, for example, due to a *bad* Employee processing the SalesOrder or a PurchaseOrder, a *bad* Vendor, a *bad* Logistics company or combinations of those. Bad quality complaints may be caused by *bad* Products, a *bad* Vendor, a *bad* Logistics company or combinations of these.

A Ticket may lead to refunds which are recorded in the form of SalesInvoice objects with negative revenue. While the constellation of a *good* Employee allocated to the Ticket and a *good* Customer may lead to low or no refund, *bad* ones will lead to higher refund. If the problem was caused by the Vendor, there will be also a refund for the company in the form of a PurchInvoice with negative expenses. While the constellation of a *good* Employee and a *good* Vendor may lead to high refund, *bad* ones will lead to lower or no refund.

Correlations: Due to the influence of master data's quality properties, any FoodBroker dataset contains complex correlations between master data instances and the process outcome. The latter can be measured in the form of a financial result. The financial result is calculated based on business transaction graphs by aggregating all revenue- and expense-related properties, in particular those of SalesInvoice and PurchInvoice. Thus, it becomes possible to analyze the influence of the different master objects (employees, customers,vendors, etc.) as well as their interaction (relationship pattern) on the financial result.

4.4.3 Implementation

The initial implementation [142] was written in Java 1.7 and executes process simulation in parallel on all available threads of a machine. The generated data is stored in a MySQL database dump with separate databases for the ERP and CIT systems. To support BIIIG's basic approach to metadata acquisition (Section 5.2.2), every class of both databases has a dedicated table with the same name and 1:n

Class	Configuration	Default Value
Master Data		
`Employee`	number of instances	$30 + 10 \times SF$
	proportions of good/normal/bad inst.	$0.1/0.8/0.1$
`Product`	number of instances	$1000 + 10 \times SF$
	proportions of good/normal/bad inst.	$0.1/0.8/0.1$
	list price range	0.5..8.5
`Customer`	number of instances	$50 + 20 \times SF$
	proportions of good/normal/bad inst.	$0.1/0.8/0.1$
`Vendor`	number of instances	$10 + 5 \times SF$
	proportions of good/normal/bad inst.	$0.1/0.8/0.1$
`Logistics`	number of instances	$10 + 0 \times SF$
	proportions of good/normal/bad inst.	$0.1/0.8/0.1$
Food Brokerage		
`Process`	number of cases	$10000 \times SF$
	date range	2014-01-01..2014-12-31
`SalesQuotation`	products per quotation (lines)	1..20
	quantity per product	1..100
	sales margin/*IM*	0.05/0.02
	confirmation probability/*IM*	0.6/0.2
	line confirmation probability	0.9
	confirmation delay/*IM*	0..20/5
`SalesOrder`	agreed delivery delay/*IM*	2..4/1
	purchase delay/*IM*	0..2/2
	invoice delay/*IM*	0..3/-2
`PurchOrder`	price variation/*IM*	0.01/-0.02
	delivery delay/*IM*	0..1/1
	invoice delay/*IM*	2..5/3
Complaint Handling		
`Ticket`	probability of bad quality/*IM*	0.05/-0.1
	sales refund/*IM*	0.1/-0.05
	purchase refund/*IM*	0.1/0.05

SF abbreviates **s**cale **f**actor
number of instance configurations are defined by linear functions $a + b \times SF$

IM abbreviates **i**mpact per **m**aster data instance
$IM > 0$ means good/bad master data instances increase/decrease value
$IM < 0$ means good/bad master data instances decrease/increase value

Table 4.2: FoodBroker Configuration Parameters

	Master Data		Transaction Data			
SF	Objects	Time	Objects	Relationships	Time	Dump Size
1	1.1K	4s	73K	380K	4s	42MB
10	1.7K	4s	725K	3.8M	25s	426MB
100	6.8K	4s	7.2M	38M	4min	4.1GB
1000	67K	8s	68M	360M	35min	39GB

Table 4.3: Measures of FoodBroker datasets for different scale factors (SF)

.

associations correspond to foreign keys in class tables where the column names represent the relationship type. m:n associations are stored in separate tables. The database dump includes SQL statements for the creation of tables. All instances of a class are inserted into the tables corresponding to their class. Domain-specific string values such as employee or product names are provided by an embedded SQLite database.

Scalability in terms of different data sizes is controlled by a scale factor (*SF*). This makes it possible to create datasets with equal criteria regarding distributions, value ranges and probabilities but with a different number of instances. The number of master data instances and the number of simulated cases are defined by linear functions. The simulation function has a default slope of 10,000 per scale factor. We take into account that master data instances do not scale proportionally to cases. For example, there is only a limited amount of logistics companies. Thus, the functions of all master data classes not only have a specific slope but also have a y-intercept to specify a minimum number of instances.

Besides the scale factor and the resulting data growth, the generation of master data but also the process simulation are customizable by several configuration parameters. For example, one can set the confirmation probability of quotations or the influence of master data quality criteria on that probability. All configurations are set within a single file. An overview of the configuration parameters is provided in table 4.2. While using FoodBroker for benchmarks requires fixed configurations and variable scale, variable configurations at a fixed scale can be used to evaluate business intelligence applications at different scenarios. Table 4.3 shows dataset measures for different scale factors using the standard configuration. With respect to runtime, our current implementation shows a linear behavior for increasing scale factors. All datasets were generated on a workstation containing an Intel Xeon Quadcore, 8GB RAM and a standard HDD.

Three years after the initial release, Kemper implemented a second version within the Gradoop framework (Section 4.3) to support a parallel execution on shared nothing clusters [92]. However, since this version creates EPGM data directly and cannot be used to evaluate the transformation of relational to graph data. The source codes of single machine[3] and distributed[4] versions are available to the public.

4.5 Conclusion

Multiple logical partitions of a single large graphs as well as collections of these are especially, but not only, valuable for business data analytics. Graph properties and respective operators are the base for complex analytical programs on graph data. With the initial motivation to serve BIIIG's requirements, we proposed the novel Extended Property Graph Model with many distinct features. With FoodBroker, we also created a data generator that allows to test correctness and scalability of graph-analytical applications. Today, EPGM is the foundation of Gradoop, a distributed system for declarative graph analytics. At the time of this dissertation Gradoop has been under ongoing development since 2014.

[3] https://github.com/dbs-leipzig/foodbroker

[4] https://github.com/dbs-leipzig/gradoop/tree/master/gradoop-flink/src/main/java/org/gradoop/flink/datagen/transactions/foodbroker

Chapter 5

Graph-based Transformation and Integration of Data

Before business data can be analyzed by graph algorithms it must be turned into graphs or collections of these. This section is dedicated to the problems related to this issue. After an overview (Section 5.1) we will study details about BIIIG's metadata management (Section 5.2), data transformation (Section 5.3) and data integration (Section 5.4). We will also present a practical algorithm to extract business transaction graphs from business information data (Sections 5.5) as well as results of experimental evaluations with FoodBroker and real ERP data (Section 5.6). Finally, there will be a brief conclusion of these essential parts of BIIIG (Section 5.7).

5.1 Overview

In the last decades business intelligence technologies have been adopted by many enterprises. Most prevailing are data warehouse and OLAP approaches based on the relational model and its descendant, the multidimensional model [33]. By contrast, graph-based business intelligence is a fairly new approach. Compared to traditional approaches, graph data models promise significant benefits in terms of analytical flexibility, in particular to evaluate relationships without a predefined and rather static data warehouse schema. In particular, they allow to integrate all relationships first and to select the relevant ones on demand. This is in particular useful for data scientists who perform experimental analyses whose results cannot be predicted prior to the data integration process. Powerful graph models such as the property graph model [157] are promising for the implementation of this approach as they allow a flexible and uniform representation of heterogeneous domain objects and their relationships.

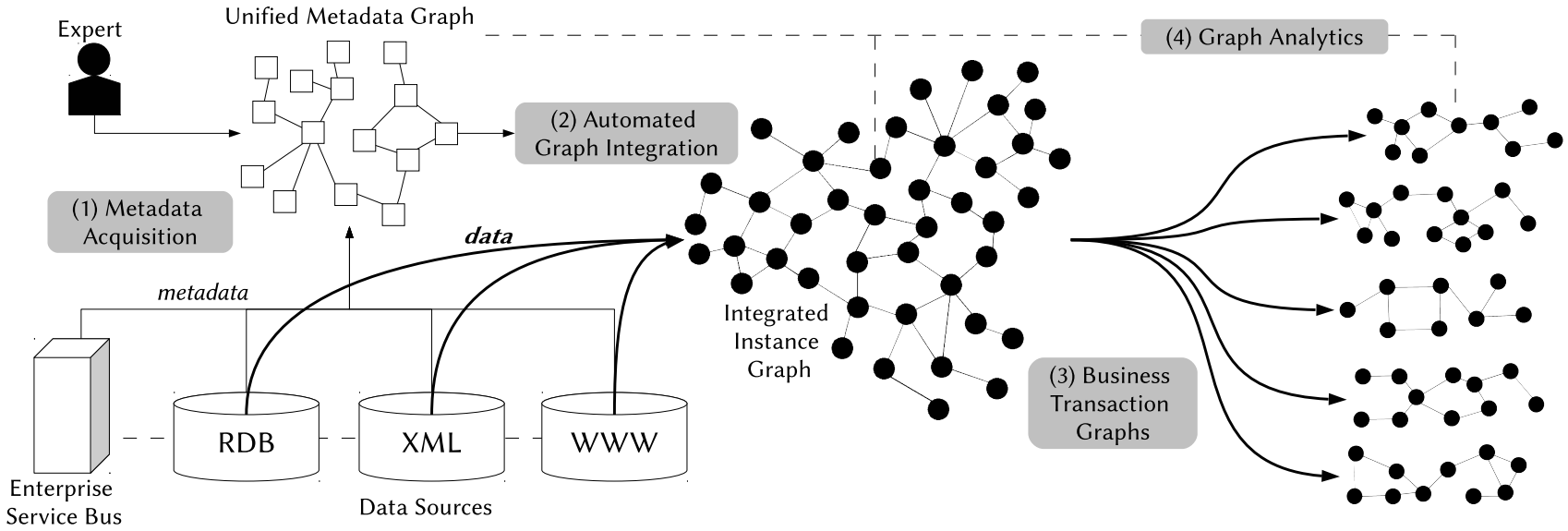


Figure 5.1: Overview of the BIIIG framework.

By a respective graph model, BIIIG is able to represent basically all structured data sources. Figure 5.1 illustrates all data processing steps that are performed to enable graph-based analytics on non-graph source data:

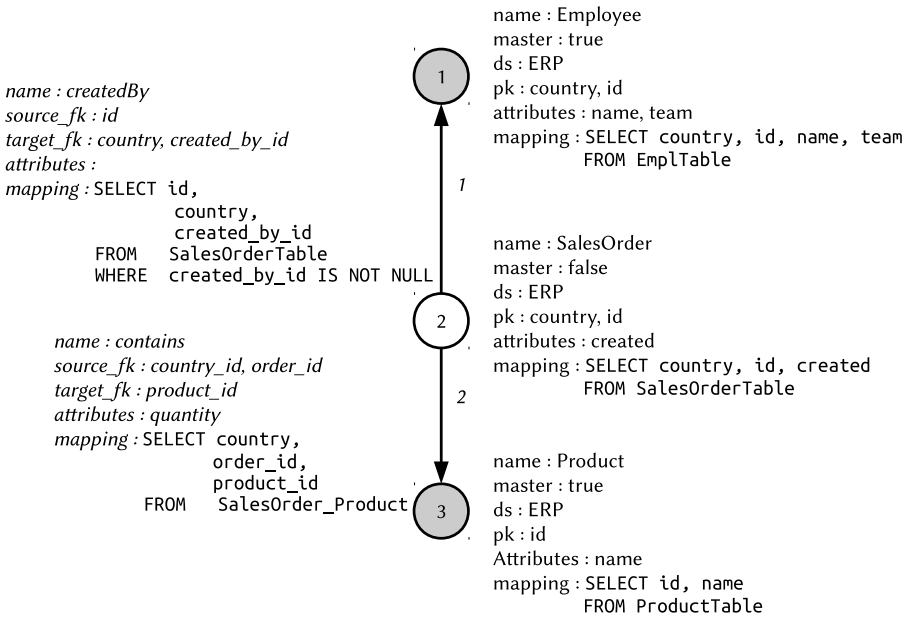
(1) In the first step (Section 5.2) the schema of all data sources is represented by a *unified metadata graph (UMG)*. In the UMG vertices represent classes and edges represent their relationship types. The UMG combines metadata from all data sources and serves BIIIG users as an intuitive and generic metadata model. For relational sources, the UMG can be created automatically in large parts based on schema information provided by databases.

(2) Afterwards (Section 5.3), instances of classes and relationship types are loaded into BIIIG's main data store, the *Integrated Instance Graph (IIG)*. In this graph vertices represent data objects and edges represent relationships. The IIG is generated automatically based on the UMG. To make the IIG data self-descriptive and to achieve a high semantic expressiveness vertices and edges are associated to classes and relationship types by reserved properties.

(3) Both, the UMG and the IIG play a role in BIIIG's approach to data integration (Section 5.4). The UMG may already contain relationship types across data sources. Such relationship types are possible for both transaction and master data. For example, tickets in a customer issue tracking system may be related to sales orders in an ERP system and master data about customers and employees can be held redundantly in multiple systems. The actual data integration happens in the IIG where relationships across systems are materialized by edges. Additionally, groups of vertices that correspond to each other are fused into a single representative *(vertex fusion)*.

(4) We present a generic algorithm (Section 5.5) to extract business transaction graphs (Section 3.2) from the IIG. We expect the algorithm to be applicable to all business information systems under no or with only minor domain-specific modifications.

(5) All of BIIIG's graph structures are valuable for graph-based business analytics. Analysts can use the UMG and the IIG to access any piece of recorded data including all of its relationships on both metadata and instance level. With a framework such as Gradoop, this can be done, for example, by declarative pattern matching queries [84] or flexible graph summaries [88]. Furthermore, graph mining techniques can be applied to both the IIG or the set of BTGs (Chapters 6, 7).



(a) Example Unified Metadata Graph (UMG). Vertices represent classes and edges represent relationship types. Source mappings are implemented as SQL queries.

Employee		
column	**type**	**PK**
country	char(2)	1
id	int	2
name	varchar	
team	varchar	

SalesOrder		
column	**type**	**PK**
country	char(2)	1
id	int	2
created	date	
created_by_id	int	
foreign key	**columns**	**target**
createdBy	[country, created_by]	Employee

contains		
column	**type**	**PK**
country	char(2)	1
order_id	int	
product_id	int	
quantity	int	
foreign key	**columns**	**target**
order	[country, order_id]	SalesOrder
product	[product_id]	Product

Product		
column	**type**	**PK**
id	int	1
name	varchar	

(b) Relational database schema of the UMG of Figure 5.2a.

Figure 5.2: Example unified metadata graph with associated relational database.

5.2 Metadata Management

BIIIG is targeting business information systems as its main data source. At the instance level these systems store domain data objects and their mutual relationships. Data objects are typically classified and there are predefined types of relationships, i.e., there is a domain model. BIIIG reflects the models of all data sources in a single *Unified Metadata Graph (UMG)*. The UMG is a uniform representation of classes and all possible relationship types. The UMG further supports relationship types across sources and, thus, plays an important role for data integration (Section 5.4). Additionally, the UMG is valuable for data analysts as it reveals available classes and relationship types in a model of a, compared to an instance graph, compact size.

Since business information systems use databases to persist data, there is also a mapping between the domain model and the database's data organisation. The UMG stores these mappings in a unified way to enable an automated data transformation (Section 5.3). For relational databases it is also possible to automate parts of UMG and mapping generation. The remainder of this section will provide more details about BIIIG's metadata management.

5.2.1 Unified Metadata Graph

The Unified Metadata Graph (UMG) represents classes and relationship types of every data source as well as relationship types across sources. The UMG is a property graph according to Definition 2.5 where vertices V represent classes of domain data objects and edges E represent possible relationship types among classes. Figure 5.2 shows an example UMG of a relational database.

The UMG has a schema, i.e., there is a set of reserved property keys Y_V for vertices s.t. $\forall y \in Y_V . \forall v \in V . \pi(v, y) \neq \varepsilon$. As illustrated by Figure 5.2a we store the properties $Y_V = \{\mathsf{name}, \mathsf{master}, \mathsf{ds}, \mathsf{pk}, \mathsf{attributes}, \mathsf{mapping}\}$, in particular the class name, a master data indicator $\pi(v, \mathsf{master}) \in \{true, false\}$, a data source identifier ds, the primary key pk, a list of attributes and a source database mapping. By analogy, there is a set of property keys Y_E for edges s.t. $\forall y \in Y_E . \forall e \in E . \pi(e, y) \neq \varepsilon$. We store the reserved properties $Y_E = \{\mathsf{name}, \mathsf{source\_fk}, \mathsf{target\_fk}, \mathsf{attributes}, \mathsf{mapping}\}$ that include the relationship type name.

The values of pk, source_fk, target_fk and attributes are lists of property keys. $\pi(v, \mathsf{attributes})$ may be empty. Note, that an empty list has a different meaning than ε. In particular, ε means the non-existence of a property but if a class or relationship type has an empty attributes value, its instances will have no attributes. More details about the role of ds,pk, source_fk and target_fk properties will be provided in Section 5.3.

The values of properties with key mapping are mappings between the classes (or relationship types) and data sources. They enable automated graph transformation (Section 5.3). Generally, mappings are database queries that return lists of data objects (or relationships), each represented by a property list. A property list that represents a data object must contain a value for pk and may contain a value for all attributes. Respectively, for every relationship there must be values for all source_fk and target_fk.

We will use the most important case, relational databases, as an example. Here, mappings have the form of SQL queries and column names of a query's result correspond to property keys. Figure 5.2 shows an example UMG with a mapped relational database. In this simple mapping, tables correspond to classes, optionally with embedded 1:n relationship types, or to m:n relationship types. For example, table SalesOrderTable persists objects of class SalesOrder as well as relationships of type createdBy and table SalesOrder_Product holds relationships of type contains. Since relational databases allow the explicit persistence of non-existence (NULL), the mapping of createdBy ensures by a NOT NULL predicate that only actually existing relationships will be selected.

5.2.2 Metadata Acquisition

Metadata mappings are specific for every data source. In particular they are not only dependent on the database type but also on the object mapping strategy of the source system [132]. In an ideal case, the metadata of a complete data source or at least a rough proposal can be acquired automatically. Therefore classes, relationship types and their original database mappings must be accessible in the source system. However, a domain expert will always be required to categorize master and transaction data, to review the proposal and to add missing relationships such as those across data sources. For the main use case of business information systems based on relational databases we propose the following automated approach to metadata acquisition:

The approach includes two phases: First, all table metadata is scanned to extract classes. Second, tables are scanned again to extract relationship types. The two-phase approach is required to ensure consistency since all classes must be known before relationship types can be added to the UMG. Algorithm 5.1 shows the class acquisition. The algorithm's inputs are a data source id (which will be added to all classes), the relational schema of the source database and the unified metadata graph. The latter will be empty, if the current one is the first acquired data source but will already contain classes and relationship types otherwise. To be passed to the second phase, the algorithm finally returns the UMG. The basic idea of this approach is that tables either represent classes or m:n relationships. Each table (line 1) is considered as a class candidate. The criterion for a table

Algorithm 5.1 Relational Metadata Acquisition (Part I - Classes)

Input: data source id (ds), relational schema (schema), unified metadata graph (umg)
Output: unified metadata graph (umg)

```
1:  for all table in schema.tables do
2:      pk = getPrimaryKey(table)
3:      if pk != null then
4:          class = new Vertex
5:          class[ds] = ds
6:          class[name] = table.name
7:          class[pk] = pk.columns
8:          fks = getForeignKeys(table)
9:          for all column in table.columns do
10:             if not pk.contains(column) and not fks.contains(column) then
11:                 class[attributes].add(column)
12:             end if
13:         end for
14:         class[mapping] = createSQLMapping(table, class)
15:         umg.vertices.add(class)
16:     end if
17: end for
18: return umg
```

to represent a class is the existence of a primary key (lines 2, 3). Primary key extraction is represented by the function *getPrimaryKey*(table). There is no general implementation of this function. On the contrary, this is the first time where the particular database system and object mapping strategy must be considered. Typical approaches are the evaluation of database metadata (e.g., PRIMARY KEY constraints), the exploitation of naming conventions (e.g., primary keys are always named id) or lookups in data dictionaries [229] of business information systems. In the case of an existing primary key, a new vertex will be created (line 4). Besides the source identifier (line 5), the table name is used as class name (line 6) and all columns of the primary key are used to set the pk property (line 7).

Analog to primary keys, the extraction of foreign keys is represented by the function *getForeignKeys*(table) and requires a system-specific implementation. Here, the same approaches such as evaluation of database metadata (e.g., FOREIGN KEY constraints), interpretation of naming conventions (e.g., foreign keys are always named {tablename}_id) or lookups in data dictionaries have to be applied. In this part of the algorithm foreign keys play the role of a blacklist for attribute extraction. More precisely, all columns (line 9) except members of the primary key and any foreign key (line 10) are added to the attributes property (line 11). Finally, the class mapping is created (line 14) and the vertex is added to the UMG (line 15). The creation of the source mapping is represented by function *createSQLMapping*(table,

Algorithm 5.2 Relational Metadata Acquisition (Part II - Relationship Types)

Input: relational schema (schema), unified metadata graph (umg)

Output: unified metadata graph (umg)

```
1:  for all table in schema.tables do
2:      pk = getPrimaryKey(table)
3:      fks = getForeignKeys(table)
4:      if pk != null and fks.length > 0 then
5:          // embedded 1:n foreign keys
6:          for all fk : fks do
7:              atts = []
8:              addRelationshipType(umg, fk.name, atts, table.name, pk, fk.targetName, fk)
9:          end for
10:     else if pk == null and fks.length == 2 then
11:         // m:n table
12:         sourceKey = fk[0]
13:         targetKey = fk[1]
14:         atts = []
15:         for all column : table.columns do
16:             if not sourceKey.contains(column) and not targetKey.contains(column) then
17:                 atts.add(column)
18:             end if
19:         end for
20:         addRelationshipType(umg, table.name, atts,
              sourceKey.targetName, sourceKey, targetKey.targetName, targetKey)
21:     end if
22: end for
23: return umg
```

class) which creates a SQL statement that selects all columns that are part of either the pk or the attributes property. Example queries are shown in Figure 5.2a.

Algorithm 5.2 shows the acquisition of relationship types. The algorithm's inputs are again the relational schema of the source database and the unified metadata graph which now contains at least all classes of the same data source. Here, tables are scanned a second time (line 1). To decide, if the table represents either a class table with potentially embedded 1:n relationships or a m:n relationship table, primary key (line 2) and foreign keys (line 3) will be determined using the same functions as Algorithm 5.1. Based on the functions' results there are two cases: In the first case of embedded 1:n relationships there must be a primary key and a non-empty set of foreign keys (line 4). In the second case of a m:n table there must be no primary key but exactly two foreign keys (line 10).

In both cases, finally the method *addRelationshipType* will be called. In the 1:n case, this happens for every foreign key (line 8) but only once per table in

Algorithm 5.3 Method *addRelationshipType*

Input: unified metadata graph (umg), relationship type name (name), relationship type attributes (attributes), source class name (sourceName), source key (sourceKey), target class name (targetName), target key (targetKey)

```
1:  source = umg.getVertexWhere(|class| -> class[name] == sourceName)
2:  target = umg.getVertexWhere(|class| -> class[name] == targetName)
3:  if  source != null and target != null  then
4:      relType = new Edge(source, target)
5:      relType[name] = name
6:      relType[attributes] = attributes
7:      relType[source_fk] = sourceKey.columns
8:      relType[target_fk] = targetKey.columns
9:      class[mapping] = createSQLMapping(table, relType)
10:     umg.edges.add(relType)
11: end if
```

the m:n case (line 20). The first difference between both cases is the type name (first parameter of *addRelationshipType*). In the 1:n case, the key name will be used (line 8). Key names are dependent on the implementation of *getForeignKeys*. In the m:n case, the table name will be used instead (line 20). The second difference refers to attributes. While there are generally no attributes in the 1:n case (line 7) all columns except those involved in the two foreign keys are considered to be attributes in the m:n case (lines 15-19). The method *addRelationshipType* further requires source and target class names as well as respective keys. For the 1:n case, the source class is set by the table name (line 8) since a class must have been derived from the same table in Algorithm 5.1. The target class corresponds to the target table of the foreign key (line 8). Analog to the target of 1:n relationship types, the target table names of both foreign keys will be used in the m:n case (line 20). The same logic applies to the passed keys which are primary as well as the current foreign key in the 1:n case and two foreign keys in the m:n case.

The actual edge creation happens within the method *addRelationshipType*. The method body is shown by Algorithm 5.3. First, source and target are queried from the UMG (lines 1,2). Here, we use lambda predicates ($\lambda : V \rightarrow \{true, false\}$) to represent queries. If both class vertices could be queried (line 3), a consistent relationship type can be created (line 4). The properties for name and attributes are taken over from the method's input (lines 5,6). Further on, the columns of both keys are set as values of source_fk and target_fk (lines 7,8). By analogy to vertices, the function *createSQLMapping*(table, relType) is used to create a mapping (line 9). The function creates a SQL statement that selects all columns that are part of either source_fk, target_fk or the attributes property. Example queries are shown in Figure 5.2a. Finally, the edge is added to the UMG (line 10).

Algorithm 5.4 Automated Graph Transformation

Input: unified metadata graph (umg)

Output: integrated instance graph (iig)

```
1:  iig = new PropertyGraph
2:  for all class in umg.vertices do
3:      for all properties in dbQuery(class[ds], class[mapping]) do
4:          object = new Vertex
5:          object[class] = class[name]
6:          object[master] = class[master]
7:          object[sid] = class[ds] + class[name]
8:          for all key in class[pk] do
9:              object[sid] = object[sid] + properties[key]
10:         end for
11:         for all key in class[attributes] do
12:             object[key] = properties[key] if properties[key] != NULL
13:         end for
14:         iig.vertices.add(object)
15:     end for
16: end for
17: for all relType in umg.edges do
18:     for all properties in dbQuery(relType.source[ds], relType[mapping]) do
19:         sourceSid = relType.source[ds] + relType.source[name]
20:         for all key in relType[source_fk] do
21:             sourceSid = sourceSid + properties[key]
22:         end for
23:         targetSid = relType.target[ds] + relType.target[name]
24:         for all key in relType[target_fk] do
25:             targetSid = targetSid + properties[key]
26:         end for
27:         source = iig.getVertexWhere(|object| -> object[sid] == sourceSid)
28:         target = iig.getVertexWhere(|object| -> object[sid] == targetSid)
29:         if source != null and target != null then
30:             relationship = new Edge(source, target)
31:             relationship[type] = relType[name]
32:             for all key in relType[attributes] do
33:                 relationship[key] = properties[key] if properties[key] != NULL
34:             end for
35:             iig.edges.add(relationship)
36:         end if
37:     end for
38: end for
39: return iig
```

5.3 Graph Transformation

BIIIG's graph transformation is fully automated based on the source-mappings provided by the UMG. The process entails two phases: First, the mappings of all classes are evaluated and for each data object a new vertex is added to the central instance graph. All vertices whose objects are identifiable within the source system will be queryable by a global source identifier. Second, the mappings of all relationships are evaluated and for each relationship a new edge will be added. During the edge creation process, source identifiers are used to query incident vertices. The remainder of this section will provide more details about BIIIG's instance model and source data transformation.

5.3.1 Integrated Instance Graph

The main data store of BIIIG is called *integrated instance graph* (IIG). In this graph each data object is represented by a vertex and each relationship is represented by an edge. Just like the UMG, the IIG is a property graph where vertices V represent data objects and edges E represent their mutual relationships.

The IIG has a partial schema, i.e., there are *reserved properties* as well as an arbitrary number of *data properties* for each vertex and edge. We store values for the keys $Y_V = \{\mathsf{master}, \mathsf{class}, \mathsf{sid}\}$. The values of master data indicator and class are taken over from an object's UMG class during data transformation. The property sid stores a globally unique so-called *source identifier* composed from the data source identifier, class name and local identifier (e.g., a table's primary key value). It provides provenance information and plays a crucial role for the automated transformation process. For edges, only a type property is mandatory. Its value is adopted from an edge's relationship type in the UMG. The properties master, class and type are actually metadata but add semantics to the generally schema-less graph structure. There are no constraints for data properties.

5.3.2 Transformation process

Algorithm 5.4 describes the transformation process in detail. The input of the algorithm is the UMG including data source mappings and the result is the IIG. First, all source data objects are turned into IIG vertices within a loop over all classes (lines 2-16). The class specific source mapping (e.g. SQL statement) is used to query its instances in the form of property lists (line 3). Every new vertex (line 4) obtains the class name (line 5) and master data indicator (line 6) from the currently processed class. The source identifier is generated in lines 7-10. Initially, it includes only data source identifier and class name (line 7). In the subsequent loop a value is added for every pk value (lines 8-10). This simple concatenation is just for illustration. In an actual implementation, a more complex multiplexing could be applied. In the

next loop all (non NULL) attribute properties are transferred to the vertex (lines 11-13). Finally, the vertex will be added to the IIG (line 14).

Second, all relationships are turned into IIG edges within a loop over all relationship types (lines 17-38). By analogy to vertices, the type-specific mapping is used to query relationships, represented by property lists (line 18). The connected vertices will be queried from the IIG by their source identifiers (lines 27, 28). These identifiers are created in the same way as lines 7-10 but from relationship information (lines 19-26). However, we will only continue with the current edge, if vertices exist for both identifiers (line 29). The new edge (line 30) obtains its type property from the currently processed relationship type's name (line 31) and receives all nonempty property values of the relationship (lines 32-34). Finally, the constructed edge will be added to the IIG (line 35).

5.4 Data Integration

BIIIG supports the integration of data from multiple sources. BIIIG's data integration process entails three steps: First, cross-system edges (relationship types) are added to the UMG. Second, cross-system edges (relationships) are automatically added to the IIG since they are based on cross-system relationship types in the UMG. Third, groups of vertices that represent the same logical entity will be fused into a single representative. In the following, we will provide more details about these steps. Note, that logical data integration problems are out of the scope of this dissertation. Instead, we will only discuss technical advantages of the graph model for data integration.

5.4.1 Metadata Integration

Figure 5.3 shows an unified metadata graph (UMG) with cross-system relationship types. There are two reasons for the existence of such edges. First, there could be *transaction references*. For example, the edge of type concerns in Figure 5.3 shows that instances of class Ticket may reference instances of class SalesOrder. In business information systems, such references play an important role. For example, if a customer reports a problem with an order, a ticket will be created. However, it is possible that the order processing is done with an enterprise resource planning (ERP) system while tickets are recorded in a customer issue tracking (CIT) system. Since a ticket cannot always be directly processed by the creating employee, a reference to the affected sales order must be stored. In this way the reference will be available to all employees processing the ticket.

