Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Query, Analysis, and Benchmarking Techniques for Evolving Property Graphs of Software Systems

Ph.D. Dissertation

Gábor Szárnyas

Thesis supervisor:
Dániel Varró, D.Sc.

Budapest
2019

Gábor Szárnyas
http://mit.bme.hu/~szarnyas/pub/szarnyasg-phd-dissertation.pdf

March 2019

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

H-1117 Budapest, Magyar tudósok körútja 2.

Declaration of own work and references

I, Gábor Szárnyas, hereby declare that this dissertation, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Szárnyas Gábor kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2019. 03. 24.

Szárnyas Gábor

Acknowledgements

During the long years of PhD school, I was lucky enough to have support from a great number of colleagues, collaborators, and friends.

Advisor First and foremost, I would like to express my gratitude towards my advisor Dániel Varró for his guidance and patience.

Colleagues I spent my PhD programme in the Fault-Tolerant Systems Research Group in the Department of Measurement and Information Systems. I am thankful to heads of department, Ákos Jobbágy and Tamás Dabóczi, along with the leaders of the research group: András Pataricza, István Majzik, and Zoltán Micskei. I would like to thank my colleagues, many of whom were important collaborators in my publications, István Ráth (co-advisor from 2014 to 2016), Gábor Bergmann, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, Szilvia Varró-Gyapay, Zoltán Szatmári, Gábor Huszerl, László Gönczy, Imre Kocsis, and András Vörös, along with Dénes Harmath (IncQuery Labs). I also had some extraordinary fellow PhD students, including Benedek Izsó, Ágnes Salánki, Oszkár Semeráth, Dániel Darvas, Csaba Debreceni, András Szabolcs Nagy, Tamás Tóth, Vince Molnár, Dávid Honfi, Márton Búr, Ákos Hajdu, Kristóf Marussy, Rebeka Farkas, and Attila Klenik. Finally, my work was greatly assisted by our enthusiastic system administrators, Áron Tóth and Dávid Cseh.

Students I was fortunate to work with many outstanding students. I learnt a lot from cooperating with them and listening to their – sometimes critical – questions, which often forced me to rethink our objectives and refine our understanding of the problem. These students include Dániel Stein, József Makai, János Maginecz, Zsolt Kővári, Soma Lucz, Dávid Szakállas, Bálint Hegyi, Márton Elekes, Tamás Nyíri, János Benjamin Antal, Petra Várhegyi, and Lehel Boér.

Open-source community I am grateful to the broader open-source community for creating a culture that fosters collaboration, with the overwhelming majority of developers responding to my issues in a timely and helpful manner. I am particularly grateful to the community of the Neo4j database. I would like to thank the engineers, researchers, and managers of Neo4j company, including Alex Averbuch, Rik van Bruggen, Alastair Green, Michael Hunger, Martin Junghanns Max Kießling, Tobias Lindaaker, Stefan Plantikow, Mats Rydberg, Petra Selmer, Andrés Taylor, and Hannes Voigt.

BME Database Laboratory I would like to thank the head of the Database Laboratory at the Department of Telecommunications and Media Informatics, Sándor Gajdos, for introducing me to the world of databases, along with members of the lab including Ákos Barabás, Levente Erős, Bence Golda, Ádám Lippai, and József Marton.

BME I would like to thank Ervin Tóth for inspiring me to start my PhD programme. I am grateful to the representatives of the PhD student union (including Rita Lippai, Lívia Hajas, László Lendvai, and Zoltán Tóth) and to Edit Halász for her efforts in organizing courses for PhD students.

LDBC I had a great time collaborating with member of the Linked Data Benchmark Council's Social Network Benchmark and Graphalytics task forces, including Peter Boncz, Orri Erling, Vlad Haprian, Alexandru Iosup, Marcus Paradies, Arnau Prat-Pérez, and Alexandru Uta.

TTC I would like to thank the organizers of the annual Transformation Tool Contests, especially Tassilo Horn, Louis Rose, Antonio García-Domínguez, Filip Křikava, and Georg Hinkel.

Research visits and collaborations I would like to acknowledge the support of institutions which hosted my research visits: the University of York, McGill University, and the University of Waterloo, along with my hosts: Dimitris Kolovos, Konstantinos Barmpis, Semih Salihoglu, Chathura Kankanamge, and Amine Mhedhbi. I would also like to thank all other researchers who provided guidance in interpreting their results or assisted using their software tools, including Milos Nikolic and Christoph Koch (DBToaster), and Muhammad Saleem for inviting me to collaborate on the analysis of RDF benchmarks.

Projects and grants I would like to express my gratitude for the support of the MONDO EU FP7 project (Scalable Modelling and Model Management on the Cloud, EU ICT-611125) [MON16], the MTA-BME Lendület Cyber-Physical Systems Research Group project (LP2015-12) [MTA18], and the ÚNKP-17-3-III New National Excellence Program of the Ministry of Human Capacities. I would also like to acknowledge the awards and generous support provided by the Schnell László Foundation, the Pro Progressio Foundation supported by Red Hat, Inc., and the ACM Student Research Competition's travel grant. Cloud computing resources for my research were provided by Digital Ocean, Inc., and by a Microsoft Azure for Research award.

Friends and family Last but definitely not least, I would like to acknowledge the continuous support of my friends and family.

Summary

The graph data model is an important tool for modelling complex, highly interconnected data sets as the human mind tends to model the world as objects (nodes) and their relationships (edges). Graphs are used extensively in computer science, including algorithms for compiling, routing, and scheduling. Still, data sets are rarely represented as a graph, mostly due to the dominance of the relational model in database management systems. In this work, we focus on two key graph representation formats:

(1) Graphs of object-oriented models provide strong metamodelling features and are used in the field of software engineering and model-driven engineering.
(2) Property graphs provide a concise and intuitive data model but lack advanced schema definition and metamodelling techniques.

Even though both formats define a typed, attributed graph data model, there has been surprisingly little convergence and exchange of ideas between these fields even on the conceptual level. One of the common challenges of these fields is efficient evaluation of *graph queries* that consist of pattern matching and traversal operations. In model-driven engineering, graph queries are used for *model validation*, i.e. repeatedly checking for *well-formedness constraints*, which allows developers to catch design flaws early and therefore improve their productivity. Graph queries are also used for running *model simulations* and maintaining *multiple viewpoints*. In databases, graph queries are used to extract information from highly interconnected data sets, e.g. for *financial fraud detection, analysing social networks,* and *serving personalized recommendations*. The use cases listed for both fields could benefit significantly from applying *incremental view maintenance* techniques to speed up the performance of repeated query evaluation. While model-driven engineering has a number of incremental query frameworks (such as Viatra Query and eMoflon), databases offer less support for such techniques.

In this dissertation, we investigate the applicability of incremental view maintenance on property graphs to improve the performance of repeated query evaluation. We show that most graph queries can be reduced to an extended nested relational algebra, which can be further transformed into a flat, incrementally maintainable format. To tackle scalability challenges that arise due to the space-time tradeoff of incremental queries, we propose a distributed architecture that allows adding multiple computing nodes to increase the resources available for query processing.

We assess the performance of the proposed approaches with the benchmarks specified in this dissertation. To this end and to assist the benchmarking efforts of others, we present two cross-technology macrobenchmarks for comparing the performance of graph processing workloads. The Train Benchmark, originating from model-driven engineering, defines a set of graph queries that repeatedly check the correctness of a railway model instance while it is continuously updated. The more database-oriented LDBC Social Network Benchmark's new Business Intelligence (BI) workload defines on OLAP-style aggregation-heavy global graph queries, which employ complex pattern matching and path traversal operations.

A key difficulty for benchmarks is to provide representative workloads. To ensure that the queries cover the difficulties of real-world queries, we define common challenges of graph queries w.r.t. expressivity and performance. To characterize the representativeness of synthetic graphs, we adapt recent findings from network science. These allow us to study the structure of graphs with edge-type information, which is often overlooked by traditional graph analytical techniques. We apply these findings to distinguish between real and synthetic graph models, and identify graph metrics that can serve as an input for generating realistic graph instances.

Összefoglaló

Gondolatainkban a minket körülvevő világot gyakran objektumokra (azaz csomópontokra) és azok kapcsolatára (azaz élekre) képezzük le. A gráf adatmodell emiatt fontos eszköz komplex, sok összeköttetéssel rendelkező adathalmazok modellezésére. A gráfokat kiterjedten használják a számítástudományban, beleértve a fordítóprogramokban, hálózati útvonalkeresőkben és az ütemezőkben használt algoritmusokat. Az *adattárolás* területén azonban még mindig a relációs adatmodell a domináns és ritkának számít, hogy egy adathalmazt gráf formátumban reprezentáljanak és tároljanak.

Értekezésemben két fő gráfreprezentációs formátumra koncentrálok: (1) Az objektum-orientált gráfmodellek erős metamodellezési képességekkel rendelkeznek és a szoftvertechnológia és a modellvezérelt szoftverfejlesztés területén elterjedtek. (2) A tulajdonsággráfok egy tömör és intuitív adatmodellt adnak, de nem nyújtanak lehetőséget séma és metamodell definiálására.

Bár mindkét formátum típusos, attribútumokkal ellátott gráf adatmodellt definiál, ezidáig meglepően kevés együttműködés történt a két terület művelői között. A területek egy fontos közös kihívása a *gráflekérdezések* hatékony kiértékelése, melyek gráfmintákból és gráfbejáró műveletekből állnak. A modellvezérelt tervezés területén ezen gráfminták egyik felhasználása a validáció, melynek során ún. *jólformáltsági kényszerek* ismételt ellenőrzésével biztosítják, hogy a modellben található hibák a fejlesztési folyamat során hamar kiderüljenek. A gráflekérdezések használatosak *modellszimulációra* és *többszörös nézeti pontok karbantartására* is. Az adatbázis-kezelés területén a gráflekérdezések lehetővé teszik, hogy információt nyerjünk ki sok összeköttetéssel rendelkező adathalmazokból, pl. *pénzügyi csalások detekciójára*, *közösségi hálózatok elemzésére* és *személyre szabott ajánlások kiszolgálására*. A két területen felsorolt alkalmazási esetek mindegyike profitálhatna *inkrementális nézetkarbantartási* technikák alkalmazásából, ezek ugyanis lehetővé tennék az ismételt lekérdezéskiértékelés teljesítményének növelését. Míg a modellvezérelt tervezés területén több inkrementális lekérdezőmotor is elérhető (pl. Viatra Query, eMoflon), a gráfadatbázisok kevésbé támogatják ezeket a technikákat.

Értekezésemben megvizsgálom az inkrementális nézetkarbantartás alkalmazhatóságát a tulajdonsággráf adatmodellre. Megmutatom, hogy a legtöbb gráflekérdezés lefordítható egy kiterjesztett beágyazott (többszintű) relációalgebra nyelvre, ami tovább transzformálható egy lapos (egyszintű), inkrementálisan karbantartható formára. Az inkrementális lekérdezésekhez alkalmazott tár–idő csere miatt felmerülő skálázhatósági kihívások megoldására egy olyan elosztott architektúrát javaslok, ami megengedi több számítási csomópont alkalmazását, ily módon növelve a lekérdezésfeldolgozásra elérhető erőforrások mennyiségét. Részletesen megvizsgálom továbbá a javasolt megközelítések teljesítményét, melyhez bemutatok két technológiafüggetlen benchmarkot (teljesítménymérési keretrendszert). Ezek lehetővé teszik adott gráflekérdező motorok különböző terhelési profilokon nyújtott teljesítményének összehasonlítását. A modellvezérelt tervezés területéről származó Train Benchmark keretrendszer olyan gráflekérdezéseket definiál, amik ismételt módon ellenőrzik folyamatosan módosított vasúti példánymodellek helyességét. Az inkább adatbázisokra fókuszáló LDBC Social Network Benchmark új „üzleti intelligencia” terhelési profilja sok aggregációt igénylő globális gráflekérdezéseket definiál, ami komplex mintaillesztési és útvonalkeresési műveleteket igényel.

A benchmarkokkal szemben támasztott egyik fő követelmény, hogy repezentatív terhelési profilt nyújtsanak. Annak érdekében, hogy a lekérdezések lefedjék a valós rendszerekben alkalmazott párjaik kihívásait, összegyűjtöttem a kifejezőerőre és teljesítményre vonatkozó gyakori kihívásokat. Ezen túl a szintetikusan generált gráfok reprezentativitásának biztosítására a hálózatkutatás eredményeit használtam. Ezekkel jobban jellemezhető az éltípusokat is tartalmazó gráfok struktúrája, amit a hagyományos gráf analitikai technikák gyakran figyelmen kívül hagynak. A szerzett ismereteket felhasználva bemutatom, hogyan lehetséges megkülönböztetni valós és szintetikus gráfokat, majd azonosítom azokat a metrikákat, amik bemenetként szolgálhatnak valósághű gráfok generálására.

Contents

1 Introduction **3**
1.1 Background . . . 3
1.1.1 A Brief History of Graph Data Processing . . . 3
1.1.2 Model-Driven Engineering . . . 4
1.1.3 Model Queries over Databases . . . 5
1.2 Benchmarks for Application Scenarios . . . 7
1.3 Challenges and Contributions . . . 8
1.3.1 Challenges . . . 9
1.3.2 Contributions . . . 9
1.4 Structure of the Dissertation . . . 10

2 Preliminaries **13**
2.1 Running Example: a Railway Model . . . 13
2.2 Conceptual Graph Data Models . . . 13
2.2.1 Untyped Graphs . . . 13
2.2.2 Graphs with Types and Labels . . . 15
2.2.3 Property Graphs . . . 15
2.2.4 Path Property Graphs . . . 16
2.3 Practical Graph Data Models . . . 16
2.3.1 Eclipse Modeling Framework (EMF) . . . 17
2.3.2 Property Graphs . . . 17
2.3.3 Semantic Graphs (RDF) . . . 17
2.3.4 Relational Model . . . 18
2.3.5 Comparison of Data Models . . . 19
2.3.6 Instance Models . . . 19
2.3.7 Graph Schema . . . 21
2.4 Basics of Relational Algebra . . . 22
2.4.1 Relations and Relational Schemas . . . 22
2.4.2 Representing Labelled Graphs as Relations . . . 23
2.4.3 Unary Operators . . . 24
2.4.4 Binary Operators . . . 25
2.5 Graph Pattern Matching and Traversal . . . 28

2.5.1 Pattern Matching . . . 28
2.5.2 Graph Traversals . . . 29
2.5.3 Graph Queries . . . 30
2.6 Graph Query Languages . . . 31
2.6.1 Cypher . . . 31
2.6.2 PGQL . . . 32
2.6.3 G-CORE . . . 32
2.6.4 SPARQL . . . 32
2.6.5 VIATRA Query Language . . . 32
2.7 Categorization of Graph Processing Workloads . . . 33
2.7.1 Graph Query Workloads . . . 33
2.7.2 Graph Analysis Workloads . . . 34
2.7.3 Summary of Graph Processing Workloads . . . 35

I Structural Analysis of Typed Graphs 37

3 Characterization of Typed Graphs 39
3.1 Introduction . . . 39
3.2 Running Example and Mapping to Edge-Typed Graphs . . . 40
3.3 Concepts for Characterizing Graphs . . . 41
3.4 Metrics for Untyped Graphs . . . 41
3.4.1 Basic Graph Metrics . . . 41
3.4.2 Clustering Metrics . . . 42
3.4.3 Summary of Metrics for Untyped Graphs . . . 44
3.5 Typed Metrics . . . 45
3.5.1 Metrics for Types . . . 45
3.5.2 Metrics for Nodes . . . 46
3.5.3 Metrics for Type-Node Pairs . . . 49
3.5.4 Metrics for Type Pairs . . . 49
3.6 Experimental Setup . . . 50
3.6.1 Domains and Instance Models . . . 50
3.6.2 Data Preparation . . . 50
3.7 Evaluation . . . 52
3.7.1 Which Metrics Are Characteristic? . . . 52
3.7.2 How Do Domains Differ? . . . 53
3.7.3 Statistical Methodology . . . 55
3.7.4 Threats to Validity . . . 56
3.7.5 Summary of Findings . . . 56
3.8 Optimization of Graph Metrics with Linear Algebra . . . 56
3.9 Related Work . . . 58
3.9.1 Graph Analytical Engines . . . 58
3.9.2 Benchmarks for Graph Analytics . . . 59
3.9.3 Metrics for Typed Graphs in Other Fields . . . 61
3.10 Conclusion and Future Work . . . 62

4 Characterizing the Realistic Nature of Graphs 65

4.1 Characterizing Randomized Graph Models . . . 65
4.1.1 Experimental Setup . . . 65
4.1.2 Evaluation . . . 66
4.2 Characterizing Graphs Synthesized by Solvers . . . 66
4.2.1 Experimental Setup . . . 67
4.2.2 Analysis of Generated Graph Models . . . 67
4.3 Related Work . . . 70
4.3.1 Generation of Graph Instances . . . 70
4.3.2 Domain-Agnostic Characterization of Realistic Graphs . . . 71
4.4 Conclusion and Future Work . . . 71

II Benchmarks for Global Queries over Evolving Property Graphs 73

5 The Train Benchmark 77
5.1 Introduction . . . 77
5.2 Query Technologies . . . 79
5.2.1 EMF Tools . . . 80
5.2.2 RDF Tools . . . 80
5.2.3 Property Graph Tools . . . 80
5.2.4 Relational Databases . . . 81
5.3 Benchmark Specification . . . 81
5.3.1 Inputs and Outputs . . . 81
5.3.2 Benchmark Phases . . . 81
5.3.3 Use Case Scenarios . . . 82
5.3.4 Specification of Queries . . . 83
5.3.5 Specification of Transformations . . . 85
5.3.6 Query and Transformations for Constraint RouteSensor . . . 85
5.3.7 Instance Model Generation and Fault Injection . . . 86
5.3.8 Ensuring Deterministic Results . . . 86
5.4 Evaluation . . . 87
5.4.1 Benchmark Setup . . . 87
5.4.2 Measurement of Execution Times . . . 88
5.4.3 How to Read the Charts? . . . 88
5.4.4 Measurement of Memory Consumption . . . 92
5.4.5 Analysis of Results . . . 93
5.4.6 Threats to Validity . . . 95
5.4.7 Summary . . . 96
5.4.8 Comparison to Related Benchmarks . . . 96
5.5 Conclusion . . . 97
5.6 Future Work . . . 98

6 The Business Intelligence Workload of the LDBC Social Network Benchmark 99
6.1 Introduction . . . 99
6.2 Benchmark Design . . . 100
6.2.1 Choke Point-Based Query Design . . . 100
6.2.2 Graph-Specific Choke Points . . . 101

6.2.3 Language Choke Points . . . 101
6.2.4 Benchmark Data Set . . . 102
6.3 Detailed Query Discussion . . . 104
6.3.1 Top Posters in a Country (Query 5) . . . 104
6.3.2 Experts in Social Circle (Query 16) . . . 105
6.3.3 Weighted Interaction Paths (Query 25) . . . 107
6.4 Benchmark Results . . . 108
6.5 Conclusion and Future Work . . . 109

7 Related Graph Benchmarks 111
7.1 Model Transformation and Graph Transformation Benchmarks . . . 111
7.1.1 Benchmarks for Graph and Model Transformation . . . 111
7.1.2 Tool Contests . . . 111
7.1.3 Assessment of Incremental Model Queries . . . 114
7.2 RDF Benchmarks . . . 114