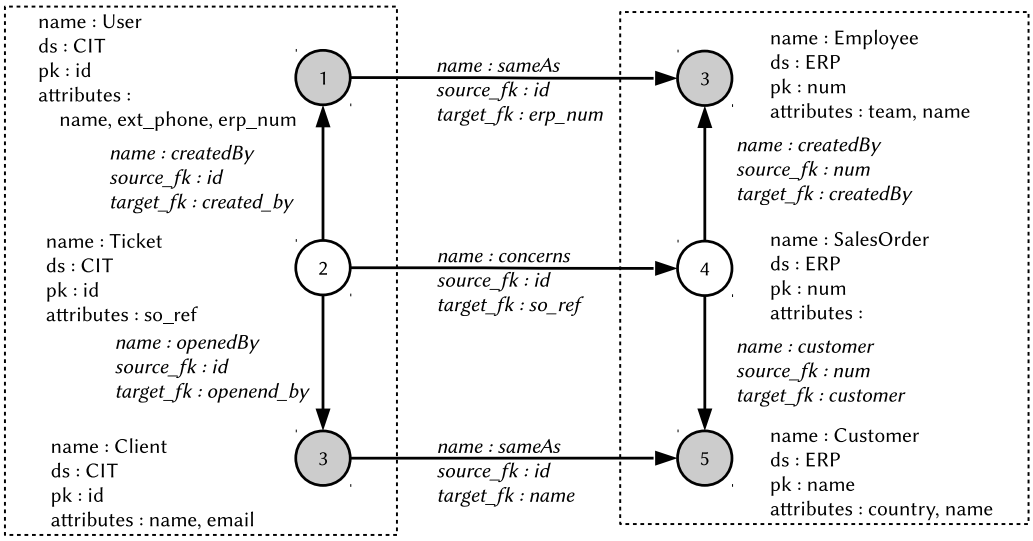


Figure 5.3: Example unified metadata graph with metadata from multiple sources. Vertices represent classes and edges represent relationship types. Gray vertices represent master data classes. Dashed rectangles represent source boundaries and correspond to the ds properties of vertices.

The second reason for cross-system relationships is the redundant storage of master data. Although avoiding redundancy is a general goal for good data management it cannot be practiced in all cases. For example, if multiple systems are used some data can be required in both systems. Figure 5.3 shows two edges of this kind, both of the reserved type *sameAs*. The following scenario illustrates the origin of such cases: Tickets are processed in a call center whose employees, for security purposes, have only access to the CIT system. Human resource management maintains all employee data in the ERP system and sales orders are recorded in the ERP system, too. However, the CIT system requires master data about employees and customers. About employees, to maintain additional CIT specific data (e.g., external phone number) and about customers to enable the call center staff to ensure the caller is really a customer.

For both transaction references and redundant master data a common implementation are dedicated database fields that store the primary key values of the target system. For example, the column so_ref of the table corresponding to the class Ticket references the primary key of SalesOrder records. There are three potential ways of adding cross-system edges to the UMG. First, they can be extracted from a middleware that is managing cross-system relationship types, for example, so-called enterprise service bus systems [122]. Second, they are added manually by a domain expert. Third, the data provided by the UMG can be made available to an approach to automated schema matching [155] whose results can be feed back. Independent of how they were added, cross-system relationship types are stored in the same format as inner-system edges. In consequence, they are normalized for the automated data transformation process (Section 5.3).

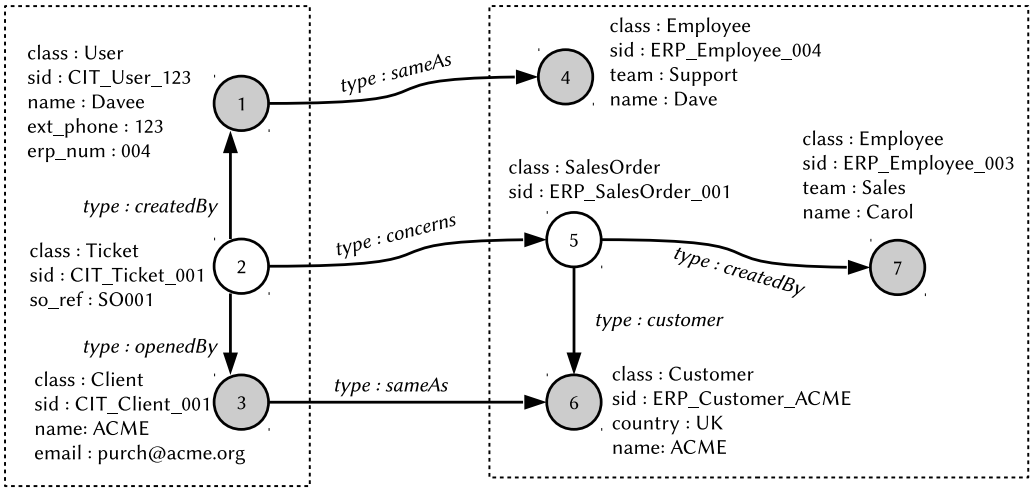


Figure 5.4: Example integrated instance graph with data from multiple sources. Vertices represent data objects and edges represent relationships. Gray vertices represent master data. Dashed rectangles represent source boundaries and correspond to the prefix of sid properties of vertices. The instance graph was generated based on the unified metadata graph shown by Figure 5.3

5.4.2 Instance Integration

Since the automated data transformation process of Section 5.3 is based on the integrated UMG cross-system relationships will be created automatically. This is particularly enabled by our concept of source identifiers which includes source systems as well as class information and, thus, ensures globally unique identifiers. Figure 5.4 shows an example integrated instance graph (IIG) based on the UMG of Figure 5.3. The *sid* properties of vertices show examples for source identifiers whose prefix associates them either to the CIT or the ERP system. For the case of data redundancy it is also possible to detect clusters of corresponding entities on the instance level. For example, as a naive solution any approach to object matching [99] could be applied to the vertex set and every correspondence would result in a new *sameAs* edge. Clusters of these edges could then be interpreted as a single logical entity. More elaborate solutions such as FAMER [163] consider correspondences and their weights only as input of more complex clustering algorithms.

These *correspondence clusters* are only an intermediate step and should be resolved eventually (*deduplication*). Our approach is to separate data integration problems from general graph problems. Thus, BIIIG only solves the latter. This approach is called *vertex fusion*. Here, correspondence clusters are turned into single representatives and all edges from the original vertices are redirected to the representative to ensure consistency. The actual clustering logic (e.g., connected components [44] of only *sameAs* edges or the result of FAMER clustering) as well as the actual data integration logic are specified by the user in the form of a user defined functions (UDFs). The first UDF maps a graph to a collection of subgraphs

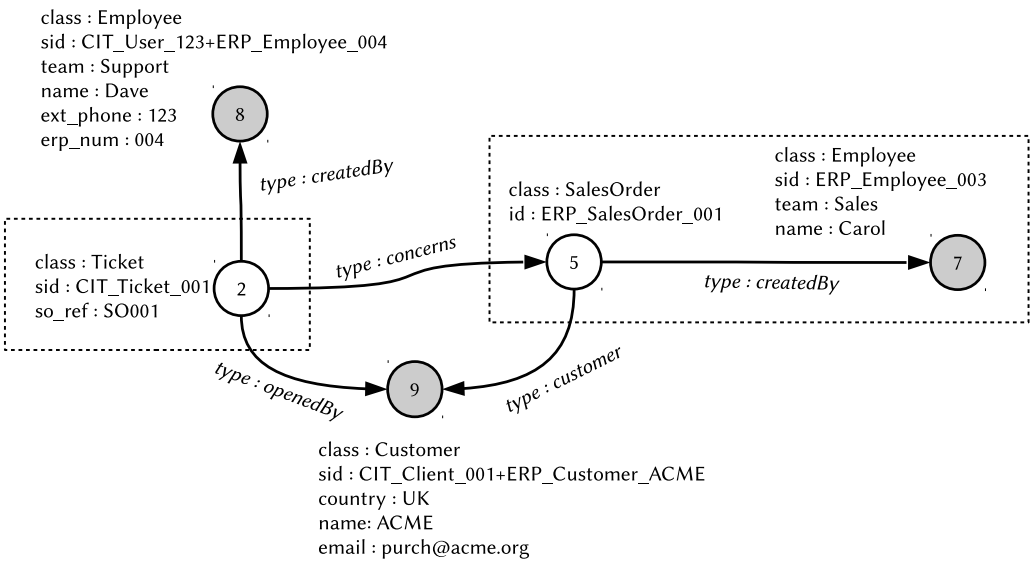


Figure 5.5: Example instance graph of Figure 5.4 after vertex fusion.

(correspondence clusters) and a second UDF maps a graph (correspondence cluster) to a single vertex (representative). Within the second function, potential conflicts such as different values for the same property key must be resolved. Further on, the representative's properties may contain all required provenance information.

Figure 5.5 shows the graph of Figure 5.4 after vertex fusion based on *sameAs* connected components as clustering. In particular, vertices 1,4 were fused into vertex 8 and vertices 3,6 were fused into vertex 9. All edges of the affected vertices were redirected respectively. It can be seen that fused vertices cannot be associated to a single system anymore. To provide provenance, they encode all original source identifiers. The example further shows, that property conflicts were resolved. For example, vertices 1 and 4 have name properties with different values. For vertex 8, the correct one was chosen by a not further specified method. Vertex fusion does not only work for pairs of vertices with a single *sameAs* edge but also for subgraphs where a path of only *sameAs* edges exists between any pair of vertices.

Algorithm 5.5 shows the pseudocode of the vertex fusion process. The algorithm's input is the IIG and user defined functions that specify the application-specific clustering and integration logic (e.g., property conflict resolution). First, the clustering UDF is applied and each cluster is processed separately (line 1). For each cluster the integration UDF is executed to create a representative (line 2) which is added to the IIG (line 3). We assume that all edges that connect vertices within the same cluster are data integration fragments (e.g., sameAs-edges) and delete them from the iig (line 4). The deletion happens before the actual vertex fusion to avoid unnecessary edge redirection. Then, every original vertex is processed in a loop (line 5). First, the original source id of the current vertex is added

Algorithm 5.5 Vertex Fusion

Input: integrated instance graph (iig), clustering function ($clusteringUdf : G \rightarrow \mathcal{P}(G)$), integration function ($integrationUdf : G \rightarrow V$)

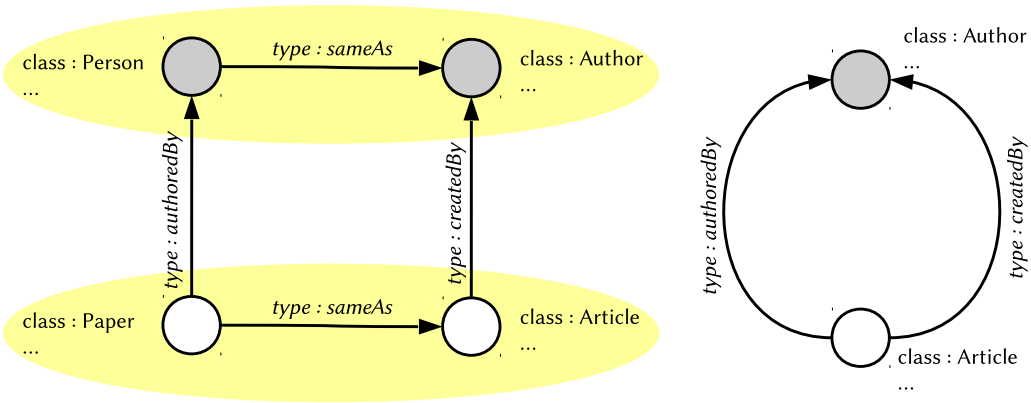
Output: integrated instance graph (iig)

```
1:  for all cluster in clusteringUdf(iig) do
2:      representative = integrationUdf(cluster)
3:      iig.vertices.add(representative)
4:      iig.edges.removeIf(|edge| -> cluster.edges.contains(edge))
5:      for all vertex in cluster.vertices do
6:          representative[sid] = multiplex(representative[sid], vertex[sid])
7:          for all edge in iig.edges.where(|edge| -> edge.source == vertex) do
8:              edge.source = representative
9:          end for
10:         for all edge in iig.edges.where(|edge| -> edge.target == vertex) do
11:             edge.target = representative
12:         end for
13:         iig.vertices.remove(vertex)
14:     end for
15: end for
```

to the representative's source id by a not further specified kind of multiplexing (line 6). Second, all edges whose source (lines 7-9) or target (line 10-12) is the original vertex will be redirected to the representative, i.e., the representative is set as source (line 8) or target (line 11). Finally, the at this point isolated original vertex will be removed from the IIG (line 13).

5.4.3 Deduplication of Relationships

It is possible that vertices of two correspondence clusters are connected by edges and that these edges may logically correspond to each other. Figure 5.6 shows an example for a bibliographic scenario. In Figure 5.6a articles and papers as well as authors and people form disjoint correspondence clusters. Figure 5.6b shows the same graph after vertex fusion. It can be seen that the resulting parallel edges in Figure 5.6b correspond to each other, too. BIIIG provides no solution to this problem yet. However, we want to address deduplication of edges (relationships) as a further important problem of research about graph-based data integration. One possible solution strategy could be the application of object matching strategies among all edges that connect different clusters.



(a) Before vertex fusion. Yellow areas represent vertex correpsondence clusters.

(b) After vertex fusion.

Figure 5.6: Example of the relationship integration problem.

5.5 Business Transaction Graphs

We proposed business transaction graphs to analyze business data (Section 3.2). Business transaction graphs represent all data related to a single business process execution (case). Thus, we developed a generic baseline algorithm for the extraction fo these graphs. The algorithm is based on business data characteristics. In particular, we observed that, with some exceptions (Section 5.5.2), transaction data objects (e.g., phone calls, sales orders and invoices) are created exactly for one case while master data objects (e.g., customer, products and employees) are involved in many cases. We further observed that direct relationships between transaction data objects reflect causal connections, i.e., are specific to a single case, too. For example, an invoice object that refers to a sales order was created to invoice the order. By contrast, relationships among master data objects (e.g., employee-department) are never related to cases. However, relationships between transaction data and master data carry specific information about cases since they provide context, for example, the customer who placed the order of a case.

5.5.1 Algorithm

Based on these observations we developed an algorithm that automatically extracts a collection of business transaction graphs from an integrated instance graph. The algorithm implements the following rules to discover a BTG.

1. Every transaction vertex belongs to exactly one BTG.
2. Every edge with at least one incident transaction vertex belongs to exactly one BTG, in particular, to the same BTG as the transaction vertex.
3. Every edge between two transaction vertices reflects a causal connection, i.e., both vertices belong to the same BTG.

Algorithm 5.6 Business Transaction Graph Isolation

Input: integrated instance graph (iig)

Output: collection of business transaction graphs (btgs)

```
1:  btgs = new GraphCollection
2:  transVertices = iig.vertices.where(|vertex| -> vertex[master] == false)
3:  while transVertices ≠ ∅ do
4:      btg = new Graph
5:      seedVertex = transVertices.getAny()
6:      btg.add(seedVertex)
7:      queue = new Queue
8:      queue.add(seedVertex)
9:      while queue ≠ ∅ do
10:         vertex = queue.poll()
11:         for all edge in iig.edges.where(
            |edge| -> edge.source == vertex || edge.target == vertex) do
12:             if not btg.edges.contains(edge) then
13:                 otherVertex = edge.source == vertex ? edge.target : edge.source
14:                 if not btg.vertices.contains(otherVertex) then
15:                     btg.vertices.add(otherVertex)
16:                     if vertex[master] == false then
17:                         queue.add(otherVertex)
18:                         transVertices.remove(otherVertex)
19:                     end if
20:                 end if
21:                 btg.edges.add(edge)
22:             end if
23:         end for
24:     end while
25:     btgs.add(btg)
26: end while
27: return btgs
```

4. Master data vertices may be part of an arbitrary number of BTGs.

5. Every edge without an incident transaction vertex has no relationship to a specific BTG.

Thus, our algorithm takes an arbitrary transaction vertex (rule 1) and traverses all edges since all of its incident edges must belong to the same BTG (rule 2). For each visited transaction vertex, the traversal will be continued since even vertices that are indirectly linked by causal connections transitively belong to the same BTG (rule 3). By contrast, the traversal is stopped for master data vertices since their incident edges may belong to another BTG (rule 4) or may be even irrelevant for BTGs (rule 5).

The pseudocode of this algorithm is shown by Algorithm 5.6. The algorithm's input is an integrated instance graphs (IIG) and its output is a new collection of business transaction graphs (line 1). In the IIG, all vertices have a master data indicator to distinguish master and transaction data (Section 5.3.1). Based on this property, we first extract a set of only transaction vertices (line 2) which have to be distributed to BTGs. Then, we process this set in a loop until it is empty (line 3). Every iteration results in a new business transaction graph (line 4). Therefore, a random transaction vertex (line 5) is added to this graph (line 6) and to a traversal queue (lines 7,8). The queue represents all vertices of a business transaction graph whose neighborhood has not been traversed yet.

The queue is processed iteratively until it is empty (lines 9). Each iteration starts with polling a vertex (line 10). Initially, this is the seed vertex of line 5. Then, all incident edges of the current vertex are queried and processed (line 11). Edges that are already contained in the current graph will not be processed again (line 12). Since every incident edge of a transaction vertex must have been created for exactly one case (rule 1), we will finally add it to the BTG (line 21) after processing its adjacent vertex (line 13-20). By analogy to edges, an adjacent vertex will only be processed if not already contained in the current BTG (line 14). Otherwise, it will be added to the BTG (line 15). Lines 16 to 18 reflect the main idea of this algorithm with regard to the stated rules: Only if the current vertex is a transaction vertex (line 16), it will be added to the queue (line 17) to trigger the traversal of its neighborhood and it will be removed from the global set of transaction vertices (line 18) since it may only be part of a single BTG.

5.5.2 Limitations

There are cases where transaction vertices can belong to more than one case. The problem was already discussed in the context of artifact choreographies [51]. For example, if a company's business includes complex logistics, it will be economical to combine shipping for multiple orders although their connection is not causal

but geographical and temporal. In this case, our algorithm would extract a *pseudo-case*, i.e., lead to a single business transaction graph that includes data of multiple independent cases. The problem can be solved by another reserved property to mark *non-exclusive* transaction data. This property must be evaluated in line 16 of Algorithm 5.6 s.t. these vertices as treated just like master data. However, the problem must not be solved by setting the master data indicator to true since the indicator might be used for further analytical steps (Chapter 3) and, thus, would blur analytical results.

In one of out use case evaluations in cooperation with a real estate platform (Section 8.1) transaction vertices represented website navigation steps and master data reflected geographical information, user account data and real estate constants (e.g., housing features). The problem of pseudo-cases was not only confirmed but even intensified. In particular, whole subgraphs of transaction vertices had to be shared across multiple cases [162]. In this example, a case was defined by property announcement but multiple cases might have been created during a single web session. However, to reach the analytical goal, session related information had to be available for all cases. Generalized, there was a nesting of two business processes and the evaluation of the inner process's cases required access to the data of the outer cases. We found no general solution but a domain-specific variant of Algorithm 5.6 could solve the problem.

5.6 Experimental Evaluation

For a functional evaluation, we used two datasets. First, we used a real-world dataset whose data originates in a real installation of the Open Source ERP system ERPNext [221]. The dataset contains fictive business activities created during application development and testing. While the database size is rather small we found the data realistic and useful for an initial proof-of-concept. Since we were not able to gain two interrelated real-world business datasets, we used a second synthetic dataset created by the FoodBroker data generator (Section 4.4). This dataset emulates data from two systems and allows to evaluate data integration. The latter was also used for public demonstration of BIIIG [143].

5.6.1 Implementation

Both datasets were stored in MySQL databases and we used Neo4j [232] in version 2.0 to store property graphs. Since Neo4j was lacking support for multiple graphs in one database, we used separated database instances for the unified metadata graph (UMG), the integrated instance graph (IIG) and the collection of business transaction graphs (BTGs). Additionally, Neo4j provides no support for logical graphs. Thus, we used a workaround and added a reserved property with key

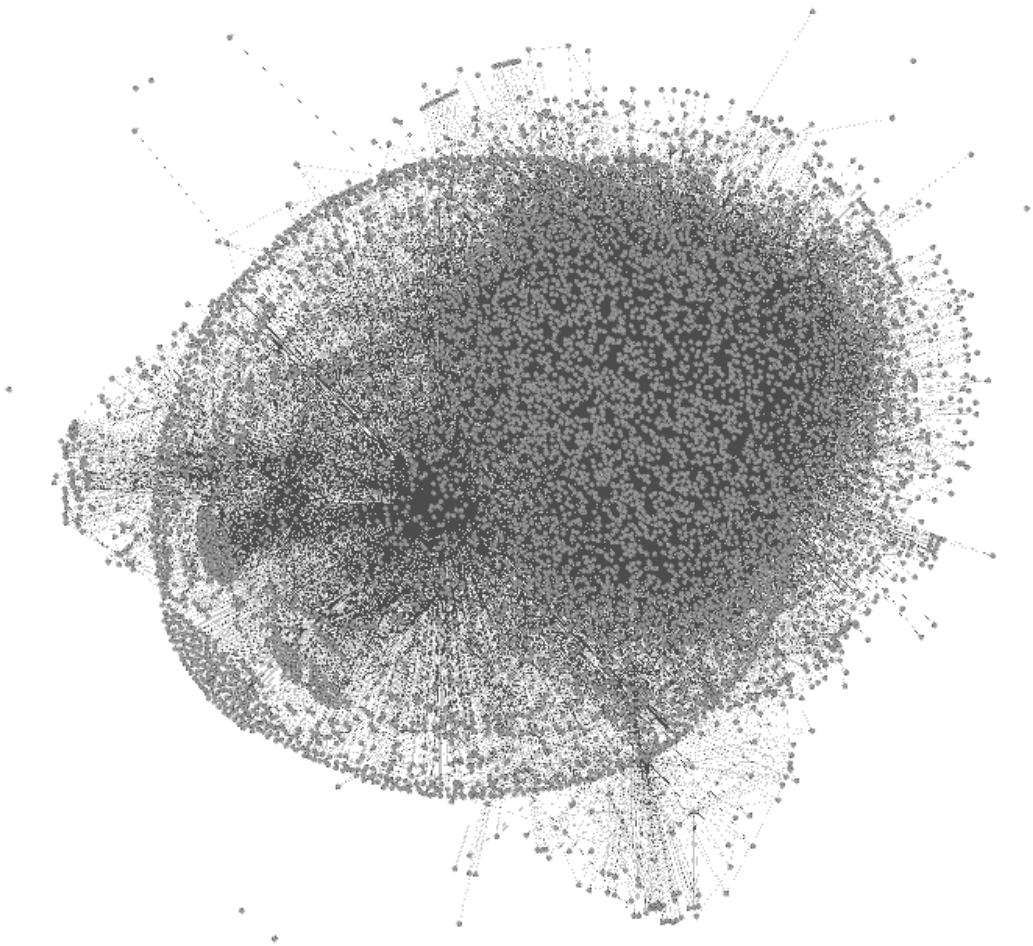


Figure 5.7: Visualization (Gephi [224]) of the integrated instance graph extracted from a dataset that was created by the real-world business information systems ERPNext.

btg_id to every vertex. Since the used version of Neo4j had no support for multi-value properties, master data was duplicated for every BTG.

All data processing tasks from metadata acquisition (Algorithm 5.1) to business transaction graph extraction (Algorithm 5.6) were implemented using Java in version 7. With regard to vertex fusion (Algorithm 5.5) the prototype contained only a simplified version with edge-wise fusion and a simple property conflict resolution instead of user defined functions. Neo4j was accessed using the native Java API which is known to provide the best performance [73]. All data processing tasks were executed remotely by a REST API. There was also a web frontend [143] based on Ruby on Rails in version 3.0 which allows UMG manipulation, for example, to rename classes and relationship types or to classify master and transaction data. Since we were using Neo4j, its stock frontend could be used to browse the graphs and to access UMG and IIG with the query language Cypher [217].

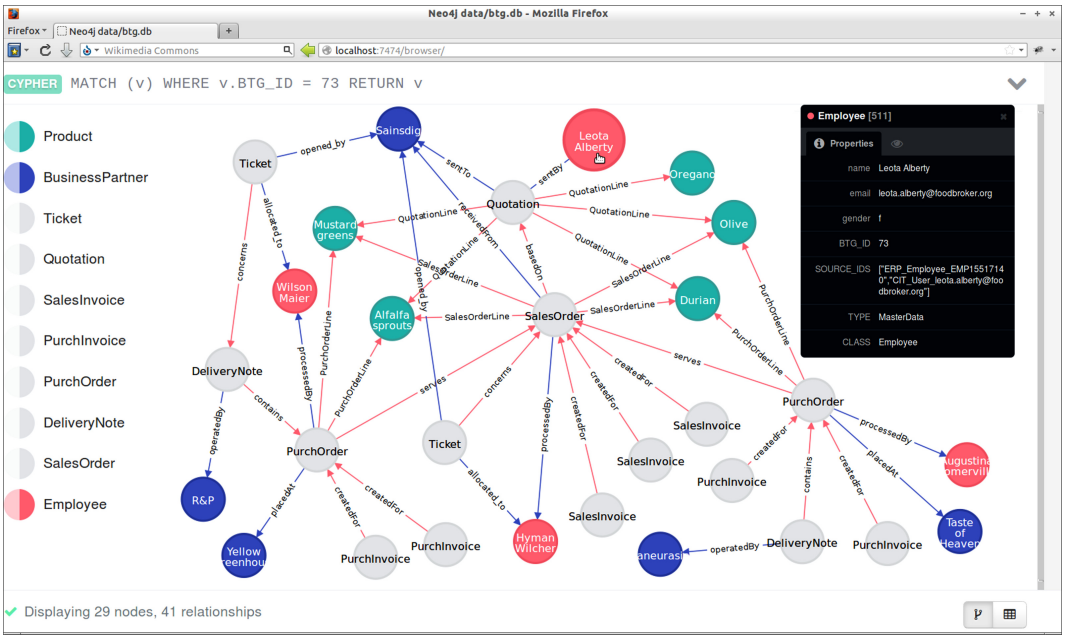


Figure 5.8: Visualization (Neo4j frontend [232]) of a business transaction graph extracted from a dataset created by the FoodBroker data generator.

5.6.2 Results

For both datasets, we were able to acquire the UMG automatically. However, we used different implementations of *getPrimaryKey* and *getForeignKeys* from Algorithms 5.1 and 5.2. For the synthetic FoodBroker dataset we could extract keys from the database schema as respective constraints were created by the data generator. We manually added cross-system relationship types including *sameAs* edges and achieved, as expected, an UMG that corresponded to the data generator's schema. For the real-world dataset of ERPNext neither key constraints nor a data dictionary were provided. Thus, we used convention exploits to gather primary and foreign keys. In particular, foreign key candidates were chosen based on naming patterns and datatypes. Candidates were automatically validated by table joins, i.e., a join with a non-empty result was considered valid. By this method we derived 102 classes and 583 relationship types.

For both datasets we could execute data transformation based on the extracted UMGs to create the respective IIGs. The IIG extracted from the real ERP data had 8,358 vertices, 38,892 edges and 87,746 nonempty properties. Figure 5.7 shows a visualization of the extracted IIG. In can be seen that data is highly interconnected. After a closer look at the graph we found that the for this are relatively few master data objects (mostly employees) that were often referenced by transaction data (mostly created by). For FoodBroker vertex and edge counts corresponded, according to the current scale factor, to the object and relationship counts of Table 4.3. Vertex fusion on FoodBroker data worked as designed.

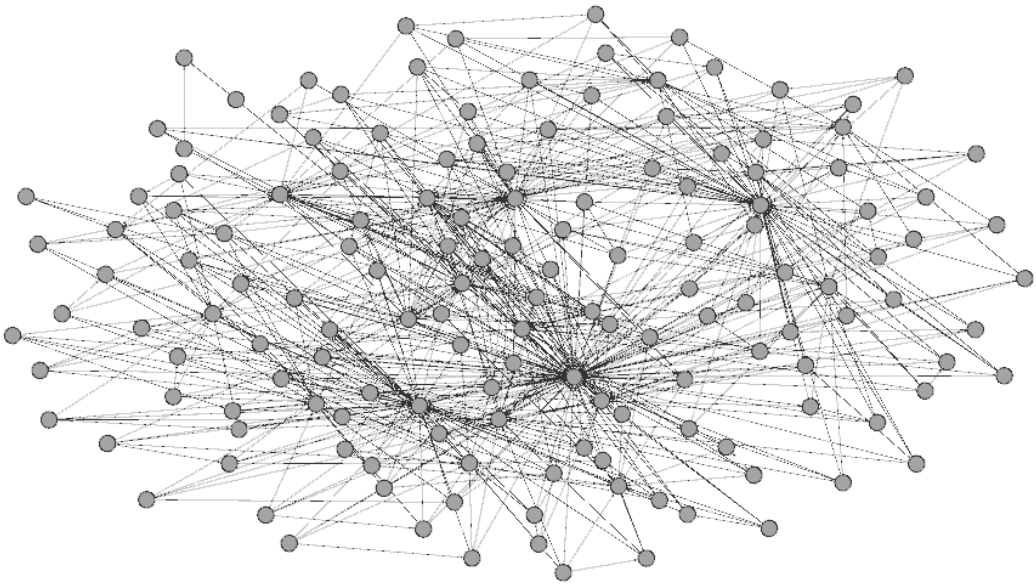


Figure 5.9: Visualization (Gephi [224]) of a business transaction graph extracted from a dataset created by a real-world business information system. Zoom factor is to low to render labels.

We also executed business transaction graph extraction for both datasets. A synthetic FoodBroker dataset contains a predictable number of business transaction graphs that is set by a scale factor (Section 4.4). We extracted exactly as many BTGs as configured for different scale factors according to Table 4.3. Figure 5.8 shows an example graph in the Neo4j frontend. From the real dataset, we isolated 1,983 BTGs. The BTG size ranges from 2 to 221 nodes and 2 to 883 edges. We expect real-world BTGs that contain ERP data to have at least the size of the biggest ones that we have extracted in our experiment. Figure 5.9 and Figure 5.10 show an example of these biggest graphs in two zoom factors. This graph was already to large for the Neo4j frontend. The clipping of Figure 5.10 is meaningful for people with background knowledge about ERP systems. Further spot tests of BTGs yielded only reasonable results in the context of business processes. In particular, we found interrelated transaction data objects from first sales activities over product purchase and invoicing up to general ledger accounting as well as involved master data such as employees, customers and products.

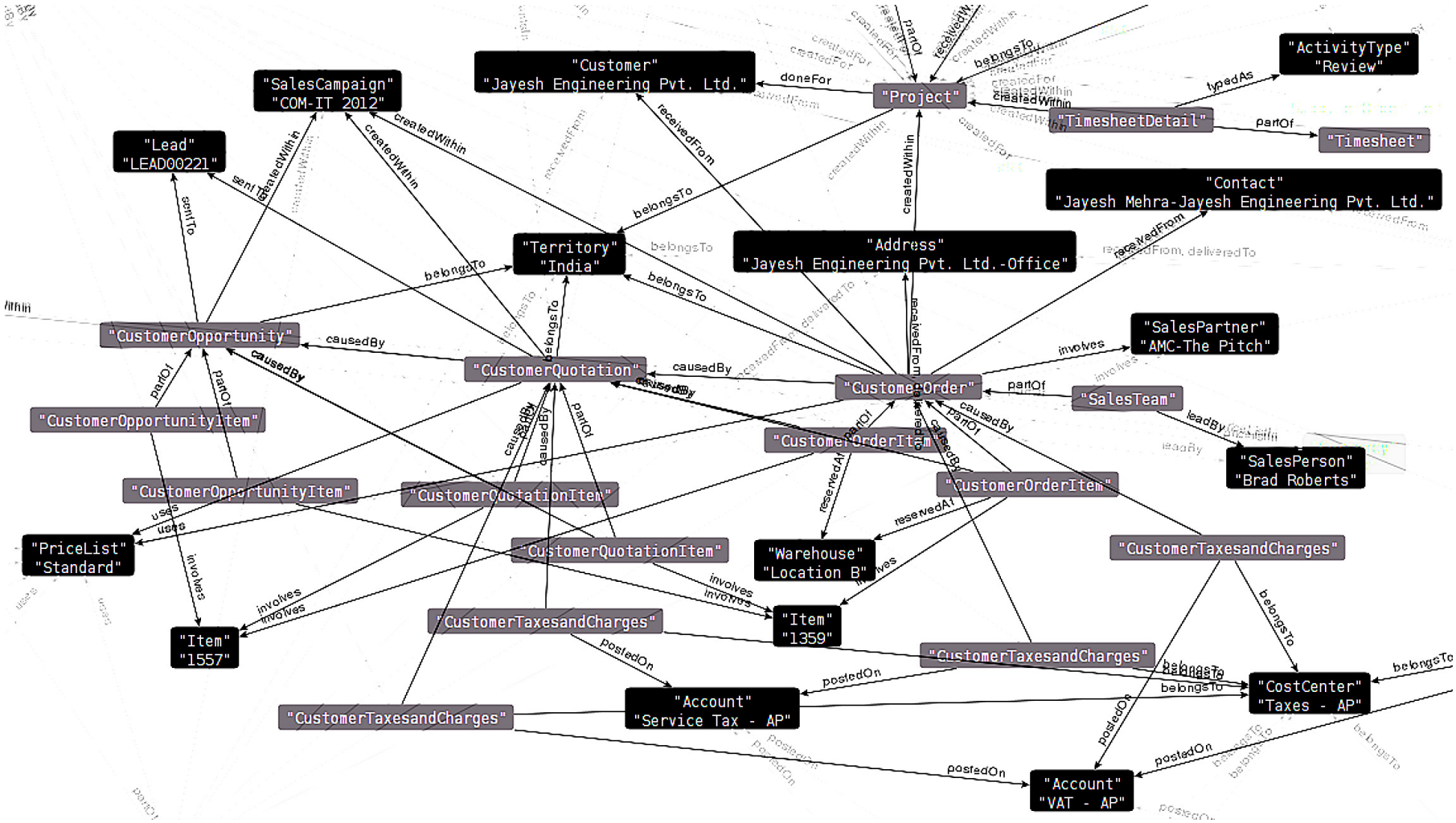


Figure 5.10: Clipped view on the graph of Figure 5.9 with a higher zoom factor (Gephi [224]). Black vertices show master data and gray vertices show transaction data. The displayed label is the class property of transaction data and a manually chosen business identifier for master data.

5.7 Conclusion

Graph-based data transformation and integration are crucial steps to enable graph-based analytics of business data. With BIIIG, we proposed practical solutions to these problems. BIIIG utilizes the capabilities of property graphs to uniformly represent metadata and instance data of diverse sources such as ERP and other business information systems. Metadata acquisition and instance integration are largely automatic and retain valuable source relationships for subsequent data analytics and data mining. We further provide a generic algorithm to automatically extract business transaction graphs in common applications. An initial prototype for relational data sources has been implemented based on an existing graph database and was successfully applied to a real ERP use case. The algorithm to extract BTGs[1] was already ported to Gradoop and evaluated in a practical application [162]. This application confirmed its value but has also shown its limitations.

[1] https://github.com/dbs-leipzig/gradoop/blob/master/gradoop-flink/src/main/java/org/gradoop/flink/algorithms/btgs/BusinessTransactionGraphs.java

Chapter 6

Frequent Subgraph Mining in Distributed Graph Collections

BIIIG aims to identify correlations between business measures and graph patterns. In the previous chapters we have already discussed how business data can be represented by graphs and how business measures can be calculated based thereon. To reach BIIIG's analytical target counting of graph pattern frequencies is a crucial primitive. The remainder of this chapter is dedicated to this problem that is usually referred to as *frequent subgraph mining* (FSM). After a motivation (Section 6.1), we will first study the frequent subgraph mining problem with regard to directed multigraphs (Section 6.2). Afterwards, we will propose an approach to FSM that is tailored to the programming model of state-of-the-art Big Data technology (Section 6.3) and provide a comparison to competitors based on MapReduce (Section 6.4). Finally, we will present results of an experimental evaluation (Section 6.5) as well as a conclusion (Section 6.6).

6.1 Motivation

Although FSM found much research interest in the last two decades [82], the range of applications was limited to chemical (e.g., chemical compounds) and biological structures (e.g., protein interaction networks). Data of these scenarios shows different characteristics compared to business data represented by business transaction graphs (Section 3.2). In consequence, existing FSM algorithms are not applicable to BIIIG for two reasons:

First, business data typically describes directed multigraphs, i.e., the direction of an edge has a semantic meaning and there may exist multiple edges between the same pair of vertices. To solve the first problem, we extended gSpan [196], a leading sequential algorithm, and developed *DMGSpan (Directed MultiGraph gSpan)* which is to the best of our knowledge the first FSM algorithm with support for directed multigraphs (Section 6.2).

Second, the data volume of business applications can be very large and sequential algorithms will not lead to acceptable execution times. An established approach to speed up complex computations on very large data volumes is parallelization, i.e., the computation is split up into smaller pieces which can be processed concurrently by multiple processors. The problem of parallel frequent subgraph mining has already been studied for multiple computing threads [120, 183], FPGAs [171] and GPUs [93]. We will show, that also DMGSpan can be parallelized with ease, as far as shared main memory is available.

However, in Big Data scenarios even parallel single machine solutions will reach their limits, for example, if either input data volume or the size of intermediate results exceed main memory. A common approach to solve this problem is spreading computation across a cluster of multiple physical machines without shared memory. The rise of this approach was strongly connected with the MapReduce [42] programming paradigm, which has also been applied to the FSM problem [10, 18, 72, 109, 114]. Besides the fact that none of these approaches provides support for directed multigraphs either, MapReduce is not well suited for complex iterative problems like FSM as it leads to a massive overhead of disk access. Further on, existing approaches still leave room for algorithmic improvements such as minimizing the data shuffled over the network and the total number of expensive isomorphism resolutions (Section 6.4).

In recent years, a new generation of advanced cluster computing systems appeared. With Apache Flink [30] and Apache Spark [202] as prominent representatives, these *distributed in-memory dataflow systems* provide a larger set of operators than MapReduce and support holding data in main memory between operators as well as during iterative calculations. We studied how FSM could be efficiently parallelized by the use of these systems and developed *DIMSpan (Distributed In Memory gSpan)*, an advanced approach to distributed FSM (Section 6.3).

DIMSpan inherits the algorithmic foundation of DMGSpan and, thus, supports directed multigraphs. In the presence of shared memory, the parallelization of FSM algorithms is result-centric, i.e., input data is globally accessible and every thread is processing a part of the result. This is not possible in the absence of shared memory. DIMSpan turns parallelization upside down and follows a data-centric approach based on the dataflow programming model, i.e., the input is distributed and the result is globally available. Additionally, we use algorithmic and technical optimizations that were not applied by former solutions based on MapReduce.

We performed experimental evaluations with real and synthetic datasets based on a prototype that was implemented within the Gradoop framework. Our results show that DIMSpan is highly scalable in terms of an increasing data volume and an increasing number of machines. Further on, we could show the positive runtime impact of our optimization techniques (Section 6.5).

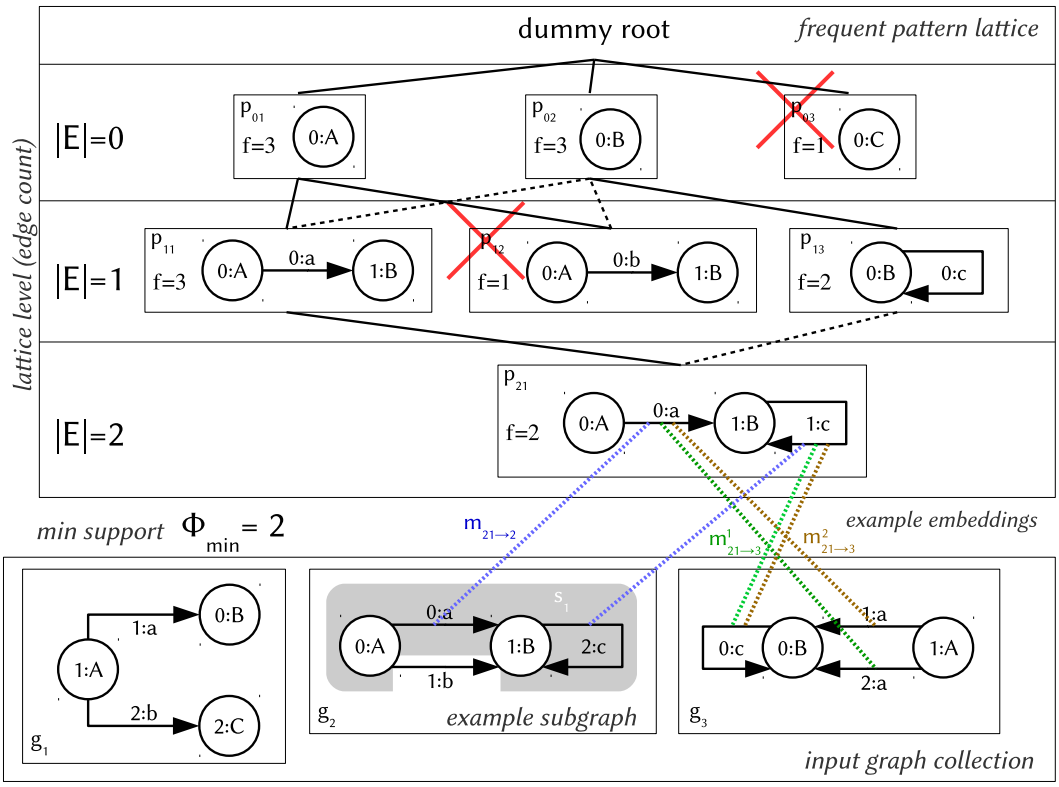


Figure 6.1: Example illustrations for a collection of labeled graphs, a subgraph (gray background), a frequent pattern lattice and embeddings. The values attached to vertices and edges represent id:label pairs

6.2 Frequent Subgraph Mining in Collections of Directed Multigraphs

The current section will study frequent subgraph mining algorithms and present our approach to support directed multigraphs.

6.2.1 Data Model

Frequent subgraph mining is a general problem and neither specific to BIIIG nor to the Extended Property Graph Model. FSM in graph collections is usually referred to as *graph transaction setting* (Section 2.8.2). The vast majority of existing FSM algorithms of this setting are based on collections $G = \{g_1, g_2, \ldots, g_n\}$ of n labeled and undirected graphs without parallel edges [82]. Our contribution is solving an extended variant of this problem whose graphs are directed and may have parallel edges (multigraphs). In this chapter we will use a minimal data model that exactly adds these features to the model that is commonly used in this field of research. In particular, we require all graphs to meet Definition 2.4, i.e., to be *labeled directed multigraphs*. However, our algorithms will be compatible to EPGM datasets with the only requirement that every vertex and edge has a reserved property with key Λ that reflects its label (Section 3.5.2). Formally, the set of labels is a subset of all property values, i.e., $L \subseteq D$ and $\forall x \in (V \cup E).\pi(x, \Lambda) = \lambda(x)$.

6.2.2 Problem Definition

Frequent subgraph mining (FSM) is a variant of frequent pattern mining [1] where patterns are graphs. There are two variants of the FSM problem. *Single graph FSM* identifies patterns that occur at least a given number of times within a single graph, while *graph transaction FSM* searches for patterns that occur in a minimum number of graphs in a collection. Our studied approach belongs to the second setting. Since there are many variations of the FSM problem we will define the problem and related terminology precisely before we introduce our actual algorithm.

Definition 6.1 (Subgraph) Let s, g be labeled directed multigraphs graphs according to Definition 2.4 then s will be a *subgraph* of g, in the following denoted by $s \sqsubseteq g$, if s has subsets of vertices $V_s \subseteq V_g$ and edges $E_s \subseteq E_g$ with matching sources and targets s.t. $\forall e \in E_s.\Big(\varsigma_s(e) = \varsigma_g(e) \wedge \tau_s(e) = \tau_g(e)\Big)$.

On the bottom of Figure 6.1, a collection of directed multigraphs $G = \{g_1, g_2, g_3\}$ and an example subgraph $s_1 \sqsubseteq g_2$ are illustrated. Identifiers and labels of vertices and edged are encoded in the format `id:label`, for example, vertex `1:A` represents v_1 with label $\lambda(v_1) = \mathsf{A}$.

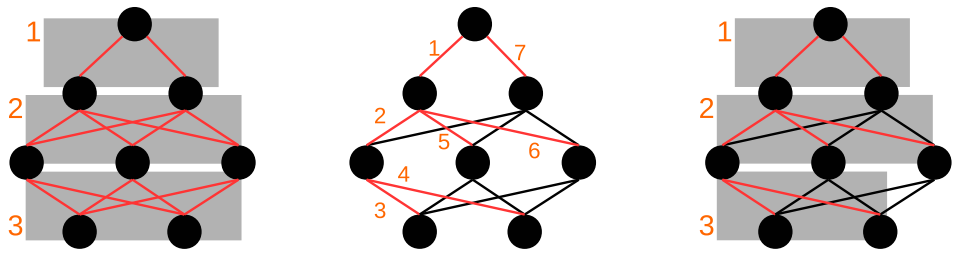
Definition 6.2 (Path) Let g be a graph then a *path* of length n in g is a n-tuple $\overline{v} = \langle v_i \mid v_i \in V\rangle_{1 \leq i \leq n}$ of vertices where at least one edge connects each pair of subsequent vertices s.t.
$\forall v_i, v_{i+1}.\exists e \in E.\Big(\varsigma(e) = v_i \wedge \tau(e) = v_{i+1} \vee \tau(e) = v_i \wedge \varsigma(e) = v_{i+1}\Big)$.

Definition 6.3 (Pattern) A *pattern* p is a connected graph. A graph with more than one vertex $|V| > 1$ will be considered *connected*, if a path exists between any pair of vertices s.t. $\forall u_1, u_2 \in V.\exists \overline{v} = \langle v_1, \ldots, v_n\rangle.(u_1 = v_1 \wedge u_2 = v_n)$.

Definition 6.4 (Isomorphism) Two graphs s, p (e.g., a subgraph and a pattern) will be considered *isomorphic*, in the following denoted by $s \simeq p$,
if two bijective mappings exist for vertices $\iota_v : V_s \leftrightarrow V_p$ and edges $\iota_e : E_s \leftrightarrow E_p$ with matching labels, sources and targets s.t. $\forall v \in V_s.\lambda_s(v) = \lambda_p(\iota_v(v))$ and
$\forall e \in E_s.\Big(\lambda_s(e) = \lambda_p(\iota_e(e)) \wedge \iota_v(\varsigma_s(e)) = \varsigma_p(\iota_e(e)) \wedge \iota_v(\tau_s(e)) = \tau_p(\iota_e(e))\Big)$.

Definition 6.5 (Embedding) Let g be a graph and p be a pattern, then an *embedding* is defined as a pair $m = \langle \iota_v, \iota_e\rangle$ of isomorphism mappings which describe a subgraph $s \sqsubseteq g$ isomorphic to p. As a graph may contain multiple subgraphs isomorphic to the same pattern (e.g., subgraph automorphisms), we use $\mu : G \times P \rightarrow \mathcal{P}(M)$ to denote an *embedding map* which associates all existing elements of embedding space M to a pair of input graph $g \in G$ and pattern $p \in P$.

$\mu(g, p) = \emptyset$ means that graph g does not contain pattern p. Figure 6.1 shows three differently colored edge mappings of example embeddings $m_{21\rightarrow 2}, m^1_{21\rightarrow 3}, m^2_{21\rightarrow 3}$.



(a) BFS **(b)** DFS **(c)** LDFS

Figure 6.2: Pattern lattice search strategies. Bullets represent patterns and lines represent parent-child relationships where parents are shown above children. Red lines are those actually processed by the particular search strategy.

Definition 6.6 (Support) Let $\mathcal{G} = \{G_1, \ldots, G_n\}$ be the space of all graph collections then the *absolute support* of a pattern p in a collection G denoted by $\phi : \mathcal{G} \times P \rightarrow \mathbb{N}$ is the number of graphs that contain at least one subgraph that is isomorphic to the pattern s.t.
$\forall G \in \mathcal{G}, p \in P. \exists G_p \subseteq G. \Big(\phi(p) = |G_p| \wedge (g \in G_p \Leftrightarrow \exists s \sqsubseteq g. s \simeq p)\Big)$.
We will use $\phi^{rel}(G, p) = {}^{\phi(G,p)}/_{|G|}$ to denote a pattern's *relative support*.

Definition 6.7 (Frequent Subgraph Mining) Let $G \in \mathcal{G}$ be a graph collection then *frequent subgraph mining* is an operator $\Phi(G, \phi_{min}) \mapsto F$ that extracts the complete set of frequent patterns $F \subseteq P$ from all contained patterns $P \in \mathcal{G}$ whose support is greater or equal than a given *minimum support threshold* ϕ_{min} s.t.
$\forall G \in \mathcal{G}, p \in P. (p \in \Phi(G, \phi_{min}) \Leftrightarrow \phi(G, p) \geq \phi_{min})$.

6.2.3 Basics of Frequent Subgraph Mining Algorithms

All patterns extracted by frequent subgraph mining (FSM) form a lattice of parent child relationships. The upper part of Figure 6.1 shows an example lattice, in particular the result of frequent subgraph mining with $\phi_{min} = 2$ for the example graph collection on its bottom. The results are the non-crossed patterns.

Definition 6.8 (Parent-Child Relationship of Graphs) A non-empty graph g_p will be considered a *parent* of a *child* graph g_c, in the following denoted by $g_c \overset{\bullet}{\sqsubseteq} g_p$, if it is a connected subgraph with one edge less. However, for graphs without edges the parent is an empty graph s.t.
$g_c \overset{\bullet}{\sqsubseteq} g_p \Leftrightarrow (|E_c| \geq 1 \wedge g_p \sqsubseteq g_c \wedge |E_c| - |E_p| = 1) \vee (|E_c| = 0 \wedge |V_p| = 0)$.

Because of the anti-monotonic property of frequent pattern mining (Section 2.8.1) children can only be frequent, if their parents are as well. Thus, all frequent subgraph mining algorithms start from an empty graph, the so-called *dummy root*, and grow only children of parents that are known to be frequent. Based on the

Algorithm 6.1 A priori (BFS) approach to frequent subgraph mining

Input: graph collection (input), minimum support threshold (minSupport)

Output: frequent pattern support mapping (output)

```
1:  output : Patterns → ℕ
2:  parents : Patterns → 𝒫(input)
3:  children : Patterns → 𝒫(input)
4:  parents = getFrequentSingleEdgePatterns(input)
5:  while parents ≠ ∅ do
6:     children.clear()
7:     children = generateCandidates(parents)
8:     parents.clear()
9:     for all ⟨child, graphs⟩ in children do
10:       for all graph : graphs do
11:          if not subgraphIsomorphism(graph, child) then
12:             graphs.remove(graph);
13:          end if
14:       end for
15:       support = graphs.size()
16:       if support ≥ minSupport then
17:          parents[child] = graphs
18:          output[child] = support
19:       end if
20:    end for
21: end while
22: return output
```

applied search strategy in the lattice, FSM algorithms can be classified into *a priori* and *pattern growth* algorithms (Section 2.8.2). Particular algorithms of both classes are further distinguished by the way they generate children and how they determine their frequency.

6.2.4 Breadth First Search

Breadth first search (BFS) in the lattice means that first all frequent patterns of one *level*, i.e., all patterns with the same edge count k will be determined before any child with $k + 1$ edges will be processed. Figure 6.2a illustrates this approach. Every bullet represents a pattern and every connection represents a parent-child relationship. Parents are shown above children. This search strategy is typically applied by a priori algorithms since these visit children by a *candidate generation* step which requires all frequent parents to be known in advance. Thus, BFS and a priori are often used synonym. The actual support is then counted by subgraph

isomorphism testing which is also knows as *graph pattern matching* [58]. The efficiency of an a priori algorithm is determined by its particular candidate generation technique and by the maximum number of executed subgraph isomorphism tests.

Algorithm 6.1 shows the pseudocode of an a priori approach. The input is a graph collection and the output are frequent patterns together with their support (line 1). The algorithm is iterative and there are two mappings that assign supporting graphs to current parents (line 2) and current children (line 3). We omitted details about the determination of frequent 0-edge and 1-edge parents since these steps are trivial. Let us just assume, that all 1-edge parents are known after method *getFrequentSingleEdgePatterns* has been called (line 4). The actual algorithm is iterative and represented by a while loop (lines 5-21) that terminates as soon as no more frequent parent exists.

As the first step of each iteration, the children-graph mapping of the previous iteration is cleared (line 6). Then, candidate generation is executed to generate children from the frequent parents determined in the last iteration (line 7). Since this step is specific to the actual algorithm we hide details behind the method *generateCandidates* which returns a mapping from child patterns to all graphs that contain all of their parents. Afterwards, we clear last round's parents since they are fully processed (line 8). Now, the actual support of every child is determined (lines 9-20). Therefore, subgraph isomorphism testing is performed for any candidate supporter (lines 10-14). In particular, all of these without a matching subgraph (line 11) will be removed from the child mapping (line 12). Now, as it corresponds to the size of the verified supporters the support is determined (line 15) and can be compared to the minimum support threshold (line 16). Finally, all frequent patterns will be added to next round's parents (line 17) and to the output (line 18).

6.2.5 Depth First Search

Depth first search (DFS) in the lattice means that children will be generated by edgewise extension (*pattern growth*). The search is called depth first because children are generated without knowledge about their parents' siblings. Since a child may have multiple independent parents the same child can be reached from multiple parents. To detect the resulting duplicates, these algorithms use canonical forms to represent graph patterns, often referred to as *canonical labels*, and count the distinct graphs for each label. Further on, they use pattern growth constraints to even avoid duplicate detection.

Since there is no synchronization among parents, pattern growth algorithms reduce the search to a tree as illustrated by Figure 6.2b. Red lines draw the actually processed search tree while black lines are those that can be skipped in comparison to Figure 6.2a. It has been shown that DFS algorithms are more efficient than BFS ones (Section 2.8.2) and the efficiency gain is originated in those

Algorithm 6.2 Pattern growth (DFS) approach to frequent subgraph mining

Input: graph collection (input), minimum support threshold (minSupport)

Output: frequent pattern support mapping (output)

```
1:  output : Patterns → ℕ
2:  parents : Patterns → 𝒫(input)
3:  children : Patterns → 𝒫(input)
4:  parents = getFrequentSingleEdgePatterns(input)
5:  while parents ≠ ∅ do
6:     children.clear()
7:     ⟨parent, graphs⟩ = parents.getAndRemove()
8:     for all graph in graphs do
9:        for all child in growChildren(parent) do
10:          children[child].add(graph)
11:       end for
12:    end for
13:    for all ⟨child, graphs⟩ in children do
14:       support = graphs.size()
15:       if support ≥ minSupport then
16:          parents[child] = graphs
17:          output[child] = support
18:       end if
19:    end for
20: end while
21: return output
```

skipped parent-child relationships. The efficiency of an particular pattern growth algorithm depends on the canonical labeling and the effectiveness of the growth constraints.

Algorithm 6.2 shows the pseudocode of a pattern growth algorithm. Input and output as well as the required data structures are the sames as those of a BFS algorithm (Algorithm 6.1). Even the outer while loop shows the same termination criterion (line 5). However, while in a BFS one iteration is processing a whole level of the lattice a DFS algorithm is only processing a single parent (line 7). The mapping from child patterns to graphs can be cleared for every round (line 6) since our search is a tree. Then, all children are grown from each parent supporter (line 8). Here, particular algorithms differ. These differences are hidden behind method *growChildren* (line 9). Afterwards, children are added to the current iteration's child mapping (line 10). After all children are grown, their support can be simply determined by counting their supporters (lines 13,14). Finally, all frequent patterns will be added to the parent queue (line 16) as well as to the output (line 17).

6.2.6 Frequent Subgraph Mining for Directed Multigraphs

Before BIIIG, there was no FSM algorithm with support for directed multigraphs (Section 2.8.2). To change this we chose a straight forward approach and reused concepts of existing algorithms. Comparative work [190, 130] has shown that runtime can be decreased by fast label generation and by holding embeddings (Definition 6.5) in main memory. Thus, we extended the fast append-only canonical labeling of gSpan (*DFS codes*) [196] to support directed graphs. However, the original gSpan algorithm stores no embeddings between iterations but only occurrence lists as shown by Algorithms 6.2. Thus, actual embeddings must be restored by subgraph isomorphism testing before they can be extended to generate children. To improve this, we developed a specific embedding format between DFS codes and input graphs to supports multigraphs.

Definition 6.9 (DFS Code) Let g be a labeled directed multigraph according to Definition 2.4 with $k \geq 1$ edges then its *DFS Code* representation is a k-tuple $X = \langle x_1, \ldots, x_k \rangle$ of extensions. For each DFS code there are two integer sets $U = \{0, \ldots, |V_s| - 1\}$ and $K = \{1, \ldots, |E_s|\}$ to represent initial *discovery times* of vertices and edges, i.e., their traversal orders. Both sets of discovery times have no gaps s.t. $\forall u \in U.\Big(u = 0 \vee (u-1) \in U\Big)$ and $\forall k \in K.\Big(k = 1 \vee (k-1) \in K\Big)$. K corresponds to extension indices, i.e., x_1 is the first and x_k is the last extension.

Definition 6.10 (DFS Embedding) Let g be a labeled directed multigraph according to Definition 2.4 and X be DFS code that describes a subgraph $s \sqsubseteq g$ then a *DFS embedding* is a pair $m = \langle \nu, \kappa \rangle$ where $\nu : U \leftrightarrow V_s$ maps vertices to discovery times and $\kappa : K \leftrightarrow E_s$ maps edges to discovery times.

Definition 6.11 (DFS Extension) Let L_v, L_e be global sets of vertex and edge labels, g be a graph, X be a DFS code and M be a set of its embeddings then an *extension* is a hextuple $x_k = \langle u_a, u_b, \ell_a, d, \ell_e, \ell_b \rangle$ which represents the traversal of an edge e with label $\ell_e \in L_e$ at time $k \in K$ from a *start* vertex discovered at time $u_a \in U$ to an *end* vertex discovered at time $u_b \in U$. The fields $\ell_a, \ell_b \in L_v$ hold the respective labels of both vertices. The value of $d \in \{\mathsf{in}, \mathsf{out}\}$ will indicate, if the edge is traversed in or against its direction s.t.
$\forall k \in K, m \in M.(d = \mathsf{out} \Leftrightarrow \varsigma(\kappa(k)) = \nu(u_a))$.

Pattern p_{21} of Figure 6.1 could be represented by the following DFS code:

$$X_{21} = \langle \langle 0, 1, \mathsf{A}, \mathsf{out}, \mathsf{a}, \mathsf{B} \rangle_1, \langle 1, 1, \mathsf{B}, \mathsf{out}, \mathsf{c}, \mathsf{B} \rangle_2 \rangle$$

Let us summarize our extensions: In contrast to DFS extensions for undirected graphs, ours include a direction indicator d. Further on, DFS embeddings for simple graphs consist simply of ν since κ could be determined transitively. In particular, every edge of a simple graph can be identified by its incident vertices v_a, v_b. Thus, a pair $\nu(u_a), \nu(u_b)$ is sufficient to map extensions to edges. However, we want to deal with multigraphs and, thus, we require κ.

DFS codes are generated in a tree search by edge-wise extension. Thus, every child has exactly one parent. This parent child relationship can be defined as follows:

Definition 6.12 (Parent-Child Relationship of DFS Codes) A DFS code $X^p = \langle x_1^p, \ldots, x_k^p \rangle$ is a parent of a DFS code $X^c \langle x_1^c, \ldots, x_{t-1}^c \rangle$ if $\forall k \in K^c : x_i^p = x_i^c$.

However, due to tree search multiple DFS codes that represent the same graph pattern may be created. To use DFS codes as a canonical form, gSpan is using a lexicographic order to determine a minimum one among all possible DFS codes [195]. This order is a combination of two linear orders. The first is defined on start and end vertex times of extensions $U \times U$, for example, a backwards growth to an already discovered vertex is smaller than a forwards growth to a new one.

The second order is defined on the labels of start vertex, edge and end vertex $L_v \times L_e \times L_v$, i.e., if a comparison cannot be made based on vertex discovery times, labels and their natural order (e.g., alphabetical) will be compared from left to right. To support directed graphs, we extended this order by direction $D = \{\mathsf{out}, \mathsf{in}\}$ with $out < in$ resulting into an order over $L_v \times D \times L_e \times L_v$, i.e., in the case of two traversals with same start vertex labels, a traversal of an outgoing edge will be considered smaller.

Definition 6.13 (Minimum DFS Code) There is an order among DFS codes s.t. $\forall X_1, X_2 : X_1 < X_2 \vee X_1 = X_2 \vee X_1 > X_2$. Let $\mathcal{X}_p$ be the set of all DFS codes that describe a pattern p and X_{min} be its minimum DFS code then $\nexists\, X_i \in \mathcal{X}_p . X_i < X_{min}$.

Some of the non-minimal DFS codes will never be generated due to pattern growth constraints [195]. We have taken them over from the original gSpan algorithm without modifications. For example, parents may only be extended from the shortest path between the vertices that have been discovered first and last (*rightmost path*). By these constraints gSpan directly mines a tree of minimum DFS codes as indicated by the solid lines in Figure 6.1 s.t. the dotted lines can be skipped. However, the constraints cannot totally prevent the generation of non-minimal DFS codes. Thus, there will be an additional *verification* step to check if a DFS code is minimal. This step is the only one that is directly facing the subgraph isomorphism problem. In the worst case, this step requires the complete recalculation of the minimum DFS code. With regard to Algorithm 6.2, this step can be executed directly after pattern growth (line 9) or after support counting (line 14). We will address this problem again in Section 6.3.2 since this decision is more critical in distributed context.

6.3 Frequent Subgraph Mining with Distributed Dataflow Systems

The current section will present DIMSpan, a frequent subgraph mining algorithm for distributed graph collections in the absence of shared memory.

6.3.1 Parallel Frequent Subgraph Mining

In the presence of shared memory, the parallelization of pattern growth algorithms such as gSpan is fairly simple [120]. For example, on a multi-processor system there are multiple computing slots (*threads*) which can perform calculations in parallel. Since search is reduced to a tree, the while loop of Algorithm 6.2 can be processed in parallel without any synchronizations. To do so, variables *output* and *parents* must be thread safe to support concurrent modification and variable *children* must be thread exclusive. The central data structure would then be *parents* which acts as a queue. In particular, in a naive implementation every idle thread simply polls a parent search tree node from the queue and adds all frequent child nodes until the queue is empty. Since *input* is globally available and read-only this approach will prevent any conflict among different threads.

However, in the absence of shared memory the situation dramatically changes. So-called Big Data technologies such as MapReduce [42], Apache Flink [30] and Apache Spark [203] have in common that they follow a *bring computation to the data* approach, i.e., data is partitioned among a cluster of machines and algorithms must be composed by local computations (e.g., map) and synchronization operations (e.g., reduce). Typically, these machines are connected via a local area network but share no resources, i.e., have separate CPUs, main memory banks and hard disk drives. The term *shared nothing cluster* is often used to denote this setting. In consequence, different threads have no access to common data structures.

In the context of FSM bringing computation to the data means that the input graph collection is partitioned across the cluster. Thus, in contrast to the shared memory parallelization, a single thread is not sufficient to process a search tree node as is has only access to a subset of input graphs. The most naive solution would be processing nodes sequentially in parallel, i.e., lines 6 to 12 of Algorithm 6.2 are executed on each partition and child supports are exchanged and aggregated after line 12. However, for a low minimum support threshold (which can lead to a huge number of patterns) and a large number of machines in the cluster data skew can lead to many idle threads, for example, if a single partition holds all supporters of a pattern. Further on, every synchronization barrier is causing network traffic to exchange data and additional waiting times. So, this approach is per se inefficient as it is most prone to load imbalances and maximizes the number of synchronization barriers.