III Incremental View Maintenance on Schema-Optional Property Graphs 117

8 Reducing Property Graph Queries to Relational Algebra for Incremental View Maintenance 119
8.1 Introduction . . . 119
8.2 Data Models . . . 123
8.2.1 The Property Graph Data Model . . . 123
8.2.2 Nested Relations . . . 123
8.3 Graph Relational Algebra . . . 124
8.3.1 The Get-Vertices Operator . . . 125
8.3.2 The Get-Edges Operator . . . 125
8.3.3 The Expand Operators . . . 125
8.3.4 Combining Pattern Matches . . . 126
8.3.5 Unwinding and Aggregation . . . 127
8.4 Transforming Graph RA to Flat RA . . . 128
8.4.1 Workflow Example . . . 130
8.4.2 From Graph RA to Nested RA . . . 130
8.4.3 From Nested RA to Flat RA . . . 130
8.5 View Maintenance on Flat RA . . . 132
8.5.1 Query Evaluation by Rete Network . . . 132
8.5.2 Cache Maintenance in the Rete Network . . . 135
8.5.3 Data Representation and Indexing . . . 135
8.5.4 Programming Model . . . 135
8.6 Evaluation . . . 135
8.6.1 Benchmark Setup . . . 136
8.6.2 Results and Analysis . . . 139
8.6.3 Threats to Validity . . . 139
8.7 Conclusion . . . 140

9 Distributed Incremental View Maintenance for Scalability 141

9.1 Running Example . . . 141
9.2 A Distributed Incremental Graph Query Framework . . . 142
9.2.1 Architecture . . . 142
9.2.2 The Rete Algorithm in a Distributed Environment . . . 144
9.3 Evaluation . . . 146
9.3.1 Benchmark Scenario . . . 146
9.3.2 Benchmark Architecture . . . 147
9.3.3 Results . . . 148
9.4 Conclusion . . . 150

10 Comparison to Related Graph Query and IVM Techniques 151
10.1 The Complex Structure of Graphs and Its Implications . . . 151
10.2 Worst-Case Optimal Join Algorithms for Subgraph Queries . . . 151
10.3 Incremental View Maintenance Techniques . . . 153
10.4 Graph Query Languages for IVM . . . 157
10.4.1 Cypher and openCypher . . . 158
10.4.2 G-CORE . . . 158
10.4.3 SPARQL . . . 159
10.4.4 VIATRA Query Language . . . 159
10.4.5 Other Approaches for Graph Querying . . . 160
10.5 Mapping Between Graph and Relational Queries . . . 160
10.5.1 Mapping from Graph Queries to Relational Queries . . . 161
10.5.2 Mapping from Relational Queries to Graph Queries . . . 162
10.6 Connecting Joins, Matrix Multiplications, and Graph Queries . . . 162

11 Conclusion and Future Work 165
11.1 Summary of Contributions . . . 165
11.1.1 Structural Analysis of Typed Graphs . . . 165
11.1.2 Benchmarks for Global Queries over Evolving Property Graphs . . . 166
11.1.3 Incremental View Maintenance on Schema-Optional Property Graphs . . . 167
11.2 Open Challenges and Future Work . . . 169

Publications 171

Bibliography 175

A Untyped Graph Metrics 211
A.1 Connectivity Metrics . . . 211
A.2 Shortest Path-Based Metrics . . . 212
A.3 Density Metrics . . . 213

B Techniques for Graph Analytics 215
B.1 Foundations of Linear Algebra . . . 215
B.1.1 Data Structures . . . 215
B.1.2 Matrix Operations . . . 215
B.1.3 Sparse Matrices . . . 217
B.2 Graph Metrics as Graph Queries . . . 217
B.3 Specialized Languages for Graph Analytics . . . 218

C Train Benchmark Queries **219**
C.1 ConnectedSegments . . . 219
C.2 PosLength . . . 220
C.3 RouteSensor . . . 220
C.4 SemaphoreNeighbor . . . 221
C.5 SwitchMonitored . . . 222
C.6 SwitchSet . . . 222

D Choke Points and Queries of the LDBC SNB's Business Intelligence Workload **225**
D.1 Choke Points . . . 225
D.1.1 Aggregation Performance . . . 225
D.1.2 Join Performance . . . 227
D.1.3 Data Access Locality . . . 227
D.1.4 Expression Calculation . . . 228
D.1.5 Correlated Sub-Queries . . . 228
D.1.6 Parallelism and Concurrency . . . 229
D.1.7 Graph-Specific Choke Points . . . 229
D.1.8 Language Choke Points . . . 229
D.2 Query Descriptions . . . 229

E Foundations of Incremental Query Evaluation **235**
E.1 Nullary Nodes . . . 235
E.2 Unary Nodes . . . 236
E.3 Binary Nodes . . . 236
E.3.1 Join Node . . . 237
E.3.2 Semijoin Node . . . 238
E.3.3 Antijoin Node . . . 238
E.3.4 Left Outer Join Node . . . 241
E.4 Incremental Performance for Query SemaphoreNeighbor . . . 243
E.4.1 Query Plans . . . 243
E.4.2 Execution Time and Memory Consumption . . . 243

Introduction and Preliminaries

Chapter 1

Introduction

1.1 Background

1.1.1 A Brief History of Graph Data Processing

As a branch of mathematics, graph theory has been in existence for almost three centuries [Die12]. While graphs were studied in the early 20th century [Kőn31; Kőn36], graph research was catalysed when a number of practical applications were discovered during the Second World War. Since the 1950s, the boom of information technology further boosted the development of graph theory and algorithms. The advancements in graph theory were also adopted by other domains, including social and natural sciences, which could use these techniques to argue about graph-shaped structures.

Starting from the 1960s, researchers studied a number of theoretical and practical aspects of graph processing and analysis. As a result, they established new fields by the 1970s, including *graph rewriting systems* [EPS73; Roz97] and *social network analysis* [Gra73; WF94]. During this time, the database management research community defined the graph-based *network data model*, published by the CODASYL (Conference on Data Systems Languages) Consortium [Dat70; Dat78]. However, it had little influence as it was soon superseded by the less implementation-oriented and more flexible *relational data model* [Cod70; DC74] that allowed system developers to decouple the query compiler from the query execution code and paved the way for using *high-level declarative query languages*.

As relational database managements systems (RDBMSs) were both novel and dominant at the same time, there was little activity in graph data management for a few years. However, starting from the mid-1980s, researchers started to design and implement a number of *object-oriented database managements systems* (OODBMSs). The 1990s saw a large number of theoretical works, research prototypes and commercial products in this space [AG08], which also exerted influence on RDBMSs, resulting in the implementation of object-relational systems. In the late 1990s, the semantic web initiative gave birth to the RDF (Resource Description Format) standard, with *semantic databases* following suit in the early 2000s. Around this time, researchers independently discovered that networks of both the World Wide Web [AJB99] and the underlying internet topology [FFF99] exhibit power-law distributions. These findings kickstarted the field of *network science* [BP16], which incorporates many of the previous results (such as those of social network analysis), but almost exclusively focuses on simple (directed or undirected) graphs.

Interestingly, none of the data models above achieved such a success in such a short amount of time as the *property graph* [HG16; Ang+18; Fra+18; Ang18], which extends *directed graphs* by only adding *labels/types* and *properties* (key-value pairs) to their nodes/edges. The first *property graph*

database system appeared in 2007, just two years before the term "NoSQL database" [SF12] was coined. While there are still significantly fewer property graph databases than other types of NoSQL systems (document databases, key-value stores, and wide column stores) [DCL18], graphs are increasingly used both for storing and processing highly interconnected data sets, and property graph databases have a dynamically growing market share.[1]

We believe that the success of the property graph data model can be attributed to the fact that it strikes a good balance between conceptual simplicity and expressiveness. As the human mind tends to interpret the world in terms of things (*nodes*) and their respective relationships to one another (*edges*) [Rod08b] along with attributes (*properties*) of these elements, property graphs provide a modelling framework that is intuitive but lightweight at the same time. Property graph instances are also better suited for visualization than other data models, such as RDF and hypergraphs [LP91]. Still, other graph data models, such as the *edge-typed graphs* are still relevant. Therefore, this dissertation discusses techniques and applications both for *graph databases* [Ang12; RWE15; Ang+17] storing property graphs and for *analytical frameworks* [Bat+15; Sak+16; Yan+17] working on typed graphs.

1.1.2 Model-Driven Engineering

Model-driven engineering (MDE) is a widely used development technique in many application domains such as automotive, avionics, and other cyber-physical systems [WHR14]. MDE facilitates the use of models in different phases of design and on various levels of abstraction. These models enable the automated synthesis of certain design artifacts (such as source code, configuration files, documentation) and help catch design flaws early by model validation techniques. MDE originates from the fields of *formal methods* and *software engineering* (strong influenced by object-oriented programming [Rum+91]), but established itself as a field of its own right in the last two decades. Many of the systems designed using MDE techniques have a long lifespan compared to most software products, e.g. a successful airliner is often produced and maintained for decades. Therefore, toolchains that build on *open-source software* (at least to some extent) are preferred to avoid vendor lock-in and aid the sustainability of the development tools.

In typical MDE tools, domain-specific models are defined as *typed, attributed graphs*. Therefore, processing and storing them requires techniques suited to efficient graph persistence and querying. A key component of MDE is *model validation*, i.e. checking whether a set of *well-formedness constraints* are satisfied on a domain-specific instance model. While RDBMSs also offer data validation in the form of *integrity constraints* or *check constraints* [RG98], these define simpler rules that enforce ranges or check for the presence of foreign key–primary key pairs. Model validation queries, on the other hand, often use *complex graph queries and traversals* for defining constraints, which are difficult to evaluate efficiently. Other workloads such as *model simulation* [RVV08] and *defining view points* [Bru+18] also rely on complex queries, often executed repeatedly.

As the *front-end components* of MDE software applications are mostly programmed in *statically typed object-oriented languages* (such as Java), they require the underlying storage and query layers to provide strong metamodelling features, including defining and enforcing a graph schema with a number of constraints such as containment hierarchy, cardinalities, etc. This is in stark contrast with graph and semantic databases, which offer a *schema-free* or *schema-optional* data model [BTL11], and provide only basic metamodelling features. In short, MDE applications demand the following features:

- **Metamodelling.** They require *strong metamodelling facilities* beyond the capabilities of currently available graph and semantic database systems, which are *schema-optional* at best.

[1] According to the statistics of the DB-Engines ranking site, the interest for graph databases between 2013 and 2019 has grown by 9.5 times (see https://db-engines.com/en/ranking_categories).

- **Complex queries.** They use *complex and global queries* for workloads such as model validaton or model simulation, consisting of numerous join operations and accessing a large fragment of the graph model. Some queries also rely on features outside relational algebra such as *transitive reachability* and determining *strongly connected components*.
- **Frequent reexecution.** Queries are repeatedly evaluated over an *evolving graph*, including the addition of new graph elements, updating attributes, and deleting existing elements.
- **Low response time.** Many of the common MDE workloads define operations where execution time is constrained by usability requirements. Model validation, model simulation, and maintaining multiple view points all necessitate efficient, low response time (preferably sub-second) query evaluation.

In the last two decades, a number of dedicated *graph and model transformation tools* [Kah+18] were created to tackle these challenges. A possible approach to mitigate the difficulty of complex graph queries is using *incremental view maintenance* (IVM) techniques, which aim to keep query results up-to-date efficiently upon changes in the graph (depicted in Fig. 1.1). These techniques have been used originally in *production systems* [For79] and *active databases* [PD99] that use *triggers* [Ast+76] to maintain *views* [Sto75] in a consistent state. In the context of model-driven engineering, IVM techniques have been employed by the Viatra tool [Var+16] (originally developed at the Fault-Tolerant Systems Research Group of the Budapest University of Technology and Economics), along with Reactive ATL [JT10], eMoflon [Lau+12], and NMF [Hin18a], However, a number of other features such as *scalability* [Kol+13], *fine-grained access control* [Deb+17], and *inconsistency tolerance* [GHM98] would be desirable to provide a usable persistence and query backend for MDE applications. Due to the lack of off-the-shelf solutions providing these features, MDE applications commonly use *custom model management layers* that implement these features to some degree.

$$\begin{tikzcd}
G \arrow[r, "+\Delta G"] \arrow[d, "Q"] & G + \Delta G \arrow[d, "Q\text{: full recomputation}"] \\
Q(G) \arrow[r, "+\Delta Q(G,\Delta G)\text{: incremental view maintenance}"] & Q(G + \Delta G)
\end{tikzcd}$$

Figure 1.1: Incremental view maintenance at a glance, adapted from [CY12]. Query Q is evaluated over graph G, returning $Q(G)$. Once changes ΔG are applied to the graph, it becomes $G + \Delta G$. The changes are subsequently reflected in the query results $Q(G + \Delta G)$, either through full recomputation or by applying $\Delta Q(G, \Delta G)$ to the previous query result.

1.1.3 Model Queries over Databases

At first sight, database systems seem a good fit for storing and processing the graph models of MDE applications. Due to their long history and sophisticated optimizations, databases are expected to be more scalable for complex queries than MDE engines. Additionally, they offer better *inconsistency tolerance*, i.e. they allow persisting incomplete and therefore malformed graph instances unlike many MDE tools. They also support multiple transactions to run read/write operations concurrently. As they are often used in enterprise environments for large collaborative projects involving multiple stakeholders, they provide *fine-grained access control*, *backup*, and *crash recovery* features. The software engineering and later the MDE communities therefore been long experimenting with adapting database technologies to their applications [EN79; ESW93; EKS93; Kin94; KD96; DTG00; Var08].

Relational database management systems There have been multiple attempts to create mapping layers between MDE tools and RDBMSs, the two most prominent being the *Eclipse Teneo* [Ecl15b] and *Connected Data Objects* (CDO) projects [Ecl19]. Still, relational databases have been repeatedly falling short in terms of performance and usability for supporting MDE workloads [BHH12; BK14; Góm+15; Sey+16; Hae+19]. The key reasons behind this are as follows:

- Using RDBMSs for storing graph models necessitates *object-relational mapping* (ORM) [Bla+06; ONe08] which suffers from the well-known *impedance mismatch* problem [CM84; Ire+09].
- Graph traversal queries are difficult to translate to SQL [VFV06b]. Even though support for recursive queries was introduced as an extension to the SQL:1999 standard [ISO99], it was only adopted by MySQL, the most popular open-source RDBMS, in 2017. Even with this language construct, recursive queries are often cumbersome to express and inefficient to evaluate.
- Recursive queries are also difficult to use from existing ORM frameworks – although they were proposed to the Hibernate framework's *Hibernate Query Language* (HQL) in paper [Szu+12], the actual implementation did not adopt them.
- Relational databases are notorious for having an excessive amount of configuration parameters to fine-tune performance [Ake+17]. While these can be used to improve performance for a given workload, getting them right for a certain workload requires a great deal of experience.
- The popular open-source implementations (MySQL and PostgreSQL) do not support incremental query evaluation, and even the proprietary ones (Oracle Database or Microsoft SQL Server) support it only to a limited extent, i.e. they cannot incrementally maintain views containing outer join operations [BHH12].

Object-oriented database management systems Following the rise in popularity of the object-oriented programming paradigm, *object-oriented database management systems* (OODBMSs or *object databases*), specializing in efficient retrieval of elements from object graphs, started to emerge in the 1980s. These systems were met with particular interest by the developers of *computer-aided design and manufacturing* (CAD and CAM) tools, which share many challenges of MDE workloads (such as using statically typed programming languages for front-end development), but aim at concrete application domains.

OODBMSs were initially successful, and by the mid-90s, more than 30 such systems were available [ZCC95], including commercial systems such as ObjectStore and O_2 [BDK92] along with research prototypes such as H-PCTE [Kel92] and GRAS [KSW95]. A new standardized query language, the *Object Query Language* (OQL) [CB00] was designed for OODBMSs, providing a syntax similar to SQL but expressing navigation operations in a *functional* nature. Despite the number of available systems, OODBMSs had limited success outside the narrow domains of CAD/CAM tools with only moderate adaptation to MDE and other tools. The reasons behind this are manifold, but we believe the key contributing factors were the following:

- **Complexity of data model.** The manifesto of OODBMSs [Atk+89] defined a large number of "mandatory" features, including *complex objects*, *object identity*, *encapsulation*, and *type hierarchies*, along with the inclusion of *operations* supporting overriding, overloading, and late binding. These requirements made the data model not only complex, but also resulted in rigid schemas, and rendered schema migration difficult.
- **Performance.** Despite attempts to standardize and formalize the data model and query languages of OODBMSs [FM95], the intricacies of the OO data model were less widely understood than those of the relatively simple relational data model. This led to a lack of efficient optimization methods.

- **Lack of open-source implementations.** For the last two decades [Ray99], developers and companies exhibited a shift in preference towards open-source solutions. Most OODBMSs did not follow this trend: even as of March 2019, only two of the top 10 object-oriented databases are available as open-source software[2], namely db4o and Perst, neither of which supports a declarative query language (only offering simple retrieval operations).

A hybrid approach was taken by *object-relational database management systems* (ORDBMS) [SM96], which used a relational backend but provided object-oriented query features. While this approach is supported in the SQL:1999 and was adopted by many RDBMSs (including PostgreSQL, IBM DB2, Microsoft SQL Server, and the Oracle Database), these solutions suffer from the same impedance mismatch problem [CM84; Ire+09] and performance issues that surfaced in the context of applying RDBMSs to MDE workloads. In short, neither RDBMSs nor OODBMSs turned out to be suitable backends for storing graph models. Therefore, following the appearance of the non-relational NoSQL and semantic databases, MDE researchers soon started to investigate them for storing and manipulating graphs.

NoSQL and semantic databases Since the realization that the requirements of MDE tools might be better met by NoSQL and semantic databases, the software engineering and MDE community has been experimenting to adapt database technologies to store object models [Rah+01; AK02; VFV06b]. In recent years, they investigated the applicability of NoSQL systems for tackling scalability challenges [Kol+13], including persistence layers (EMFStore [KH10], Morsa [ECM11], NeoEMF [Dan+17]), query evaluation (Mogwaï [DSC18]), and model indexing (Hawk [BK13]). The *Open Services for Lifecycle Collaboration* (OSLC) [BGL12], an open community started in 2008, aimed to harness semantic technologies to assist software engineering efforts. To this end, it provides a common approach for tool integration that build on top of the RDF standard [RS14], the Linked Data method [BHB09], and the REST (Representational State Transfer) software architecture style [BB08]. While many of these efforts were successful to some extent, the overall results and their adoption suggests that NoSQL systems are not yet fully capable of supporting complex MDE workloads, particularly struggling to provide sufficient performance for the graph queries required by such workloads.

The complexity of MDE workloads In short, to sufficiently handle complex MDE workloads, users require *continuously updatable graph database that supports strong metamodelling, can be queried with an expressive declarative query language, and can evaluate complex global queries with close to real-time response.* At the beginning of my research in 2014, such systems did not yet exists–even the concept of a "graph data warehouse" has only been explored in research works [Zha+11; LV13], and as of 2019, such systems are still not available yet.

1.2 Benchmarks for Application Scenarios

To study the challenges of handling complex graph query workloads and conduct research with impact on industry tools, we need to consider real-life use cases. This itself is a challenge as it is impossible to obtain (let alone publish) real workloads in model-driven engineering and graph data processing. The reason behind this is that both *real data sets* and even *queries specifications* constitute considerable *intellectual property*. Data sets might hold sensitive personal information (such as financial and medical data) or identify subcontractors (e.g. merely from the structure of an automotive model). In many cases, the queries are also difficult to obtain, especially in cases when exposing

[2] https://db-engines.com/en/ranking/object+oriented+dbms

them might be considered a risk (e.g. the queries used for financial fraud detection can give hints on how to circumvent such anti-fraud measures). The best practice to gather the challenges in a field is to use *commonly accepted benchmark specifications*. Such benchmarks have a number of additional advantages: they make *competing products and approaches* comparable, thus stimulate research and accelerate technical progress in their field [Pat12], as the Transaction Processing Performance Council's TPC benchmarks [TPC10; TPC17; TPC18] have done for the RDBMS industry during the past 30+ years.