Algorithm 6.3 Level-wise pattern growth (LDFS) approach to FSM

Input: graph collection (input), minimum support threshold (minSupport)

Output: frequent pattern support mapping (output)

```
1:  output : Patterns → ℕ
2:  parents : Patterns → 𝒫(input)
3:  children : Patterns → 𝒫(input)
4:  parents = getFrequentSingleEdgePatterns(input)
5:  while parents ≠ ∅ do
6:     children.clear()
7:     for all ⟨parent, graphs⟩ in parents do
8:        for all graph in graphs do
9:           for all child in growChildren(parent) do
10:             children[child].add(graph)
11:          end for
12:       end for
13:    end for
14:    parents.clear()
15:    for all ⟨child, graphs⟩ in children do
16:       support = graphs.size()
17:       if support ≥ minSupport then
18:          parents[child] = graphs
19:          output[child] = support
20:       end if
21:    end for
22: end while
23: return output
```

To solve these problems we use an approach that we call *Level-wise Depth First Search (LDFS)*. In this approach, we process all parent nodes of the same size (same *level*) in one iteration, i.e., per iteration every thread is processing all current nodes but only for its partition. Under the assumption that the support of particular patterns can be skewed but every partition contains at least a similar number of embeddings of any frequent pattern, LDFS is minimizing the likeliness of idle times. Further on, it limits the number of synchronization barriers to a minimum. In particular, we require exactly one iteration per extension. A fewer number of synchronization barriers could only be achieved by giving up level-wise support pruning, i.e., to grow even children from parents whose support is unknown. However, this would eliminate the most effective pruning technique based on the anti-monotonic property that is applied by all efficient frequent subgraph mining algorithms.

Algorithm 6.4 DIMSpan dataflow.

Input: graph collection (input), minimum support threshold (minSupport)

Output: frequent pattern support mapping (output)

```
1:  output : Patterns → ℕ
2:  parents : Patterns → ℕ
3:  graphs : input → 𝒫(Patterns) → 𝒫(Embeddings)
4:  graphs = initSingleEdgePatterns(input)
5:  repeat
6:      reports = graphs.flatMap(report supported patterns)
7:      parents = reports.combine(count partition support)
8:      parents = parents.reduce(sum global support)
9:      parents = parents.filter(support ≥ minSupport)
10:     parents = parents.filter(false positive verification)
11:     output = output.union(parents)
12:     broadcast(parents)
13:     graphs = graphs.map(grow children)
14:     graphs = graphs.filter(embedding map not empty)
15: until parents = ∅
16: return output
```

Figure 6.2c illustrates LDFS. It shows that although we apply level-wise support pruning the approach is different to BFS since we logically perform many independent depth first searches in parallel and still benefit from the efficiency of the DFS tree search. For example, in Figure 6.1 we apply the support pruning of p_{10}, p_{11}, p_{12} in parallel within the same iteration but use search constraints (Section 6.3.3) to grow only from p_{10} to p_{20}. The general idea of LDFS is shown by Algorithm 6.3. To focus on the search itself, the pseudocode shows a sequential version of LDFS. More details about its parallelization on shared nothing clusters will follow in Section 6.3.2.

Lines 1 to 6 are equivalent to those of a BFS algorithm (Algorithm 6.1) or a DFS algorithm (Algorithm 6.2). However, in comparison to BFS there is no candidate generation step and in comparison to DFS all parents are processed within a single iteration (lines 7). The actual pattern growth and support counting steps are taken over from DFS (lines 8-12 of Algorithms 6.2 and 6.3 are equal). In contrast to DFS, at LDFS the mapping *parents* does not act as a queue and is reset every iteration (line 14). The actual process of feeding parents back and putting them out based on childrens' support is the same as DFS again (lines 15-21). However, while *children* in a DFS contains only children of a single parent in a LDFS it will contain all children of the same level.

6.3.2 Distributed Dataflow

Common Big Data systems such as MapReduce, Apache Flink and Apache Spark apply the so-called *distributed dataflow* programming model, i.e., programs are represented by datasets and transformations of those (Section 2.3). Since frequent subgraph mining is an iterative problem, MapReduce is not suitable as it requires writing data back to disk after each iteration. By contrast, the more recent generation of systems such as Flink and Spark can hold data in main memory between iterations. Additionally, they offer a wider range of operations. For these reasons, we choose to develop an algorithm tailored to this class of systems. Further on, it shall be based on DMGSpan (Section 6.2) to support directed multigraphs. Thus, we called the approach *DIMSpan (Distributed In Memory gSpan).* Besides the use of an in memory system to minimize disk access it also applies LDFS to minimize synchronization barriers and idle time. We also applied further optimization techniques to minimize the amount of data exchanged over the network and the total number of isomorphism resolutions. These will be discussed in the remainder of this chapter.

To provide an overview, Algorithm 6.4 shows the dataflow of DIMSpan. Inputs are a dataset of graphs and the minimum support threshold. The output is a dataset of frequent patterns and their support. Just to avoid confusion: In line 1 frequent patterns and their support are represented by a mapping according to the actual meaning but the actual implementation is a dataset of $\langle \text{pattern}, \text{support} \rangle$ pairs. Graphs are partitioned over multiple machines (*workers*) and so are their embeddings. Thus, in comparison to Algorithm 6.3 the variable *parents* maps patterns only to their support but not to embeddings or occurrence lists (line 2). This information is kept attached to graphs instead, i.e., for every graph, we store a mapping from patterns to embeddings (line 3). These maps are initially set with 1-edge patterns and their embeddings (line 4). Then, we start a loop (line 5) that terminates as soon as no more frequent parents exist (line 15). Each iteration body represents one level of the LDFS.

In the beginning of each iteration every graph reports all k-edge ($k \geq 1$) supported patterns (DFS codes), i.e., the keys of the last iteration's embedding map, in a *flatmap* transformation (line 6). Then, we use a *combine* operation to count the pattern support per partition (line 7) and aggregate its sum in a *reduce* transformation (line 8). This is the first synchronization barrier and the first time where data is shuffled among workers, i.e., support counts are repartitioned according to their patterns s.t. all counts of the same pattern will be processed by the same thread. Using a previous combine operation is reducing the exchanged data volume enormously because not every single occurrence but only partition supports are sent along the network.

Then, we *filter* frequent patterns (line 9) and verify them to be no false positives (line 10). The latter is done by recalculating the minimum DFS code of the current DFS code and comparing the both. Since there will be a mismatch for false positives (non-minimal ones), only the matching ones will pass this step. Since this is the only part of the algorithm resolving the isomorphism problem, reducing its cardinality may reduce total runtime [195]. Thus, we placed the verification step after support pruning. Hence, false positives are counted and shuffled but the total number of isomorphism resolutions is minimized. Our experimental evaluation will confirm the effectiveness of this decision (Section 6.5). After support pruning and verification, the current frequent patterns are added to the output by a binary *union* transformation (line 11).

Now, once the current level's frequent patterns are determined they must be made available to all workers. Therefore Apache Flink and Apache Spark provide a technique called broadcasting[1][2]. By their application basically a copy of all frequent patterns is sent to the main memory of all workers (line 12). They will then be used in the pattern growth process (line 13). In particular, although all embeddings of the current level are available at this stage, only those of frequent patterns will be grown for the next iteration. Finally, we apply another *filter* operation (line 14) and only graphs with non-empty embedding maps will pass. Thus, our main data structure's content may potentially shrink in each iteration, if only a subset of graphs accumulates frequent patterns.

6.3.3 Constrained Pattern Growth

Besides gSpan's canonical labels (DFS codes) we also adopted its growth constraints to skip parent-child relationships in the pattern lattice (dotted lines in Figure 6.1). We refer to [195] for theoretical background and will discuss only our adaptation to the distributed dataflow programming model. There are two constraints for growing children of a parent embedding. The first, in the following denoted by *time constraint*, dictates that forwards growth is only allowed starting from the rightmost path and backwards growth only from the rightmost vertex, where *forwards* means an extension to a vertex that is not contained in the parent and *backwards* means an extension to a contained one. The *rightmost vertex* is the parent's latest discovered vertex and the *rightmost path* is the path of forward growths from the initial start vertex to the rightmost one. The second constraint, in the following denoted by *branch constraint*, commands that the minimum DFS code of an edge needs to be greater than or equal to the parent's branch. The *branch* of a DFS code is its 1-edge (grand-) parent code, i.e., the initial extension of a DFS code.

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html #broadcast-variables

[2] http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables

Algorithm 6.5 Pattern growth map function

Input: graph (graph), parent pattern-embeddings map (parentMap), sorted frequent patterns (parents)

Output: child pattern-embeddings map (childMap)

```
1:  childmap in Patterns → P(Embeddings)
2:  minBranch = new DFSCode
3:  extensionCandidates = graph.edges // sorted
4:  for parent in parents do
5:    parentEmbeddings = parentMap[parent]
6:    if parentEmbeddings not empty then
7:      parentBranch = parent[0]
8:      if parentBranch > minBranch then
9:        minBranch = parentBranch
10:       extensionCandidates.delete(|e| -> branch(e) < minBranch )
11:     end if
12:     for parentEmb in parentEmbeddings do
13:       for edge in extensionCandidates do
14:         if not parentEmb.contains(e) and time constraint satisfied then
15:           childEmb = parentEmb.extend(edge)
16:           childMap[childEmb.pattern].add(childEmb)
17:         end if
18:       end for
19:     end for
20:   end if
21: end for
22: return childMap
```

Algorithm 6.5 shows our adaption of these constraints to the distributed dataflow programming model, i.e., details about line 13 of Algorithm 6.4. Technically, it is a map-transformation that replaces the pattern-embeddings map of pattern size k (variable *parentMap*) by a map of pattern size $k + 1$ (variable *childMap*) in line 1. Therefore, the previously broadcasted list of frequent patterns of size k and the graph itself are required as further inputs. A naive solution would be testing possible growth for the cross of supported frequent patterns' embeddings and the graph's edges. As an optimization, we use a merge strategy based on the branch constraint to reduce the number of these tests. Therefore, frequent patterns and edges are sorted according to their branch. Sorting happens only once per graph in a preprocessing step as well as once per worker and pattern on broadcast reception. At the same time, we also cache rightmost paths for each pattern.

For each execution of the map function, we keep a current minimum branch which initially represents the dummy root (line 2) and an extension candidate set which initially corresponds to the graph's edges (line 3). First, we will check for every frequent pattern (line 4) if it is supported by the current graph (lines 5,6). If a pattern is supported and its branch is greater than the current minimum (line 8), we will update the minimum branch (line 9) to shrink extension candidates (line 10) since only those may lead to minimum DFS codes. Thus, only for the cross of embeddings (line 12) and branch-validated edges (line 13) parent containment and time constraint need to be checked (line 14). In the case of a successful growth (line 15) the resulting pattern and its embedding will be added to the output (line 16).

6.3.4 Preprocessing and Dictionary Coding

Before executing the dataflow shown in Algorithm 6.4, we apply a preprocessing that includes label-frequency based pruning, string-integer dictionary coding and sorting edges according to their 1-edge minimum DFS codes. The original gSpan algorithm already used these concepts but we improved the first two and adapted the third to our level-wise DFS strategy. In the first preprocessing step, we determine frequent vertex labels and broadcast a dictionary to all workers. Afterwards, we drop all vertices with infrequent labels as well as their incident edges. Then, we determine frequent edge labels, in contrast to the original gSpan algorithm, only based on the remaining edges. Thus, we can potentially drop more edges, for example, e_1 of g_2 in Figure 6.1 would be removed. This would not be the case by just evaluating its edge label since without dropping v_2 of g_1 before (because v_2 has an infrequent label C) the frequency of edge label b would be 2, i.e., considered frequent.

After dictionaries for vertex and edge labels are made available to all workers by broadcasting, we do not only replace string labels by integers to save memory and to accelerate comparison but also sort edges according to their minimum DFS code (branch). Technically, we use sorted arrays instead of sets to store edges. We benefit from the resulting sortedness in every execution of the constrained pattern growth (Section 6.3.3) as the effort of determining branch-valid edge candidates (line 10 of Algorithm 6.5) is reduced from a set filter operation to a simple increment of the minimum edge index.

6.3.5 Data Structures and Compression

We do not only use minimum DFS codes as canonical labels but also a data structure based thereon to support all pattern operations (counting, growth and verification) without format conversions. We further store graphs as sorted lists of 1-edge DFS codes to allow a direct comparison at the lookup for the first valid

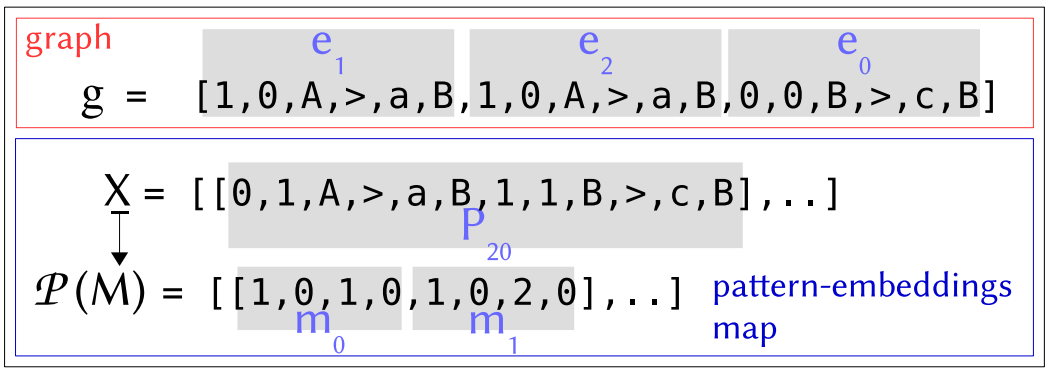


Figure 6.3: Dataset element representing graph g_3, pattern p_{21} and embedding map $\mu(g_3, p_{21})$ of Figure 6.1.

edge of a branch in the pattern growth process (line 10 of Algorithm 6.5). Figure 6.3 illustrates a single element of *graphs* in Algorithm 6.4 representing g_3 from Figure 6.1 together with its embedding map in the 2nd iteration. Graphs and patterns are stored as DFS codes according to Definition 6.9 but encoded in integer arrays where all 6 elements store a graph's edge or a pattern's extension. For the sake of readability we use alphanumerical characters in Figure 6.3. The embedding map $\mu : \mathcal{X} \rightarrow \mathcal{P}(M)$ is stored as a pair of nested integer arrays where equal indexes map embeddings to patterns. All embeddings of the same pattern p represented by a DFS Code X are encoded in a single multiplexed integer array where all $|V_p| + |E_p|$ elements store a single embedding. Here, indexes relative to their offset relate vertex ids to their initial discovery time and edge ids to extension numbers.

This data structure does not only allow fast pattern operations but also enables lightweight and effective integer compression. Therefore, we exploit the predictable value ranges of our integer arrays. First, we use dictionary coding with limited dictionary sizes. Second, vertex discovery times are bound by the maximum edge count k_{max}. Thus, the array's values may only range from $0..(max(k_{max}, |L_v|, |L_e|) - 1)$ where L_v, L_e are sets of distinct vertex and edge labels. In the context of FSM, the maximum value will typically be much less than the integer range of 2^{32}. There are compression techniques benefiting from low-valued integer arrays by the suppression of leading zeros [39]. In preliminary experiments we found that Simple16 [206] allows very fast compression and gives an average compression ratio of about 7 over all patterns found in our synthetic test dataset (see Section 6.5.2). Since also graphs and embeddings have low maximum values we apply integer compression also to them to further decrease memory usage. Embeddings and graphs are only decompressed on demand and at maximum for one graph at the same time. All equality-based operations (map access and support counting) are performed on compressed values.

	Pre.	**Map 1**	**Reduce 1**	**Map 2**	**Reduce 2**	**Post.**
I-FSM [72] (iterative)		read subgraphs	shuffle subgraphs, pattern growth, write subgraphs	read subgraphs, **add canonical labels**	shuffle subgraphs, find frequent labels, filter subgraphs by label, write subgraphs	
MR-FSE [114] (iterative)		read pattern-embeddings map, read frequent patterns, **pattern growth**, write pattern-embeddings map		read pattern-embeddings map, extract patterns	shuffle patterns, count and filter, write frequent patterns	
F&R [109] (2-phase)		read graphs, **FSM for each partition**	shuffle partition supports, filter candidates, write candidates	read graphs, read candidates, **refine partition supports**	shuffle patterns, count and filter, write frequent patterns	
DIMSpan (iterative)	read graphs	receive frequent patterns, pattern growth, update pattern-embeddings map		extract patterns from pattern-embeddings map	count partition supports, shuffle partition supports, count and filter, **verify frequent patterns**, send frequent patterns	write frequent patterns

Table 6.1: Methodical comparison of DIMSpan and approaches based on MapReduce.

6.4 Comparison to MapReduce-based Approaches

To the best of our knowledge, before DIMSpan there were only five approaches to transactional FSM based on shared nothing clusters [72, 114, 10, 109, 18] and all were based on MapReduce. Since [10, 18] show relaxed problem definitions in comparison to Definition 6.7, we compare DIMSpan only to I-FSM [72], MR-FSE [114] and the filter-refinement (F&R) approach of [109]. In experimental evaluations the authors of MR-FSE and F&R have each shown that their approaches are faster than I-FSM. Initially, we wanted to reproduce evaluation results of MR-FSE and F&R on our own cluster. Unfortunately, MR-FSE is not available to the public. Regarding F&R, only binaries[3] are accessible. However, there is no sufficient English documentation and the binaries rely on an outdated non-standard Hadoop installation. Thus, we were not able to execute the binaries without errors despite support of the author.

For this reason, we qualitatively compare the main execution costs of the MapReduce approaches to DIMSpan w.r.t volume of disk access, network data exchange (shuffling) and the total number of isomorphism resolutions. In contrast to an experimental evaluation this comparison is independent from software development skills, framework configurations, programming language and other implementation details. From our own experience we can report that in Big Data environments, which are highly complex software systems, these factors may impact total runtime at least as much as the actual concept.

6.4.1 Methodical Comparison

Table 6.1 compares the considered methods w.r.t. the steps of preprocessing, two map-reduce phases and postprocessing. All approaches except one are iterative, i.e., perform a level-wise search. For these iterative methods, the map-reduce phases of Table 6.1 represent a single iteration's body. By contrast, F&R is partition-based and requires only two map-reduce phases to extract frequent patterns of all sizes. In the following, we briefly describe the MapReduce approaches with regard to Table 6.1:

I-FSM is using full subgraphs (structure and canonical labels) as its main data structure. In map phase 1 (Map 1) k-edge subgraphs of the previous iteration are read from disk. In reduce phase 1 (Reduce 1), subgraphs are shuffled by graph id and graphs are reconstructed by a union of all subgraphs. Afterwards, $k+1$-edge subgraphs are generated and written to disk. In Map 2 they are read again and a canonical label is calculated from scratch for every subgraph. In Reduce 2, all subgraphs are shuffled again according to the added label and label supports are counted. Finally, all subgraphs showing a frequent label are written to disk.

[3] `https://sourceforge.net/projects/mrfsm/`

MR-FSE is using pattern-embedding maps as its main data structure. In Map 1 k-edge maps of the previous iteration are read from disk. Additionally, all k-edge frequent patterns are read by each worker. Then, graphs are reconstructed based on embeddings, pattern growth is applied and updated maps are written back to disk. Reduce 1 is not used. In Map 2 the grown maps are read again and a record for each pattern and supporting graph is extracted. In Reduce 2, these records are shuffled to count their support. After filtering, frequent patterns are written to disk.

F&R reads graphs from disk and runs a modified version of Gaston [129], an efficient single-machine algorithm, on each partition in Map 1. Then, a statistical model is used to report partition supports of patterns. In Reduce 1, local supports are evaluated for each pattern and a set of candidate patterns P including some support information are written to disk. In Map 2 graphs and information about candidate patterns are read from disk. For some partitions, local pattern supports may be unknown at this stage. Thus, they are refined by subgraph-isomorphism testing. In Reduce 2, refined pattern supports are summed up, filtered and written to disk.

6.4.2 Cost Comparison

Table 6.2 shows a comparison of upper bounds for the three stated dimensions. We consider our way of comparing iterative and non-iterative methods as valid since with regard to upper bounds every step can be considered as union of all k-edge results, e.g., $P = P_1 \cup .. \cup P_k$.

Disk access: I-FSM uses the most voluminous data structure of full subgraphs S. Additionally, these subgraphs are read and written twice. Thus, I-FSM clearly has the highest cost for disk access. MR-FSE uses embedding maps μ as its main data structure, which is with regard to vertex- and edge labels an irredundant version of S that describes subgraphs by patterns and embeddings (Section 6.2.2). This map is written once and read twice. Additionally, patterns P are read and written once. Thus, MR-FSE is superior to I-FSM. F&R reads graphs twice but writes no intermediate results despite rather small pattern information. Since the volume of G roughly corresponds to the one of S_1 or μ_1, F&R requires the lowest disk access of the three MapReduce approaches. However, DIMSpan further reduces disk access to a minimum as it is based on a distributed in-memory system. In particular, graphs are read only once from disk before the iterative part and patterns are written only once to disk afterwards.

	Pre	M1	R1	M2	R2	Post
`disk access`						
I-FSM		$\uparrow S$	$\downarrow S$	$\uparrow S$	$\downarrow S$	
MR-FSE		$\uparrow \mu, P \downarrow \mu$		$\uparrow \mu$	$\downarrow P$	
F&R		$\uparrow G$	$\downarrow P$	$\uparrow G, P$	$\downarrow P$	
DIMSpan	$\uparrow G$					$\downarrow P$
`network traffic`						
I-FSM			S		S	
MR-FSE		$w \cdot P$			$\lvert G\rvert \cdot P$	
F&R			$w \cdot P$		$w \cdot P$	
DIMSpan					$2w \cdot P$	
`isomorphism resolution`						
I-FSM				$\lvert S\rvert$		
MR-FSE		$\lvert S\rvert$				
F&R		$w \cdot \lvert P\rvert$		$(\lvert G\rvert - 1) \cdot \lvert P\rvert$		
DIMSpan					$\lvert P\rvert$	

w: number of worker threads (partitions, $w = |W|$)
P : set of all grown patterns
G : set of input graphs ($G \gg P$)
μ : all grown patterns and their embeddings ($\mu \gg G$)
S : all grown subgraphs (unit of pattern and embedding, $S > \mu$)

Table 6.2: Cost comparison of DIMSpan and approaches based on MapReduce.

Network traffic: Since I-FSM shuffles the complete set of subgraphs twice, it clearly causes the most network traffic. All other approaches only exchange pattern information. However, since MR-FSE is neither partition-based like F&R nor uses a combine operation like DIMSpan, a record for each pattern and graph ($|G| \cdot P$) may be shuffled among physical machines. With regard to network traffic, F&R and DIMSpan are comparable to each other, especially since both are using compression to further reduce the volume of the few exchanged records.

Isomorphism resolution: All of the four compared approaches resolve the subgraph isomorphism problem in different ways and with different cardinalities. The respective steps are highlighted by bold font in Table 6.1. I-FSM calculates a (in [72] not further specified) canonical label from scratch for each grown subgraph and, thus, the isomorphism problem is resolved with maximum cardinality $|S|$. MR-FSE is using DFS codes like DIMSpan but in [114] it is clearly stated that no verification is performed at any time. In Section 6.4.3 we will show that their proposed solution may lead to an incomplete result. However, with a small modification, their basic idea could be fixed and false positives would be detected by enumerating all DFS

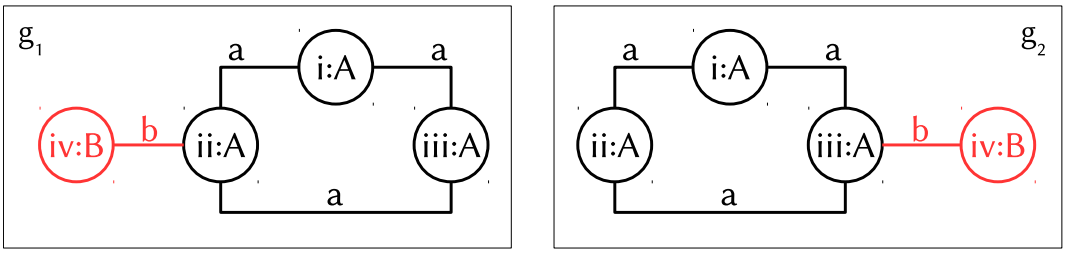


Figure 6.4: Illustration of our couterexample showing two graphs g_1, g_2. Each one contains a 3-edge subgraph with automorphisms (black lines) and an extension to a 4-edge subgraph (red lines). Roman numbers are vertex identifiers.

code permutations of each distinct edge set (subgraph) to choose the minimal code. We assume the presence of this fix and, thus, even assume isomorphism to be resolved for each subgraph, i.e., $|S|$ times. F&R is facing the isomorphism resolution problem in two steps. First, when running FSM for each partition ($w \cdot |P|$) and, second, when counting patterns by a priori like subgraph isomorphism testing in the refinement step. Since the local frequency of each pattern must be known for at least one partition, the upper bound is not fully $|G| \cdot |P|$. For this dimension, DIMSpan is clearly superior because no a priori like operations are applied at any time and every pattern is verified only once.

Summary: DIMSpan shows the lowest costs with regard to all of the stated dimensions. Besides this, DIMSpan is the only approach that provides source code to the public, supports directed multigraphs and already applies first pruning steps in a preprocessing (Section 6.3.4).

6.4.3 Disprove of Isomorphism-free Verification of MRFSE

In this subsection we will provide a disprove by counterexample to show that the so-called *isomorphism-free verification* of gSpan's minimum DFS codes proposed in [114] is not correct. According to the information provided in the paper the approach will fail for subgraphs that contain automorphisms. The author did not answer a source code request.

Proposition: Due to the condition in line 10 of Algorithm 2 of [114] and Lemma V of [114], a pair of DFS code and embedding g_e^s will only be added to the output, if a pair in hashset $genG$ covering exactly the same edge set does not already exists. In consequence, only the first g_e^s (notation of [114]) for each distinct edge set will be added to the output. Thus, the authors propose that for every graph G and every minimal k-edge DFS code X_{min} it is possible to generate the complete set of minimal $k+1$-edge children based on exactly one embedding which maps a subgraph of G to X_{min}.

X^3_{min} $\langle\langle 0,1,A,a,A\rangle, \langle 1,2,A,a,A\rangle, \langle 2,0,A,a,A\rangle\rangle$
X^4_{min} $\langle\langle 0,1,A,a,A\rangle, \langle 1,2,A,a,A\rangle, \langle 2,0,A,a,A\rangle, \langle 2,3,A,b,B\rangle\rangle$

	k-edge emb.	$k+1$-edge emb.	extension	minimal
g_1 :				
m_{11}	$\langle i, ii, iii\rangle$	$\langle i, ii, iii, iv\rangle$	$\langle 1,3,A,b,B\rangle$	no
m_{12}	$\langle i, iii, ii\rangle$	$\langle i, iii, ii, iv\rangle$	$\langle 2,3,A,b,B\rangle$	yes
m_{13}	$\langle ii, i, iii\rangle$	$\langle ii, i, iii, iv\rangle$	$\langle 0,3,A,b,B\rangle$	no
m_{14}	$\langle ii, iii, i\rangle$	$\langle ii, iii, i, iv\rangle$	$\langle 0,3,A,b,B\rangle$	no
m_{15}	$\langle iii, i, ii\rangle$	$\langle iii, i, ii, iv\rangle$	$\langle 2,3,A,b,B\rangle$	yes
m_{16}	$\langle iii, ii, i\rangle$	$\langle iii, ii, i, iv\rangle$	$\langle 1,3,A,b,B\rangle$	no
g_2 :				
m_{21}	$\langle i, ii, iii\rangle$	$\langle i, ii, iii, iv\rangle$	$\langle 2,3,A,b,B\rangle$	yes
m_{22}	$\langle i, iii, ii\rangle$	$\langle i, iii, ii, iv\rangle$	$\langle 1,3,A,b,B\rangle$	no
m_{23}	$\langle ii, i, iii\rangle$	$\langle ii, i, iii, iv\rangle$	$\langle 2,3,A,b,B\rangle$	yes
m_{24}	$\langle ii, iii, i\rangle$	$\langle ii, iii, i, iv\rangle$	$\langle 1,3,A,b,B\rangle$	no
m_{25}	$\langle iii, i, ii\rangle$	$\langle iii, i, ii, iv\rangle$	$\langle 0,3,A,b,B\rangle$	no
m_{26}	$\langle iii, ii, i\rangle$	$\langle iii, ii, i, iv\rangle$	$\langle 0,3,A,b,B\rangle$	no

Table 6.3: Embeddings and DFS codes during the pattern growth from 3-edge subgraphs (black lines) to 4-edge subgraphs (red lines) in the graphs of Figure 6.4.

Counterexample: Given the two graphs g_1, g_2 of Figure 6.4, in the 3rd iteration the two black-lined subgraphs are both represented by a single minimum DFS code X^3_{min}. Extending both by the red edges should result in the same minimal DFS code X^4_{min}. Both minimum DFS codes are listed on top of Table 6.3. The table further lists all 6 possible embeddings $m_{11}, .., m_{16}$ and $m_{21}, .., m_{26}$ for each of the two graphs before and after pattern growth. We see that not all possible extensions lead to a minimum DFS code (e.g., m_{11} does not). These are the ones that have to be filtered out in a verification step to be neither added to the result nor to become extended in the subsequent iteration.

In [114], only the first discovered embedding for each distinct edge subset and minimum DFS code is stored, i.e., only one of $m_{11}, .., m_{16}$ and one of $m_{21}, .., m_{16}$. Let m_{11} and m_{26} be the stored ones, then the frequency of X^4_{min} will be incorrect as the false positive code of m_{11} will be counted instead. Let m_{11} and m_{22} be the stored ones then the correct minimum DFS code will never be generated. We see, a single embedding per distinct edge set and DFS code cannot guarantee to generate all minimal children. Our counterexample shows that extending DFS codes using an isomorphism-free append-only approach based on only a single embedding cannot guarantee the correct result.

Contradiction: The isomorphism-free verification of [114] potentially fails for all subgraphs containing at least one automorphism. Subgraphs similar to our counterexample occur inter alia in molecular databases, for example, cycloalkanes[4].

6.5 Experimental Evaluation

In this section we present the results of a performance evaluation of DIMSpan based on a real molecular dataset of simple undirected graphs and a synthetic dataset of directed multigraphs. We evaluate scalability for increasing volume of input data, increasing result sizes (decreasing minimum support) and variable cluster size. For all experiments, we evaluate the improvement gained by our optimizations.

6.5.1 Implementation

We evaluated DIMSpan using Java 1.8, Apache Flink 1.2 and Hadoop 2.6.0. More precisely we used Flink's DataSet API[5] for all transformations and its *bulk iteration* for the iterative part. Apache Flink was chosen because an integration of DIMSpan into the Gradoop framework (Section 4.3) was one of the initial requirements. We further used the Simple16 implementation of JavaFastPFOR[6] [106] for compression. The source code of DIMSpan is available on GitHub[7] under Apache licence, version 2.0 (Alv2).

To show the impact of our optimizations, we made them configurable. In all evaluations, the term *baseline* refers to a configuration without preprocessing, without compression and local pattern verification before reporting, i.e., resolving isomorphism $|G| \cdot |P|$ times. We use Flink's aggregation to count pattern frequencies (lines 7,8 of Algorithm 6.4). This aggregation operator is already highly optimized by Apache Flink. Thus, to disable the combine step, we would have had to re-implement aggregation using the external API without these internal optimizations. Since this would have significantly blurred a potential comparison, the baseline already contains the combine operation.

[4] `https://en.wikipedia.org/wiki/Cycloalkane`

[5] `https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html`

[6] `https://github.com/lemire/JavaFastPFOR`

[7] `https://github.com/dbs-leipzig/gradoop`; org.gradoop.examples.dimspan

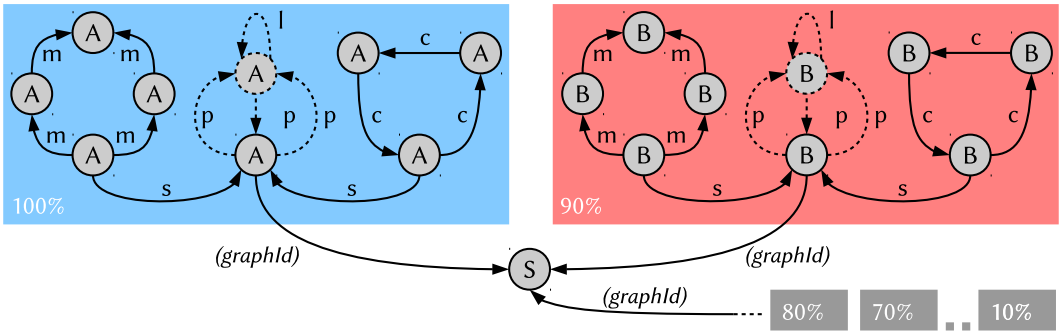


Figure 6.5: Example graph of Gradoop's predictable transactions graph generator. Colored rectangles represent subgraphs and their guaranteed support.

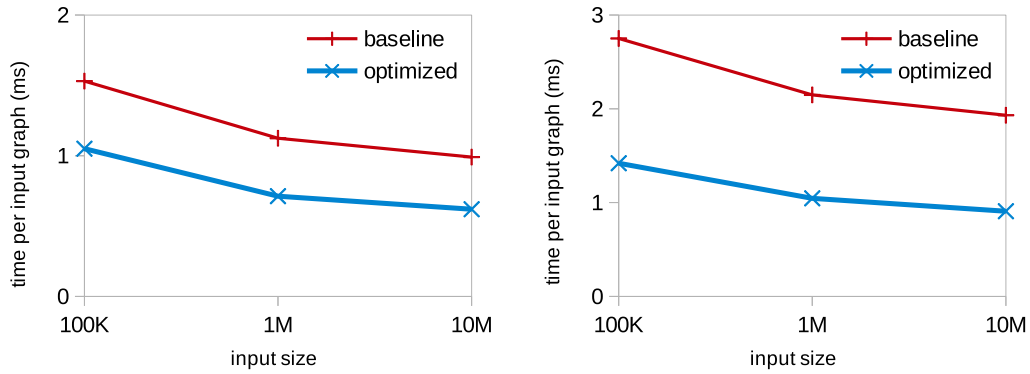
6.5.2 Datasets

We evaluated three data-related dimensions that impact the runtime of a distributed FSM algorithm: structural graph characteristics, *input size* $|G|$ and *result size* $|F|$. To show scalability for one of these dimensions, the other two need to be fixed. While $|F|$ can be increased by decreasing the minimum support threshold, varying the other two dimensions separately is less trivial. Thus, we decided to use two base datasets with divergent structural characteristics and just copy every graph several times to increase $|G|$ under preservation of structural characteristics and $|F|$, i.e., an increased number of embeddings at the same number of patterns.

The first base dataset is *yeast-active*[8], in the following denoted by *molecular*, a real dataset from anti-cancer research. It was chosen to represent molecular databases because structural characteristics among them do not fundamentally differ due to the rules of chemistry. For example, all molecular databases describe simple undirected graphs with only few different edge labels (e.g., single and double bond) and most of the frequent patterns are paths or trees [129]. The base dataset contains around 10K graphs (9567) and is scaled up to datasets containing around 100K to 10M graphs. We did not use an optimized version of DIMSpan for undirected graphs but provide an according parameter. If the parameter is set to undirected, the direction indicator (see Section 6.2.3) will just be ignored. Dedicated application logic will only be used, if it is unavoidable, for example, an 1-edge DFS code that describes a non-loop edge with two equal vertex labels (automorphism) leads to two embeddings in undirected mode.

The second category of datasets, in the following denoted by *synthetic*, was created by our own data generator[9]. It generates unequally sized connected directed multigraphs where each 10th graph has a different size ranging from $|V| = 10, |E| = 14$ to $|V| = 91, |E| = 140$. There are 11 distinct vertex and $5 + |G|/1000$ distinct edge labels. Figure 6.5 drafts their structure.

[8] `https://www.cs.ucsb.edu/~xyan/dataset.htm`
[9] org.gradoop.flink.datagen.transactions.predictable



(a) Molecular dataset
$|W| = 96$, $\phi^{rel}_{min} = 5\%$

(b) Synthetic dataset
$|W| = 96$, $\phi^{rel}_{min} = 70\%$

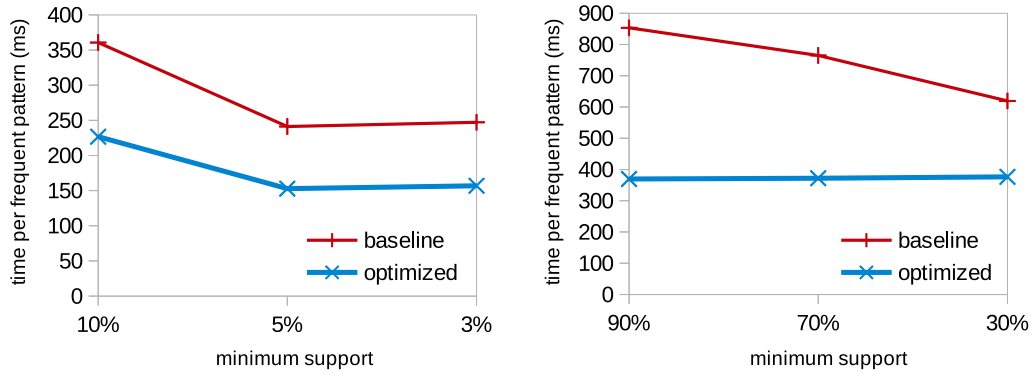
dataset	molecular			synthetic		
$t_b/|G|$ (input size)	100K	1M	10M	100K	1M	10M
baseline runtime total t_b (sec)	153	1124	9902	275	2148	19315
optimized runtime total t_o (sec)	105	712	6193	142	1045	9076
$t_b/|G|$ (ms)	1.5	1.1	1.0	2.8	2.1	1.9
$t_o/|G|$ (ms)	1.1	0.7	0.6	1.4	1.0	0.9
improvement $(t_b - t_o)/t_b$	31%	37%	37%	48%	51%	53%

Figure 6.6: Scalability for varying input size.

For each vertex label, there is an equivalent subgraph whose vertices all show this label. In particular, all graphs contain the subgraph labeled with As but only 90% the one labeled with Bs and so on. Although all subgraphs are connected via a vertex with label S this will not lead to frequent patterns since the respective edges are labeled by a graph identifier, i.e., are distinct per graph. The result is predictable and contains 702 frequent patterns with 1 to 13 edges for each min support decrement of 10% (i.e., 702 for 100%, 1404 for 90% , ..). The patterns contain loops, parallel edges (in and against direction), different subgraph automorphisms (e.g., "rotated" and "mirrored") separately as well as in all combinations. The data generator was not only designed for comparative evaluations but also for testing the correctness of implementations. To verify the number of contained frequent patterns we implemented a simple pruning-free brute-force FSM algorithm and manually verified all patterns of sizes 1..4, 12,13.

6.5.3 Experimental Results

All experiments were performed on our in-house cluster with 16 physical machines equipped with an Intel E5-2430 2.5 Ghz 6-core CPU, 48 GB RAM, two 4 TB SATA disks and running openSUSE 13.2. The machines are connected via 1 Gigabit Ethernet.



(a) Molecular dataset
$|W| = 96$, $|G| = 1M$

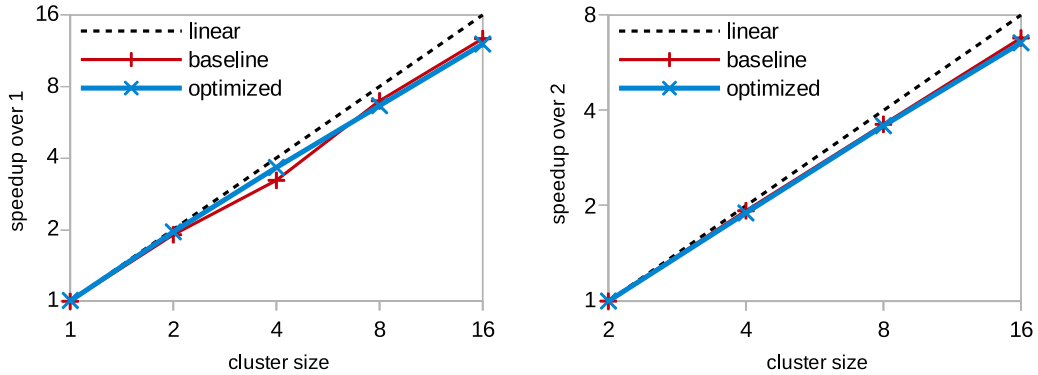
(b) Synthetic dataset
$|W| = 96$, $|G| = 1M$

dataset	molecular			synthetic		
ϕ^{rel}_{min} (minimum support)	10%	5%	3%	90%	70%	30%
$|F|$ (result size)	1270	4660	12807	1404	2808	5616
baseline runtime total t_b (sec)	458	1124	3166	1198	2148	3478
optimized runtime total t_o (sec)	288	712	2010	519	1045	2115
$t_b/|F|$ (ms)	361	241	247	853	765	619
$t_o/|F|$ (ms)	227	153	157	370	372	377
improvement $(t_b - t_o)/t_b$	37%	37%	37%	57%	51%	39%

Figure 6.7: Scalability for varying result size.

Input Size: Figure 6.6 shows measurement results for increasing input size $|G|$ for both datasets under fixed minimum support thresholds on a cluster with 16 machines and 96 worker threads ($|W| = 96$). To compare runtimes for different input sizes the charts show the average time to process a single input graph for the molecular (6.6a) and the synthetic dataset (6.6b). This time is constantly decreasing with an increasing input size for both workloads. The reason is the optimization strategy to verify DFS codes after counting (Section 6.3.2). Increasing input volume leads only to more embeddings but not to more frequent patterns. Thus, the number of isomorphism resolutions is only dependent on the result size which is fixed in this benchmark. For the same reason, the *improvement* of our optimized configuration is slightly increasing for larger datasets in comparison to the baseline (last row of the table in Figure 6.6). This outcome confirms a minor positive effect of minimizing the total number of isomorphism resolutions.

Result Size: Figure 6.7 shows measurement results for decreasing minimum support, i.e., increasing result size $|F|$, for both datasets under fixed input size on a cluster with 16 machines. The charts show the average time to extract a single frequent pattern for the molecular (6.7a) and the synthetic dataset (6.7b). Except for small result size on the molecular dataset, this time is constant for the



(a) Molecular dataset
$|G| = 1M$, $\phi^{rel}_{min} = 5\%$

(b) Synthetic dataset
$|G| = 1M$, $\phi^{rel}_{min} = 70\%$

dataset	molecular					synthetic				
physical machines	1	2	4	8	16	1	2	4	8	16
worker threads $\|W\|$	6	12	24	48	96	6	12	24	48	96
baseline runt. (sec)	14292	7521	4437	2054	1124	35586	14538	7540	4022	2148
optimized runt. (sec)	8588	4394	2354	1293	712	16470	6829	3599	1909	1045
improvement	40%	42%	47%	37%	37%	54%	53%	52%	53%	51%
baseline speedup	1.0	1.9	3.2	7.0	12.7		1.0	1.9	3.6	6.8
optimized speedup	1.0	2.0	3.6	6.6	12.1		1.0	1.9	3.6	6.5

Figure 6.8: Horizontal scalability for varying cluster sizes.

optimized version on both workloads, while the baseline time is decreasing for increasing input size. This shows, that the total runtime of the optimized version only depends on the result size, which is a desirable behavior. In contrast to the molecular dataset, the improvement of the synthetic workload is decreasing for larger results. The reason is, that due to its label diversity a relatively large part of the input data can be pruned during preprocessing for the synthetic dataset while rather rare as well as extremely frequent patterns in the molecular database contain the same atoms (vertex labels) and bonds (edge labels).

Cluster Size: Figure 6.8 shows measurement results for a variable cluster size, i.e., increasing number of worker threads $|W|$, for both datasets with fixed input size and under fixed minimum support thresholds. The charts show the speedup gained over one machine for the molecular (6.8a) and over two machines for the synthetic (6.8b) dataset. The latter was chosen since we achieved a superlinear speedup from 1 to 2 machines. Similar effects occur for 10K and 100K synthetic graphs as well as for different minimum support thresholds. We cannot explain these effects and thus attribute them to Apache Flink's program execution. For larger cluster sizes, we see that DIMSpan scales sligtly sublinear but still achieves notable speedups on both datasets for an increasing number of machines. The

slight decrease compared to an optimal speedup is influenced by the fact that the baseline already contains our efficient data structure and a combine operation that minimizes network traffic for pattern counting. Further on, the number of shuffled records in the counting phase is smaller for the baseline since false positives are verified before sending them over the network.

6.6 Conclusion

Frequent subgraph mining in graph collections is an important primitive to enable graph-based business analytics. Existing approaches provided no support for directed multigraphs and were either not scalable for large data volumes or not available to the public.

We presented DMGSpan, an approach that extends the popular gSpan algorithm i.a. by support for directed multigraphs, and DIMSpan, the first approach to parallel transactional FSM that combines its effective search space pruning with the technical advantages of state-of-the-art distributed in-memory dataflow systems. A functional comparison to approaches based on MapReduce has shown that DIMSpan is superior in terms of network traffic, disk access and the number of isomorphism resolutions. Our experimental evaluation showed the high scalability of DIMSpan for large datasets, low minimum support thresholds and increasing cluster size. We found that different optimizations depend on each other and should be chosen with regard to dataset characteristics. The source codes of both implementations are available to the public: DMGSpan is part of Directed Multigraph Miner[10] and DIMSpan is part of Gradoop (Section 4.3).

[10] `https://github.com/p3et/dmgm`

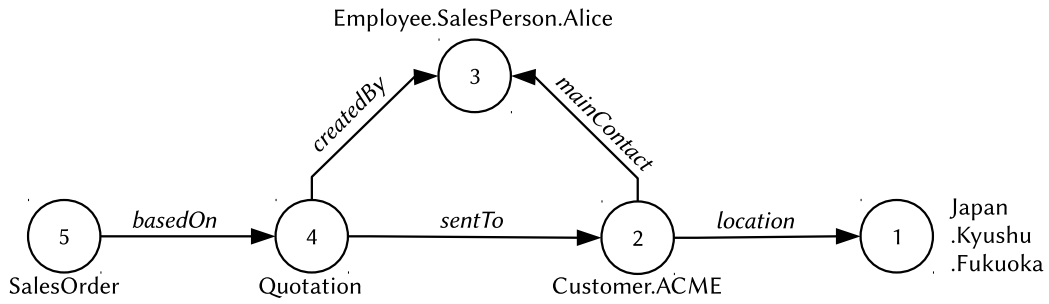
Chapter 7

Generalized Multidimensional Frequent Subgraph Mining

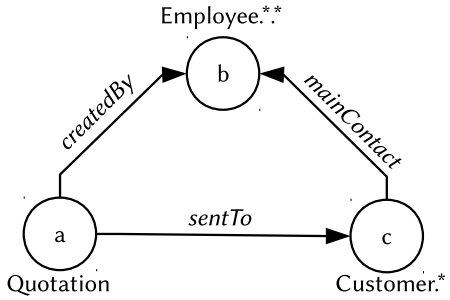
After business transaction graphs have been normalized for graph pattern mining (Section 3.4) their vertex labels represent dimension values. Often, dimension values can be attached to taxonomies, for example, a *city* can be assigned to a *region* and a region to a *country*. In this section we will study how our frequent subgraph miner DMGSpan (Section 6.2) can be extended to extract patterns at different levels of multiple dimension taxonomies. The respective problem is called *generalized multidimensional frequent subgraph mining (GM-FSM)*. We will first motivate the topic (Section 7.1) and provide formal introductions to problem, data model and terminology (Section 7.2). Afterwards, we will propose two methods that can be applied to solve this problem (Sections 7.3,7.4) and present the results of an experimental comparison (Section 7.5). Finally, we will end with a conclusion (Section 7.6).

7.1 Motivation

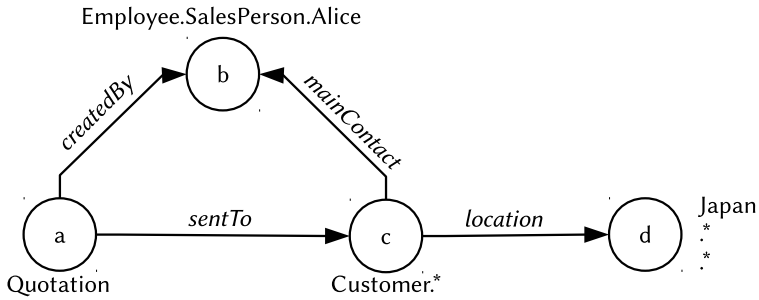
Frequent pattern mining (Section 2.8) is an important research problem that has found much interest since the early nineties. In a general notion a pattern is a collection of labels that are attached to a data structure such as itemsets, sequences, trees or graphs. In many applications these labels represent dimension values and can be assigned to taxonomies. In this case mining patterns at different taxonomy levels may reveal interesting patterns. For example, the pattern $\{\text{bread}, \text{butter}\}$ could be infrequent while the more general one $\{\text{bakery product}, \text{milk product}\}$ is frequent. In some cases, analysts also want to analyze patterns across levels. For example, the pattern $\{\text{wholegrain bread}, \text{butter}\}$ can be more interesting than just $\{\text{bread}, \text{butter}\}$. Finally, users also want to analyze patterns in the context of multiple dimensions, for example, to find out that $\{\text{bread}, \text{butter}\}$ is mostly bought in the morning in suburban stores.



(a) Subgraph of a business transaction graph.



(b) Generalize pattern (trivial).



(c) Generalized pattern (interesting).

Figure 7.1: Example multidimensional subgraph and patterns.

These simple examples show that frequent pattern mining across different levels of multiple dimensional taxonomies provides a high analytical value. However, a respective approach to generalized multidimensional pattern mining has only been studied for sequences [149]. With regard to graphs, generalization has already been investigated [76] but under the assumption that all vertices belong to the same semantic class (e.g., atoms). To the best of our knowledge, we developed the first approach to *generalized multidimensional frequent subgraph mining (GM-FSM)*. Further on, our approach supports directed multigraphs instead of simple undirected graphs and is based on an efficient pattern growth algorithm instead of an a priori approach.

Let us start with a look on an example application and the resulting graph patterns: Figure 7.1a shows an example subgraph of a business transaction graph. Edges and most vertices show simple labels (e.g., Quotation, createdBy). However, some vertices are labeled by *taxonomy paths* because they are attached to dimen-

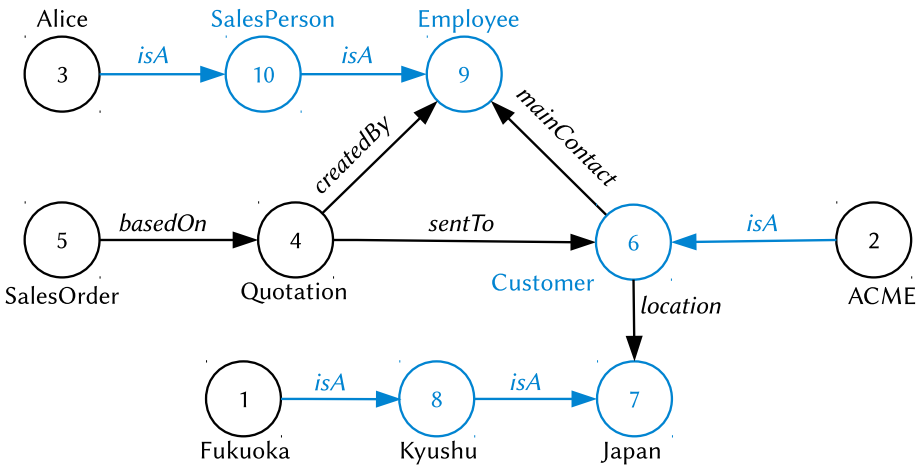


Figure 7.2: Path-substitution method: Taxonomy paths are represented by dedicated vertices and edges (blue lines).

sional taxonomies. Taxonomy paths are represented by the format [top-level].· · · .[bottom-level] where the *bottom-level* is the most discriminative dimension value. For example, vertex 3 not only represents Alice (bottom-level) but also more generally a SalesPerson and an Employee (top-level). This information is taken from a respective taxonomy to which Alice is assigned. Further on, vertex 1 represents Fukuoka, a city on the island Kyushu, and so on. Taxonomy paths can be generated in a normalization step (Section 3.5). Since the subgraph's vertices are associated to different taxonomies its contained patterns are considered *multidimensional*.

The problem of frequent subgraph mining (Definition 6.7) is the extraction of graph patterns that occur in at least a minimum number of graphs in a collection (*minimum support*). In scenarios where bottom-level labels (e.g., Alice) may have low frequencies, mining frequent graph patterns on the bottom-level will barely lead to results. By contrast, considering only the top-levels (e.g.,Employee.*.*) leads only to very general patterns such as the one of Figure 7.1b. The pattern expresses that a quotation was sent by an employee to a customer, where the latter has the sending employee as its main contact. However, since this pattern might occur in all business transaction graphs that represent sales process executions it is considered *trivial*. To find more *interesting* patterns, respective labels must be considered at arbitrary level combinations. By doing so, we can extract patterns as the one in Figure 7.1c. It expresses that the specific employee Alice sent a quotation to a customer from Japan. Here, labels at different levels are combined, in particular, the top-level label Japan.*.* and the bottom-level label Employee.Sales.Alice.

The naive approach to extract such patterns would be to mine bottom-levels first with a very low minimum support threshold and generalize patterns in a post processing step. In order to guarantee completeness with this approach the minimum support threshold must be zero. However, since frequent subgraph min-

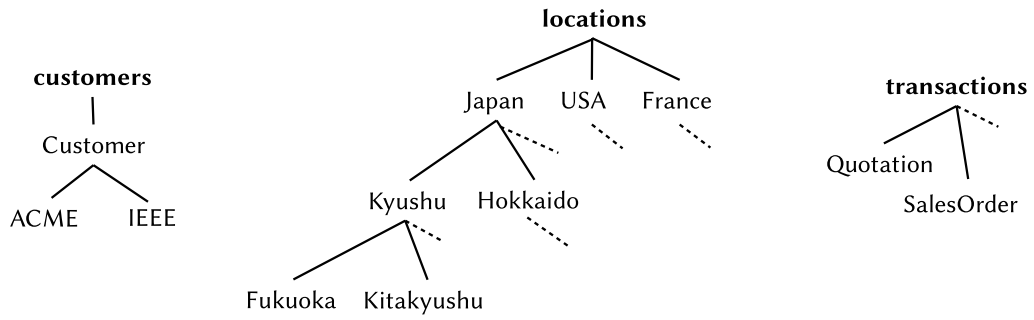


Figure 7.3: Example taxonomies of Figures 7.1 and 7.2. Bold fonts highlight dummy roots.

ing (FSM) includes the NP-complete subgraph isomorphism problem, a threshold close to zero would lead to an exploding result size as well as a dramatic increase of response time. For this reason, we developed two novel methods to GM-FSM which take advantage of special characteristics of taxonomy paths. Both methods are based on DMGSpan (Section 6.2) to extract frequent subgraphs from directed multigraphs.

In the first method (Section 7.3), we use a preprocessing step to integrate taxonomy paths into the graph structure as shown in Figure 7.2. Here, taxonomy paths are represented by actual paths in a graph that consist of only edges that show a dedicated label isA. The method requires two further modifications to find all results and to remove false-positives. In the second method (Section 7.4), we decompose the problem into FSM and *generalized frequent vector mining (GFVM)*. Therefore we decompose patterns into a top-level (highest general) graph structure and a vector of lower-level tails. In particular, top-level labels are attached to the graph structure, lower level tails of taxonomy paths are stored in a vector and vertices are mapped to vector fields. For example, vector $\langle$ Kyushu.Fukuoka, ACME, SalesPerson.Alice $\rangle$ can be derived from the subgraph spanned by v_1, v_2, v_3 in Figure 7.1a. The extraction happens in two steps: First, we use DMGSpan to identify frequent structural patterns only based on top-level labels and attach a set of lower-level vectors to each pattern. Second, we apply GFVM to refine the result.

7.2 Problem, Data Model and Terminology

The difference between FSM and GM-FSM is the definition of *pattern support*. In FSM, a graph will support a pattern, if there is an isomorphic subgraph with equal labels for all mapped pairs of vertices and edges (Definition 6.7). In GM-FSM, we additionally consider patterns which are *generalizations* of a subgraph to be supported. In the following, we will introduce a data model to describe multidimensional graphs and important terminology to formally introduce the GM-FSM problem.

Definition 7.1 (Taxonomy/Taxonomy Path) A *taxonomy* is defined as a quadruple $T = \langle L, \ell_{root}, \eta, \rho \rangle$ that contains a label set $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$, a *dummy root* label $\ell_{root} \notin L$, a parent-child mapping $\eta : L \rightarrow (L \cup \{\ell_{root}\})$ and a function $\rho : L \rightarrow \mathcal{P}(L)$ that associates a *taxonomy path* (an ordered set of labels) to every label. The *level* of a label is defined as the length of its taxonomy path denoted by $|\rho(\ell)|$ and defines an order among labels s.t. $\ell_1 < \ell_2 \Leftrightarrow |\rho(\ell_1)| < |\rho(\ell_2)|$. A valid taxonomy path contains exactly one label from all levels above the lowest one s.t. $\forall \ell \in L. \forall 1 \leq i \leq |\rho(\ell)|. \exists! \ell_p \in \rho(\ell). |\rho(\ell_p)| = i$ and for all labels except the one with the highest level its parent is contained in the path s.t.
$\forall \ell \in L. \forall \ell_i \in \rho(\ell). (\eta(\ell_i) \in \rho(\ell) \vee \eta(\ell_i) = \ell_{root})$.

To some readers, our formal introduction of taxonomy may seem strange and overcomplicated to describe a tree. However, we assume the reader to know the concept of a tree and tailored our definition to a most clear usage of taxonomy path function ρ in the remainder of this chapter.

Definition 7.2 (Label Generalization) Let T be a taxonomy and $\ell_s, \ell_g \in L_T$ be two of its labels then ℓ_g will be a *generalization* of ℓ_s, denoted by an operator $<_T$, if $\ell_g <_T \ell_s \Leftrightarrow \ell_g \neq \ell_s \wedge \ell_g \in \rho(\ell_s)$. We will further use a *generalized or equal* operator $\leq_T$ that includes equality, i.e., $\ell_g \leq_T \ell_s \Leftrightarrow \ell_g \in \rho(\ell_s)$.

Definition 7.3 (Top-level Label) Let T be a taxonomy then $\ell_{top} \in L$ will be a *top-level* label, if it has no generalizations. We will use $\omega : L \rightarrow \{true, false\}$ to denote a top-level label s.t. $\omega(\ell_{top}) = true \Leftrightarrow \nexists \ell \in L. \ell <_T \ell_{top}$.

Definition 7.4 (Bottom-level Label) Let T be a taxonomy then $\ell_{bot} \in L$ will be *bottom-level* label, if it is no generalization of any other label. We will use
$\beta : L \rightarrow \{true, false\}$ to denote a bottom-level label s.t.
$\beta(\ell_{bot}) = true \Leftrightarrow \nexists \ell \in L. \ell_{bot} <_T \ell$.

Figure 7.3 shows some examples of taxonomies, where customers, locations and transactions are dummy roots. The practical reason for the use of dummy roots is the following: We want to allow the user to decide, if a taxonomy contains either one or multiple top-level labels. However, formally a taxonomy must have a root. Thus, we decided to distinguish between (dummy) root and top-level. While the *root* is only a formal necessity a *top-level* label is the highest general label that may appear in patterns. For example, the highest general semantics of vertices that belong to the taxonomy with root customers of 7.3 is the *appearance of a customer*. By contrast, the highest general meaning of vertices that are mapped to the taxonomy with root locations is the *actual country* but not the bare appearance of a country. Thus, our model allows both to represent a logical single root by a single top-level label but also to avoid meaningless over-generalizations [76] by the use of multiple top-level labels.

In illustrations such as Figure 7.1 we use sequences of $*$ symbols to express the level difference of an actual label and its deepest specialization. For example, Japan.$*$.$*$ indicates the existence of a specialization whose taxonomy path has length 3. Further on, we express that this path is parent of all paths that show the same head but values instead of one or more $*$ symbols. For example, Japan.Kyushu.$*$ and Japan.Kyushu.Fukuoka are both specializations of Japan.$*$.$*$.

Definition 7.5 (Multidimensional Graph) A multidimensional graph is defined as a triple $g = \langle \mathfrak{g}, \mathcal{T}, \zeta \rangle$ of a labeled directed multigraph $\mathfrak{g}$ according to Definition 2.4, a set of taxonomies (dimensions) $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ and a mapping $\zeta : V_{\mathfrak{g}} \rightarrow \mathcal{T}$ that associates every vertex to a taxonomy that contains its vertex label s.t. $\forall v \in V_{\mathfrak{g}}.\lambda(v) \in L_{\zeta(v)}$.

Definition 7.6 (Graph Generalization) Given two multidimensional graphs g, s then g will be a *generalization* of s denoted by an operator $<_{\mathcal{T}}$, if there is a bijective edge mapping $\iota_e : E_g \leftrightarrow E_s$ with matching edge labels, sources and targets s.t. $\forall e \in E_g.\Big(\lambda_g(e) = \lambda_s(\iota_e(e)) \wedge \iota_v(\varsigma_g(e)) = \varsigma_s(\iota_e(e)) \wedge \iota_v(\tau_g(e)) = \tau_s(\iota_e(e))\Big)$ and there is a vertex mapping $\iota_v : V_g \leftrightarrow V_s$ where all labels are generalized or equal s.t. $\forall v \in V_g.\Big(\zeta_g(v) = \zeta_s(\iota_v(v)) \wedge \lambda_g(v) \leq_T \lambda_s(\iota_e(v))\Big)$ and there is at least one generalized label s.t. $\exists v \in V_g.\lambda_g(v) <_T \lambda_s(\iota_e(v))$.

Definition 7.7 (Top-level Graph) A *top-level graph* is a multidimensional graph g with only top-level labels s.t. $\forall v \in V.\lambda(v) = \omega(\lambda(v))$.

Definition 7.8 (Bottom-level Graph) A *bottom-level graph* is a multidimensional graph g with only bottom-level labels s.t. $\forall v \in V.\lambda(v) = \beta(\lambda(v))$.

Definition 7.9 (Generalized Multidimensional Frequent Subgraph Mininig) Let $G \in \mathcal{G}$ be a graph collection of multidimensional graphs then *generalized multidimensional frequent subgraph mining* is an operator $\Phi_{\mathcal{T}}(G, \phi_{min}) \mapsto F$ that extracts the complete set of frequent patterns $F \subseteq (P \cup P_{\mathcal{T}})$ from all contained patterns according to Definition 6.3 $P \in \mathcal{G}$ and all of their generalizations $P_{\mathcal{T}} \in \mathcal{G}$ whose support is greater or equal than a given minimum support threshold ϕ_{min} s.t. $\forall G \in \mathcal{G}, p \in (P \cup P_{\mathcal{T}}).(p \in F \Leftrightarrow \phi(G, p) \geq \phi_{min})$.

7.3 Path Substitution Method

We propose two methods to solve the problem of Definition 7.9. The first one is called *path substitution method* or simply *substitution method*. In this method we exploit the fact that (taxonomy) paths can be directly represented as parts of input graphs and result patterns. We consider this method to be the naive approach. The basic idea of the substitution method is to replace vertices whose labels are not the top-level of their associated taxonomy by their taxonomy path. Based thereon, we apply a modified variant of DMGSpan (Section 6.2) with additional pre- and postprocessing steps.