Micro- and macrobenchmarks *Microbenchmarks* focus on the performance of small operators such as querying a single node type in a graph [LBV18]. This might reveal important information on low-level implementation details such as caching and the effect of warmup on certain operations, however, the requirements of application developers are often better served by *macrobenchmarks*, which consist of complex operations and are more similar to a real application. While there are a number of benchmarks available for graph workloads [VSV05; Zün08; Sch+09; Arm+13], none of them provides comparative measurements for complex graph queries on realistic data. At the beginning of my research in 2014, macrobenchmarks targeting such workloads – the Train Benchmark and the LDBC SNB's Business Intelligence workload – were in an early stage of research. This dissertation makes major contributions to both benchmarks.

Designing representative workloads On the conceptual level, a benchmark specification defines (1) data sets, e.g. graph instances of increasing sizes, (2) a set of queries, and (3) a scenario that prescribes the operations to be performed. A key requirement for benchmarks is that they should be *representative* [Gra93], i.e. their data sets, queries, and scenarios should resemble real use cases that are relevant to the interest of benchmark users. In this dissertation, we aim to assist benchmark designers in this goal by proposing a set of *abstract characteristics for characterizing a given workload.*

As discussed previously, users often cannot share the queries used in their workloads. However, they might be able to answer questions such as *"What is the maximum number of joins used in a query?"*, *"What percentage of the database is accessed by a query?"* and *"Which types of aggregation operations are the bottleneck in the queries?"*. These inputs can be used to construct similar queries.

It would be logical to apply the same approach for creating representative data sets. Existing graph benchmarks often use random graph models [Tae+07] that exhibit a highly regular structure or graphs that correspond to a relational data set [BS09]. Even the most recent and advanced approaches such as gMark [Bag+17] only consider a narrow set of characteristics (e.g. degree distributions can be controlled for a given edge type, but the interplay between types cannot be tuned). While there would definitely be interest for realistic graph generators [SLO18], both *characterizing real graphs* and *synthesizing realistic graphs* are highly non-trivial problems. In this dissertation, we adapt recent results of *the multidisciplinary field of network science* to obtain metrics that describe the structure of graphs. Synthesizing realistic graphs is an open research challenge and state-of-the-art approaches offer limited scalability, only supporting graphs up to a few hundred elements [You+18].

1.3 Challenges and Contributions

As we concluded in Sec. 1.1.3, MDE tools could greatly benefit from using high-performance graph database systems, especially ones supporting *incremental view maintenance* over property graphs or semantic graphs. Other users also expressed their interest for running continuous queries over an evolving graph data set. While such systems were already proposed in research works in the

1990s [KSW95], no graph database system offered incremental query evaluation at the beginning of my research in 2014. This still holds true as shown by recent survey [Sah+17], which interviewed users from industry and academia about their graph data managements interest and practices. In fact, this survey confirmed both *the demand for incremental graph processing techniques* (more than a third of respondents indicated that they perform incremental computations on their graphs) and *the lack of such systems* (the 22 software products included in the survey have limited or no support for incremental computations), suggesting that users who indicated that they rely on IVM use either computations with limited incrementality and/or implemented problem-specific ad-hoc solutions.

1.3.1 Challenges

To create the building blocks of an incremental graph query engine that work both in the MDE and property graph domains, we investigated the challenges of *scalable incremental view maintenance for graph queries*. To derive representative performance results for such systems, we looked for *representative macrobenchmarks to measure the performance of global graph queries*. Finally, we aimed to *characterize realistic graph models*.

Ch1. **Queries over evolving property graphs.** A common approach to speed up the evaluation of queries on continuously changing data sets is *incremental view maintenance* (IVM), which defines a view for each query and maintains its results upon changes. While IVM techniques have been developed for more than three decades, the feasibility of incremental operations on graph data structures is still actively studied from the theoretical perspective [FHT17]. Additionally, the sophisticated *property graph data model* introduces even more challenges for IVM techniques.

Ch2. **Representative macrobenchmarks for graph querying.** To capture challenging aspects of real workloads, we need some *abstract aggregative descriptions* to provide a summary of why they are difficult. This characterization is important for two key reasons: (1) it allows benchmark designers to define *synthetic but representative workloads* without requiring access to confidential information such as queries and data sets (both protected by intellectual property rights and non-disclosure agreements) and (2) it makes possible to combine different workloads into a single one. These abstract descriptions include ones that target the data set, such as *typed graph metrics*, and also ones that define query language features or stress performance aspects.

Ch3. **Domain-specific characterization of realistic graph models.** Creating realistic graph instances for benchmarks and distinguishing synthetic graphs from real ones requires an approach to characterize graph models in a certain domain. The field of *network science* has studied a number of graph metrics (such as *degree distribution*) to characterize real networks, however, devoted less attention to metrics that characterize graphs with type information, which is necessary to describe domain-specific graph models.

1.3.2 Contributions

My research contributions presented in this dissertation are as follows:

Co1. **Structural analysis of typed graphs:** I proposed various graph metrics and statistical analysis techniques to characterize domain-specific engineering models. I used selected metrics to distinguish between real models and synthetic ones produced by automated generators.

Co2. **Benchmarks for global queries over evolving property graphs:** I contributed to the design of two benchmark frameworks for global property graph queries: the Train Benchmark that

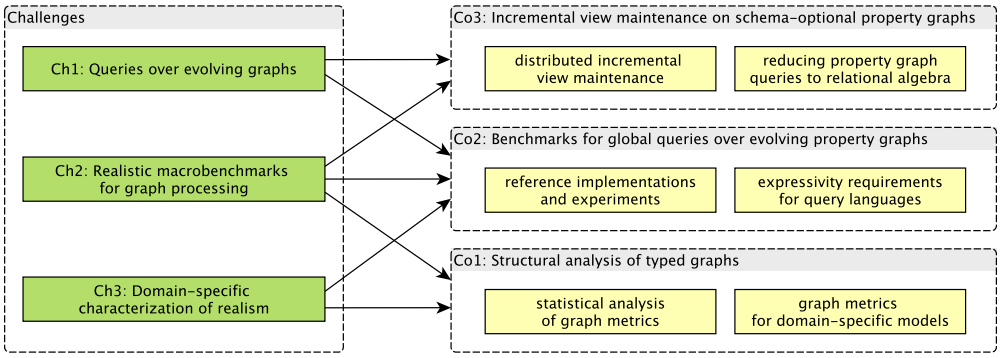


Figure 1.2: A graph of the key challenges and their relations to the contribution groups. Note that the challenges on the left are numbered as Ch1 to Ch3 from top to bottom, while the contributions on the right are numbered as Co1 to Co3 from bottom to top.

defines a continuous model validation workload and the LDBC SNB's Business Intelligence workload that consists of complex aggregation-heavy global graph queries.

Co3. **Incremental view maintenance on schema-optional property graphs:** I developed scalable incremental view maintenance techniques for evolving schema-optional graphs. I designed and implemented two prototypes, IncQuery-D for distributed incremental view maintenance over RDF (semantic) graphs and *ingraph* for incremental evaluation of property graph queries.

Fig. 1.2 lays out the challenges and connects them to the proposed contributions. It shows that the central themes of this dissertation are *global graph queries*, *macrobenchmarks*, and *realistic workloads*, with benchmarks closely related to each of the three contribution groups. Fig. 1.3 shows an overview of the groups of contributions, including their key results and prototype tools, positioning them w.r.t. data processing system types (such as *model query engines* and *relational databases*), industry tools, and workload categories.

1.4 Structure of the Dissertation

The dissertation is structured as follows.

Introduction and Preliminaries Chapter 2 defines the basic concepts and graph data models used in this dissertation through the example of the Train Benchmark. It also presents the basic operators of relational algebra and gives an overview of different types of graph processing.

Part I: Structural analysis of typed graphs In the first group of contributions, we apply recent findings in network science on graph models to characterize their internal structure in Chapter 3. We place a particular emphasis on investigating the interplay between edge types of the graph model, an aspect which is often overlooked by traditional tools of network science. Then, we use a subset of these metrics to distinguish between real vs. synthetic graphs in Chapter 4. The main chapters of this contribution group are complemented by an overview of metrics for untyped graph in Appendix A and a summary of graph computation techniques in Appendix B.

Part II: Benchmarks for global queries over evolving property graphs In the second group of contributions, we present two benchmark frameworks that allow their users to run experiments for

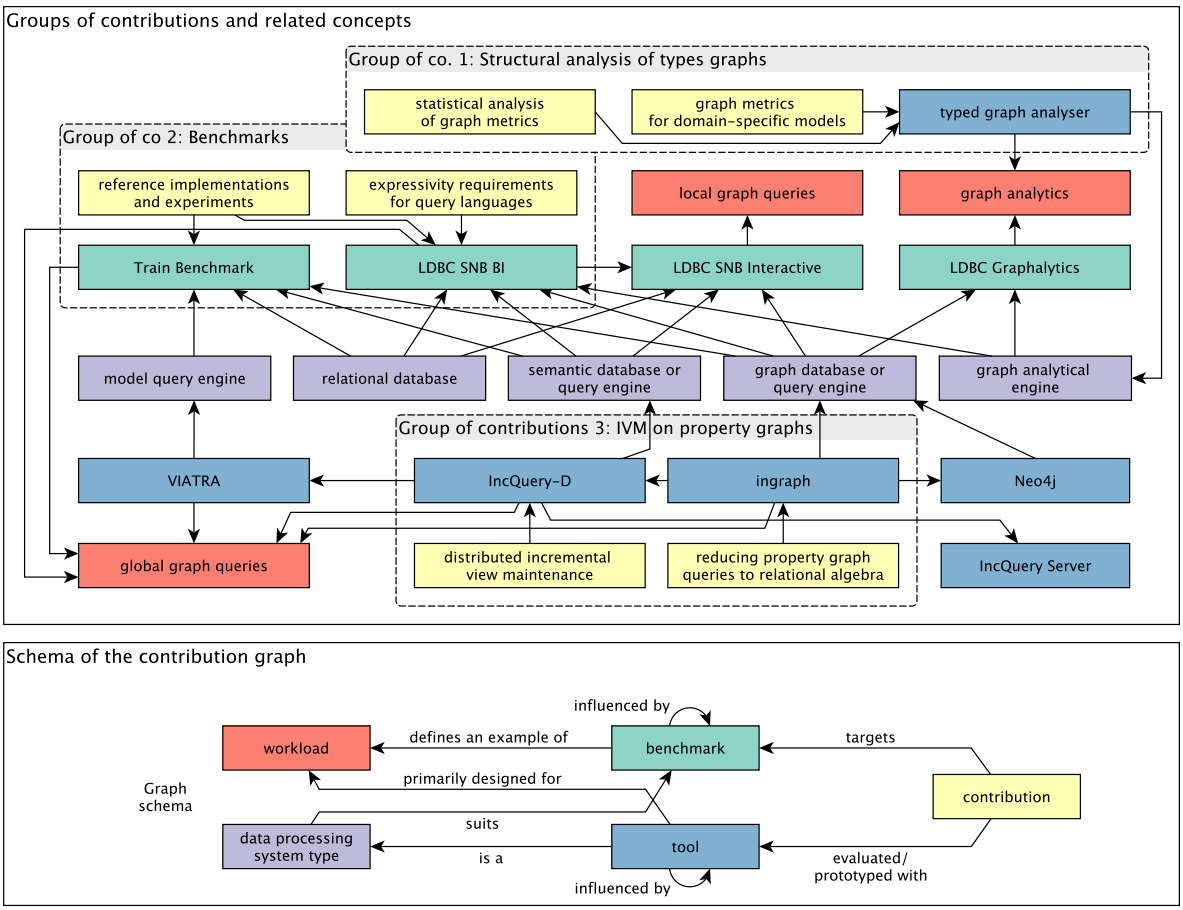


Figure 1.3: A graph of the key contributions and prototype systems presented in this dissertation, along with their relations to other systems and concepts. Contribution groups 1–3 are described in Parts I–III. The LDBC Social Network Benchmark Interactive workload and the LDBC Graphalytics benchmark are described in [Erl+15] and [Ios+16], respectively.

assessing the performance of global graph queries. Chapter 5 presents the Train Benchmark, then Chapter 6 discusses the LDBC Social Network Benchmark's new Business Intelligence workload (LDBC SNB BI). We review related benchmarks in Chapter 7. The appendices contain the detailed specifications of the benchmark workloads: Appendix C defines the queries and transformations of Train Benchmark, while Appendix D presents the choke points and queries for the LDBC SNB BI.

Part III: Incremental view maintenance on schema-optional property graphs The third and final group of contributions tackles the challenges of *incremental view maintenance on schema-optional graph data models* such as property graphs and semantic (RDF) graphs. In Chapter 8, we present an approach to reduce property graph queries to relational algebra and evaluate the results using the LDBC Social Network Benchmark's BI workload. Then, in Chapter 9, we show how to perform incremental view maintenance in a distributed setup and use the Train Benchmark to assess the scalability of the proposed approach. We discuss related view maintenance techniques in Chapter 10. As supplementary material, Appendix E presents the foundations of incremental query evaluation in Rete networks, including detailed derivations of delta expressions.

Conclusion and Future Work To conclude the dissertation, Chapter 11 summarizes the results and highlights potential future research directions.

Chapter 2

Preliminaries

In this chapter, we define the concepts used throughout this dissertation. The majority of examples, definitions, and terms presented here are involved in more than one contribution group (Parts I–III).

2.1 Running Example: a Railway Model

We use an example from the domain of the railway network design, defined by the Train Benchmark framework [j2] (Chapter 5). Fig. 2.1 shows an example: Fig. 2.1a illustrates the domain with a (partial) network, and Fig. 2.1b shows its metamodel.

A train Route is a logical path of the railway, which requires a set of Sensors for safe operation. The occupancy of Track Elements (Segments and Switches) is monitored by sensors. A route follows certain Switch positions (straight or diverging) which define the *prescribed* position of a switch belonging to the route. Different routes can specify different positions for the same switch. A route is active if all its switches are in the position prescribed by the switch positions followed by the route. Each route has a Semaphore on its entry and exit points.

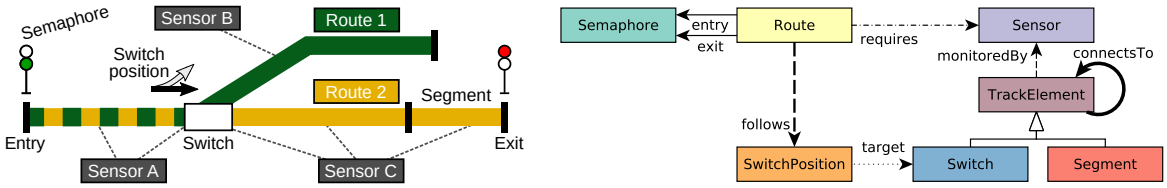
2.2 Conceptual Graph Data Models

In this section, we give a brief overview of related concepts in graph theory. In this dissertation, we only consider *directed graphs* in the data model ($G = (V, E, \ldots)$) and capture the undirected semantics of certain edges in the *graph queries* (see Sec. 2.5.3). This approach makes the data model conceptually simpler and is used commonly in practice, e.g. in graph databases [Web12].

2.2.1 Untyped Graphs

We define graph data models of increasing expressive power. We start with the simplest one, *untyped graphs*, which hold no type information. Formally:

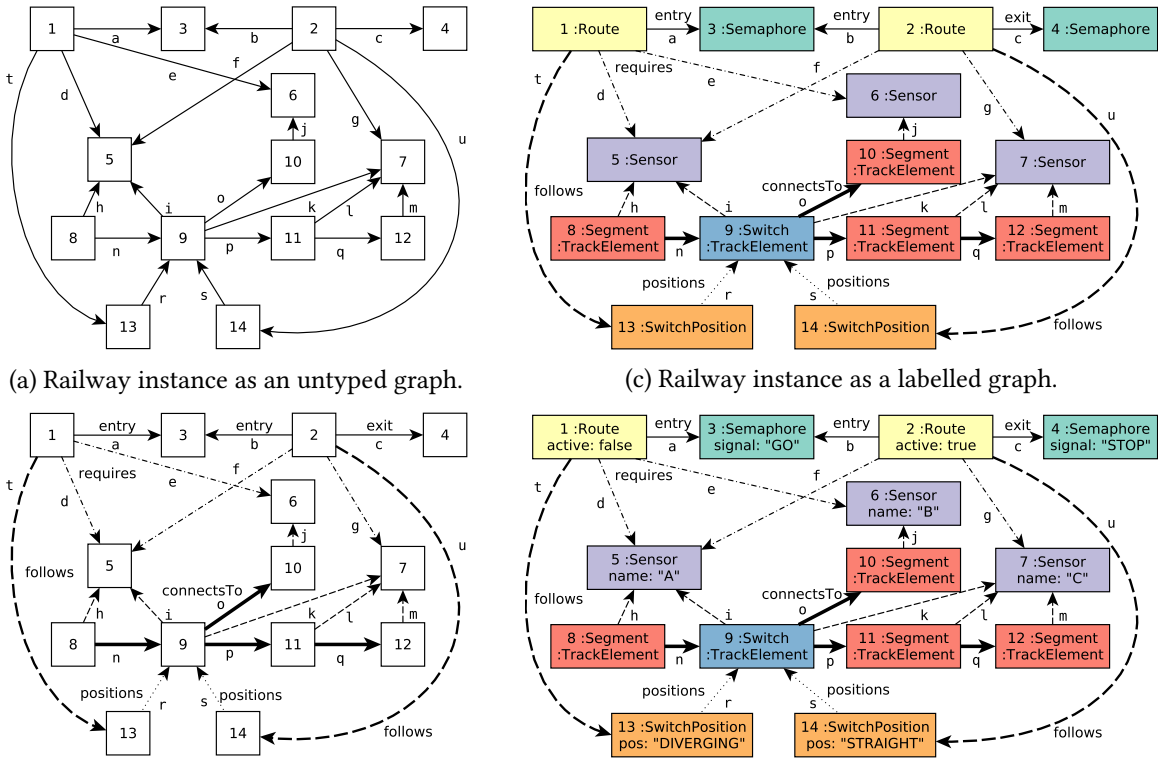
> **Definition 1 (untyped graph or homogeneous graph)** An *untyped graph* is defined as
>
> $$G = (V, E, \mathit{src}, \mathit{trg}),$$
>
> where V is a set of *nodes* and E is a set of *edges*. Functions $\mathit{src}, \mathit{trg} : E \to V$ are *total functions*, respectively assigning the *source* and *target* node to each edge.



(a) Illustration of the domain concepts.

(b) Metamodel of the railway graph.

Figure 2.1: Example railway network and its metamodel.



(a) Railway instance as an untyped graph.

(c) Railway instance as a labelled graph.

(b) Railway instance as an edge-typed graph.

(d) Railway instance as a property graph.

Figure 2.2: Example railway network represented with data models of decreasing expressivity: as an untyped graph (Def. 1) as an edge-typed graph (Def. 4), as a labelled graph (Def. 6), and as a property graph (Def. 7).

Fig. 2.2a shows the example graph as an untyped graph.

Definition 2 (graph elements) We refer to the union of *nodes* and *edges* as *graph elements*.

Many practical applications, e.g. logistics and routing, operate on graphs where cost of traversing a particular edge is of interest. To represent these, we can assign *weights* to graph edges.

Definition 3 (weighted graph) A *weighted graph* is a graph where each edge is assigned a *weight* $w \in \mathbb{R}$.

Note on terminology. Due to their applications in many scientific fields, the terminology of graphs

uses different name for the same (or closely related similar) concepts. For example, nodes are often called *vertices*, and edges are often called *relationships* or *arcs*. Also the terms *graph* and *network* are used interchangeably in many works, including this dissertation. Additionally, we use the term *model* as a shorthand for *graph model* and note explicitly when we refer to other kinds of models (e.g. *textual models*).
Compared to the *untyped graph*, there are numerous graph data models offering greater expressive power, to be introduced later in this section. To distinguish these from the untyped graph model, the latter is informally called the *textbook graph* model, referring to the fact that most university textbooks on graph and algorithm theory only consider untyped graphs.