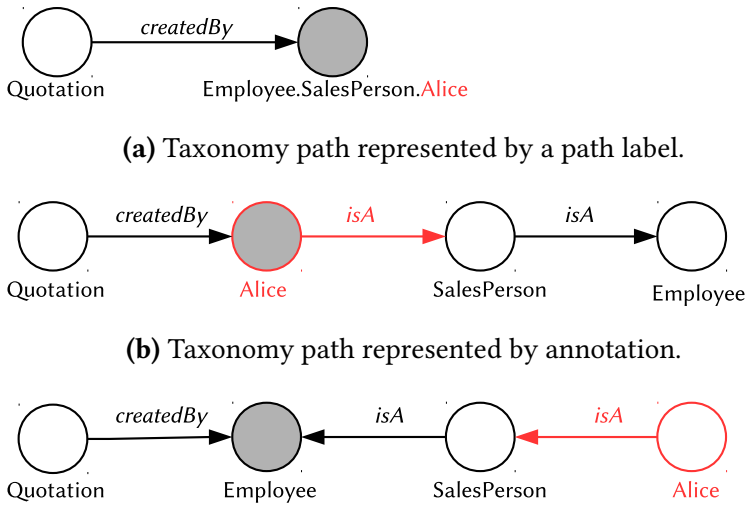


Figure 7.4: Comparison of taxonomy path representations in frequent graph patterns. The gray filled vertex can be assigned to a taxonomy path. Red color indicates infrequent parts of a pattern.

7.3.1 Taxonomy Path Substitution

Path substitution fundamentally differs from semantic annotations. Figure 7.4 illustrates the differences. Figure 7.4a shows a subgraph that represents a single logical relationship between a pair of entities (Quotation and Alice). The taxonomy path label at the vertex representing Alice indicates that there are two generalizations of Alice, in particular, SalesPerson and Employee. Red color indicates infrequent parts of the patterns, i.e., Quotation $\xrightarrow{createdBy}$ Employee and Quotation $\xrightarrow{createdBy}$ SalesPerson are frequent but Quotation $\xrightarrow{createdBy}$ Alice is not.

Figure 7.4b shows the same pattern but the taxonomy path of Alice is represented by a path attached to the vertex that represents the logical entity (*annotation*). All efficient frequent subgraph mining algorithms (Section 6.2.3) extract only those patterns whose parents were already frequent. This requires extractable patterns to be connected. In the context of embedded taxonomy paths generalizations must be subgraphs of their specializations. This is not the case for annotations since infrequent vertices or subpatterns such as the vertex Alice may cut frequent generalized patterns. For example, there is no connected subpattern in Figure 7.4b that represents Quotation $\xrightarrow{createdBy}$ SalesPerson.

Figure 7.4c represents the taxonomy path by *path substitution*. Here, the taxonomy path is embedded into the graph by the same number of vertices and edges but the other way around. In particular, vertex Alice was replaced by its highest generalization Employee. Thus, it will be possible to discover the general pattern first and extend it by specialization-edges as long as they are frequent. This approach would work with both pattern growth and a priori algorithms.

7.3.2 Extensions of FSM Algorithms

Extending a frequent subgraph algorithm, for example, DMGSpan, to support the path substitution method requires three steps: First, there must be a preprocessing step that performs the actual substitution. To preserve provenance and to make the process reversible a reserved edge label must be used for isA relationships. Second, there must be a postprocessing step that applies the reverse process to the extracted patterns by collapsing all isA paths to the most special label. Additionally, if vertex-only patterns shall not be extracted it will also be necessary to filter false positives, i.e., patterns of only isA labels, in this step. Third, implementations of FSM algorithms typically include a maximum pattern size (edge count) parameter to limit computations. This filter must be modified to consider only "real" edges, i.e., those representing actual relationships but not the isA ones. In contrast to the original algorithm this means that a higher number of iterations is required since taxonomy paths are processed just like regular edges.

7.4 Pattern Decomposition Method

In contrast to basic frequent subgraph mining GM-FSM is facing an increased number of label-combinations. For example, the subgraph

$\mathsf{Customer.ACME} \xrightarrow{location} \mathsf{Japan.Kyushu.Fukuoka}$

of Figure 7.1 has the following generalizations:

$\mathsf{Customer.*} \xrightarrow{location} \mathsf{Japan.Kyushu.Fukuoka}$
$\mathsf{Customer.*} \xrightarrow{location} \mathsf{Japan.Kyushu.*}$
$\mathsf{Customer.*} \xrightarrow{location} \mathsf{Japan.*.*}$
$\mathsf{Customer.ACME} \xrightarrow{location} \mathsf{Japan.Kyushu.*}$
$\mathsf{Customer.ACME} \xrightarrow{location} \mathsf{Japan.*.*}$

Applying the path substitution method of Section 7.3 shifts the identification of frequent combinations to the FSM algorithm since the algorithm makes no difference between logical and technical edges. This may lead to a dramatically increased number of isomorphism resolutions. For example, extending DMGSpan means that every additional isA edge leads to a multiple of DFS code verifications (Section 6.2.6). Thus, we consider the naive substitution method inefficient and investigated a second *pattern decomposition method* or simply *decomposition method*.

In this method we exploit the fact that a specialized graph can only be frequent if its highest generalization (top level graph, Definition 7.7) is already frequent. In particular, all specializations have the same topology as their highest generalization and, thus, isomorphism must only be resolved once per distinct top-level generalization. Thus, we first extract frequent top-level patterns by DMGSpan (Section 6.2) and identify frequent specializations by frequent vector mining (Section 7.4.2), i.e., we decompose the problem into two sub-problems.

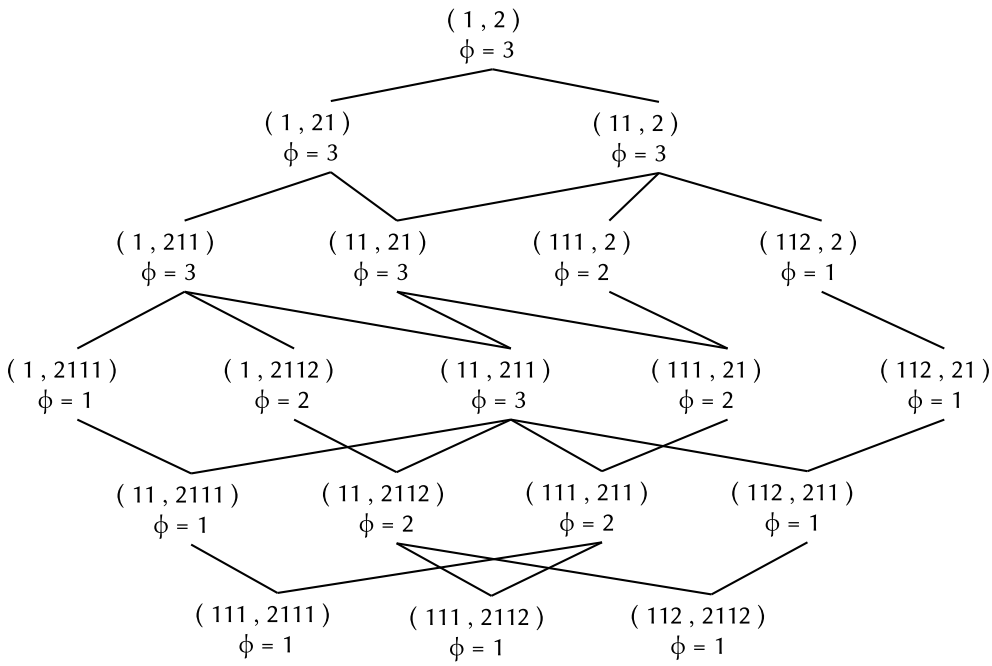


Figure 7.5: Generalization search lattice for a 2-dimensional example vector set $\mathcal{L} = \{(111, 2111), (111, 2112), (112, 2112)\}$. Common prefixes indicate label generalizations (e.g., $11 <_T 112$). Edges represent vector generalization from bottom to top.

However, to identify a minimum DFS code (Section 6.2.6) a global order of labels must ensure that the first order criterion is the taxonomy itself., i.e., we require the following constraint:

Definition 7.10 (Global Order of Labels) Let $\mathbb{T} = \{T_1, T_2, \ldots, T_n\}$ be the global space of taxonomies with an order s.t. $\forall T_1, T_2 \in \mathbb{T}.(T_1 < T_2 \vee T_1 = T_2 \vee T_1 > T_2)$, $\mathbb{L} = \{\ell_1, \ell_2, \ldots, \ell_n\}$ be the global alphabet of labels and $\xi : \mathbb{L} \rightarrow \mathbb{T}$ be a mapping that associates every label exclusively to a taxonomy then there is a transitive order among labels s.t. $\forall \ell_1, \ell_2 \in \mathbb{L}.(\xi(\ell_1) < \xi(\ell_2) \Leftrightarrow \ell_1 < \ell_2)$.

7.4.1 Extensions of DMGSpan

We extended DMGSpan to support the decomposition method. First of all, to implement the global order of labels according to Definition 7.10 we added a prefix to every vertex label that represents its taxonomy. This step happens before dictionary coding int the preprocessing phase. We order all vertex labels according to their label to ensure that the natural order of integers corresponds to the one of the original vertex labels.

Second, we use taxonomy paths as vertex labels. These are represented by integer arrays with one field for each level's encoded label. At the pattern growth step (line 9 of Algorithm 6.2), we consider only the top-level label, i.e., the first field

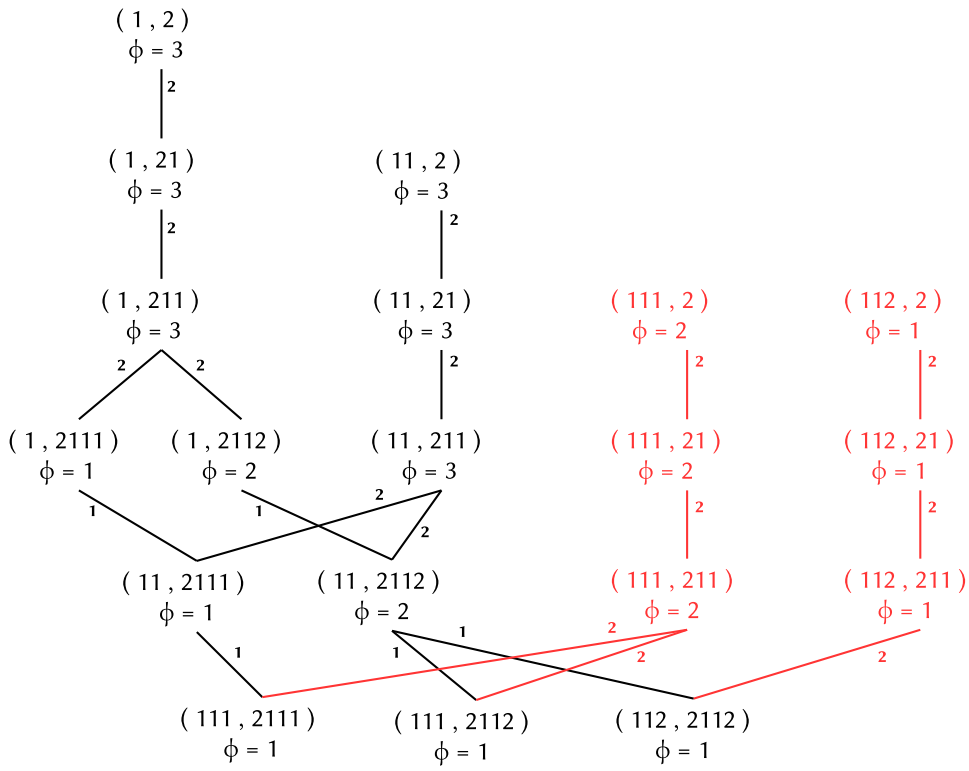


Figure 7.6: Bottom-up search in the example lattice of Figure 7.5. Edge labels correspond to u_{min} of Algorithm 7.1. Red lines indicate paths that have been traversed unnecessarily at $\phi_{min} = 3$.

of the array, to form a DFS Code that represents a top-level pattern (Definition 7.7). At the same time taxonomy paths of all vertices are added to each embedding. In this way they can be evaluated in the subsequent vector mining step. Before this, the support of the top-level pattern will be calculated and it will be verified. This ensures that vector mining is only applied for frequent and valid top-level patterns.

7.4.2 Generalized Frequent Vector Mining

For each frequent top-level pattern with u vertices and k edges there is a DFS Code $X = \langle x_1, x_2, \ldots, x_k \rangle$ to represent a pattern and a set of taxonomy path vectors with $|U|$ fields whose indices $u \in U$ correspond to vertex discovery times U from Definition 6.9, for example, the 1st vector field stores $\rho(\lambda(\nu(1)))$. Based on this set we apply *generalized frequent vector mining (GFVM)*:

Definition 7.11 (Taxonomy Path Vector) Let g be a multidimensional graph, X be a DFS code and $m_g^X = \langle \nu, \kappa \rangle$ be an embedding of X in g then a *taxonomy path vector* $\vec{\ell} = \langle \overline{\ell}_1, \overline{\ell}_2, \ldots, \overline{\ell}_u \rangle$ that is attached to m_g^X represents the taxonomy paths of its vertex labels s.t. $\overline{\ell}_u = \rho_{\zeta(\nu(u))}(\lambda(\nu(u)))$. The *vector space* of a taxonomy path vector corresponds to the taxonomies associated to the mapped vertices, i.e., $\zeta(\nu(1)) \times \zeta(\nu(2)) \times \cdots \times \zeta(\nu(u))$.

Algorithm 7.1 Bottom-up search GFVM

Input: vectors set (input), minimum support threshold (ϕ_{min}), vector indices U

Output: set of frequent vectors (output)

```
1:  specs ⊆ ℒ × U
2:  φ : ℒ → ℕ
3:  for all bottom in input do
4:      specs.add(⟨bottom, 1⟩)
5:      φ(bottom)++
6:  end for
7:  while specs ≠ ∅ do
8:      gens ⊆ ℒ × U
9:      for all ⟨spec, u_min⟩ in specs do
10:         for all u_min ≤ u ≤ |U| do
11:             if spec.isGeneralizableAt(u) then
12:                 gen = spec.generalize(u)
13:                 gens.add(⟨gen, u⟩)
14:                 φ(gen) += φ(spec)
15:             end if
16:         end for
17:     end for
18:     specs = gens
19: end while
20: for all ⟨vec, support⟩ in φ do
21:     if support ≥ φ_min then
22:         output.add(spec)
23:     end if
24: end for
25: return output
```

Definition 7.12 (Vector Generalization) Let $\vec{\ell}_s, \vec{\ell}_g$ be two taxonomy path vectors of the same DFS Code with vertex discovery times U then $\vec{\ell}_g$ will be a *generalization* of $\vec{\ell}_s$ denoted by an operator $\vec{<}_{\mathcal{T}}$, if all labels are generalized or equal s.t. $\forall u \in U.(\overline{\ell}_{gu} \subseteq \overline{\ell}_{su})$ and there is at least one generalized label s.t. $\exists u \in U.(\overline{\ell}_{gu} \subset \overline{\ell}_{su})$.

Definition 7.13 (Generalized Frequent Vector Mininig) Let $\mathcal{L}$ be a graph collection of taxonomy path vectors of the same vector space then *generalized frequent vector mining* aims to extract all frequent generalizations, i.e., all generalizations whose occurrence count is greater than or equal to a given minimum support threshold.

Based on Definition 7.12, the search space of all possible generalizations can be described by a lattice. Figure 7.5 shows a respective lattice for an example vector set. Taxonomy paths are represented by a canonical labeling for trees [111], e.g., label 111 could represent Employee.SalesPerson.Alice.

7.4.3 Bottom-up search

To efficiently extract all frequent generalizations it must be avoided to visit lattice nodes multiple times. We developed two algorithms that satisfy this requirement. The first one is based on level-wise generalization of the input vector list (*bottom-up* search). An example search graph is shown in Figure 7.6. To avoid multiple node visits, we only *generalize to the right*, i.e., a field will only be generalized, if itself, none or a field with smaller index was generalized before. Thus, a minimum generalization index u_{min} is passed among lattice nodes. In Figure 7.6 this index is represented by edge labels. For example, starting from (111, 2111) we generalize at $u = 1$ to (11, 2111) and at $u = 2$ to (111, 211). At the generalization of (111, 211) only fields right of $u_{min} = 2$ will be processed to avoid a second visit of (11, 211).

Algorithm 7.1 shows the pseudocode of the bottom up search. Inputs are a vector set, the minimum support threshold an the vertex discovery times of the DFS Code, i.e., the field numbers of all vectors. There are two main data structures: First, there is a set of specializations and their u_{min} (line 1) and, second, a support map (line 2). The data structures are initialized by values of each bottom level vector (line 3). In particular, we add every distinct vector to the set of specializations (line 4) and count its support (line 5). Then, we iteratively generalize specializations until no more generalization can be found (line 7). In the iteration body, we first initialize a set of generalizations (line 8) which will be the next iteration's input (line 18). Then, we process each distinct specialization (line 9) and create all possible generalizations (lines 11,12) *to the right* of field u_{min} (line 10). All generalizations will be added to the next iteration's dataset (line 13) and their support will be increased by the current specialization's support (line 14). After all generalizations were found, the frequent ones will be added to the output (lines 20-24).

We assume that in the standard use case all taxonomies are balanced rooted trees and all vector fields represent bottom-level labels. In this case, every lattice node will only be visited once (line 9) because our main data structure is a set and generalizations are only made to the right. Otherwise multiple visits of the same node are possible. However, we determine frequent vectors at the end of the algorithm. Thus, even unbalanced inputs will lead to the correct result because the frequency of the same node can be incremented in different iterations. Further on, the algorithm sums up frequencies (line 14) and, thus, is more efficient than just generating and counting all generalizations for each input vector. However, since

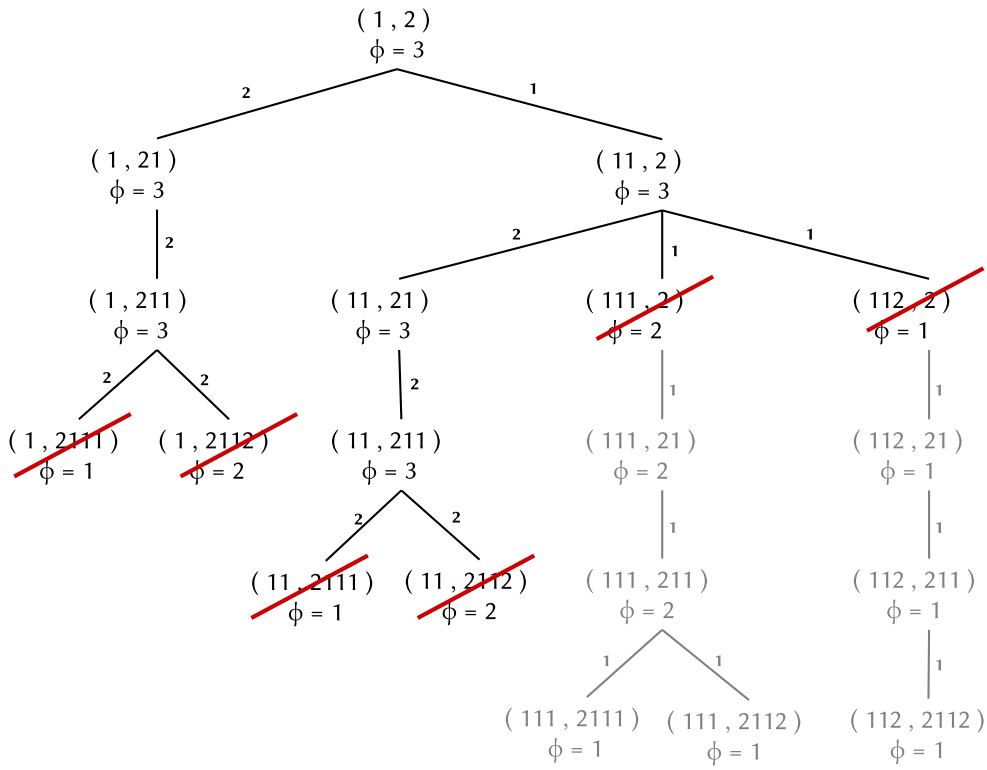


Figure 7.7: Top-down search in the example lattice of Figure 7.5. Edge labels correspond to u_{min} of Algorithm 7.2. Gray lines indicate pruned paths at $\phi_{min} = 3$.

frequent generalizations are determined at the end of the algorithm and there is no pruning during the search every possible node of the lattice has to be visited. The red lines in Figure 7.6 illustrate example search paths that lead only to infrequent generalizations. To overcome both limitations, i.e., to visit only as many nodes as possible and visit every node only once even for unbalanced inputs, we developed a second algorithm based on a top-down search.

7.4.4 Top-down search

The second approach is based on level-wise specialization of the input vector list (*top-down* search). An example search graph is shown in Figure 7.7. To avoid multiple node visits, we only *specialize to the right*, i.e., a field will only be specialized, if itself, none or a field with smaller index was specialized before. Thus, just like in the bottom-up search, a minimum generalization index u_{min} is passed among lattice nodes. In Figure 7.7 this index is represented by edge labels. For example, starting from (1, 2) we specialize at $u = 1$ to (11, 2) and at $u = 2$ to (1, 21). At the specialization of (1, 21) only fields right of $u_{min} = 2$ will be processed to avoid a second visit of, for example, (11, 21).

Algorithm 7.2 shows the pseudocode of the top-down search. Inputs are equivalent to the bottom-up search but one of the two main data structures differs: First, there is a set of, this time, generalizations and their u_{min} (line 1). Second, we do

Algorithm 7.2 Top-down search GFVM

Input: vectors set (input), minimum support threshold (ϕ_{min}), vector indices U

Output: set of frequent vectors (output)

```
1:  gens ⊆ ℒ × U
2:  bottomMap : ℒ → 𝒫(ℒ)
3:  for all bottom in input do
4:      top = maxGeneralization(bottom)
5:      gens.add(⟨top, 1⟩)
6:      bottomMap[top].add(bottom)
7:  end for
8:  while gens ≠ ∅ do
9:      specs ⊆ ℒ × U
10:     for ⟨gen, u_min⟩ in gens do
11:         bottoms = bottomMap[gen]
12:         if bottoms.size() ≥ φ_min then
13:             output.add(gen)
14:             for bottom in bottoms do
15:                 for all u_min ≤ u ≤ |U| do
16:                     if bottom.isSpecializableAt(gen, u) then
17:                         spec = specialize(gen, bottom, u)
18:                         specs.add(⟨spec, u⟩)
19:                         bottomMap[spec].add(bottom)
20:                     end if
21:                 end for
22:             end for
23:         end if
24:     end for
25:     gens = specs
26: end while
27: return output
```

not only store the support of generalizations but all bottom-level vectors which are their specializations (line 2) as well. The data structures are again initialized by values of each bottom level vector (line 3). However, in the top-down search we create the highest generalization (line 4), add it to the working set (line 5) and link it to the bottom level vector (line 6). Then, we specialize in a loop until no more specializations can be found (line 8). Similar to the bottom-up search, in each iteration we first initialize a set of specializations (line 9) which will be the next iteration's input (line 25). Then, we process all current generalizations (line 10). In contrast to the bottom-up search, we evaluate the support of a generalization directly in the search (line 11) and will only start specializing, if the generalization is still frequent (line 12) and add it directly to the output (line 13). Now, we must

process every assigned bottom-level vector (line 14) and check, if it can be specialized to the right (line 15) in the context of its current generalization (line 16). If so, we specialize (line 17), send the specialization to the next iteration (line 18) and map it to the current bottom-level (line 19).

The major advantage of this approach in comparison to bottom-up search is that support-based pruning is applied during the search (line 12) and, thus, only the first infrequent specialization of every search path must be visited. The red crossings in Figure 7.7 mark these nodes and gray lines indicate the pruned search paths. Since we transitively also traverse taxonomies top-down we need to visit every node only once even for unbalanced inputs. However, in comparison to the bottom-up search there is the additional cost of the bottom-level mapping (line 2) and respective access times (lines 6, 19). The map could be replaced by an on demand pattern matching in the input vector list where the current generalization is the pattern. However, this would make lines 6 and 19 even more expensive, i.e., the map acts as an index. In an experimental evaluation, we will show that this method outperforms the others, taxonomy path substitution method and the bottom-up search.

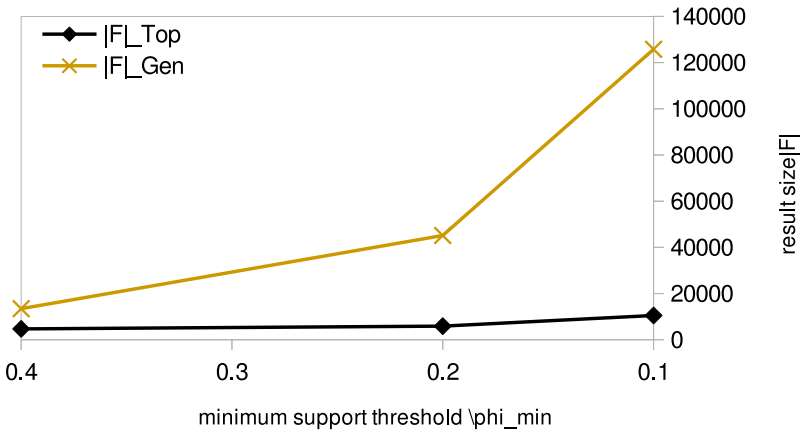
7.5 Experimental Evaluation

We implemented a prototype with Java 1.8 and compared the path substitution method *(PS)* to both decomposition methods, i.e., bottom-up *(BU)* and top-down *(TD)* search. The core of all methods was an early sequential implementation of DMGSpan. All experiments were run on a machine equipped with an Intel i7-4770 CPU, 16GB RAM, SSD and running Ubuntu 14.04.

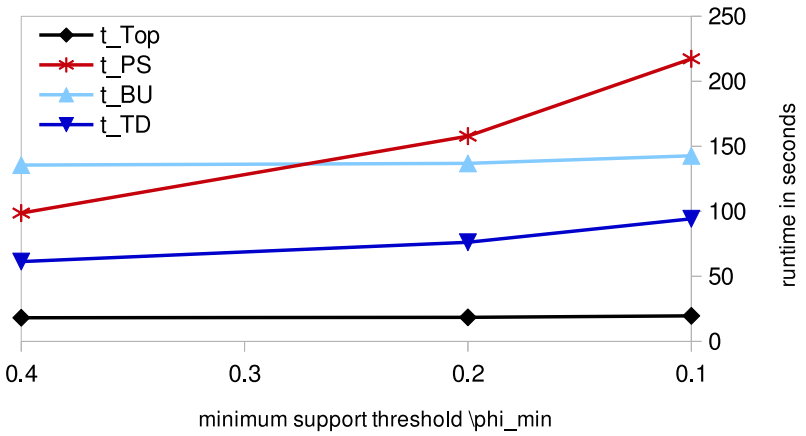
7.5.1 Dataset

The used dataset was generated by the Gradoop implementation of FoodBroker [92] which also contains master data properties that represent taxonomies. For example, products belong to different groups. Data was exported to files in TLF format[1]. To measure the particular impact of the applied generalization strategy we required a dataset for which frequent specializations impact the result size more than topological varieties, i.e., with an increasing minimum support threshold the number of frequent specialization should grow faster than the number of frequent top-level patterns. The creation of a respective dataset was very challenging since FoodBroker was originally not designed for this scenario and there are no other datasets available that satisfy this property. By a custom configuration and a specific graph selection we could create a respective dataset of 1333 graphs with 19

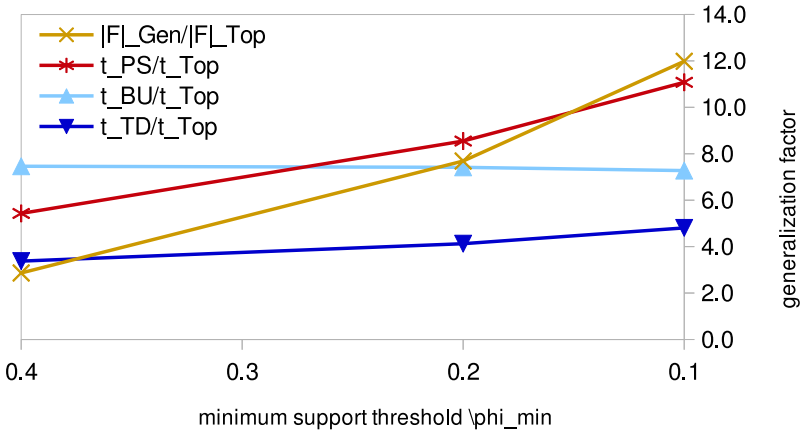
[1] `package org.gradoop.flink.io.impl.tlf`



(a) Result size.



(b) Runtime.



(c) Generalization factor.

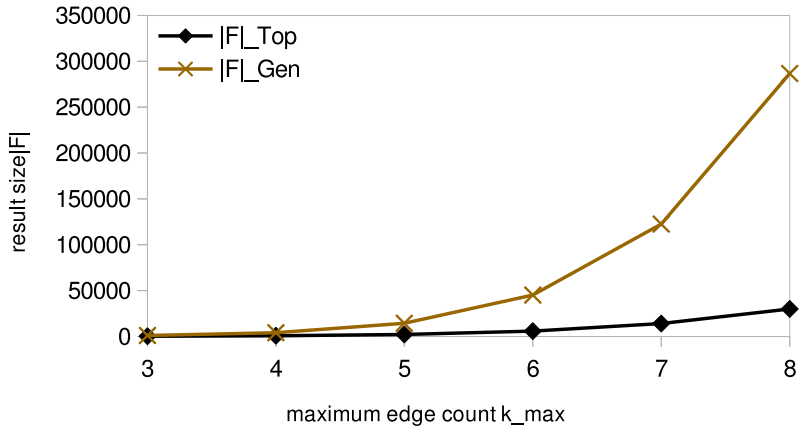
Figure 7.8: GM-FSM evaluation results for variable minimum support threshold ϕ_{min} and a fixed maximum edge count $k_{max} = 6$.

vertices and 22 edges each. However, all graphs were more or less completely isomorphic on the top-level which is the worst case for any frequent subgraph mining algorithm. Thus, we had to use a maximum edge count k_{max} parameter to limit the result size.

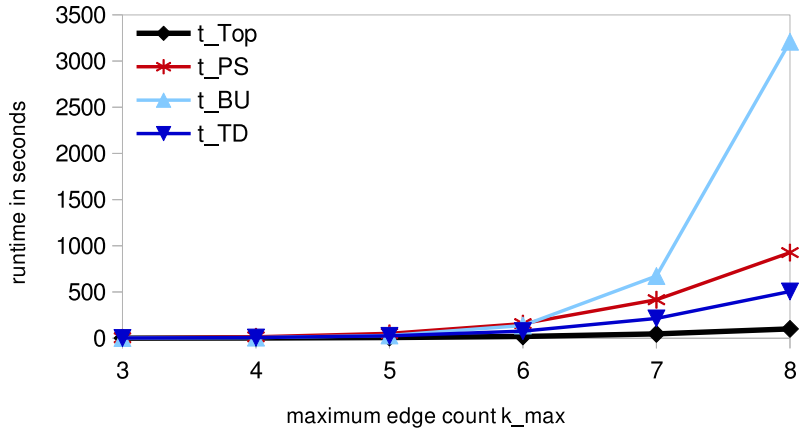
7.5.2 Results

We performed two experiments to measure the scalability of the different algorithms. In the first one we used a fixed maximum edge count k_{max} and a variable minimum support threshold ϕ_{min} to increase the result size. We measured data for $k_{max} = 3$ to $k_{max} = 8$ and will discuss the results of a medium value of $k_{max} = 6$. Figure 7.8a shows the result growth for a changing ϕ_{min}. Due to our dataset characteristics GM-FSM results $|F|_{Gen}$ grow exponentially while the number of top-level patterns $|F|_{Top}$ remains nealy constant. For an efficient algorithm we expect a runtime to increase slower than the result size. Figure 7.8b shows the runtime development for mining only top-levels t_{Top} and for our three proposed algorithms $t_{PS/BU/TD}$. As expected, t_{Top} is nearly constant since the number of frequent top-level patterns is bounded by k_{max}. Both decomposition methods show better runtimes and a less steep slope than the PS. At the comparison of both decomposition methods BU has a longer runtime but a less steep slope than TD.

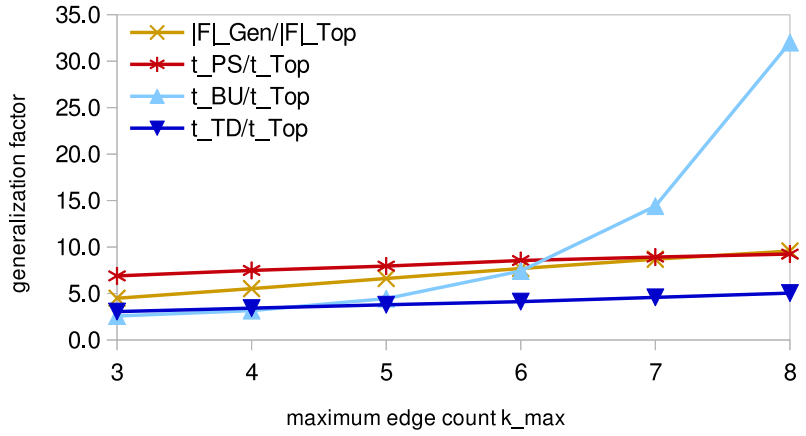
To compare slopes of results growth and runtimes we used a measure called *generalization factor* that is defined as the ratio of GM-FSM results and top-level FSM results $|F|_{Gen}/|F|_{Top}$ for the result size and as the ratio of GM-FSM time and top-level FSM time $t_{PS/BU/TD}/t_{Top}$ for runtimes. Figure 7.8c shows generalization factors for the result size and runtimes in a common chart. If a curve is below $|F|_{Gen}/|F|_{Top}$ it will be considered efficient. Except for $\phi_{min} = 0.4$ this is always the case for TD. We assume that the reason therefore lies in the overhead of the vector mining in the case of only few frequent specializations. However, the naive PS method is behaving constantly worse and even becomes inefficient for $\phi_{min} = 0.1$. In this figure, the BU approach seems to be increasingly efficient with increasing result size. Unfortunately, with the used dataset it was not possible to increase the result size further by lowering the minimum support threshold. However, the results of our second experiment will show that this is only the case for small vectors bounded by k_{max}.



(a) Result size.



(b) Runtime.



(c) Generalization factor.

Figure 7.9: GM-FSM evaluation results for fixed minimum support threshold $\phi_{min} = 0.2$ and a variable maximum edge count k_{max}.

In the second experiment, we used a fixed intermediate $\phi_{min} = 0.2$ and varied k_{max} to increase the result size. Figure 7.9a shows the development of result size over k_{max}. This time, the increase of $|F|_{Gen}$ is not only caused by the number of frequent specializations but also by an increasing number of frequent top-level patterns $|F|_{Top}$. Both show an exponential growth. Analog to our first experiment, we show runtimes (Figure 7.9b) and generalization factors (Figure 7.9c) for top-level FSM and our three GM-FSM algorithms. In this experiment, the BU method clearly shows the worst curves. Although it is the fastest approach for small k_{max} it becomes very inefficient for larger values. In particular, it is the only method whose slope is steeper than the one of the result size. The PS method is also inefficient but shows an improving behaviour for larger k_{max}. For values $k_{max} \geq 5$ the TD search shows lowest runtimes and best efficiency. For example, in comparison to top-level FSM it requires only about 5 times longer to extract 10 times more results for $k_{max} = 8$ while this ratio is about 1:1 for the PS method.

7.6 Conclusion

We presented the first study about multidimensional generalized frequent subgraph mining. To represent different dimensions, vertex labels are assigned to multiple taxonomies. Besides our conceptual contribution, we proposed three algorithms that solve this problem based on our data model. In an experimental evaluation the decomposition into frequent subgraph mining and frequent vector mining with top-down search has shown the best scalability. This confirms our expectation that decomposition is more efficient than embedding taxonomy paths into the graph structure and mining specializations by a standard frequent subgraph mining algorithm. In real-world evaluations (Sections 8.1 and 8.2) we found further indications that confirm this observation.

Chapter 8

Real-World Applications, Conclusion and Outlook

In the last chapter of this dissertation, we will report our practical experience from two real-world applications (Sections 8.1 and 8.2), provide a final summary (Section 8.3) and state some future research directions (Section 8.4).

8.1 BIIIG for Real Estate Fraud Detection

In the context of his bachelor's thesis [162] Saalmann performed the first real world evaluation of BIIIG in cooperation with Immowelt AG[1], a German company that runs multiple real estate platforms. The evaluation served as use case of the company's fraud detection team. Following, we will describe the analytical scenario, his technical solution and the evaluation results in more detail.

8.1.1 Analytical Scenario

The purpose of real estate platforms is to connect real estate providers with people who want to rent or buy properties. In the following, we will use the terms *agents* and *customers* to denote both groups. Other important domain entities of these platforms are *properties* and their *offers*. Further on, a platform has *visitors* that perform *actions* such as creating or viewing offers. There are different kinds of fraud by agents to cheat customers. Immowelt is already using different techniques to extract patterns that are typical to fraud by mining data that is known to be related to fraud. These patterns are then applied to monitor new offers and to automatically detect suspicious ones. The latter will then be verified by humans.

The primary goal of the evaluation was the identification of previously unknown fraud patterns by the use of BIIIG. As a second goal, all implementations should be done with Gradoop (Section 4.3) to evaluate its usability and to gain practical experience that can be used to improve the system.

[1] https://www.immowelt-group.com/

8.1.2 Application and implementation of BIIIG

Saalmann used Gradoop to implement a BIIIG workflow that includes characteristic subgraph mining (Section 3.5). At the time of the evaluation Gradoop was not able to import data directly from relational databases. Thus, he used a detour via CSV files that were exported from the relational source database and imported them into Gradoop by a custom logic to directly generate a single large graph. This graph contained master data (e.g., properties and locations) as well as transactional data (actions). Then, he applied the business transaction graph algorithm (Section 5.5) to extract graphs that represent single offers and all data related to their creation. As already discussed in Section 5.5.2 he added domain-specific extensions to the algorithm. Offers contained a property that marked them to be fraud. This property could be used to categorize (Section 3.5.1) graphs accordingly.

By contrast, the normalization process (Section 3.5.2) included multiple steps. First, vertex properties that represented relevant dimension values were replaced by dedicated vertices and edges, for example, Property $\xrightarrow{type}$ House. Second, to also evaluate action sequences the temporal order of transactional data was represented by dedicated edges in the format ActionType1 $\xrightarrow{then}$ ActionType2. Third, he applied taxonomy path substitution (Section 7.3) for dimensions that could be associated to taxonomies (e.g., locations). This solution was chosen because GM-FSM was not ported to Gradoop. Further on, there was no efficient version of characteristic subgraph mining. Thus, he chose the naive approach and used DIMSpan (Section 6.3) to mine frequent patterns with low minimum support thresholds separately for both categories (*fraud* and *no fraud*) and evaluated interestingness of patterns in a subsequent step.

8.1.3 Evaluation results

Saalmann stated that the functional evaluation of the characteristic subgraph mining workflow lead to meaningful results. Most but not all of the extracted patterns were already known by the domain experts. This shows that the method works. However, he noted that all interesting patterns had only the nature of sequences or itemsets and he called into question, if graph pattern mining is required to answer the analytical question of this scenario. However, he also stated that characteristic subgraph mining is very suitable for initial mining tasks whose results can be the base of data models for simpler frequent pattern mining techniques.

He also executed a performance evaluation on an in house cluster with 4 computing nodes. He acknowledged a good horizontal scalability but a bad overall performance. He identified two bottlenecks. The first one is the technical conversion of a single large graph into a collection of small graphs after the business transaction graph extraction. In particular, all vertices and edges of the initial single large graph have to be grouped by their graph id to create technical transactions that assure that all vertices and edges of the same graph are held on the same par-

tition. Logically, the problem cannot be avoided. However, the actual bottleneck is originated in the implementation of the distributed group by operation of Apache Flink. In particular, grouping is solely based on key hashing. Thus, in the worst case, even if all vertices and edges of the one graph are located on the same partition, the group by operation will send them to another one. In consequence, the complete graph collection will be shuffled among all workers.

The second bottleneck was the subgraph mining that only allowed to mine samples of the actual dataset. We see three reasons: The first reason, also stated by Saalmann himself, was an insufficient normalization process due to a lack of development time. Performance could have been improved dramatically by the elimination of frequent trivial subgraphs that lead to a potentiation of intermediate results without providing additional information. Second, he applied the path substitution method (Section 7.3) to extract generalized patterns, too. In Section 7.5 we reported that this approach is inefficient. Third, he used the naive approach to extract characteristic patterns. In Section 3.5.3 we stated that this is inefficient, too. However, the blame is not with Saalmann since one of the evaluation's restrictions was the use of Gradoop and at the time of his evaluation more efficient implementations were not available.

8.2 BIIIG for Security Threat Analysis

We performed a functional evaluation in cooperation with Siemens[2], a German company that is active in different areas of business. The evaluation served as use case of the company's security department. In the following, we will describe the analytical scenario, our technical solution and the evaluation results in more detail.

8.2.1 Analytical Scenario

"The mission of the Siemens Security Department is to ensure the long-term protection of Siemens' employees and assets against threats having the potential to impact the former."[3] Since the company has branches and customer projects in many parts of the world respective threats must be identified worldwide. One strategy for threat identification is analyzing and mining incident data. Here, the term *incident* refers to a mostly localized event that has an actual or potential impact on security such as bomb attacks, protests or other specific threats. Our goal was the application of BIIIG to extract knowledge that could not be extracted by other techniques that are already applied by the company. There are different companies that sell incident data. Due to nondisclosure we must neither name incident data providers nor established analytical techniques.

[2] `www.siemens.com`

[3] the department's official description, text quote from an internal document

Every incident can be assigned to different dimensions such as location, time or involved actors. Most of the dimensions are m:n relationships and some can be assigned to taxonomies. The first analytical question was the extraction of patterns that characterize a fixed dimension. For example, which pattern are characteristic for particular actors. The second question was the development of patterns for a fixed dimension over time. For example, how does the behaviour of a particular actor change over time. Finally, results should be ranked according to their statistical significance.

8.2.2 Application and implementation of BIIIG

We spent 12 person weeks (PW) for an in-house evaluation in the company's headquarters in Munich. About 1PW was required for organizational purposes and 9PW were spent for transformation and integration of source data. The remaining 2PW were used to implement analytical workflows that apply characteristic subgraph mining (CSM, Section 3.5) to the two analytical questions. The result was a prototype with two respective components. The prototype was written in Java 8 and its source code is closed. However, for all actual graph mining tasks we used *Directed Multigraph Miner (DMGM)*, an Open Source[4] Java library that includes an in-memory implementation of EPGM's property graph collections (Definition 4.1) as well as thread-parallel implementations of DMGSpan (Section 6.2) and GM-FSM in pattern decomposition variant with top-down search (Section 7.4.4). Further on, for both mining algorithms DMGM supports specific frequency determination and pruning techniques (Section 3.5.3) as they are required to implement CSM.

As already stated, most of the time was spent for turning data source into analyzable graphs. This included entity extraction from texts, data cleaning and data integration. There were three data sources in the formats CSV, XML and JSON. The data sources were directly transformed into an integrated graph collection. Since our focus lay on the evaluation of our data mining techniques and due to the lack of a relational data source the transformation and integration process of Chapter 5 was not applied. All data processing steps that were performed to create the graph collection were pragmatic data engineering without the application of any scientific approaches. Thus, we omit further details about them.

The starting point of the actual evaluation was a collection of about 188K acyclic graphs. Each of the graphs contained $1 + d$ vertices that represent an incident and all d available dimensions. Dimension types were represented by edge labels and taxonomy paths were integrated into vertex labels, for example, $\mathsf{Incident} \xrightarrow{location} \mathsf{Iraq.Saladin.Tikrit}$. Sizes varied from about $d = 5..20$ vertices and $d + 1$ edges. Some dimensions were present for every incident (e.g., date) while other dimensions occurred in variable quantities from $0..d$ (e.g., actors).

[4] `https://github.com/p3et/dmgm`

We implemented two configurable command line applications to answer our two analytical questions. Both programs included the DMGM implementation of generalized multidimensional CSM. The difference of both programs was the preprocessing of the graph collection, in particular, categorization and removal of vertices and edges that corresponded to a category. The latter was applied to suppress trivial patterns. For example, if a graph is categorized by terrorism, it will contains an edge Incident $\xrightarrow{type}$ Terrorism. In this case all contained patterns of this graph could be extended by this edge since all dimensions are connected to the incident vertex. The resulting patterns like 2017 $\xleftarrow{date}$ Incident $\xrightarrow{type}$ Terrorism will only increase result size but will not provide additional semantic value. Thus, we removed them to decrease runtime and to avoid meaningless results.

To answer the first question (dimension characteristics) we added the dimension value as a graph property and removed vertex and edge that represented this value before. In the case of more than one value for the selected dimension, we duplicated the graphs. This leads to correct results for the characteristic mining process. However, in the subsequent pattern selection (Section 3.5.4) we used the original collection size before duplication to calculate correct relative values.

To answer the second analytical questions (dimension development) we filtered graphs according to the containment of a particular dimension (e.g., terrorism), added a taxonomy path of the time dimension in desired granularity (e.g., YYYY.MM) and removed the vertex and edge that represent the date. The results could then be represented by a support-matrix of dimension values (time) and patterns to reflect their development.

To rank result patterns we reimplemented GraphRank [71] by He and Singh. In this approach every pattern is transformed into a feature vector and a p-value is calculated based on the features' probabilities to represent a patterns significance. In particular, we implemented the simplified model (Section 4 of [71]) and used an implementation of the regularized beta function provided by Apache Commons Math[5]. To make features configurable, we designed a generic approach that allows the user to define arbitrary features (Section 2 of [71]). Actual implementations that we have used in the evaluation were *edge labels*, i.e., which dimensions appeared, *vertex labels*, i.e., which dimension values/entities appeared and the *vertex label co-occurrences*, i.e., the common appearance of dimension values. The probabilities were calculated based on the input collection, for example, the vertex label co-occurrence feature Islamic State ⇔ Terrorism had a high probability.

The output format were CSV files whose columns represented patterns as well as their support and significance measures. Patterns were represented by ASCII characters similar to Neo4j Cypher.

[5] `http://commons.apache.org/proper/commons-math/javadocs/api-3.3/org/apache/commons/math3/special/Beta.html`

8.2.3 Evaluation results

Our major goal was a functional evaluation of generalized multidimensional characteristic subgraph mining. The mining results were presented to an experienced analyst of the security domain. He stated that the extracted knowledge corresponds to the professional knowledge of security experts. On the one hand, this means that we failed our goal to extract new knowledge. On the other hand, it shows that our method is correct since CSM requires no domain knowledge as input but extracts domain knowledge by a general algorithm. He further stated that the result could be useful for someone who is new to the domain and for monitoring. For example, if CSM is applied on a daily basis to observe particular dimensions (e.g., countries) new patterns will be detected automatically.

As a minor goal, we aimed to evaluate result ranking of graph patterns. Unfortunately, the p-value calculation of GraphRank failed for patterns with very low probabilities. In particular, very frequent and trivial patterns had p-values close to 1 and some less likely patterns had low p-values as expected. However, unlikely but characteristic patterns lead to p-values with value `NaN` as the result of a floating point calculation [227]. We were able to trace the problem back to the Apache Commons implementation of the regularized beta function. In previous internal evaluations we observed the same effect with the p-value calculation of [123, 145] by Micale et al. for large patterns with very rare labels. Since we are neither experts in statistics nor in the implementation of statistical measures we cannot just report our experience but have no explaination. As a work around we just passed probabilities of all features to the user who could then apply a custom result ranking using Microsoft Excel.

Due to a lack of time we could not carry out a systematic performance evaluation. However, we can report an overall positive impression of DMGM's performance. All evaluations were made on a Intel i5 dual core laptop with 16GB RAM. Response times of our analytical programs ranged from tens of seconds to a few minutes even for very low minimum support thresholds such as 0.1%.

8.3 Conclusion

This dissertation investigated the usage of graph data models and graph data mining for the analysis of business data to support decision-making. Therefore, we developed a conceptional framework called BIIIG (Business Intelligence with Integrated Instance Graphs). To provide a powerful example application we proposed a novel method called Characteristic Subgraph Mining (CSM). Here, interrelated domain data is represented by a collection of small graphs to extract patterns that are correlated with business indicators (e.g., financial result). To enable CSM multiple subproblems of diverse nature were solved:

First of all, our approach requires to represent collections of attributed graphs which may have attributes themselves to hold aggregated measure values and graph dimensions. By an overview of recent database systems we showed that there is neither a system nor a graph data model available that meets this requirement. Thus, we proposed to extend the property graph model by the support for graph properties and overlapping elements. Based on this data structure, we further proposed the operations aggregation, selection and property transformation that are essential to implement CSM. The resulting Extended Property Graph Model was the base of Gradoop, a distributed system from declarative graph analytics. Gradoop has been under ongoing development since 2014. In June 2018 the project had 18 contributors and 115 stars on GitHub[6] which are remarkable numbers for an academic prototype without full time professional developers.

BIIIG's prime motivation is the analysis of business data. In particular data of business information systems is most interesting for the evaluation of BIIIG but, unfortunately, most guarded by companies. Even in established cooperations we were never able to access this type of data and we found only a single vendor of an ERP system that gave us at least a test dataset. Thus, we developed FoodBroker, a data generator based on business process simulation whose datasets imitate those of business information systems.

Another crucial problem to which BIIIG provides an elaborate solution is the transformation of relational data into property graphs. Neither previous approaches nor the solutions provided by database vendors reach the functionality of BIIIG's metadata-driven graph transformation. In particular, our approach supports edge properties, the transformation of multiple databases into a single graph and a flexible management of schema mappings. Although an all-encompassing solution was never implemented single prototypes from academic and real-world evaluations have shown that the concept can not only be applied to relational databases but also to other structured data such as XML or JSON. BIIIG also includes a data integration strategy that allows to link data from multiple sources and to represent entities that correspond to each other by a single vertex. With our algorithm to extract business transaction graphs, we also presented a good starting point to turn a graph of domain objects into a meaningful collection of graph transactions.

To the best of our knowledge, before BIIIG frequent subgraph mining in the graph transaction setting was only applied to molecular datasets which are collections of simple undirected graphs. Since this is not the case for business data such as business transaction graphs, we extended the popular gSpan algorithm to support directed multigraphs and to hold embeddings in main memory to improve runtime. To make the algorithm applicable to scenarios where millions of graphs have to be evaluated we even studied a horizontally scalable variant. With

[6] `https://github.com/dbs-leipzig/gradoop`

DIMSpan we presented a solution based on state-of-the-art Big Data technology and with multiple optimizations. In experimental evaluations we were able to show that DIMSpan is not only suitable for business intelligence but also performs well in the classic scenario of molecular data.

Basic frequent subgraph mining is only of limited values since many interesting patterns appear only at generalization. Thus, we presented the first study about multidimensional generalized frequent subgraph mining. We proposed multiple algorithms to solve this problem. The most elaborate solution is exploiting the fact that large parts of the combinatorial problem can be solved in the absence of isomorphism resolution by decomposing it into frequent subgraph mining of highly generalized patterns and an efficient determination of all frequent specializations by frequent vector mining. In an experimental evaluation and by practical experience we showed that this approach performs much better than a naive solution that integrates generalizations into the graph structure.

We reported two real-word applications of characteristic subgraph mining. The diversity of applications shows that BIIIG is not only useful for business intelligence but can also help to detect patterns of fraud and security threats. As a major result we state that the approach leads to correct, meaningful and interesting results. We see the most valuable application in the initial extraction of knowledge from unknown data without a domain-specific hypothesis, i.e., it can discover unexpected correlations in complex data. Thus, BIIIG's findings can be the basis of "hard-wired" business intelligence solutions such as tailored data warehouse models and productive data mining tools. However, we have also seen that usability as well as the performance of distributed workflows should be improved and that result ranking still requires manual input.

We made the implementations of most of our prototypes available to the public. In particular, there is Directed Multigraph Miner[7] which provides parallel single machine implementations for frequent subgraph mining in standard and generalized variant as well as characteristic subgraph mining based on both variants. In June 2018 the library contained no further functionality than a database API and the mining algorithms. For now, complex graph workflows must be expressed in plain Java. However, there are programming interfaces that make the contribution of operators straight forward. By contrast, there is Gradoop which supports diverse data formats and includes business transaction graph extraction, FoodBroker, DIMSpan and all EPGM operators that have been proposed in this dissertation. However, due to the notable implementation effort in distributed systems we never found the time to port the generalized and characteristic variants.

[7] `https://github.com/p3et/dmgm`

8.4 Future Research Directions

As already mentioned earlier in this dissertation there are still open problems around graph-based data integration. First, there is the edge deduplication problem of Section 5.4.3. Further on, our current approach to vertex fusion (Section 5.4.2) includes a loss of information about inner cluster relationships and original property values. To improve provenance, the application of a nested graph model such as a recent approach [17] proposed by Bergami et al. could be investigated. With a respective approach, correspondence clusters would be preserved as they would just be nested into their representative and could be accessed or ignored according to the current application's demand.

Frequent subgraph mining is a very powerful technique but it is still the greatest bottleneck of our analytical approach. Thus, in particular the horizontal scalability should be improved further. Currently, all existing solutions follow synchronized approaches, i.e., all workers perform a certain task concurrently and synchronize after execution. This is the case for iterative solutions (e.g., DIMSpan) as well as for the filter and refinement approach. Negative side-effects are an increased memory usage of the level-wise search as all embeddings of one level are kept in main memory at the same time (iterative approaches) or a potentially huge number of subgraph-isomorphism testings (filter and refinement). The reason for both issues are originated in constraints of the distributed dataflow programming model. Thus, it should be investigated if, just like in graph processing, asynchronous iterative approaches can be used to improve distributed FSM by reducing waiting times and memory consumption.

Characteristic subgraph mining is a novel approach to analyze data of different source data models by the usage of graphs. Our functional evaluations in real-world applications have shown that the approach leads to meaningful and interesting results. However, our current implementation takes only few advantages of category specific supports. In particular, we still use the anti-monotonic property of frequent subgraph mining as our major pruning approach, i.e., follow a top-down approach. Thus, we must potentially mine many trivial parent patterns first before characteristic children can be found. Thus, future work should investigate, if bottom-up or hybrid approaches that directly apply a user-defined interestingness measure as pruning criterion could improve performance.

In the context of multidimensional generalized frequent subgraph mining we have shown that a decomposition that minimizes isomorphism resolutions can increase efficiency, i.e., graphs are powerful and have a unique expressiveness but when it comes to data mining their usage should be reduced to a minimum. Thus, an approach similar to our decomposition method could be applied to FSM in general. Since FSM algorithms evaluate only a single label per vertex every dimension value must be presented by a single vertex, too. However, in many scenarios a

single logical entity can be assigned to multiple dimensions. For example, in the evaluation with Immowelt Saalmann has expanded such multidimensional vertices to star-shaped subgraphs. Thus, a frequent subgraph mining algorithm with support for multiple labels per vertex should be developed. By doing so, a decomposition strategy similar to GM-FSM could be applied. Finally, both approaches could even be combined.

Bibliography

[1] Aggarwal, C. C., and Han, J., Eds. *Frequent Pattern Mining*. Springer, 2014.

[2] Agrawal, R., Imielinski, T., and Swami, A. N. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.* (1993), pp. 207–216.

[3] Agrawal, R., and Srikant, R. Fast algorithms for mining association rules in large databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile* (1994), pp. 487–499.

[4] Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M. J., Schelter, S., Höger, M., Tzoumas, K., and Warneke, D. The stratosphere platform for big data analytics. *VLDB J. 23*, 6 (2014), 939–964.

[5] Ambler, S. W. Mapping objects to relational databases: What you need to know and why. *Ronin International* (2000).

[6] Angles, R. A comparison of current graph database models. In *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012* (2012), pp. 171–177.

[7] Angles, R., Arenas, M., Barceló, P., Boncz, P. A., Fletcher, G. H. L., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J. F., van Rest, O., and Voigt, H. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), pp. 1421–1432.

[8] Angles, R., Boncz, P. A., Larriba-Pey, J., Fundulaki, I., Neumann, T., Erling, O., Neubauer, P., Martínez-Bazan, N., Kotsev, V., and Toma, I. The linked data benchmark council: a graph and RDF industry benchmarking effort. *SIGMOD Record 43*, 1 (2014), 27–31.

[9] Angles, R., and Gutiérrez, C. Survey of graph database models. *ACM Comput. Surv. 40*, 1 (2008), 1:1–1:39.

[10] Aridhi, S., D'Orazio, L., Maddouri, M., and Mephu, E. A novel MapReduce-based approach for distributed frequent subgraph mining. In *Reconnaissance de Formes et Intelligence Artificielle (RFIA) 2014* (France, June 2014).

[11] Arora, R., Goel, S., and Mittal, R. K. Using dependency graphs to support collaboration over github: The neo4j graph database approach. In *Ninth International Conference on Contemporary Computing, IC3 2016, Noida, India, August 11-13, 2016* (2016), pp. 1–7.

[12] Bader, D. A., and Madduri, K. Gtgraph: A synthetic graph generator suite. Technical report, 2006.

[13] Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G. H. L., Lemay, A., and Advokaat, N. gmark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng. 29*, 4 (2017), 856–869.

[14] Bedi, P., and Sharma, C. Community detection in social networks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 6*, 3 (2016), 115–135.

[15] Beheshti, S., Benatallah, B., and Motahari-Nezhad, H. R. Scalable graph-based OLAP analytics over process execution data. *Distributed and Parallel Databases 34*, 3 (2016), 379–423.

[16] Beheshti, S., Benatallah, B., Nezhad, H. R. M., and Sakr, S. A query language for analyzing business processes execution. In *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings* (2011), pp. 281–297.

[17] Bergami, G., Petermann, A., and Montesi, D. Thosp: an algorithm for nesting property graphs. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018* (2018), pp. 8:1–8:10.

[18] Bhuiyan, M., and Hasan, M. A. An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Trans. Knowl. Data Eng. 27*, 3 (2015), 608–620.

[19] Bizer, C. D2R MAP - A database to RDF mapping language. In *Proceedings of the Twelfth International World Wide Web Conference - Posters, WWW 2003, Budapest, Hungary, May 20-24, 2003* (2003).

[20] Bizer, C., and Schultz, A. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst. 5*, 2 (2009), 1–24.

[21] Bleco, D., and Kotidis, Y. Business intelligence on complex graph data. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops, Berlin, Germany, March 30, 2012* (2012), pp. 13–20.

[22] Bleco, D., and Kotidis, Y. Graph analytics on massive collections of small graphs. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.* (2014), pp. 523–534.

[23] Boncz, P. A. LDBC: benchmarks for graph and RDF data management. In *17th International Database Engineering & Applications Symposium, IDEAS '13, Barcelona, Spain - October 09 - 11, 2013* (2013), pp. 1–2.

[24] Borgelt, C., and Berthold, M. R. Mining molecular fragments: Finding relevant substructures of molecules. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan* (2002), pp. 51–58.

[25] Brin, S., Motwani, R., Ullman, J. D., and Tsur, S. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.* (1997), pp. 255–264.

[26] Bringmann, B., and Nijssen, S. What is frequent in a single graph? In *Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference, PAKDD 2008, Osaka, Japan, May 20-23, 2008 Proceedings* (2008), pp. 858–863.

[27] Burdick, D., Calimlim, M., and Gehrke, J. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany* (2001), pp. 443–452.

[28] Calders, T., Ramon, J., and Dyck, D. V. Anti-monotonic overlap-graph support measures. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy* (2008), pp. 73–82.

[29] Canim, M., and Chang, Y. System G data store: Big, rich graph data analytics in the cloud. In *2013 IEEE International Conference on Cloud Engineering, IC2E 2013, San Francisco, CA, USA, March 25-27, 2013* (2013), pp. 328–337.

[30] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull. 38*, 4 (2015), 28–38.

[31] Cattell, R. Scalable sql and nosql data stores. *SIGMOD Rec. 39*, 4 (May 2011), 12–27.

[32] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004* (2004), pp. 442–446.

[33] Chaudhuri, S., Dayal, U., and Narasayya, V. R. An overview of business intelligence technology. *Commun. ACM 54*, 8 (2011), 88–98.

[34] Chen, C., Yan, X., Zhu, F., Han, J., and Yu, P. S. Graph OLAP: towards online analytical processing on graphs. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy* (2008), pp. 103–112.

[35] Cheng, H., Yan, X., and Han, J. Mining graph patterns. In Aggarwal and Han [1], pp. 307–338.

[36] Cheng, J., Ke, Y., and Ng, W. Graphgen — a synthetic graph data generator. `http://www.cse.ust.hk/graphgen/`. Accessed: 2018-06-10.

[37] Codd, E. F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.

[38] Cook, D. J., and Holder, L. B. Substructure discovery using minimum description length and background knowledge. *J. Artif. Intell. Res. 1* (1994), 231–255.

[39] Damme, P., Habich, D., Hildebrandt, J., and Lehner, W. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.* (2017), pp. 72–83.

[40] Das, S., Srinivasan, J., Perry, M., Chong, E. I., and Banerjee, J. A tale of two graphs: Property graphs as RDF in oracle. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.* (2014), pp. 762–773.

[41] De Virgilio, R., Maccioni, A., and Torlone, R. Converting relational to graph databases. In *First International Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2013), GRADES '13, ACM, pp. 1:1–1:6.

[42] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[43] Denis, B., Ghrab, A., and Skhiri, S. A distributed approach for graph-oriented multidimensional analysis. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA* (2013), pp. 9–16.

[44] Diestel, R. *Graph Theory, 4th Edition*, vol. 173 of *Graduate texts in mathematics*. Springer, 2012.

[45] Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vañó, A., Gómez-Villamor, S., Martínez-Bazan, N., and Larriba-Pey, J. Survey of graph database performance on the HPC scalable graph analysis benchmark. In *Web-Age Information Management - WAIM 2010 International Workshops: IWGD 2010, XMLDM 2010, WCMT 2010, Jiuzhaigou Valley, China, July 15-17, 2010, Revised Selected Papers* (2010), pp. 37–48.

[46] Džeroski, S. Multi-relational data mining: An introduction. *SIGKDD Explor. Newsl. 5*, 1 (July 2003), 1–16.

[47] Eavis, T., and Zheng, X. Multi-level frequent pattern mining. In *Database Systems for Advanced Applications, 14th International Conference, DASFAA 2009, Brisbane, Australia, April 21-23, 2009. Proceedings* (2009), pp. 369–383.

[48] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. GRAMI: frequent subgraph and pattern mining in a single large graph. *PVLDB 7*, 7 (2014), 517–528.

[49] Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., and Boncz, P. A. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), pp. 619–630.

[50] Erling, O., and Mikhailov, I. RDF support in the virtuoso DBMS. In *Networked Knowledge - Networked Media - Integrating Knowledge Management, New Media Technologies and Semantic Systems*. 2009, pp. 7–24.