2.2.2 Graphs with Types and Labels

Untyped graphs allow users to capture *homogeneous* real-life networks. However, such networks usually do not exist in isolation, instead they are tightly connected with other networks, and emerge through the interplay between edges of different types. To capture some of the heterogeneity in real networks, we first assign a *type* to each edge. Formally:

Definition 4 (edge-typed graph or multiplex graph) An *edge-typed graph* is defined as

$$G = (V, E, \mathit{src}, \mathit{trg}, T, \mathit{type}),$$

where T is a set of edge types, and function $\mathit{type} : E \to T$ assigns a *single type* to each edge.

Fig. 2.2b shows the example graph as an edge-typed graph. This data model allows us to use the tools of network science for multiplex graphs, which often reveal more insights compared to traditional graph analysis on untyped graphs, while still "abstracting away" some information (the labels of nodes and properties of graph elements).

In the *node-labelled graph* data model, we assign a *set of labels* to each node. Formally:

Definition 5 (node-labelled graph) A *node-labelled graph* is defined as

$$G = (V, E, \mathit{src}, \mathit{trg}, L, \mathit{labels}),$$

where L is a set of node labels, and function $\mathit{labels} : V \to 2^L$ assigns a *set of labels* to each node.

The *labelled graph* data model uses both edge types and node labels:

Definition 6 (labelled graph) A *labelled graph* is defined as

$$G = (V, E, \mathit{src}, \mathit{trg}, L, T, \mathit{labels}, \mathit{type})$$

Fig. 2.2c shows the example graph as a labelled graph. This example highlights that the labelled graph model offers quite rich modelling capabilities.

2.2.3 Property Graphs

To capture the *properties* in the graph, let S be a set of scalar literals, $\mathrm{FBAG}(S)$ denote the set of all finite bags of elements from S, and let $D = S \cup \mathrm{FBAG}(S)$ be the value domain for the PG.

Definition 7 (property graph) A *property graph* is defined as

$$G = (V, E, \mathit{src}, \mathit{trg}, L, T, \mathit{labels}, \mathit{type}, P_v, P_e)$$

Additionally to the concepts already presented, P_v and P_e are defined as:

- P_v is the set of node properties. $p \in P_v$ is a function $p : V \to D$, which assigns a property value $d \in D$ to node $v \in V$, if v has property p, otherwise returns NULL.
- P_e is the set of edge properties. $p \in P_e$ is a function $p : E \to D$, which assigns a property value $d \in D$ to an edge $e \in E$, if e has property p, otherwise returns NULL.

Fig. 2.2d shows the example graph as a property graph. It is easy to see that this instance holds the most information and is the most faithful representation of the example railway network of Fig. 2.1a.

2.2.4 Path Property Graphs

The *path property graph* data model is an extension of *property graphs* (Def. 7) introduced by the G-CORE language (Sec. 2.6.3). The goal of the language is to treat *paths* (Sec. 2.5.2) as first-class citizens by introducing an explicit *set of paths* in the data model, with each path having its own set of labels and properties. As the *path property graph* data model is not used in this work, we refrain from providing a formal definition and refer the reader to paper [Ang+18]. However, we expect this data model to gain more significance in the near future through the influence of the G-CORE language.

Note on terminology. Works in network science commonly refer to concepts identical to *edge-typed graphs* as multidimensional networks [Ber+13], multilayer(ed) networks [BSK11; Bró+12; Kiv+14], and multiplex networks [BNL14; NL15]. Additionally, database researchers use the term *edge-labelled graph* [Ang+17] as well. Conversely, *labelled graphs* are also called typed graphs [Lin+16], heterogeneous networks [HB15], and heterogeneous information networks [Shi+17]. For *property graphs*, the closest related concept is that of *attributed graphs* [SWZ99], which predate property graphs by more than a decade [KG89].

2.3 Practical Graph Data Models

In this work, we consider four substantially different data models, each with different metamodelling and model representation support. To discuss these models, we use the model of the Train Benchmark (Chapter 5) to illustrate how these technologies define the metamodel (e.g. the supertype hierarchy) and how they store the instance models as graphs.

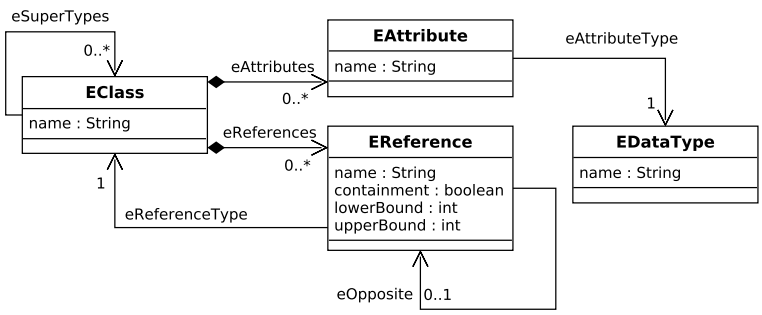


Figure 2.3: The kernel of the Ecore metamodel.

2.3.1 Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework provides Ecore, one of the *de facto* standard industrial metamodelling environments, used for defining several domain-specific languages and editors. Ecore enables developers to define metamodels and automates the generation of a wide range of tools. Ecore is discussed in detail in [Ste+09], here we only present its core kernel in Fig. 2.3.

- EClass intances represent classes. EClasses are identified by their name and may have several attributes and references. To support inheritance, a class can refer to a number of *supertype* classes (eSuperTypes).
- EAttribute instances represent attributes which contain data elements of a class. They are identified by their name and have a *data type* (eAttributeType).
- EDataType represents simple data types that are treated as atomic (their internal structure is not modelled) and identified by their name.
- EReference represents a unidirectional edge between EClasses and it is identified by its name. Lower and upper multiplicities (lowerBound and upperBound) can be specified. It is also possible to mark a reference as a containment that represents composition of model elements. Bidirectional associations can be modelled as two EReference instances that are mutually connected via their *opposite* references (eOpposite).

The EMF metamodel of the example railway graph is shown in Fig. 2.4. Compared to the simple metamodel of Fig. 2.1b, it enriches the schema with numerous additional features, such as cardinality constraints, containment relations, class hierarchies, and opposite edges.

2.3.2 Property Graphs

Variants of the property graph data model (Def. 7) are common in *graph databases* such as Neo4j[1], OrientDB[2], and JanusGraph[3] (originally introduced as Titan[4]). Currently, most graph database systems support no or optional metamodelling features. However, schema description languages were proposed (such as gTop, see Sec. 10.5.1) and they are under active development.

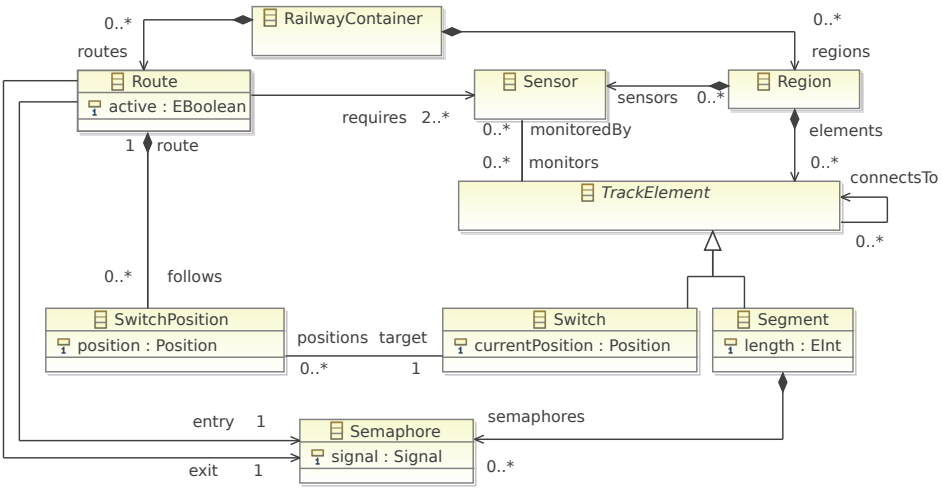
2.3.3 Semantic Graphs (RDF)

The Resource Description Framework (RDF) [RS14; Wyl+18] is a family of W3C (World Wide Web Consortium) specifications originally designed as a *metadata data model* for *semantic web* applications. In the following, we use the terms RDF and *semantic graph* interchangeably.

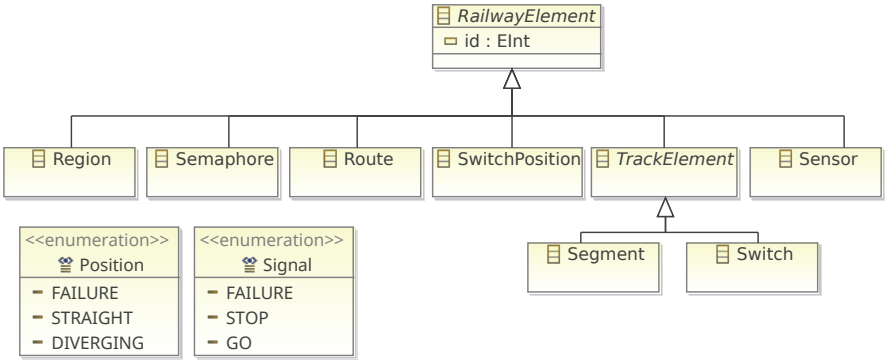
The RDF data model makes statements about *resources* (nodes/objects) in the form of triples. A *triple* is composed of a *subject*, a *predicate* and an *object*, e.g. "John is-type-of Person", "John has-an-age-of 34".[5] Both the *ontology* (metamodel) and the *facts* (instance model) are represented as triples and stored together in the *knowledge base*.

The knowledge base is typically persisted in specialized databases tailored to store and process triples efficiently, called *triplestores*. Some triplestores are capable of *reasoning*, i.e. inferring logical

[1] https://neo4j.com/
[2] https://orientdb.com/
[3] http://janusgraph.org/
[4] http://titan.thinkaurelius.com/
[5] *Predicates* are sometimes called *properties* [Cho+05; Özs16], which should not be confused with the concept of the same name in *property graphs* (Sec. 2.3.2). Predicates/properties in RDF are more generic than those in property graphs as they also represent edges between nodes (and not only attributes).



(a) Containment hierarchy and references.



(b) Supertype relations.

Figure 2.4: The metamodel of the Train Benchmark.

consequences from a set of facts or axioms. RDF supports knowledge representation languages with different expressive power, ranging from RDFS [GB14b] to OWL 2 [Gro12].

It is non-trivial to capture semantic graphs with the theoretical models presented in Sec. 2.2. Hence, semantic graphs can be modelled as *labelled graphs* (Def. 4), where nodes are the resources and literals in the semantic graphs and each edge type is a URI.

2.3.4 Relational Model

Relational database management systems (commonly abbreviated as RDBMSs) have been dominating the database landscape for more than 40 years. Building on the formal foundations of the *relational data model* and the accessibility of the SQL language, RDBMSs are still the most widely used database systems, with many free and commercial products. Due to their long history, these systems are stable and mature with sophisticated tools for operational tasks such as access control and data import.

Database schema For a given database, the sum of the table schemas – each defining attribute names and types along with primary and foreign key relations (PK/FK), and integrity constraints –

makes up the *database schema*. While the term *metamodel* is not commonly associated with relational databases, the database schema fulfils the same role as the metamodel.

Mapping object-oriented concepts Many object-oriented concepts such as class hierarchies are difficult to map to the relational data model. In fact, the *object-to-relational mapping* (ORM) is a well-known challenge in software engineering [Bla+06; ONe08]. We discuss ORM approaches in more detail in Sec. 10.5.

2.3.5 Comparison of Data Models

OO (UML)	OO (EMF)	Property graph	Semantic graph (RDF)	Relational (SQL)
class definition	EClass instance	node label	rdfs:Class	table definition
reference definition	EReference instance	edge label	owl:ObjectProperty	FK constraint
attribute definition	EAttribute instance	property name	owl:DataTypeProperty	column definition
type	EDataType instance	(only primitives)	rdfs:Datatype	(only primitives)
class attributes	eAttributes ref.	–	rdfs:domain	table columns
class reference	eReferences ref.	–	rdfs:domain	FKs
attribute type	eAttributeType ref.	property type	rdfs:range	column type
reference type	eReferenceType ref.	–	rdfs:range	–
superclasses	eSuperTypes ref.	–	rdfs:subClassOf	(various mappings)
composition	containment flag	–	–	–
object	EObject instance	node	resource	table row
concrete reference	object ref.	edge	triple to a resource	FK instances
attribute value	attribute value	property value	triple to a literal	attribute value
MOF model	Ecore model	graph schema ⊛	ontology ⊛	relational schema

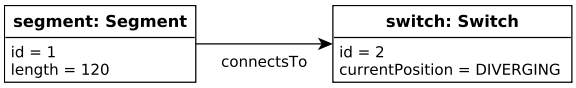
Table 2.1: Mapping concepts between data models. Notation – *OO:* object-oriented, *ref.:* reference, *FK:* foreign key, ⊛: the data model is schema-optional.

Tab. 2.1 defines the mapping from object-oriented concepts to the various metamodelling frameworks used in the Train Benchmark. The table shows that the *object-oriented* and *SQL* data models require an explicit schema, while other formats make this optional. As one of the common defining features of the *semantic* and *property graph* data models, we use the term *schema-optional graph* to denote these models [BTL11].

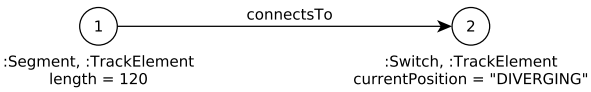
2.3.6 Instance Models

For each data model, we present a *small example instance model*, consisting of a Segment (id: 1, length: 120), a Switch (id: 2, currentPosition: "DIVERGING"), and a connectsTo edge from the segment to the switch. The instance models are shown in Fig. 2.5.

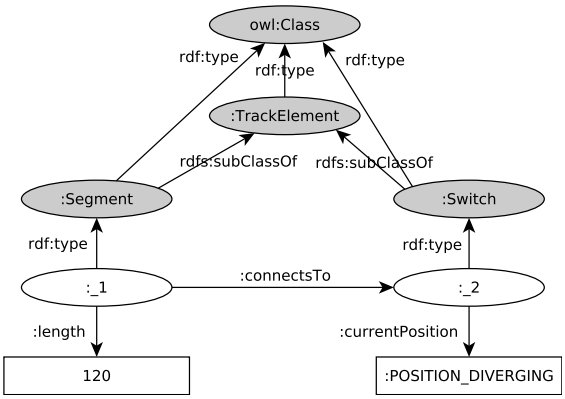
EMF An EMF instance model is shown in Fig. 2.5a. By default, EMF does not use numeric unique identifiers, instead (1) it uses references for the in-memory representation, and (2) it relies on XPath expressions for serialized models. However, developers may mark an attribute as an identifier. In the EMF metamodel of the Train Benchmark, we defined every class as a subtype of class RailwayElement which has an explicit id attribute, serving as a unique numeric identifier.



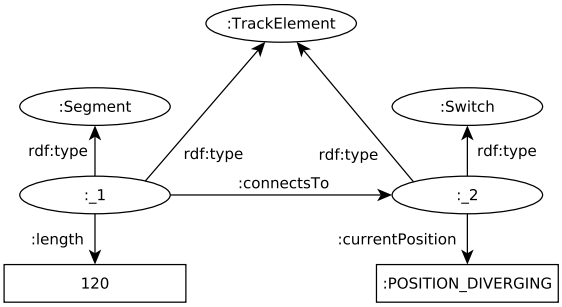
(a) An EMF instance model.



(b) A property graph instance model.



(c) An RDF instance model *with meta-level resources inserted in the model.* The nodes for the (relevant part of the) metamodel are depicted in grey.



(d) An RDF instance model *with inferred triples.* Note that the direct rdf:type edges to TrackElement node are explicitly inserted in the model.

Figure 2.5: Instance models in EMF, property graph and RDF data models.

Property Graph The property graph instance model is shown in Fig. 2.5b. The nodes are typed with *labels*, e.g. node 1 is labelled as both Segment and TrackElement, while node 2 is labelled as both Switch and TrackElement.

Semantic Graphs (RDF) As discussed in Sec. 2.3.3, semantic graphs are typically stored in triple-stores, which might provide *reasoning* features that allow them to feature type information at the cost of reduced query performance. Based on the reasoning capabilities of the semantic engine and considering performance requirements, database designers need to decide whether to store the metamodel or inferred (redundant) edges in the database. The two approaches work as follows:

- *Storing the metamodel.* The metamodel (described in OWL2 [Gro12] and designed in Protégé [Gen+03]) is added to each instance model. Such an instance model is shown in Fig. 2.5c. During query evaluation, the engine runs the inferencer to determine *inferred triples.*
- *Storing the inferred triples.* For a resource of a given type, all supertypes are explicity asserted in the model. For example, a resource with the type Segment also has the type TrackElement. Such an instance model is shown in Fig. 2.5d. Note that the `_1` and `_2` resources not only have types Segment and Switch, but also type TrackElement.

RDF uses Universal Resource Identifiers (URIs) [URI01] to identify nodes (resources) in the graph. To assign a numeric identifier to each resource, the URIs follow pattern `http://www.semanticweb.org/ontologies/2015/trainbenchmark#_x`, where `x` represents a unique identifier. The resulting SQL instance model follows trivially from the schemas and is omitted for the sake of conciseness.

SQL The metamodel of the Train Benchmark is mapped to SQL tables with a standard *object-relational mapping* (ORM) solution that assigns *a separate table to each class* [Bla+06; ONe08]. A class and its superclass(es) are connected by *foreign keys.* Many-to-many references are mapped to *junction tables* (also known as *association tables* or *join tables*). SQL data instances use primary keys for storing

data model	data model feature					
	nodes		edges		paths	
	types	props	types	props	types	props
simple, directed, and weighted graph	○	○	○	○	○	○
node-labelled graph	⊗	○	○	○	○	○
edge-typed graph or multiplex network [BNL14]	○	○	⊗	○	○	○
labelled graph or heterogeneous network [Shi+17]	⊗	○	⊗	○	○	○
property graph [HG16]	⊗	⊗	⊗	⊗	○	○
path property graph [Ang+18]	⊗	⊗	⊗	⊗	⊗	⊗
semantic graph (RDF) [PAG09]	⊗	⊗	⊗	○	○	○
object-oriented graph [Rum+91] and EMF [Ste+09]	⊗	⊗	⊗	○	○	○

Table 2.2: Features available to describe graph elements (nodes, edges, and paths) in various graph data models. For each graph element, column "types" shows whether it can be described with types (or labels), while column "props" shows whether it can be described with properties. Notation: ⊗ feature available, ○ feature not available.

unique identifiers. For the small example, the mapping in the Train Benchmark results in three tables with the following schemas: *TrackElement*(id), *Segment*(id, length), and *Switch*(id, currentPosition).

2.3.7 Graph Schema

Both relational databases and object-oriented systems [Rum+91] require their users to define the *schema* of the data *a priori*. SQL supports this by its *Data Definition Language* (DDL) with keywords such as `CREATE TABLE` and `PRIMARY KEY`. Object-oriented systems define their graph schemas by *meta-modelling* [VP03], which specifies a model (the *abstract syntax*) that defines the structure of a modelling language. On the other side of the spectrum, most property graph and semantic web systems are *schema-free* [BTL11], i.e. they do not require an explicit schema and rely on the *implicit schema* of the data set. This makes many user operations simpler, as populating the database, migrating between different versions can be achieved with less implementation effort – at least initially. However, many use cases, such as storing business critical data, need to enforce that the data complies with a predefined schema. Therefore, many practical systems are *schema-optional* [BTL11], i.e. they do not mandate a schema, but allow users to define it.

The schema inferencing algorithm The lack of a predefined schema is so prevalent in practice that we designed the *schema inferencing* algorithm to work around this problem by *extracting the relevant part of the schema from the query specification* (see Sec. 8.4.3).

Summary of Data Models

Tab. 2.2 shows the summary of theoretical and practical graph data models with their features. Comparison of database and MDE concepts are shown in Tab. 2.3.

Comparison category	Model-driven engineering	Graph data processing	Sim.
data modelling	object-oriented model	property graph	⊘
	metamodel	graph schema	⊘
	instance model	graph instance	⊘
	well-formedness contraints	integrity constraints	⊘
	model validation	data validation	⊘
	objects (classes)	nodes (labels)	⊗
	references (typed)	edges (typed)	⊗
	attributes	properties	⊗
	opposite references	undirected edges	⊗
data processing	model traversal code	stored procedures	○
	model queries	graph queries	⊗
	incremental queries	incremental view maintenance	⊗
	model transformation	graph updates	⊘
	model obfuscation	data anonymization	⊘
	code generation	–	–
	–	graph analytics	–
serialization / querying	XMI	GraphML, GraphSON, CSV	–
	VQL, OCL	Cypher, PGQL, G-CORE	–
non-incremental engines	Eclipse OCL, Epsilon, ATL	Neo4j, PGX, JanusGraph	–
incremental engines ⊛	Viatra, NMF, Reactive ATL	Graphflow	–
bidirectional engines	eMoflon, MoTe, UML-RSDS	–	–

Table 2.3: Comparison of concepts in model-driven engineering and graph processing fields. Incremental engines also offer non-incremental query evaluation. Concepts in the same row are similar, but often do not have an exact one-to-one correspondence. Column *"Sim."* describes the strength of similarity between concepts in the same row: ⊗ strongly similar ⊘ somewhat similar ○ weakly similar. Remark "⊛": incremental engines typically include support for non-incremental query evaluation as well.

2.4 Basics of Relational Algebra

2.4.1 Relations and Relational Schemas

In relational database theory [GUW09], a *relation* is a subset of the Cartesian product of domains, i.e. it contains a set of *tuples*. Each tuple has the same number of *attribute values* which correspond to the *relational schema*, the set of *attribute names* of the relation. We denote relations in *italic*, attributes in sans serif, and represent tuples as $\langle x_1, \dots, x_n \rangle$.

Bag relations Instead of building the algebra on *set semantics*, relations can be generalized for *bags*, also known as *multisets* [Syr00]. In this case, a tuple can occur more than once in a *bag relation*. For a detailed discussion on their properties, see [GUW09, Section 5.1: Relational Operations on Bags].

Note on order of attributes In the database literature, some authors define a relational schema as a *list of attributes* [EN00], while others define it as a *set of attributes* [GUW09; Mai83]. In this work,

we define it as a *set of attributes*, which allows us to formalize queries in a more succinct way.

2.4.2 Representing Labelled Graphs as Relations

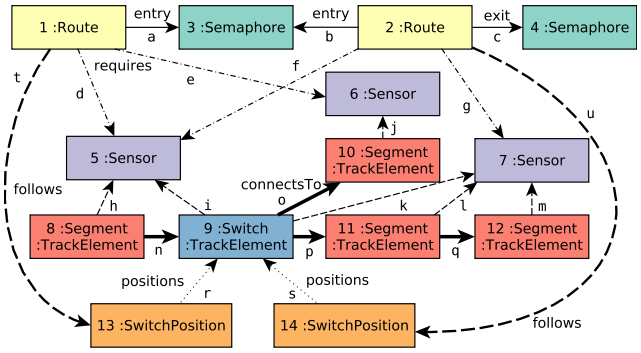


Figure 2.6: Railway instance as a labelled graph.

We map *nodes* and *edges* in the graph to *tuples*. For each node label and edge type, we define a separate relation. For the sake of simplicity, we presume that both nodes and edges in the graph have a unique identifier. Below, we use the labelled graph of Fig. 2.2c as an example, repeated in Fig. 2.6.

Mapping nodes to tuples Nodes can be trivially mapped to relations of 1-tuples by introducing a relation for each label. The nodes of the labelled graph in Fig. 2.6 constitute the following relations:

- *Route*(route) = $\{\langle 1\rangle, \langle 2\rangle\}$
- *Segment*(segment) = $\{\langle 8\rangle, \langle 10\rangle, \langle 11\rangle, \langle 12\rangle\}$
- *Semaphore*(semaphore) = $\{\langle 3\rangle, \langle 4\rangle\}$
- *Sensor*(sensor) = $\{\langle 5\rangle, \langle 6\rangle, \langle 7\rangle\}$
- *Switch*(switch) = $\{\langle 9\rangle\}$
- *SwitchPosition*(switchPosition) = $\{\langle 13\rangle, \langle 14\rangle\}$
- *TrackElement*(te) = $\{\langle 8\rangle, \langle 9\rangle, \langle 10\rangle, \langle 11\rangle, \langle 12\rangle\}$

Mapping edges to tuples Each edge is represented by a triple ⟨source node, edge, target node⟩. The edges of the labelled graph in Fig. 2.6 constitute the following relations:

- *connectsTo*(te1, connectsTo, te2) = $\{\langle 8, n, 9\rangle, \langle 9, o, 10\rangle, \langle 9, p, 11\rangle, \langle 11, q, 12\rangle\}$
- *entry*(route, entry, semaphore) = $\{\langle 1, a, 3\rangle, \langle 2, b, 3\rangle\}$
- *exit*(route, exit, semaphore) = $\{\langle 2, c, 4\rangle\}$
- *follows*(route, follows, switchPosition) = $\{\langle 1, t, 13\rangle, \langle 2, u, 14\rangle\}$
- *requires*(route, requires, sensor) = $\{\langle 1, d, 5\rangle, \langle 1, e, 6\rangle, \langle 2, f, 5\rangle, \langle 2, g, 7\rangle\}$
- *monitoredBy*(te, mB, sensor) = $\{\langle 8, h, 5\rangle, \langle 9, i, 5\rangle, \langle 10, j, 6\rangle\langle 9, k, 7\rangle, \langle 11, l, 7\rangle, \langle 12, m, 7\rangle\}$
- *target*(switchPosition, target, switch) = $\{\langle 13, r, 9\rangle, \langle 14, s, 9\rangle\}$

Notation In the representation above, attribute names of tuples are only denoted in the relational schemas and not in the tuples of the relation. Without the schema, we cannot determine the *semantics* of tuples, e.g. we cannot decide if the elements in $\langle 8, n, 9\rangle$ and $\langle 1, t, 13\rangle$ have the same labels. Hence, we often specify the attribute names for each tuple:

- $\langle$te1 : 8, connectsTo : n, te2 : $9\rangle$,
- $\langle$route : 1, follows : t, switchPosition : $13\rangle$.

Using this notation, the *follows* relation can be represented as:

$$\textit{follows} = \{\langle \mathsf{route} : 1, \mathsf{follows} : t, \mathsf{switchPosition} : 13\rangle, \langle \mathsf{route} : 2, \mathsf{follows} : u, \mathsf{switchPosition} : 14\rangle\}\,.$$

Next, we revisit the definitions of basic operators along with an example for each operator using the example graph of Fig. 2.6. For a detailed discussion of basic relational algebra operators, we refer the reader to database textbooks [EN00; SKS05]. Certain extension were based on the definitions of [GUW09, Section 5.2: Extended Operators of Relational Algebra].

#ops.	Notation	Name	Props.	Schema
1	$\sigma_\theta(r)$	selection	i	$\mathrm{sch}\,(r)$
	$\pi_{x_1/y_1,\ldots,x_n/y_n}(r)$	projection	i	$\langle y_1, \ldots, y_n\rangle$
	$\delta(r)$	duplicate-elimination	i	$\mathrm{sch}\,(r)$
2	$r \cup s$	union	c, a	$\mathrm{sch}\,(r)$
	$r \uplus s$	bag union	c, a	$\mathrm{sch}\,(r)$
	$r - s$	set minus	–	$\mathrm{sch}\,(r)$
	$r \bowtie s$	natural join	c, a	$\mathrm{sch}\,(r) \parallel (\mathrm{sch}\,(s) - \mathrm{sch}\,(r))$
	$r \ltimes s$	semijoin	–	$\mathrm{sch}\,(r)$
	$r \overline{\ltimes} s$	antijoin	–	$\mathrm{sch}\,(r)$
	$r ⟕ s$	left outer join	–	$\mathrm{sch}\,(r) \parallel (\mathrm{sch}\,(s) - \mathrm{sch}\,(r))$

Table 2.4: Number of operands, properties and result schemas of basic relational algebra operators. A unary operator α is idempotent (i), iff $\alpha(x) = \alpha(\alpha(x))$ for all inputs. A binary operator β is commutative (c), iff $x\ \beta\ y = y\ \beta\ x$ and associative (a), iff $(x\ \beta\ y)\ \beta\ z = x\ \beta\ (y\ \beta\ z)$. For schema transformations, append is denoted with $\parallel$, while removal is marked with –.

2.4.3 Unary Operators

Definition 8 (projection) The *projection* operator π filters the *attributes* (columns) of a relation by only keeping a certain set of them: $t = \pi_{x_1,\ldots,x_n}(r)$. In addition, the *extended projection* can perform computations and produce *new attributes*, e.g. $t = \pi_{x_1 \to a, 2 \to b}$ returns a relation of schema (a, b) with attribute b having a constant value 2.

Example 1 Return route nodes that have an outgoing requires edge.

$$\pi_{\mathsf{route/r}}(\textit{requires}) = \{\langle \mathsf{r} : 1\rangle, \langle \mathsf{r} : 2\rangle\}$$

Definition 9 (selection) The *selection* operator σ filters the relation according to some criteria: $t = \sigma_\theta(r)$, where predicate θ is a propositional formula. The operator *selects* all tuples in r for which θ holds.

Example 2 Return requires edges starting from route 1.

$$\sigma_{\mathsf{route}=1}(\mathit{requires}) = \{\langle \mathsf{route} : 1, \mathsf{requires} : d, \mathsf{sensor} : 5\rangle\}, \langle \mathsf{route} : 1, \mathsf{requires} : e, \mathsf{sensor} : 6\rangle\}$$

For *bag relations*, it is often required to remove duplications, i.e. to enforce set semantics.

Definition 10 (duplicate elimination) The *duplicate-elimination* operator δ eliminates duplicate tuples in a bag.

Example 3 Return all TrackElements with an outgoing connectsTo edge.

$$\delta\left(\pi_{\mathsf{te1}} \mathit{connectsTo}\right) = \{\langle \mathsf{te1} : 8\rangle, \langle \mathsf{te1} : 9\rangle, \langle \mathsf{te1} : 11\rangle\}$$

2.4.4 Binary Operators

Additive and Subtractive Operators

Definition 11 (union) The *union* of two relations produces the *set union* of the tuples in the relations.

Definitions across the literature vary regarding the requirements posed against the schema of the input relations of the union operator. While some definitions only require the input relations to have schemas with the same number of elements, we require the two input relations of the operation to have the same schema.[6]

Example 4 (Get Segments and Switches)

$$\left(\pi_{\mathsf{segment/te}}\ \mathit{Segment}\right) \cup \left(\pi_{\mathsf{switch/te}}\ \mathit{Switch}\right) = \{\langle \mathsf{te} : 8\rangle, \langle \mathsf{te} : 9\rangle, \langle \mathsf{te} : 10\rangle, \langle \mathsf{te} : 11\rangle, \langle \mathsf{te} : 12\rangle\}$$

A variant of the union operator targets *bag relations*.

Definition 12 (bag union) The *bag union* of two relations produces the *bag (multiset) union* of the tuples in the relations.

Example 5 (Get TrackElements and Switches, uniqueness not required)

$$\mathit{TrackElement} \uplus \left(\pi_{\mathsf{switch/te}}\ \mathit{Switch}\right) = \{\langle \mathsf{te} : 8\rangle, \langle \mathsf{te} : 9\rangle, \langle \mathsf{te} : 9\rangle, \langle \mathsf{te} : 10\rangle, \langle \mathsf{te} : 11\rangle, \langle \mathsf{te} : 12\rangle\}$$

Definition 13 (set minus) The *set minus* operation on two relations removes the tuples present in the second relation from the first relation.

Example 6 (Get TrackElements that are not Switches)

$$\mathit{TrackElement} - \left(\pi_{\mathsf{switch/te}}\ \mathit{Switch}\right) = \{\langle \mathsf{te} : 8\rangle, \langle \mathsf{te} : 10\rangle, \langle \mathsf{te} : 11\rangle, \langle \mathsf{te} : 12\rangle\}$$

[6] Practical implementations, such as some SQL engines, require the two schemas to have the same types (but some allow different names).

Join-Like Operators

Join-like operators are the primary means for databases to *connect data elements* and rebuild the heavily normalized pieces of information during query execution. Here, we define the most common join-like operators from the *Cartesian product* to *left outer join.*

The $\times$ operator produces the Cartesian product $t = r \times s$, where t holds all tuples that are the concatenation of exactly one tuple from r and exactly one tuple from s. Formally, using relational calculus [GUW09]:

Definition 14 (Cartesian product)

$$r \times s = \{\langle r_1, \dots, r_m, s_1, \dots, s_n\rangle \mid \langle r_1, \dots, r_m\rangle \in r \wedge \langle s_1, \dots, s_n\rangle \in s\}$$

Example 7 Generate any combinations of the *follows* and *exit* edges:

$$\begin{aligned}
&\textit{follows} \times \textit{exit} = \{\\
&\quad \langle \mathsf{route}_1 : 1, \mathsf{follows} : t, \mathsf{switchPosition} : 13, \mathsf{route}_2 : 2, \mathsf{exit} : c, \mathsf{semaphore} : 4\rangle,\\
&\quad \langle \mathsf{route}_1 : 2, \mathsf{follows} : u, \mathsf{switchPosition} : 14, \mathsf{route}_2 : 2, \mathsf{exit} : c, \mathsf{semaphore} : 4\rangle\\
&\}
\end{aligned}$$

The result of the *join* or *natural join* operator $\bowtie$ is determined by creating the Cartesian product of the relations, then filtering those tuples which are equal on the attributes that share a common name. The combined tuples are projected to remove duplicate attributes, i.e. from the attributes present in both of the two input relations, we only keep a single one. Thus, the join operator is defined as:

Definition 15 (join or natural join)

$$r \bowtie s \equiv \pi_{\mathrm{sch}(r) \cup \mathrm{sch}(s)} \left(\sigma_{r.A_1 = s.A_1 \wedge \dots \wedge r.A_n = s.A_n} \left(r \times s\right)\right),$$

where $\{A_1, \dots, A_n\}$ are the attributes in $\mathrm{sch}\,(r) \cap \mathrm{sch}\,(s)$, i.e. the set of attributes that occur in both schemas.

Note that if the set of common attributes is empty, the join operator is equivalent to the Cartesian product of the relations. The join operator is both commutative and associative: $r \bowtie s = s \bowtie r$ and $(r \bowtie s) \bowtie t = r \bowtie (s \bowtie t)$. In graph queries, the join operator can connect nodes, edges, and subgraphs to each other:

Example 8 Subgraphs of three nodes and two edges along the follows and exit edges:

$$\textit{follows} \bowtie \textit{exit} = \{\langle \mathsf{route} : 2, \mathsf{follows} : u, \mathsf{switchPosition} : 14, \mathsf{exit} : c, \mathsf{semaphore} : 4\rangle\}$$

Semijoin The *semijoin* operator $\ltimes$ takes its left input and only keeps tuples which have a matching pair on its right input in a join. Formally:

Definition 16 (semijoin) $r \ltimes s \equiv \pi_{\mathrm{sch}(r)}\,(r \bowtie s)$.

The semijoin operator can express positive structural conditions.

Example 9 The triples for follows edges that *have* an exit edge on their route:

$$\mathit{follows} \ltimes \mathit{exit} = \{\langle \mathsf{route} : 2, \mathsf{follows} : u, \mathsf{switchPosition} : 14 \rangle\}$$

Antijoin The antijoin operator $\overline{\ltimes}$ (also known as *left anti semijoin*) collects the tuples from the left relation r which have no matching pair in the right relation s [GK98]. Formally:

Definition 17 (antijoin)

$$t \equiv r \overline{\ltimes} s = r - (r \ltimes s)$$

The antijoin operator can express negative structural conditions.

Example 10 The triples for follows edges that *do not have* an exit edge on their route:

$$\mathit{follows} \overline{\ltimes} \mathit{exit} = \{\langle \mathsf{route} : 1, \mathsf{follows} : t, \mathsf{switchPosition} : 13 \rangle\}$$

Left outer join The *left outer join* operator $⟕$ produces the join of its input relations, then adds tuples from the left relation that did not have a pair in the right relation and pads them with NULL values [SKS05]. Formally:

Definition 18 (left outer join)

$$r ⟕ s \equiv (r \bowtie s) \cup \left((r \overline{\ltimes} s) \times \langle \mathsf{NULL} \rangle_{|\mathrm{sch}(s) - \mathrm{sch}(r)|} \right),$$

where $\langle \mathsf{NULL} \rangle_k$ denotes a tuple of k NULL values.

Example 11 The triples for follows edges that *either have or do not have* an exit edge on their route:

$$\begin{aligned} &\mathit{follows} ⟕ \mathit{exit} = \{ \\ &\quad \langle \mathsf{route} : 2, \mathsf{follows} : u, \mathsf{switchPosition} : 14, \mathsf{exit} : c, \mathsf{semaphore} : 4 \rangle, \\ &\quad \langle \mathsf{route} : 1, \mathsf{follows} : t, \mathsf{switchPosition} : 13, \mathsf{exit} : \mathsf{NULL}, \mathsf{semaphore} : \mathsf{NULL} \rangle \\ &\} \end{aligned}$$

Theta-joins All join operators presented so far ($\bowtie$, $\ltimes$, $\overline{\ltimes}$, $⟕$) follow the semantics of *natural join*, i.e. they perform the matching according to the set of common attributes of the input relations. In practice, many cases require a join operation performed against a different set of attributes. These can be conveniently handled with the *theta-join* operator, which performs the matching according to a predicate θ, a propositional formula (similarly to the selection predicate in Def. 9).

Definition 19 (theta-join or θ-join)

$$r \underset{\theta}{\bowtie} s \equiv \sigma_\theta (r \times s)$$

Example 12 Pairs of different connectsTo edges starting from the same node, based on relations *ct1* = *ct2* = *connectsTo* = $\{\langle 8, n, 9\rangle, \langle 9, o, 10\rangle, \langle 9, p, 11\rangle, \langle 11, q, 12\rangle\}$.

$$ct1 \underset{\substack{(ct1.\mathsf{te1}=ct2.\mathsf{te1})\wedge \\ (ct1.\mathsf{te2}\neq ct2.\mathsf{te2})}}{\bowtie} ct2 = \{\langle 9, o, 10, 9, p, 11\rangle, \langle 9, p, 11, 9, o, 10\rangle\}$$

Variants of the $\ltimes$, $\overline{\ltimes}$, and $\rtimes\!\!\bowtie$ operators can be defined analogously. For example:

Definition 20 (theta-antijoin)

$$r \underset{\theta}{\overline{\ltimes}} s \equiv r - (r \underset{\theta}{\ltimes} s), \quad \text{where} \quad r \underset{\theta}{\ltimes} s \equiv \pi_{\mathrm{sch}(r)}(r \underset{\theta}{\rtimes\!\!\bowtie} s).$$

Example 13 Get connectsTo edges that have no pair starting from the same node, based on relations *ct1* = *ct2* = *connectsTo*.

$$ct1 \underset{\substack{(ct1.\mathsf{te1}=ct2.\mathsf{te1})\wedge \\ (ct1.\mathsf{te2}<ct2.\mathsf{te2})}}{\overline{\ltimes}} ct2 \equiv ct1 - \Big(ct1 \underset{\substack{(ct1.\mathsf{te1}=ct2.\mathsf{te1})\wedge \\ (ct1.\mathsf{te2}<ct2.\mathsf{te2})}}{\ltimes} ct2\Big) = ct1 - \{\langle 9, o, 10\rangle, \langle 9, p, 11\rangle\} =$$
$$\{\langle 8, n, 9\rangle, \langle 11, q, 12\rangle\}$$

As a shorthand, we can pass a set of attributes $X = \{x_1, \dots, x_n\}$ instead of a predicate θ. The set of attributes should be a subset of the schemas of both relations (i.e. given relations r and s, $X \subseteq \mathrm{sch}(r) \cap \mathrm{sch}(s)$). Using this notation, the join-like operator checks the equivalence for all attributes:

$$r \underset{X}{\overline{\ltimes}} s \equiv r \underset{(r.x_1=s.x_1)\wedge\ldots\wedge(r.x_n=s.x_n)}{\overline{\ltimes}} s$$

2.5 Graph Pattern Matching and Traversal

Next, we define the concepts of *graph pattern matching* and *graph traversal* based on [Ang+17].