[51] Fahland, D., de Leoni, M., van Dongen, B. F., and van der Aalst, W. M. P. Many-to-many: Some observations on interactions in artifact choreographies. In *3rd Central-European Workshop on Services and their Composition, Services und ihre Komposition, ZEUS 2011, Karlsruhe, Germany, February 21-22, 2011. Proceedings* (2011), pp. 9–15.

[52] Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., and Wu, Y. Graph pattern matching: From intractable to polynomial time. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 264–275.

[53] Fazzinga, B., Flesca, S., Furfaro, F., Masciari, E., Pontieri, L., and Pulice, C. How, who and when: Enhancing business process warehouses by graph based queries. In *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016* (2016), pp. 242–247.

[54] Fong, J. Converting relational to object-oriented databases. *SIGMOD Record 26*, 1 (1997), 53–58.

[55] Fong, J., Wong, H. K., and Cheng, Z. Converting relational database into XML documents with DOM. *Information & Software Technology 45*, 6 (2003), 335–355.

[56] Fowler, M. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[57] Freeman, L. C. Centrality in social networks conceptual clarification. *Social Networks 1*, 3 (1978), 215 – 239.

[58] Gallagher, B. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS 6* (2006), 45–53.

[59] Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., and Jacobsen, H. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 1197–1208.

[60] Ghrab, A., Romero, O., Skhiri, S., Vaisman, A. A., and Zimányi, E. A framework for building OLAP cubes on graphs. In *Advances in Databases and Information Systems - 19th East European Conference, ADBIS 2015, Poitiers, France, September 8-11, 2015, Proceedings* (2015), pp. 92–105.

[61] Golfarelli, M., Dario, M., and Rizzi, S. The dimensional fact model: A conceptual model for data warehouses. *International Journal of Cooperative Information Systems 07*, 02n03 (1998), 215–247.

[62] Golfarelli, M., Rizzi, S., and Cella, I. Beyond data warehousing: what's next in business intelligence? In *DOLAP 2004, ACM Seventh International Workshop on Data Warehousing and OLAP, Washington, DC, USA, November 12-13, 2004, Proceedings* (2004), pp. 1–6.

[63] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012* (2012), pp. 17–30.

[64] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014), pp. 599–613.

[65] Gouda, K., and Zaki, M. J. Efficiently mining maximal frequent itemsets. In *Proceedings of the 2001 IEEE International Conference on Data Mining, 29 November - 2 December 2001, San Jose, California, USA* (2001), pp. 163–170.

[66] Gupta, A. Generating large-scale heterogeneous graphs for benchmarking. In *Specifying Big Data Benchmarks - First Workshop, WBDB 2012, San Jose, CA, USA, May 8-9, 2012, and Second Workshop, WBDB 2012, Pune, India, December 17-18, 2012, Revised Selected Papers* (2012), pp. 113–128.

[67] Han, J., and Fu, Y. Mining multiple-level association rules in large databases. *IEEE Trans. Knowl. Data Eng. 11*, 5 (1999), 798–804.

[68] Han, J., Kamber, M., and Pei, J. *Data Mining: Concepts and Techniques, 3rd edition.* Morgan Kaufmann, 2011.

[69] Han, J., Pei, J., and Yin, Y. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.* (2000), pp. 1–12.

[70] Hartig, O., and Thompson, B. Foundations of an alternative approach to reification in RDF. *CoRR abs/1406.3399* (2014).

[71] He, H., and Singh, A. K. Graphrank: Statistical modeling and mining of significant subgraphs in the feature space. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China* (2006), pp. 885–890.

[72] Hill, S., Srichandan, B., and Sunderraman, R. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *ACM International Conference on Bioinformatics, Computational Biology and Biomedicine, BCB' 12, Orlando, FL, USA - October 08 - 10, 2012* (2012), pp. 661–666.

[73] Holzschuher, F., and Peinl, R. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Joint 2013*

EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings (2013), ACM, pp. 195–204.

[74] Huan, J., Wang, W., and Prins, J. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003), 19-22 December 2003, Melbourne, Florida, USA* (2003), pp. 549–552.

[75] Huan, J., Wang, W., Prins, J., and Yang, J. SPIN: mining maximal frequent subgraphs from graph databases. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25, 2004* (2004), pp. 581–586.

[76] Inokuchi, A. Mining generalized substructures from a set of labeled graphs. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004), 1-4 November 2004, Brighton, UK* (2004), pp. 415–418.

[77] Inokuchi, A., Washio, T., and Motoda, H. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery, 4th European Conference, PKDD 2000, Lyon, France, September 13-16, 2000, Proceedings* (2000), pp. 13–23.

[78] Iordanov, B. Hypergraphdb: A generalized graph database. In *Web-Age Information Management - WAIM 2010 International Workshops: IWGD 2010, XMLDM 2010, WCMT 2010, Jiuzhaigou Valley, China, July 15-17, 2010, Revised Selected Papers* (2010), Springer, pp. 25–36.

[79] Iosup, A., Hegeman, T., Ngai, W. L., Heldens, S., Prat-Pérez, A., Manhardt, T., Chafi, H., Capota, M., Sundaram, N., Anderson, M. J., Tanase, I. G., Xia, Y., Nai, L., and Boncz, P. A. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB 9*, 13 (2016), 1317–1328.

[80] Jain, N., Liao, G., and Willke, T. L. Graphbuilder: scalable graph ETL framework. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013* (2013), p. 4.

[81] Jayanthi, B., Duraiswamy, K., et al. A novel algorithm for cross level frequent pattern mining in multidatasets. *International Journal of Computer Applications (0975–8887) Vol 37* (2012).

[82] Jiang, C., Coenen, F., and Zito, M. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review 28*, 1 (2013), 75–105.

[83] Jindal, A., and Madden, S. Graphiql: A graph intuitive query language for relational databases. In *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014* (2014), pp. 441–450.

[84] Junghanns, M., Kiessling, M., Averbuch, A., Petermann, A., and Rahm, E. Cypher-based graph pattern matching in GRADOOP. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017* (2017), pp. 3:1–3:8.

[85] Junghanns, M., Kiessling, M., Teichmann, N., Gómez, K., Petermann, A., and Rahm, E. Declarative and distributed graph analytics with GRADOOP. *PVLDB to appear* (2018).

[86] Junghanns, M., Petermann, A., Gómez, K., and Rahm, E. GRADOOP: scalable graph data management and analytics with hadoop. *CoRR abs/1506.00548* (2015).

[87] Junghanns, M., Petermann, A., Neumann, M., and Rahm, E. Management and analysis of big graph data: Current systems and open challenges. In *Handbook of Big Data Technologies*. 2017, pp. 457–505.

[88] Junghanns, M., Petermann, A., and Rahm, E. Distributed grouping of property graphs with GRADOOP. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings* (2017), pp. 103–122.

[89] Junghanns, M., Petermann, A., Teichmann, N., Gómez, K., and Rahm, E. Analyzing extended property graphs with apache flink. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA@SIGMOD 2016, San Francisco, California, USA, July 1, 2016* (2016), pp. 3:1–3:8.

[90] Junghanns, M., Petermann, A., Teichmann, N., and Rahm, E. The big picture: Understanding large-scale graphs using graph grouping with GRADOOP. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings* (2017), pp. 629–632.

[91] Ke, Y., Cheng, J., and Yu, J. X. Efficient discovery of frequent correlated subgraph pairs. In *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009* (2009), pp. 239–248.

[92] Kemper, S., Petermann, A., and Junghanns, M. Distributed foodbroker: Skalierbare generierung graphbasierter geschäftsprozessdaten. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Workshopband* (2017), pp. 105–110.

[93] Kessl, R., Talukder, N., Anchuri, P., and Zaki, M. J. Parallel graph mining with gpus. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, August 24, 2014* (2014), pp. 1–16.

[94] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013* (2013), pp. 169–182.

[95] Khetrapal, A., and Ganesh, V. Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University* (2006), 22–28.

[96] Kimball, R., and Ross, M. *The data warehouse toolkit: the complete guide to dimensional modeling.* John Wiley & Sons, 2011.

[97] Koller, D., and Friedman, N. *Probabilistic Graphical Models - Principles and Techniques.* MIT Press, 2009.

[98] Koop, D., Freire, J., and Silva, C. T. Visual summaries for graph collections. In *IEEE Pacific Visualization Symposium, PacificVis 2013, February 27 2013-March 1, 2013, Sydney, NSW, Australia* (2013), pp. 57–64.

[99] Köpcke, H., and Rahm, E. Frameworks for entity matching: A comparison. *Data Knowl. Eng. 69*, 2 (Feb. 2010), 197–210.

[100] Kotidis, Y. Extending the data warehouse for service provisioning data. *Data Knowl. Eng. 59*, 3 (2006), 700–724.

[101] Kuramochi, M., and Karypis, G. Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining, 29 November - 2 December 2001, San Jose, California, USA* (2001), pp. 313–320.

[102] Kuramochi, M., and Karypis, G. Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov. 11*, 3 (2005), 243–271.

[103] Kyrola, A., Blelloch, G. E., and Guestrin, C. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems*

Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012 (2012), pp. 31–46.

[104] Lam, D. N., Liu, A. Y., and Martin, C. E. *Graph-Based Data Warehousing Using the Core-Facets Model.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 240–254.

[105] Lee, S., Park, B. H., Lim, S., and Shankar, M. Table2graph: A scalable graph construction from relational tables using map-reduce. In *First IEEE International Conference on Big Data Computing Service and Applications, BigDataService 2015, Redwood City, CA, USA, March 30 - April 2, 2015* (2015), pp. 294–301.

[106] Lemire, D., and Boytsov, L. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper. 45*, 1 (2015), 1–29.

[107] Lenzerini, M. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2002), PODS '02, ACM, pp. 233–246.

[108] Lijffijt, J., Papapetrou, P., and Puolamäki, K. A statistical significance testing approach to mining the most informative set of patterns. *Data Min. Knowl. Discov. 28*, 1 (2014), 238–263.

[109] Lin, W., Xiao, X., and Ghinita, G. Large-scale frequent subgraph mining in mapreduce. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014* (2014), pp. 844–855.

[110] Ling, Z. J., Tran, Q. T., Fan, J., Koh, G. C. H., Nguyen, T., Tan, C. S., Yip, J. W. L., and Zhang, M. GEMINI: an integrative healthcare analytics system. *PVLDB 7*, 13 (2014), 1766–1771.

[111] Liu, J., and Zhang, X. X. Dynamic labeling scheme for XML updates. *Knowl.-Based Syst. 106* (2016), 135–149.

[112] Liu, Y., Dighe, A., Safavi, T., and Koutra, D. Graph summarization methods and applications: A survey. *CoRR abs/1612.04883* (2018).

[113] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB 5*, 8 (2012), 716–727.

[114] Lu, W., Chen, G., Tung, A. K. H., and Zhao, F. Efficiently extracting frequent subgraphs using mapreduce. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA* (2013), pp. 639–647.

[115] Lysenko, A., Roznovat, I. A., Saqi, M., Mazein, A., Rawlings, C. J., and Auffray, C. Representing and querying disease networks using graph databases. *BioData Mining 9* (2016), 23.

[116] Maatuk, A. M., Ali, M. A., and Rossiter, B. N. Converting relational databases into object-relational databases. *Journal of Object Technology 9*, 2 (2010), 145–161.

[117] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010* (2010), pp. 135–146.

[118] Martínez-Bazan, N., Gómez-Villamor, S., and Escale-Claveras, F. DEX: A high-performance graph database management system. In *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany* (2011), pp. 124–127.

[119] Marton, J., Szárnyas, G., and Varró, D. Formalising opencypher graph queries in relational algebra. In *Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings* (2017), pp. 182–196.

[120] Meinl, T., Wörlein, M., Fischer, I., and Philippsen, M. Mining molecular datasets on symmetric multiprocessor systems. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Taipei, Taiwan, October 8-11, 2006* (2006), pp. 1269–1274.

[121] Meng, X., Bradley, J. K., Yavuz, B., Sparks, E. R., Venkataraman, S., Liu, D., Freeman, J., Tsai, D. B., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M. J., Zadeh, R., Zaharia, M., and Talwalkar, A. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research 17* (2016), 34:1–34:7.

[122] Menge, F. Enterprise service bus. In *Free and open source software conference* (2007), vol. 2, pp. 1–6.

[123] Micale, G., Giugno, R., Ferro, A., Mongiovì, M., Shasha, D. E., and Pulvirenti, A. Fast analytical methods for finding significant labeled graph motifs. *Data Min. Knowl. Discov. 32*, 2 (2018), 504–531.

[124] Miliaraki, I., Berberich, K., Gemulla, R., and Zoupanos, S. Mind the gap: large-scale frequent sequence mining. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 797–808.

[125] Miller, J. J. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA* (2013), vol. 2324, p. 36.

[126] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A. Introducing the graph 500. *Cray Users Group (CUG) 19* (2010), 45–74.

[127] Nagumey, A. Book review : Parallel and distributed computation: Numerical methods: Dimitri p. bertsekas and john n. tsitsiklis. 1989. englewood cliffs, new jersey: Prentice-hall. 715 pp. $40. *IJHPCA 3*, 4 (1989), 73–74.

[128] Nentwig, M., Hartung, M., Ngomo, A. N., and Rahm, E. A survey of current link discovery frameworks. *Semantic Web 8*, 3 (2017), 419–436.

[129] Nijssen, S., and Kok, J. N. The gaston tool for frequent subgraph mining. *Electr. Notes Theor. Comput. Sci. 127*, 1 (2005), 77–87.

[130] Nijssen, S., and Kok, J. N. Frequent subgraph miners: Runtime don't say everything. In *Proceedings of the International Workshop on Mining and Learning with Graphs (MLG 2006* (2006), pp. 173–180.

[131] Nooijen, E. H. J., van Dongen, B. F., and Fahland, D. Automatic discovery of data-centric and artifact-centric processes. In *Business Process Management Workshops - BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers* (2012), pp. 316–327.

[132] O'Neil, E. J. Object/relational mapping 2008: Hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 1351–1356.

[133] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[134] Park, Y., Shankar, M., Park, B., and Ghosh, J. Graph databases for large-scale healthcare systems: A framework for efficient data management and data services. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014* (2014), pp. 12–19.

[135] Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. Discovering frequent closed itemsets for association rules. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings.* (1999), pp. 398–416.

[136] Pei, J., and Han, J. Constrained frequent pattern mining: a pattern-growth view. *SIGKDD Explorations 4*, 1 (2002), 31–39.

[137] Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., and Hsu, M. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Trans. Knowl. Data Eng. 16*, 11 (2004), 1424–1440.

[138] Petermann, A. Graph pattern mining for business decision support. In *Proceedings of the VLDB 2017 PhD Workshop co-located with the 43rd International Conference on Very Large Databases (VLDB 2017), Munich, Germany, August 28, 2017.* (2017).

[139] Petermann, A., and Junghanns, M. Scalable business intelligence with graph collections. *it - Information Technology 58*, 4 (2016), 166–175.

[140] Petermann, A., Junghanns, M., Kemper, S., Gómez, K., Teichmann, N., and Rahm, E. Graph mining for complex data analytics. In *IEEE International Conference on Data Mining Workshops, ICDM Workshops 2016, December 12-15, 2016, Barcelona, Spain.* (2016), pp. 1316–1319.

[141] Petermann, A., Junghanns, M., Müller, R., and Rahm, E. BIIIG: enabling business intelligence with integrated instance graphs. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014* (2014), pp. 4–11.

[142] Petermann, A., Junghanns, M., Müller, R., and Rahm, E. Foodbroker - generating synthetic datasets for graph-based business analytics. In *Big Data Benchmarking - 5th International Workshop, WBDB 2014, Potsdam, Germany, August 5-6, 2014, Revised Selected Papers* (2014), pp. 145–155.

[143] Petermann, A., Junghanns, M., Müller, R., and Rahm, E. Graph-based data integration and business intelligence with BIIIG. *PVLDB 7*, 13 (2014), 1577–1580.

[144] Petermann, A., Junghanns, M., and Rahm, E. Dimspan: Transactional frequent subgraph mining with distributed in-memory dataflow systems. In *Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies* (New York, NY, USA, 2017), BDCAT '17, ACM, pp. 237–246.

[145] Petermann, A., Micale, G., Bergami, G., Pulvirenti, A., and Rahm, E. Mining and ranking of generalized multi-dimensional frequent subgraphs. In *2017 Twelfth International Conference on Digital Information Management (ICDIM)* (Sept 2017), pp. 236–245.

[146] Pham, M., Boncz, P. A., and Erling, O. S3G2: A scalable structure-correlated social graph generator. In *Selected Topics in Performance Evaluation and Benchmarking - 4th TPC Technology Conference, TPCTC 2012, Istanbul, Turkey, August 27, 2012, Revised Selected Papers* (2012), pp. 156–172.

[147] Pham, M., Passing, L., Erling, O., and Boncz, P. A. Deriving an emergent relational schema from RDF data. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015* (2015), pp. 864–874.

[148] Pinto, H., Han, J., Pei, J., Wang, K., Chen, Q., and Dayal, U. Multi-dimensional sequential pattern mining. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, November 5-10, 2001* (2001), pp. 81–88.

[149] Plantevit, M., Laurent, A., Laurent, D., Teisseire, M., and Choong, Y. W. Mining multidimensional and multilevel sequential patterns. *TKDD 4*, 1 (2010), 4:1–4:37.

[150] Poulovassilis, A., and Levene, M. A nested-graph model for the representation and manipulation of complex objects. *ACM Trans. Inf. Syst. 12*, 1 (1994), 35–68.

[151] Power, D. J., Sharda, R., and Burstein, F. Decision support systems. *Wiley Encyclopedia of Management 7* (2015), 1–4.

[152] Qiao, F., Zhang, X., Li, P., Ding, Z., Jia, S., and Wang, H. A parallel approach for frequent subgraph mining in a single large graph using spark. *Applied Sciences 8*, 2 (2018), 230.

[153] Qu, Q., Zhu, F., Yan, X., Han, J., Yu, P. S., and Li, H. Efficient topological OLAP on information networks. In *Database Systems for Advanced Applications - 16th International Conference, DASFAA 2011, Hong Kong, China, April 22-25, 2011, Proceedings, Part I* (2011), pp. 389–403.

[154] Rahimian, F., Payberah, A. H., Girdzijauskas, S., and Haridi, S. Distributed vertex-cut partitioning. In *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings* (2014), pp. 186–200.

[155] Rahm, E., and Bernstein, P. A. A survey of approaches to automatic schema matching. *The VLDB Journal 10*, 4 (Dec. 2001), 334–350.

[156] Ranu, S., and Singh, A. K. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China* (2009), pp. 844–855.

[157] Rodriguez, M. A., and Neubauer, P. Constructions from dots and lines. *Bulletin of the Association for Information Science and Technology 36*, 6 (2010), 35–41.

[158] Roy, A., Bindschaedler, L., Malicevic, J., and Zwaenepoel, W. Chaos: scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), pp. 410–424.

[159] Rudolf, M., Paradies, M., Bornhövd, C., and Lehner, W. The graph story of the SAP HANA database. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings* (2013), pp. 403–420.

[160] Rudolf, M., Paradies, M., Bornhövd, C., and Lehner, W. Synopsys: Large graph analytics in the sap hana database through summarization. In *First International Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2013), GRADES '13, ACM, pp. 16:1–16:6.

[161] Rudolf, M., Voigt, H., Bornhövd, C., and Lehner, W. Synopsys: Foundations for multidimensional graph analytics. In *Enabling Real-Time Business Intelligence - International Workshops, BIRTE 2013, Riva del Garda, Italy, August 26, 2013, and BIRTE 2014, Hangzhou, China, September 1, 2014, Revised Selected Papers* (2014), pp. 159–166.

[162] Saalmann, E. Fallstudie zu graphbasierter Business Intelligence mit Gradoop am Beispiel der Immobilienwirtschaft. In *Bachelor's Thesis* (2017), University of Leipzig.

[163] Saeedi, A., Peukert, E., and Rahm, E. Comparative evaluation of distributed clustering schemes for multi-source entity resolution. In *Advances in Databases and Information Systems* (Cham, 2017), M. Kirikova, K. Nørvåg, and G. A. Papadopoulos, Eds., Springer International Publishing, pp. 278–293.

[164] Saigo, H., Krämer, N., and Tsuda, K. Partial least squares regression for graph mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008* (2008), pp. 578–586.

[165] Salihoglu, S., and Widom, J. GPS: a graph processing system. In *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013* (2013), pp. 22:1–22:12.

[166] Sequeda, J. F., Arenas, M., and Miranker, D. P. On directly mapping relational databases to RDF and OWL. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012* (2012), pp. 649–658.

[167] Shahrivari, S., and Jalili, S. Distributed discovery of frequent subgraphs of a network using mapreduce. *Computing 97*, 11 (2015), 1101–1120.

[168] Soussi, R., Cuvelier, E., Aufaure, M., Louati, A., and Lechevallier, Y. DB2SNA: an all-in-one tool for extraction and aggregation of underlying social networks from relational databases. In *The Influence of Technology on Social Network Analysis and Mining*. Springer, 2013, pp. 521–545.

[169] Srikant, R., and Agrawal, R. Mining generalized association rules. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.* (1995), pp. 407–419.

[170] Srikant, R., and Agrawal, R. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings* (1996), pp. 3–17.

[171] Stratikopoulos, A., Chrysos, G., Papaefstathiou, I., and Dollas, A. Hpc-gspan: An fpga-based parallel system for frequent subgraph mining. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014* (2014), pp. 1–4.

[172] Stutz, P., Bernstein, A., and Cohen, W. W. Signal/collect: Graph algorithms for the (semantic) web. In *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I* (2010), pp. 764–780.

[173] Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G., and Xie, G. T. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), pp. 1887–1901.

[174] Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. Arabesque: a system for distributed graph mining.

In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), ACM, pp. 425–440.

[175] Thomas, L. T., Valluri, S. R., and Karlapalem, K. MARGIN: maximal frequent subgraph mining. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China* (2006), pp. 1097–1101.

[176] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. From "think like a vertex" to "think like a graph". *PVLDB 7*, 3 (2013), 193–204.

[177] Tian, Y., Hankins, R. A., and Patel, J. M. Efficient aggregation for graph summarization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008* (2008), pp. 567–580.

[178] van der Aalst, W. M. P. Process mining: Overview and opportunities. *ACM Trans. Management Inf. Syst. 3*, 2 (2012), 7:1–7:17.

[179] Vanetik, N., Gudes, E., and Shimony, S. E. Computing frequent graph patterns from semistructured data. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan* (2002), pp. 458–465.

[180] Vasilyeva, E., Thiele, M., Bornhövd, C., and Lehner, W. Leveraging flexible data management with graph databases. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013* (2013), p. 12.

[181] Verma, S., Leslie, L. M., Shin, Y., and Gupta, I. An experimental comparison of partitioning strategies in distributed graph processing. *PVLDB 10*, 5 (2017), 493–504.

[182] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., and Wilkins, D. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference, 2010, Oxford, MS, USA, April 15-17, 2010* (2010), p. 42.

[183] Vo, B., Nguyen, D., and Nguyen, T. A parallel algorithm for frequent subgraph mining. In *Advanced Computational Methods for Knowledge Engineering - Proceedings of 3rd International Conference on Computer Science, Applied Mathematics and Applications - ICCSAMA 2015, Metz, France, 11-13 May, 2015* (2015), pp. 163–173.

[184] Wang, C., Wang, W., Pei, J., Zhu, Y., and Shi, B. Scalable mining of large disk-based graph databases. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25, 2004* (2004), pp. 316–325.

[185] Wang, T., Huang, H., Lu, W., Peng, Z., and Du, X. Efficient and scalable mining of frequent subgraphs using distributed graph processing systems. In *Database Systems for Advanced Applications - 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21-24, 2018, Proceedings, Part I* (2018), pp. 891–907.

[186] Wang, X., Zhang, D., Gu, T., and Pung, H. K. Ontology based context modeling and reasoning using OWL. In *2nd IEEE Conference on Pervasive Computing and Communications Workshops (PerCom 2004 Workshops), 14-17 March 2004, Orlando, FL, USA* (2004), pp. 18–22.

[187] Wang, Z., Fan, Q., Wang, H., Tan, K., Agrawal, D., and El Abbadi, A. Pagrol: Parallel graph olap over large-scale attributed graphs. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014* (2014), pp. 496–507.

[188] Webb, G. I. Discovering significant patterns. *Machine Learning 68*, 1 (2007), 1–33.

[189] Weske, M. *Business Process Management - Concepts, Languages, Architectures, 2nd Edition.* Springer, 2012.

[190] Wörlein, M., Meinl, T., Fischer, I., and Philippsen, M. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In *Knowledge Discovery in Databases: PKDD 2005, 9th European Conference on Principles and Practice of Knowledge Discovery in Databases, Porto, Portugal, October 3-7, 2005, Proceedings* (2005), pp. 392–403.

[191] Xia, Y., Tanase, I. G., Nai, L., Tan, W., Liu, Y., Crawford, J., and Lin, C. Graph analytics and storage. In *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014* (2014), pp. 942–951.

[192] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. Graphx: a resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013* (2013), p. 2.

[193] Xirogiannopoulos, K., and Deshpande, A. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (2017), pp. 897–912.

[194] Yan, X., Cheng, H., Han, J., and Yu, P. S. Mining significant graph patterns by leap search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008* (2008), pp. 433–444.

[195] Yan, X., and Han, J. gspan: Graph-based substructure pattern mining. `http://cs.ucsb.edu/~xyan/papers/gSpan.pdf`. Technical Report 2002-10-08.

[196] Yan, X., and Han, J. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan* (2002), pp. 721–724.

[197] Yan, X., and Han, J. Closegraph: mining closed frequent graph patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003* (2003), pp. 286–295.

[198] Yen, S., and Chen, A. L. P. A graph-based approach for discovering various types of association rules. *IEEE Trans. Knowl. Data Eng. 13*, 5 (2001), 839–845.

[199] Yin, M., Wu, B., and Zeng, Z. Hmgraph OLAP: a novel framework for multi-dimensional heterogeneous network analysis. In *DOLAP 2012, ACM 15th International Workshop on Data Warehousing and OLAP, Maui, HI, USA, November 2, 2012, Proceedings* (2012), pp. 137–144.

[200] Yu, C., and Chen, Y. Mining sequential patterns from multidimensional sequence data. *IEEE Trans. Knowl. Data Eng. 17*, 1 (2005), 136–140.

[201] Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., and Liu, L. Triplebit: a fast and compact system for large scale RDF data. *PVLDB 6*, 7 (2013), 517–528.

[202] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012* (2012), pp. 15–28.

[203] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on*

Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010 (2010).

[204] Zaki, M. J. Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada* (2002), pp. 71–80.

[205] Zaki, M. J., and Hsiao, C. CHARM: an efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002* (2002), pp. 457–473.

[206] Zhang, J., Long, X., and Suel, T. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008* (2008), pp. 387–396.

[207] Zhang, N., Tian, Y., and Patel, J. M. Discovery-driven graph summarization. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA* (2010), pp. 880–891.

[208] Zhang, Z. J. Graph databases for knowledge management. *IT Professional 19*, 6 (2017), 26–32.

[209] Zhao, P., Li, X., Xin, D., and Han, J. Graph cube: on warehousing and OLAP multidimensional networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011* (2011), pp. 853–864.

[210] Zhao, X., Chen, Y., Xiao, C., Ishikawa, Y., and Tang, J. Frequent subgraph mining based on pregel. *Comput. J. 59*, 8 (2016), 1113–1128.

[211] Zhu, F., Yan, X., Han, J., and Yu, P. S. gprune: A constraint pushing framework for graph pattern mining. In *Advances in Knowledge Discovery and Data Mining, 11th Pacific-Asia Conference, PAKDD 2007, Nanjing, China, May 22-25, 2007, Proceedings* (2007), pp. 388–400.

[212] Zou, R., and Holder, L. B. Frequent subgraph mining on a single large graph using sampling techniques. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG '10, Washington, D.C., USA, July 24-25, 2010* (2010), pp. 171–178.

[213] AllegroGraph. `http://franz.com/agraph/allegrograph/`. Accessed: 2016-03-10.

[214] APB-1 OLAP Benchmark. `http://www.olapcouncil.org/research/bmarkly.htm`. Accessed: 2018-05-23.

[215] Key Features - ArangoDB. https://www.arangodb.com/key-features/. Accessed: 2016-03-10.

[216] The bigdata RDF Database. https://www.blazegraph.com/whitepapers/bigdata_architecture_whitepaper.pdf. Whitepaper May 2013.

[217] Cypher Query Language. http://neo4j.com/docs/stable/cypher-query-lang.html. Accessed: 2016-03-16.

[218] DB-Engines Ranking of Graph DBMS. http://db-engines.com/en/ranking/graph+dbms. Accessed: 2018-05-28.

[219] DB-Engines Ranking of RDF Stores. https://db-engines.com/en/ranking/rdf+store. Accessed: 2018-05-28.

[220] Method of calculating the scores of the DB-Engines Ranking. https://db-engines.com/en/ranking_definition. Accessed: 2018-05-28.

[221] ERPNext, ERP system. www.erpnext.com. Accessed: 2014-01-13.

[222] FOAF Vocabulary Specification 0.99. http://xmlns.com/foaf/spec/. Namespace Document 2014-01-14.

[223] Gelly: Flink Graph API. https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/gelly.html. Accessed: 2016-03-15.

[224] Gephi - The Open Graph Viz Platform. https://gephi.org/. Accessed: 2018-04-19.

[225] Apache Giraph. http://www.giraph.apache.org. Accessed: 2016-03-10.

[226] GraphDB: At Last, the Meaningful Database. http://ontotext.com/documents/reports/PW_Ontotext.pdf. Whitepaper July 2014.

[227] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.

[228] InfiniteGraph: The Distributed Graph Database. http://www.objectivity.com/wp-content/uploads/Objectivity_WP_IG_Distr_Benchmark.pdf. Whitepaper 2012.

[229] ISO 8000-2:2017; Data quality – Part 2: Vocabulary. Standard, International Organization for Standardization, 2017.

[230] Apache Jena - TBD. https://jena.apache.org/documentation/tdb/. Accessed: 2016-03-09.

[231] MarkLogic Semantics. http://www.marklogic.com/resources/marklogic-semantics-datasheet/. Datasheet March 2016.

[232] The Neo4j Developer Manual. https://neo4j.com/docs/developer-manual/3.2/. Version 3.3.

[233] Tutorial: Import Data Into Neo4j . https://neo4j.com/developer/guide-importing-data-and-etl/. Accessed: 2018-04-19.

[234] Oracle Spatial and Graph: Advanced Data Management. http://www.oracle.com/technetwork/database/options/spatialandgraph/spatial-and-graph-wp-12c-1896143.pdf. Whitepaper September 2014.

[235] Big Data Spatial and Graph User's Guide and Reference. http://docs.oracle.com/cd/E69290_01/doc.44/e67958/toc.htm. Accessed: 2016-03-16.

[236] Why OrientDB? http://orientdb.com/why-orientdb/. Accessed: 2016-03-10.

[237] Relational to Graph Database in Ten Minutes Flat .

[238] Resource Description Framework (RDF): Concepts and Abstract Syntax. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. W3C Recommendation 2004-02-10.

[239] RDF 1.1 N-Quads. http://www.w3.org/TR/2014/REC-n-quads-20140225/. W3C Recommendation 2014-02-25.

[240] RDF Schema 1.1. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/. W3C Recommendation 2014-02-25.

[241] Stardog 4 - The Manual. http://docs.stardog.com/. Accessed: 2016-03-10.

[242] TinkerPop Compendium. http://tinkerpop.apache.org/docs/3.3.0/. Version: 3.3.0.

[243] TITAN: Distributed Graph Database. http://thinkaurelius.github.io/titan/. Accessed: 2016-03-10.

[244] TPC-H Decision Support Benchmark. http://www.tpc.org/tpch/. Accessed: 2014-03-21.