2.5.1 Pattern Matching

A *basic graph pattern* is a graph-structured query that is evaluated against the content of a graph database. *Complex graph patterns* extend this concept with operations such as projection, union, optional matching, and difference.

Matching a graph pattern against a graph instance is widely known as the *graph isomorphism* problem [Ull76; Cor+04; Lee+12], which requires a bijective mapping between elements of the pattern and the matched subgraph. In practice, this often places too many restrictions on the results and more relaxed *matching semantics* are used instead:

- *isomorphism-based semantics* constrain some kinds of repetitions:
  - *no-repeated-node semantics* is known as *fully isomorphic matching* and is identical to the restrictions used in the *graph isomorphism* problem,
  - *no-repeated-edge semantics* is known as *edge-isomorphic matching*.[7]
- *homomorphism-based semantics* is the most relaxed one, defining no constraints on repetitions.

[7] Note that *no-repeated-node* implies *no-repeated-edge* semantics as repeated edges would necessitate their endpoints to be repeated.

The trade-offs for different matching semantics vary: homomorphism-based matching semantics are more straightforward to implement as they require no additional filtering operations. Meanwhile, isomorphism-based semantics are often deemed more intuitive by users. For example, when running a "friends-of-friends" query on a social network for person p, engines using *isomorphic* semantics will not return person p, but *homomorphic* engines will. Moreover, *no-repeated-edge* semantics (see the comparison graph query languages in Tab. 2.5)

2.5.2 Graph Traversals

One of the distinguishing features of graph queries is that they allow users to define transitive navigations across the graph. According to survey [Ang+17], these type of queries fall into the category of *navigational expressions*, but alternative terms such as *graph exploration* [Ma+16] and *graph traversal* [Rod15] are also common. In this section, we discuss the definitions used to refer to the travelled sequences of nodes and edges. We start with the graph theoretical concept of a *walk*, then limit the repetition of specific graph elements. Finally, we discuss variants of *paths*.

> **Definition 21 (walk)** A *walk* in a graph is a sequence of nodes and edges, with both endpoints of an edge appearing adjacent to it in the sequence.

> **Definition 22 (trail)** A *trail* in a graph is a *walk* with no repetition of edges.

Note that *walks* correspond to *no constraints on repetitions* semantics, while the *trail* concept embraces the same idea as the *no-repeated-edge* semantics in pattern matching. Trails can be restricted further:

> **Definition 23 (path (strict) or simple path)** A *path* in a graph is a *trail* with no repetition of nodes.

This is again an application of a previously discussed semantics in pattern matching, namely, *no-repeated-nodes*. However, practical implementations often use a relaxed definition of paths:

> **Definition 24 (path (relaxed))** Same as *walk*, see Def. 21.

In this work, we use the *relaxed definition of paths* (Def. 24), in accordance to the convention of modern graph database systems, see e.g. [Ang+17]. Next, we present further refinements for *paths*:

> **Definition 25 (shortest path or geodesic)** A *shortest path* between two nodes is a *path* that consists of the least possible edges.

Of course, there may be more than one different shortest path (of the same length) between two given nodes, and the term *shortest path* can refer to any of them. Enumerating all such paths is the *all shortest paths* problem.

Weighted shortest paths *Shortest path* on *weighted graphs* (Def. 3) are often defined as the path with the lowest total sum of edge weights. To avoid ambiguity, this is often referred to as the *weighted shortest path*. It is easy to see that this is a generalization of the previous problem, as a *weighted shortest*

path algorithm can determine *shortest paths* with all edge weights set to a constant non-negative value (e.g. a value of 1).[8]

> **Definition 26 (distance)** The *distance* between two nodes is the length of a *shortest path* between the two nodes.

Remark Applying the restriction of *shortest path* annuls the difference between the strict (Def. 23) and relaxed (Def. 24) path variants: shortest paths do not contain any repetitions by definition else they could be shortened by the repeated sequence.

> **Definition 27 (cycle or circle)** A *cycle* in a graph is a walk which starts and ends in the same node.

Similarly to paths, some cycles can be restricted to disallow repetitions of nodes or edges. These do not pose much importance in the context of this dissertation, hence we omit the distinction. However, it is important to further refine the definition of cycles in directed graphs. A *directed cycle* is one which corresponds to a *directed walk*. Directed graphs *that do note have a directed cycle* are commonly referred to as DAGs (directed acyclic graphs). Evaluating *graph queries* that define a *cyclic graph pattern* is often more challenging then acyclic ones. We discuss the detailed reasons behind this in Sec. 10.2.

2.5.3 Graph Queries

The term *graph query* are queries that contain *graph patterns* and/or *path expressions*. Often these queries also contain other data processing operators such as projection, filtering, negation, etc.

Recursive queries *Recursive queries* are queries that reference themselves (see e.g. in [RG03]), and are evaluated up to reaching a fixpoint. In general, the evaluation of recursive queries is highly non-trivial, not only from a computational, but also from a conceptual perspective, as some recursive queries might have multiple fixpoints. Therefore, practical implementations often limit the use of certain features (e.g. negation, aggregation, etc.). However, even with typical limitation applied, recursive queries are sufficiently expressive to define paths in a graph.

Regular path queries *Regular path queries* (RPQs) [MW95; Tet+18] are a restricted category of recursive queries that define paths using *regular expressions*, allowing concatenation, disjunction, and repetition of paths. These allow a concise formulation of graph queries (that would be otherwise difficult or impossible to express), but are currently only supported by a few systems.

Path unwinding The *path unwinding* [Ang+17] operation allows users to define additional operations over a path in the graph. Such operations can include aggregating value over the path (such as calculating its total weight) or even performing additional pattern matching operations (see Sec. D.2).

[8] While most modern graph data processing systems support the simpler *shortest path* problem, they often provide little support for expressing *weighted shortest path* queries. In many cases, the lack of language support makes the calculation of such queries inefficient or even infeasible.

Relational languages for graph queries In academic works, graph queries are often expressed in Datalog [CGT89]. Extensions for RA that define fixpoint computations have also been proposed [Ros+86; Agr88], but gained little traction. Support for recursive queries was introduced as an extension to the SQL:1999 standard in the form of the `WITH RECURSIVE` keywords [ISO99], which is expressive enough to formulate most graph queries. However, it was introduced rather slowly in popular open-source implementations, it was first adopted by PostgreSQL in 2009, followed by SQLite in 2014, and by MySQL in 2017.

Usages In the context of this work, *well-formedness constraints* (WFCs) on graph models – which are similar to *integrity constraints* on graph databases – are captured and checked by *graph queries*.

2.6 Graph Query Languages

In this section, we briefly introduce graph query languages relevant to this dissertation. It is interesting to note that while most languages are significantly different from SQL, all of them reuse some of its keywords (such as `WHERE`, `AS`, and `GROUP BY`).

feature	Cypher	PGQL	G-CORE	SPARQL	VQL
query result	table	table	graph / table	table / graph	table
data model	PG	PG	PPG	RDF	EMF
pattern matching sem.	no-repeated-edge ⊛	homom.	homom.	homom.	homom.
default path sem.	no-repeated-edge ⊙	arbitrary ⊙	shortest path	arbitrary	arbitrary
path unwinding	⊗	⊗	⊗	○	○

Table 2.5: Comparison of graph query language features. Notation – *sem.:* semantics, *homom.:* homomorphism-based semantics (allowing arbitary repetitions). Remark "⊛": homomorphism-based matching semantics can be achieved by using multiple `MATCH` clauses. Remark "⊙": the language allows the use of different path semantics (e.g. *shortest paths*). *PG:* property graph, *PPG:* path property graph.

2.6.1 Cypher

Cypher [Fra+18] is a high-level declarative graph query language introduced and supported by the Neo4j graph database [Web12]. It allows users to specify graph patterns with an ASCII art-style syntax visually resembling a graph, which makes queries easy to comprehend. The goal of the *openCypher* project[9] is to provide a standardized specification of the Cypher language.

Path support Cypher supports *paths*: users can define queries that look for transitive reachability (both with a fixed and an unlimited upper bound) or shortest paths, and paths can be returned as a result of the query. Cypher supports *path unwinding* with the `UNWIND` keyword and *list comprehensions*. Regular path queries (RPQs) are not yet fully supported, however, they are proposed in the openCypher language as *path patterns*.[10]

[9] http://www.opencypher.org/

[10] https://github.com/thobe/openCypher/blob/b95eec108ce4ec07eedfe13b3e5fff0e94f789a4/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc

2.6.2 PGQL

PGQL (Property Graph Query Language) [Res+16] is a declarative query language designed by Oracle Labs. It combines SQL with graph pattern matching using a Cypher-like syntax. PGQL is implemented in Oracle Labs' PGX.D analytical graph processing engine [Hon+15]. PGX.D is capable of evaluating both graph analytical computations and graph queries, however, it operates on a read-only snapshot of the graph and hence does not allow updates.

Path support PGQL supports RPQs, allowing users to define expressions over vertices and edges along paths. Both *reachability* and – in PGQL's terminology – *path finding* queries are supported.

2.6.3 G-CORE

G-CORE [Ang+18] is a *design language* created by the Linked Data Benchmark Council's (LDBC) *Graph Query Language* task force. The G-CORE language does not strive to be a standard, and instead aims to "guide the evolution of both existing and future graph query languages". The language was designed to meet two key goals, lacking from popular graph query languages available at the time: (1) allow composition of queries and (2) treat paths as first class citizens. To these ends, (1) G-CORE queries return graphs as their results (which allows composability), and (2) they are defined over the *path property graph* data model (Sec. 2.2.4). Syntax-wise, G-CORE borrows numerous constructs from openCypher and PGQL.

Path support G-CORE supports RPQs, using the regular path expression syntax of PGQL [Res+16].

2.6.4 SPARQL

SPARQL [SP08] (a recursive acronym for *SPARQL Protocol and RDF Query Language*) is the standard query language of the Linked Data community and is implemented by several semantic database management systems. A formal definition of the language is given in [PAG09].

Path support Since version 1.1, SPARQL supports a variant of RPQs with *property paths* [HS13][11], which allow sequences, alternatives, inversions, and transitive edges. However, only the endpoints of a path can be accessed as variables, and SPARQL does not support operations on the path such as *determining the length of a given path* or *path unwinding*.

2.6.5 VIATRA Query Language

The Viatra Query Language (VQL) [Ujh+15a] (formerly known as IncQuery Pattern Language) is a declarative language for defining graph patterns based on Datalog [Ber+11]. It is used in the Viatra Query framework (formerly known as EMF-IncQuery [Ujh+15a]), which will be presented later in Sec. 10.4.4. While VQL and Viatra Query were originally designed for querying EMF models (Sec. 2.3.1), they can be adapted to support other data models such as the MPS language workbench [Sza+18].

[11] It might be more intuitive to think of the term *property path* as "*predicate path*". The precise differences between *predicates* and *properties* are outside of the scope of this work, and even the standard only discusses them superficially [HS13].

Path support VQL supports recursive queries in the form of *recursive pattern calls*. Using these, VQL can express transitive reachability and constraints on the path [Ber+12], but does not support *determining the length of a given path* or *path unwinding*.

2.7 Categorization of Graph Processing Workloads

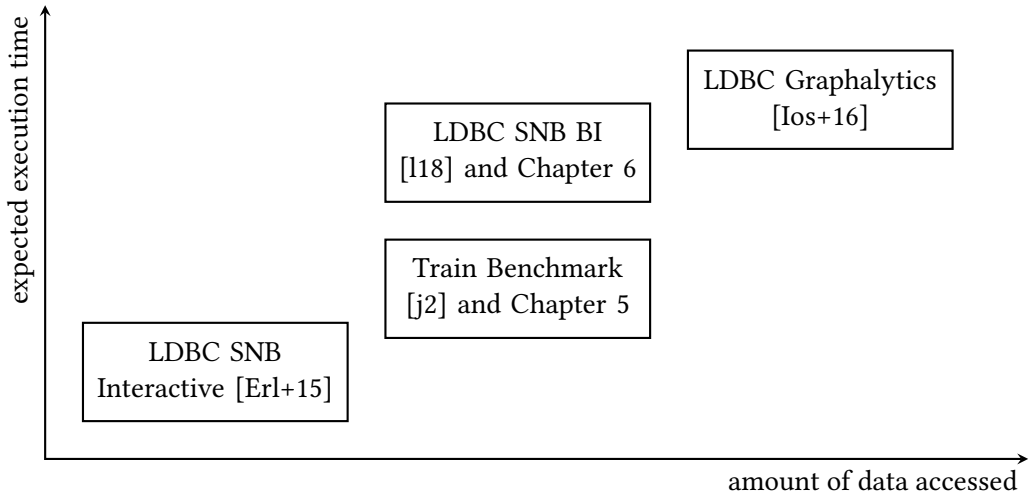


Figure 2.7: Characteristics of the workloads in the LDBC Social Network Benchmark (SNB), the LDBC Graphalytics benchmark, and the Train Benchmark.

This dissertation focuses on several different aspects of graph data processing, a field that gained a lot of momentum in the last decade. However, as of 2019, the landscape of graph processing is rather fragmented with most works only tackling a narrow scope of the domain. To get a basic overview of the field, we first present the categorization of the Linked Data Benchmark Council (LDBC) benchmarks, shown in Fig. 2.7. According to LDBC, there are two main categories of graph processing:

- **Graph query workloads** typically run graph pattern matching and navigation operations on semantic or property graphs. The *LDBC Social Network Benchmark* defines such workloads.
- **Graph analysis workloads** define *analytical computations* on graphs with little to no information stored as attributes (i.e. untyped, edge-typed, labelled, or weighted graphs). The *LDBC Graphalytics* benchmark defines such a workload.

Next, we briefly introduce these categories.

2.7.1 Graph Query Workloads

Graph query workloads aim to extract information from the graph queries that perform *graph pattern matching* and *graph navigation* operations. While most pattern matching operators can be expressed in relational algebra (using operators such as aggregation, selection and joins), navigations often require higher expressive power and recursive queries (Sec. 2.5.3) for transitive reachability. The cost of evaluating a certain graph query greatly depends on its *scope*, i.e. the number of nodes and edges it touches. Based on their scope, most graph queries – and on a bigger scale, graph query workloads – fall into one of the following categories:

- *Local graph queries* define relatively simple computations on a limited set of data (e.g. the neighbourhood of a single node), analogously to OLTP-style database systems (online transaction

processing). In systems with many simple reads, queries are they often posed concurrently by multiple users and under continuous updates. Due to the ever changing set of queries and data sets, using *one-time (batch) query evaluation* over the latest state of the database is preferable. The LDBC SNB Interactive workload [Erl+15] defines a graph OLTP workload.

- *Global graph queries* define OLAP (online analytical processing) [Bac13; CL14] workloads, where queries touch on a significant portion of the graph and often perform complex aggregations on the data. Some systems perform OLAP computations on a read-only view of the graph (updated periodically), while others evaluate them on the latest state of the database (which reflect the latest changes in the graph).

For databases evaluating (1) complex global queries that are known a-priori over (2) a continuously changing data set, it is easy to make an argument for *incremental view maintenance* (IVM) techniques [Sah+17], which is a central theme of Part III. Both benchmarks presented in Part II, namely, the Train Benchmark [j2] (Chapter 5) and the LDBC SNB Business Intelligence workload [l18] (Chapter 6) define graph OLAP workloads.

Languages and databases As discussed in Sec. 2.5.3, the SQL:1999 standard introduced the `WITH RECURSIVE` constructs to handle recursive expressions [ISO99]. Using these, most graph queries can be formulated in modern SQL dialects (see Sec. 10.5). However, the resulting queries are often cumbersome, difficult to maintain and optimize.[12] In contrast, specialized *graph query languages* (Sec. 2.6) were designed to allow users to express their queries in a concise format that can also be compiled to an efficiently evaluable representation. Internally, many graph database systems rely primarily on relational algebra extended with graph-specific constructs that support (a subset of) recursive queries.

Graph query workloads are typically evaluated in *database systems*. While RDBMs are widely used for this role, *graph databases* [RWE15] are gaining more and more momentum.[13] As of 2019, most graph database systems are single-node, but even many distributed implementations restrict query evaluation to a single node (the IncQuery-D system, presented in Chapter 9 is one of the exceptions). While local and global graph queries pose significantly different requirements to the underlying database system, there are currently no dedicated graph OLAP systems available.[14]

Benchmarks Related benchmarks for graph and semantic databases are presented in Sec. 7.2. Related database systems are compared as part of the discussion on query languages in Sec. 10.4.

2.7.2 Graph Analysis Workloads

Graph analysis workloads perform computations on the graph to derive *graph metrics* that characterize certain aspects of the graph structure, such as centrality (Sec. A.2), clustering (Sec. 3.4.2), and connectivity (Sec. A.1). These workloads often consider the connections in the entire graph or even transitive paths (e.g. to calculate *shortest paths* from a single node to all other nodes). Therefore, they are typically more expensive to run than *query workloads* and are often evaluated in batches on read-only graphs.

[12] Readers familiar with SQL might have experienced the difficulty of comprehending new SQL queries. For example, when looking at a query for the first time (without being familiar with the schema), it is often not obvious whether an attribute represents a connection (i.e. it is a foreign key) or a property. For many-to-many edges, *junction tables* make this distinction explicit, but add more noise.

[13] https://db-engines.com/en/ranking_categories

[14] Recently, database vendors started to use the term HTAP (hybrid transactional and analytical processing) [GS18], referring to capability of their systems to run both write-heavy transactional workloads and complex analytical queries.

Languages and computation framework There is a strong continuity between *global queries* and *graph analytics* as both evaluate complex computations that span over the entire graph. In fact, many graph analytical computations can be expressed with existing query languages, especially if the language supports recursive queries (Sec. 2.5.3). Consequently, claims against using specialized graph analytical systems have been made repeatedly [FRP15; Jin+15; ZY17] often in favour of RDBMSs using procedural SQL dialects such as T-SQL [Itz+09].

On the one hand, while numerous graph metrics can indeed be expressed as queries (see e.g. Cypher query that calculates the *local clustering coefficient*, listed in Sec. B.2), their execution is usually inefficient. This renders their application infeasible for graphs of moderate sizes (i.e. a few million nodes/edges). On the other hand specialized *graph analytical languages* were designed to accommodate graph analytical workloads and usually result in more efficient evaluation. However, these languages are rare and only supported by a few systems, as listed in Sec. B.3.

To achieve high performance, graph analytical computations are often translated to the *language of linear algebra* (Sec. 3.8) and are evaluated with linear algebra libraries that produce heavily optimized CPU or GPU code, ranging from BLAS [Law+79] to modern systems [Shi+18]. Due to their complexity, graph analytical workloads are almost exclusively executed as batch computations on a read-only snapshot of the graph. While *graph databases* need to store data for a prolonged period of time, many *graph analytical engines* are purely in-memory (and often distributed) tools, which do not keep a persistent copy of the graph. For more details, see [Zha+15] for a survey on in-memory data processing.

Benchmarks The LDBC Graphalytics benchmark [Ios+16] defines a graph analytical workload of 6 metrics, to be calculated on untyped and weighted graphs. We discuss this and other graph analytical benchmarks in Sec. 3.9.

Terminology Data analysis tasks are often called *analytics* or *analytical tasks*. The differences between these expression are subtle (e.g. some sources claim that analysis is a subset of analytics), and pose little importance w.r.t. this work. Hence, we use the terms *analysis* and *analytics* interchangeably.

2.7.3 Summary of Graph Processing Workloads

Tab. 2.6 shows a summary of typical graph processing workloads w.r.t. the support/applicability of given features and systems. [15] For more extensive analyses, we refer the readers to surveys [Lu+14; Bat+15; Sah+17; Hei+18] which give in-depth discussions on graph processing challenges, systems, and their applications.

[15] The applicability of various systems (databases and data processing frameworks) for certain workloads was inspired by the FOSDEM 2014 presentation on LDBC SNB: see https://www.slideshare.net/ldbcproject/ldbc-fosdem and https://archive.fosdem.org/2014/schedule/event/graphdevroom_ldbc/.

		Local queries	Global queries	Graph analytics
features	transaction profile	OLTP	OLAP	OLAP
	graph data model	P/S	P/S	U/W/E/L
	data updates	⊗	⊘	⊘
	IVM applicable	○	⊗	○
	distributed evaluation	○	⊘	⊗
systems	graph databases	⊗	⊗	⊘
	graph analysis frameworks	○	⊘	⊗
	relational databases	⊗	⊗	⊘
	semantic databases	⊗	⊗	○
most representative LDBC benchmark		SNB Interactive	SNB BI	Graphalytics

Table 2.6: Overview of features and systems that are applicable or support certain graph processing workloads, along with the most representative LDBC benchmark. Notation – ⊗ fully supported/applicable ⊘ supported/applicable to some degree ○ not supported/not applicable. Graph data models: **P**roperty, **S**emantic, **U**ntyped, **W**eighted, **E**dge-typed, and **L**abelled graph. OLTP: online transaction processing, OLAP: online analytical processing, IVM: incremental view maintenance.

Part I

Structural Analysis of Typed Graphs

Chapter 3

Characterization of Typed Graphs

3.1 Introduction

Context In this chapter, we present our results on applying typed graph metrics to graph models used in model-driven engineering (MDE). We study the properties of graph models with the goal of characterizing their structure at a fine-granularity level. This allows us to gain greater insight into the interplay between various types in the graph structure, which in turn would provide valuable input towards creating a realistic graph generator. The content of this chapter is primarily based on [c5].

Motivation While empirical software engineering highly relies on the source code repositories of large open-source projects, scalability assessment of MDE tools – i.e. model-driven engineering workbenches that operate on large engineering models – has been much more problematic. On the one hand, real complex industrial models are rarely available to public as intellectual property rights of all parties need to be protected. On the other hand, faithfulness of scalability evaluations using synthetic, auto-generated models are frequently considered questionable as these often generate models with a highly regular structure. Meanwhile, there is an increasing interest in model generators to be used for validating, testing, or benchmarking tools [BEC12; HHL14; Ara+15a].

Problem *But what makes a graph realistic?* Any engineer can distinguish an auto-generated model from a manually designed model by inspecting attributes (e.g. names). But what if we abstract from all attributes of the model and inspect only the (typed) graph structure? How can we characterize and distinguish systems engineering models (e.g. Capella [BBE15], AutoFOCUS [Ara+15b]) from models reverse engineered from source code, for instance?

Method In this chapter, we identify and evaluate graph-based metrics from other disciplines to decide which can best describe the characteristics of real graph models taken from software and systems engineering. We calculate these metrics on 83 graph models, and carry out an initial evaluation using statistical and visual exploratory data analysis. The output of our evaluation includes recommendations on characteristic metrics and hints for constructing future graph generators of realistic domain-specific graphs.[1]

[1] We use the term *metric* in an engineering sense, i.e. a metric is some descriptive value for a given scope (such as a node, an edge type, a node-type pair, etc.). Most of our metrics are neither traditional metric functions in the mathematical sense (which are non-negative, non-degenerate, symmetric, and satisfy the triangle inequality), nor measures (which are non-negative, return 0 for an empty set, and are Sigma-additive).

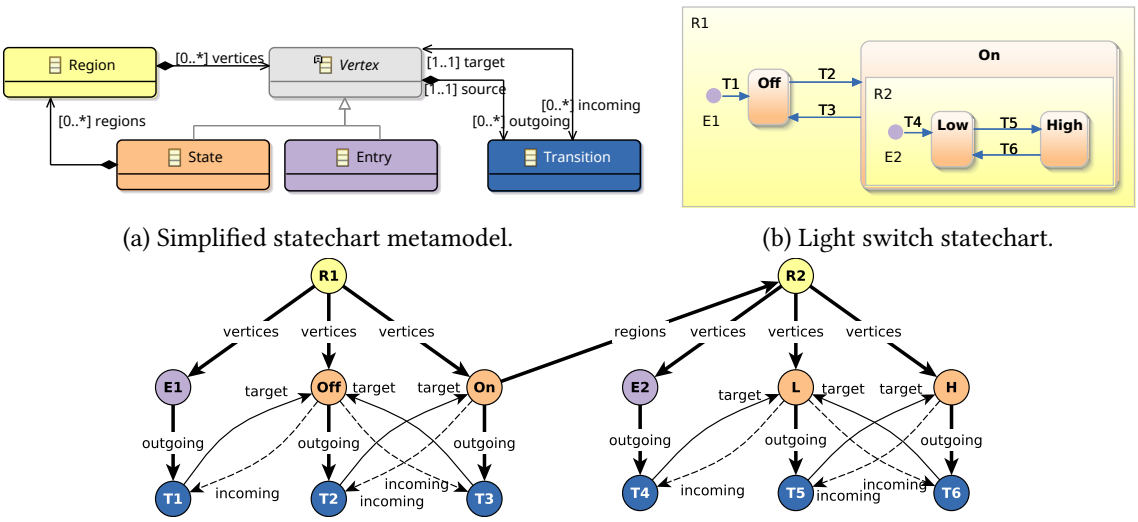
We reuse several graph metrics of network science [New03; WS98; DM02] already used in other disciplines (e.g. biology, social network analysis, neuroscience [Bat+18]) to reveal hidden structural properties of complex systems, and observe structural differences between them. However, in traditional network science, most of the analysed networks are *untyped*, i.e. they only contain edges of a single type, their direct adaptation to MDE models may not be sufficient due to the strongly typed nature of the latter. Therefore, we also collected and evaluated several graph metrics for typed networks [NL15]. Such *typed graph metrics* [Ber+13; BNL14; NL15] express structural properties w.r.t. a type, and how different types emerge together.

Contributions This chapter presents the following novel contributions:

- We collected 17 graph metrics from different disciplines (network science, physics, and social network analysis).
- We presented optimization techniques to speed up the calculation of different *clustering coefficient* metrics.
- We evaluated these metrics over 6 different modelling tools (domains) on 83 graph models.
- We carried out statistical and visual exploratory data analysis to identify *characteristic* metrics.
- Based on our findings, we give some hints for using these metrics in future graph generators.

3.2 Running Example and Mapping to Edge-Typed Graphs

Our running example in this chapter is a statechart (Fig. 3.1b) describing the behaviour of a light switch. The statechart is defined over a simplified metamodel of the Yakindu statechart modelling tool [Yak18] (Fig. 3.1a).



(a) Simplified statechart metamodel.

(b) Light switch statechart.

(c) Typed graph of the statechart. Note that *containment edges* are marked with thicker lines and *inverse edges* are marked with dashed lines.

Figure 3.1: The statechart example, the statechart metamodel and the typed graph representing the statechart.

Mapping to Edge-Typed Graphs Each model is an instance of a metamodel defined in Ecore, the metamodelling language of EMF (Eclipse Modeling Framework) [Ste+09]. The models are mapped to *edge-typed graphs* (Def. 4) using the following rules:

- Each object is mapped to a node ($v \in V$).
- Each reference type is mapped to a type ($t \in T$).
- Each reference instance is mapped to a directed edge between nodes, $(v, w, t) \in E$. Nodes v and w are the corresponding nodes of the reference's source and target objects. The type t is determined by the reference type.
- Object types are omitted. As a consequence, the graph does not contain information on the classes in the graph model.
- Redundant edges are removed: derived references are omitted along with opposite edges of containment references (see Sec. 3.6.2 for the rationale behind this).
- All attributes are removed (including derived ones).

We illustrate the mapping using the example statechart: Fig. 3.1c shows the statechart model as an edge-typed graph. The types include the reference types vertices, outgoing, incoming, target, and regions. Note that we excluded the source type as it is the inverse of the outgoing containment reference and is therefore redundant.

3.3 Concepts for Characterizing Graphs

Next, we introduce concepts for characterizing the connections in both untyped and typed graphs.

Definition 28 (connectedness in untyped graphs) In an untyped graph (Def. 1), nodes $v, w \in V$ are *connected* if there is an edge from v to w or vice versa in the graph. Formally,

$$Conn(v, w) \iff (v, w) \in E \vee (w, v) \in E.$$

Definition 29 (connectedness in an edge-typed graph) In an edge-typed graph, nodes $v, w \in V$ are *connected* in a type $t \in T$ if they have an edge in that type. Formally,

$$Conn(v, w, t) \iff \exists e \in E : \big(type(e) = t\big) \wedge \big((src(e) = v \wedge trg(e) = w) \vee (src(e) = w \wedge trg(e) = v)\big)$$

Definition 30 (activity in an edge-typed graph) In an edge-typed graph, node $v \in V$ is *active* in a type $t \in T$ if the node has at least one connection in that type:

$$Act(v, t) \iff \exists w \in W : Conn(v, w, t)$$

In the statechart graph (Fig. 3.1c), node E1 is *connected* to node R1 along type vertices, and to T1 along type outgoing. Hence, node R1 is *active* in types vertices, and outgoing.

3.4 Metrics for Untyped Graphs

3.4.1 Basic Graph Metrics

The most simple metrics for untyped graphs are the *number of nodes* $|V|$ and *the number of edges* $|E|$. For a node $v \in V$, the *in-degree* $Degree^{in}(v)$ is the number of incoming edges and the *out-degree*

Name	Notation	Nd.
Number of nodes	$\lvert V\rvert$	○
Number of edges	$\lvert E\rvert$	○
Average degree	$\langle \mathit{Degree} \rangle$	○
Density	D	⊗
Triangle count	–	○
Radius	r	○
Diameter	d	○
Global clustering coefficient	GCC	⊗
Global triangle count	–	○

(a) Graph-level metrics.

Name	Notation	Nd.
Degree	$\mathit{Degree}(v)$	○
In-degree	$\mathit{Degree}^{in}(v)$	○
Out-degree	$\mathit{Degree}^{out}(v)$	○
Local triangle count	–	○
Local clustering coefficient	$LCC(v)$	⊗
Eccentricity	$\epsilon(v)$	○
Betweenness centrality	$g(v)$	⊘
PageRank	$\mathrm{PR}(v)$	⊘
Component size	–	○

(b) Node-level metrics.

Table 3.1: Examples of metrics for untyped graphs. We only specified the *notation* for cases where there is a commonly used convention for denoting the given metric. *Nd.:* the metric normalized in the range of $[0, 1]$.

$\mathit{Degree}^{out}(v)$ is the number of outgoing edges. The *degree* metric, $\mathit{Degree}(v)$, is equal to the total number of the incoming and outgoing edges of node v, i.e. $\mathit{Degree}(v) = \mathit{Degree}^{in}(v) + \mathit{Degree}^{out}(v)$. The *average degree* of a graph is $\langle \mathit{Degree} \rangle = \frac{2|E|}{|V|}$, where $\langle\rangle$ denotes the average.

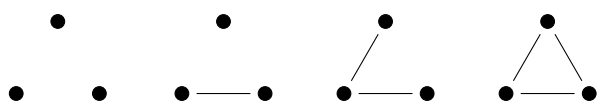
Example 14 In the statechart graph model in Fig. 3.1c, the metrics take the following values: $|V| = 14$, $|E| = 25$, $\mathit{Degree}^{in}(\mathsf{R2}) = 1$, $\mathit{Degree}^{out}(\mathsf{R2}) = 3$, and $\mathit{Degree}(\mathsf{R2}) = 4$. The average degree across the graph is $\langle \mathit{Degree} \rangle \approx 3.57$ and the density is $D \approx 0.14$.

3.4.2 Clustering Metrics

Clustering metrics are used to describe how likely triangles are to form in a given graph. It is well-known that real graphs tend to exhibit clusteredness, e.g. in a social network two friends of a person are significantly more likely to be friends with each other than two randomly picked individuals.

To handle potential triangles in the graph, we first introduce the concept of *triads*:

Definition 31 (triad) A *triad* is a subgraph formed by 3 nodes.



(a) empty (b) one-edge (c) two-path (d) triangle

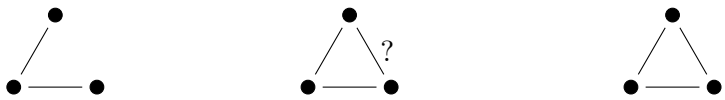
Figure 3.2: Undirected triads. Note that missing edges, such as the one in (c), are not optional, but explicitly *do not exist* in the subgraph induced by the nodes of the triad.

For undirected graphs, there are four possible triads, shown in Fig. 3.2. Triads containing at least a two-path are categorized based on whether the two ends of the path connect (Fig. 3.3). This is captured by the concept of *transitivity*. We differentiate between three categories of triads:

Definition 32 (intransitive triad or open wedge) An *intransitive triad* is a triad with end nodes that are not connected (Fig. 3.3a).

Definition 33 (potentially transitive triad or wedge) A *potentially transitive triad* is a triad with end nodes that might or might not be connected (Fig. 3.3b).

Definition 34 (transitive triad or closed wedge or triangle) A *transitive triad* is a triad with end nodes that are connected (Fig. 3.3c).



(a) intransitive triad or open wedge (b) potentially transitive triad or wedge (c) transitive triad or closed wedge or triangle

Figure 3.3: Transitivity for undirected triads and their relationship to open/indefinite/closed wedges. A *transitive triad* (*closed wedge*) is often simply referred to as a *triangle*.

Note on terminology. Different sources use different definitions for the *triad* concept. Lecture [Sni12] defines triads as "subgraphs formed by 3 nodes", and introduces the *potentially transitive*, *intransitive*, and *transitive* variants defined above. These definitions also used in Wasserman and Faust's seminal book "Social Network Analysis" [WF94]. On the contrary, paper [BNL14] defines triads as "connected triples of nodes, which close into triangles", which is equivalent to the concept of *potentially transitive triad* (Def. 33) presented in this work. A third definition is used in [ABG15], which introduces the term *wedge* as "a path of length two". Its authors differentiate between the (mutually exclusive) concepts of an *open wedge* and a *closed wedge*, which correspond to our definitions of an *intransitive triad* and a *transitive triad*, respectively. The paper then uses the term *triad* as a synonym for an *open wedge* "that is known not to form a triangle", i.e. an *intransitive triad*.

A coarse-granularity metric to characterize is the number of its triangles:

Definition 35 (global triangle count) The *global triangle count* is the total number of triangles in the graph.

At a finer granularity, the number of triangles can be determined for each node:

Definition 36 (local triangle count) The *local triangle count* for a node v is the number of triangles that contain v.

While these metrics provide some insight into the structure of the graph, neither of them is able to capture the clusteredness of a given graph, i.e. how likely its nodes tend to form triangles. To determine this, we calculate the *global clustering coefficient*, which is defined as the ratio of *triangles* (Def. 34) and *wedges* (Def. 33):

Definition 37 (global clustering coefficient)

$$GCC(G) = \frac{\text{number of triangles in } G}{\text{number of wedges in } G}$$

On a vertex level, the *local clustering coefficient* $LCC(v)$ measures the probability that the neighbours of a node $v \in V$ are connected to each other [WS98]. Formally, it can be defined as

Definition 38 (local clustering coefficient)

$$LCC(v) = \frac{\left|a, b \mid Conn(v, a) \wedge Conn(v, b) \wedge Conn(a, b)\right|}{\left|a, b \mid Conn(v, a) \wedge Conn(v, b)\right|}$$

Let us observe that this is equivalent to applying the *GCC* definition for wedges centered in a given node v:

$$LCC(v) = \frac{\text{number of triangles centered in } v}{\text{number of wedges centered in } v}$$

$LCC(v)$ is normalized to the interval $[0, 1]$, equalling 1 if every neighbour of v is connected to each other and 0 if there are no connections between the neighbours of v.

Example 15 Due to the absence of triangles n the example graph of Fig. 3.1c, the *local triangle count* of each node is 0 and the *global triangle count* of the graph is also 0. Consequently, the *global clustering coefficient* of is $GCC = 0$, and the *local clustering coefficient* is 0 for every $v \in V$, i.e. $LCC(v) = 0$. This demonstrates that despite being a widely used metric in network science, clustering coefficients often fail to capture the structure of the graph – or provide any information at all.

Related algorithms Determining the *triangle count* metrics efficiently remains a hot topic in the high-performance computing community. Recent works include a GraphBLAS-based implementation [Dav18] and an algorithm that *does not require matrix multiplication* [Low+17].

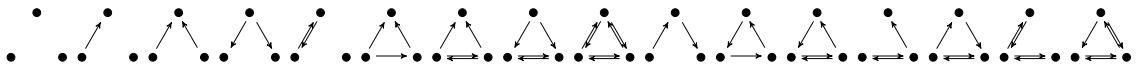


Figure 3.4: Possible triads for directed graphs, as presented in [WF94].

Clustering for directed graphs Clustering coefficients can be generalized for *directed graphs*. However, this makes their computation and interpretation significantly more complex. While the 4 possible triads for undirected graphs are simple to grasp (Fig. 3.2), there are 16 possible triads for directed graphs (Fig. 3.4).

3.4.3 Summary of Metrics for Untyped Graphs

Tab. 3.1 summarizes the global and local untyped graph metrics discussed here. At this point, it is worth noting that some metrics generalize well for typed graphs, e.g. the *number of nodes* and the *triangle count* of the graph are still meaningful, even though they discard some information about the distribution of the types in the graph. However, some metrics cannot be adapted well: calculating *betweenness centrality* or *PageRank* for an edge-typed graph – while omitting the type information – provides little insight, as the semantics of the resulting values are often unclear.

3.5 Typed Metrics

In the following, we introduce a set of *typed graph metrics* that characterize the structure of the graph with taking type information into account. This allows us to gain greater insight into the microscopic structure of the graph and study the interplay between the types in the graph. Tab. 3.2 shows the typed metrics used in this chapter. The *Scope* column lists whether the metric is defined on *types* (Sec. 3.5.1), *nodes* (Sec. 3.5.2), *type-node pairs* (Sec. 3.5.3), or *type pairs* (Sec. 3.5.4).

Name	Introduced as	Reference	Notation	Scope	Nd.
Node type activity	Layer activity	[NL15]	$NTA(t)$	T	○
Node type connectivity	–	[Ber+13]	$NTC(t)$	T	⊗
Node exclusive type connectivity	–	[Ber+13]	$NETC(t)$	T	⊗
Edge type activity	–	–	$ETA(t)$	T	○
Edge type connectivity	–	[Ber+13]	$ETC(t)$	T	⊗
Node activity	–	[NL15]	$NA(v)$	N	○
Typed participation coefficient	Multiplex partic. coeff.	[BNL14]	$TPC(v)$	N	⊗
Typed local clustering coefficient	Clustering coeff. C_1, C_2	[BNL14]	$TCC(v)$	N	⊗
Interdependence	–	[BNL14]	$\lambda(v)$	N	⊗
Typed degree	Degree	[Ber+13]	$Degree(v,t)$	T × N	○
Pairwise type connectivity	Pairwise multiplexity	[NL15]	$PTC(t_1,t_2)$	T^2	⊗

Table 3.2: Summary of metrics for typed graphs. *Introduced as:* the metric was originally introduced in the referred paper (*Reference*) under a different name. *Notation:* $v \in V$; $t, t_1, t_2 \in T$. *Scope*: N = Nodes, T = Types, T × N = Type-node pairs, T^2 = Type pairs; *Nd.*: shows whether the metric values are normalized.

Example 16 Fig. 3.1 has 5 types: $T = \{\text{vertices}, \text{regions}, \text{target}, \text{incoming}, \text{outgoing}\}$.

3.5.1 Metrics for Types

First, we present metrics that characterize each type in the graph. The number of types tends to be low for most domains, hence these metrics have the advantage of being quite "compact", i.e. the description for a graph in a given domain does not grow with the size of the graph.

Definition 39 (node type activity) *Node type activity* (*NTA*), a.k.a. *layer activity* [NL15], characterizes a type $t \in T$, and equals the number of nodes that are *active* in type t:

$$NTA(t) = \left|\{v \in V \mid Act(v,t)\}\right|.$$

Definition 40 (node type connectivity) *Node type connectivity* (*NTC*) [Ber+13] shows the ratio of nodes that belong to type t:

$$NTC(t) = \frac{NTA(t)}{|V|}.$$

Definition 41 (node exclusive type connectivity) The *node exclusive type connectivity* (*NETC*) [Ber+13] is similar to *node type connectivity*, but it calculates the ratio of nodes that belong *exclusively* to type t. In other words, it shows the ratio of nodes that only have types t:

$$NETC(t) = \frac{\left|\{v \in V \mid Act(v,t) \wedge \neg Act(v, T \backslash \{t\})\}\right|}{|V|}.$$

Definition 42 (edge type activity) *Edge type activity* (*ETA*) determines the number of edges that belong to type $t \in T$:

$$ETA(t) = \left|\{(v,w,t) \in E \mid v,w \in V\}\right|.$$

Definition 43 (edge type connectivity) *Edge type connectivity* (*ETC*) [Ber+13] determines the ratio of edges of type $t \in T$:

$$ETC(t) = \frac{ETA(t)}{|E|}.$$

Example 17 In the graph of Fig. 3.1c:

- $NTA(\text{outgoing}) = 12$ and $NTC(\text{outgoing}) = 0.85$, implying that the majority of nodes are active in type outgoing.
- $NETC(\text{outgoing}) = 0$ as there are no nodes exclusively active in this type.
- $ETA(\text{outgoing}) = 6$ and $ETC(\text{outgoing}) = 0.24$, meaning that 24% of edges are in type outgoing.

3.5.2 Metrics for Nodes

Next, we present metrics that are calculated for each node of the graph. As these metrics encode more information for large graphs than the ones that only consider types, they are more expensive to calculate and analyse. However, they provide greater insight into the structure of the network.

Definition 44 (node activity) *Node activity* (*NA*) [NL15] identifies the number of types in which node $v \in V$ is active. Formally, it is defined as:

$$NA(v) = \left|\{t \in T \mid Act(v,t)\}\right|.$$

Example 18 In the example graph (Fig. 3.1c), $NA(\mathsf{L}) = 4$, as node L is active in 4 types.

Definition 45 (typed participation coefficient) The *typed participation coefficient* (*TPC*), introduced as *multiplex participation coefficient* [BNL14], measures whether the connections of node $v \in V$ are uniformly distributed among types T:

$$TPC(v) = \frac{|T|}{|T|-1}\left[1 - \sum_{t \in T}\left(\frac{Degree(v,t)}{Degree(v,T)}\right)^2\right].$$

$TPC(v)$ takes values in $[0, 1]$, equalling 0 if all the edges of v belong to a single type, and 1 if v has exactly the same number of edges on each of types T.

Example 19 $TPC(\mathsf{R1}) = 0$ means that all edges belonging to R1 are of a single type, but

$$TPC(\mathsf{R2}) = \frac{5}{4} \cdot \left[1 - \left(\underbrace{\left(\frac{1}{4}\right)^2}_{\text{regions}} + \underbrace{\left(\frac{3}{4}\right)^2}_{\text{vertices}}\right)\right] \approx 0.47,$$

thus types belonging to node R2 (regions and vertices) are not uniformly distributed.

Typed Clustering Metrics

Similarly to the *local clustering coefficient* metric $LCC(v)$ (Sec. 3.4), the *typed local clustering coefficients* $TCC_1(v)$ and $TCC_2(v)$ [BNL14] measure the ratio of *triangles* to *wedges* centered in node $v \in V$, but also take the type information into account. To generalize the notion of the clustering coefficient, we first introduce typed variants of the *wedge* and *triangle* concepts [BNL14], illustrated in Fig. 3.5.

(a) one-wedge (b) two-wedge (c) two-triangle (d) three-triangle

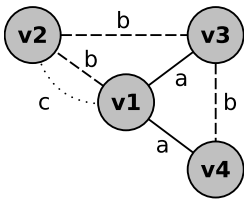
Figure 3.5: Typed wedges and triangles. The types t_1, t_2, t_3 denote different types.

Definition 46 (one-wedge) A *one-wedge* $u - v - w$ centered in node v is a *wedge* in which both edges $u - v$ and $v - w$ have the same types (Fig. 3.5a).

Definition 47 (two-wedge) A *two-wedge* $u - v - w$ centered in node v is a *wedge* in which edges $u - v$ and $v - w$ have different types (Fig. 3.5b).

Definition 48 (two-triangle) A *two-triangle* is a *triangle* which is formed by an edge of a certain type and two edges of another type (Fig. 3.5c).

Definition 49 (three-triangle) A *three-triangle* is a *triangle* which is formed by edges of different types (Fig. 3.5d).



	v_1	v_2	v_3	v_4
$LCC(v)$	0.67	1.00	0.67	1.00
$TCC_1(v)$	1.00	0.50	0.00	0.00
$TCC_2(v)$	0.25	1.00	0.50	0.00

Figure 3.6: Example graph and values of clustering coefficient metrics.

Definition 50 (typed local clustering coefficient 1) $TCC_1(v)$ is the ratio of two-triangles and one-wedges centered in node v. Formally,

$$TCC_1(v) = \frac{\left|u, w \mid Conn(v, u, t_1) \wedge Conn(v, w, t_1) \wedge Conn(u, w, t_2) \wedge t_1 \neq t_2\right|}{\left|u, w \mid Conn(v, u, t_1) \wedge Conn(v, w, t_1)\right|}$$

The example graph in Fig. 3.1c does not have any triangles, hence we use a different example to demonstrate the clustering coefficients (Fig. 3.6). In this example, $TCC_1(v_1) = 1$ as only v_3 and v_4 are connected to v_1 on the same type (a), while v_3 and v_4 are connected on a different type (b).

Definition 51 (typed local clustering coefficient 2) $TCC_2(v)$ is the ratio of three-triangles and two-wedges centered in node v. Formally,

$$TCC_2(v) = \frac{\left|u, w \mid Conn(v, u, t_1) \wedge Conn(v, w, t_2) \wedge Conn(u, w, t_3) \wedge t_1 \neq t_2 \wedge t_2 \neq t_3 \wedge t_1 \neq t_3\right|}{\left|u, w \mid Conn(v, u, t_1) \wedge Conn(v, w, t_2) \wedge t_1 \neq t_2\right|}$$

Example 20 In the example graph, the two-wedges centered in v_1 could be completed to a three-triangle in 4 possible ways:

1. (v_1, v_2, b), (v_1, v_3, a) with (v_2, v_3, c)
2. (v_1, v_2, c), (v_1, v_3, a) with (v_2, v_3, b) (this one exists)
3. (v_1, v_2, b), (v_1, v_4, a) with (v_2, v_4, c)
4. (v_1, v_2, c), (v_1, v_4, a) with (v_2, v_4, b)

Only a single one of these three-triangles exists in the graph (Item 2), hence $TCC_2(v_1) = \frac{1}{4} = 0.25$.

Typed Shortest Path Metrics

Paths in typed graphs may lead through different types and may exhibit very different properties when investigated in the context of a single type, a type pair or any types. The *interdependence* metric aims to capture how shortest paths run in the graph:

Definition 52 (interdependence) *Interdependence* [Nic+13; BNL14] investigates the number of *shortest paths* (Def. 25) w.r.t. types. For a given node, it calculates the ratio of the (1) shortest paths from node v that are *active* in at least two types and (2) all shortest paths from node v. Formally, interdependence $\lambda(v)$ is defined as:

$$\lambda(v) = \sum_{w \neq v} \frac{\psi_{vw}}{\sigma_{vw}},$$

where ψ_{vw} is the number of paths between nodes v and w that are *active* in at least two types, while σ_{vw} is the total number of shortest paths between nodes v and w.

Note that this metric is somewhat similar to the (untyped) *betweenness centrality* (Def. 64) as both investigate the ratio of shortest paths for a given node. However, there are is an important difference: interdependence considers the ratio of multi-typed and single-typed shortest paths that start from node v, while *betweenness centrality* investigates the ratio of (untyped) shortest paths passing through v and shortest paths between any pair of nodes s, t.

3.5.3 Metrics for Type-Node Pairs

In this category, we calculate metrics for each type-node pair. Compared to other calculation scopes (nodes, types, or type pairs), this category encodes the most information. However, it fails to take into account the interplay between the types, and is consequently not a generally useful metric as we will show in Sec. 3.7.

Presented as a redefinition of *Degree* in [Ber+13], the *typed degree* metric is determined by the number of neighbours of a node v with respect to type t:

Definition 53 (typed degree)

$$Degree\,(v,t) = \left|\left\{w \in V \mid Conn(v,w,t)\right\}\right|$$

The metric can be generalized for a set of a types $T' \subseteq T$:

Definition 54 (typed degree for type sets)

$$Degree\left(v,T'\right) = \sum_{d \in T'} Degree\,(v,t)\,.$$

Note that for $T' = T$, the typed degree of a node v equals its untyped degree $Degree(v)$.

Example 21 In the example graph of Fig. 3.1c, typed degree values include $Degree\,(\mathsf{R2}, \mathsf{vertices}) = 3$ and $Degree\,(\mathsf{T3}, \{\mathsf{outgoing}, \mathsf{incoming}\}) = 1$.

3.5.4 Metrics for Type Pairs

In the category, we define metrics for *type pairs*. This results in $|T|^2$ metric values, which is relatively compact for graphs that do not have an excessive amount of types. Still, it often provides good insight into the *interplay between different types*.

The *pairwise type connectivity* metric (*PTC*), introduced as *pairwise multiplexity* in [NL15], is defined for a pair of types, $t_i, t_j \in T$, where $1 \leq i, j \leq |T|$. Its value determines the ratio of nodes in the network, which are active in both types t_i and t_j. Intuitively, the more mutual nodes the two types have, the higher their *pairwise type connectivity* is.

Definition 55 (node activity) The *node activity* binary vector a_v ($v \in V$) is defined as:

$$a_v = \left\{a_v^{[t_1]}, a_v^{[t_2]}, \dots, a_v^{[t_{|T|}]}\right\}, \text{where } a_v^{[t]} = \begin{cases} 1, & \text{if } Act(v,t), \\ 0, & \text{otherwise.} \end{cases}$$

Using this vector, we define a metric for each type pair:

Definition 56 (pairwise type connectivity)

$$PTC(t_i,t_j) = \frac{1}{|V|} \sum_{v \in V} a_v^{[t_i]} a_v^{[t_j]},$$

$PTC(t_i,t_j)$ takes values from the $[0,1]$ interval, and equals 1 if the activity vectors $a_v^{[t_i]}$ and $a_v^{[t_j]}$ are identical, i.e. when t_i and t_j belong to the same nodes.

> **Example 22** In the example graph (Fig. 3.1c), $PTC(\mathsf{incoming}, \mathsf{outgoing}) = 0.71$, as these two types often appear together. This can be explained by the fact that every State node belongs to both types. However, the value is less than 1 as Entry nodes are never active in type incoming.

3.6 Experimental Setup

To study the selected set of metrics and their applicability on engineering models, we analysed the characteristics of 83 graph models by evaluating single and typed metrics on them.

3.6.1 Domains and Instance Models

Models were taken from six different domains:

- *AutoFOCUS* [Ara+15b][2] is an MDE systems engineering tool for designing distributed, embedded software systems.
- *Building Information Model (BIM)* [Eas+08] is a representation format for architecture designs. BIM models were provided by Uninova, an industrial partner in the MONDO EU FP7 project [MON16].
- *Capella* [BBE15][3] is a graphical modelling workbench for model-based systems engineering developed at Thales to support the Arcadia engineering method.
- *JaMoPP* [Hei+11][4] parses Java source code into EMF-based models and vice versa by constructing abstract syntax trees (ASTs) from the source code with the extension of cross-references (e.g. method calls, variable access).
- *Yakindu Statecharts Tools* [Yak18][5] is an integrated modelling environment developed by Itemis AG. It can be used for the specification and development of reactive systems using statecharts.
- The *Train Benchmark* [j2] is a cross-technology benchmark, presented in Chapter 5, which measures the performance of continuous model validation on graph-based models in a railway system domain. We used 4 synthetic models from the Train Benchmark in experiments, while all models from other domains were real models created by engineers.

Tab. 3.3 shows the basic graph characteristics of the models. Each domain contains several instance models (3–34) with different sizes, where BIM and JaMoPP models are the largest (up to $10\mathrm{M}$ nodes). The average degree $\langle Degree \rangle$ ranges from 2.2 to 7.8 which shows a significant difference between our models and social networks [Bró+12; Cos+11] where the average degree is often ten times as much. The number of types $|T|$ varies across domains: while Yakindu or Train Benchmark models are built from 4–12 types, Capella models of similar size may contain 10 times more types. The ratio of containment edges, $ETC\ (containment)$, is higher for AutoFOCUS and JaMoPP models which means fewer cross-references between the objects. This also explains the smaller average degrees for these models. Note that the BIM models are flat graphs without a containment hierarchy.

3.6.2 Data Preparation

When we first evaluated the typed metrics on these models, we observed outlying values along several distribution functions, which were caused by some extremities of models. Therefore, we carried out some data preparation and cleaning prior to the actual data analysis:

[2] https://af3.fortiss.org/
[3] https://www.polarsys.org/capella/
[4] https://github.com/DevBoost/JaMoPP
[5] https://www.itemis.com/en/yakindu/

Domain	#	$\lvert T\rvert$	$\lvert V\rvert$	$\langle Degree\rangle$	$ETC\,(containment)$
AutoFOCUS	24	16 – 74	10 – 1k	2.2 – 3.2	0.7 – 0.92
BIM	34	51 – 117	10k – 10M	2.2 – 5.2	0 – 0
Capella	3	103 – 182	1k – 10k	4.2 – 5.0	0.41 – 0.48
JaMoPP	9	67 – 98	100k – 1M	2.6 – 2.6	0.8 – 0.8
Yakindu	9	4 – 4	10 – 1k	3.2 – 4.6	0.41 – 0.52
Train Benchmark	4	12 – 12	1k – 10k	7.2 – 7.8	0.16 – 0.16

Table 3.3: Characteristics of the instance models. #: number of instance models; $|T|$: number of types; $|V|$: number of nodes; $\langle Degree\rangle$: average degree; $ETC(containment)$: edge type connectivity of containment edges.

- *Omitting layout information.* Some modelling tools (AutoFOCUS and Yakindu) persisted graphical information of diagram elements to the model itself. As several metrics were dominated by the large number of such elements, we decided to remove the layout information from these models (except for BIM where graphics is the key information in the models).
- *Omitting models of extreme sizes.* We omitted models that were very small (e.g. overly simple example models) compared to all other models of the domain and therefore distorted metric values and thus the results of the analysis.
- *No redundant edges.* All derived edges were removed, along with inverse edges of containment.

The resulting graph – extracted from Fig. 3.1c – is shown in Fig. 3.7.

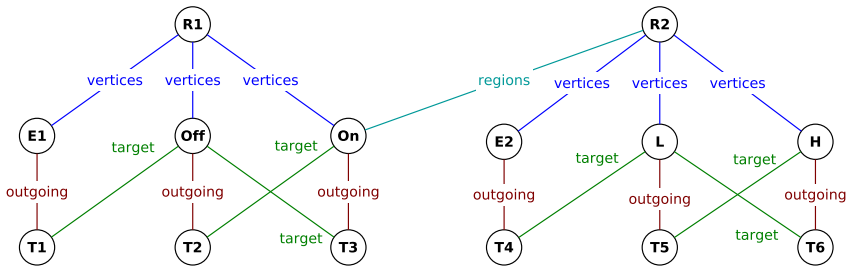


Figure 3.7: Edge-typed graph with inverse edges and edge directions omitted (extracted from the example statechart graph of Fig. 3.1c).

Performance considerations for typed metrics This approach can be adapted to capture *typed local clustering coefficient* metrics. For space considerations, we omit the details of how to adapt the metrics here, but highlight some related findings:

- Due to the presence of types, each matrix only represents a subset of the graph. Consequently, matrices are more sparse then for untyped graphs and matrix multiplication operations are often cheaper to execute.
- For typed metrics, the number of required matrix multiplications is significantly larger. For example, TCC_1 uses *two-triangles*, which have to be determined for each different type pair and therefore necessitate $|T|\cdot(|T|-1)$, i.e. $\mathcal{O}\left(|T|^2\right)$, matrix multiplications. TCC_2 uses *three-triangles* and therefore requires $\mathcal{O}\left(|T|^3\right)$ multiplications.

- As of 2019, there are very few well-optimized programming libraries for sparse matrices.[6] While matrix libraries are typically able to process *dense matrices* in parallel, they often lack support for multi-threaded operations on *sparse matrices*. However, this can be worked around by running multiple, single-threaded matrix multiplication operations at the same time. As the number of operations is (at least) a quadratic function of the number of types, 10+ types are enough to fully stress even a 64-core server CPU.

3.7 Evaluation

We calculated the values of typed metric of Sec. 3.3 for every instance model which yielded over 160 million data records as input for our analysis.[7] In this dissertation, we present selected plots that demonstrate interesting findings. The complete data sets and detailed plots are available online.[8]

In the following, we investigate three research questions, which are highly important for (1) understanding the structural differences between real vs. synthetic models and (2) parameterizing future model generators to create realistic models.

3.7.1 Which Metrics Are Characteristic?

For describing the structure of model graphs, we consider a metric *characteristic* if it has both of the following properties.

- *Homogeneity:* models *within* a specific domain have similar distribution of this metric.
- *Distinctiveness:* models from *different domains* can be distinguished based on their distribution of this metric.

Metrics ranking high in one aspect do not necessarily perform well in the other one: values belonging to even very narrow ranges can overlap entirely with each other (indicating indistinguishable domains) and diverse domains can be separated efficiently, if they are different enough.

Tab. 3.4 contains the *homogeneity values* for each metric–domain pairs. Cells with a black background indicate that the metric is highly homogeneous within a certain domain, while white cells mark that it is heterogeneous. Grey cells usually indicate domains containing outlier models which do not fall into previous categories (with homogeneity values between 0.3 and 0.7). For example, AutoFOCUS and Yakindu models are heterogeneous along each type-related metric, while Train Benchmark models are highly homogeneous here (as expected) due to their synthetic nature.

Fig. 3.8 summarizes the *distinctiveness* of metrics for each pair of domains. Red cells indicate that a certain domain pair can be separated with a high confidence using the metric (e.g. by visually inspecting the shape of their distribution or applying unsupervised learning algorithms), while black cells indicate indistinguishable domain pairs. For example, Capella and JaMoPP models have similar characteristics in edge-related metrics such as *NTC* and *PTC* but can be distinguished based on their *TPC* distribution.

In Tab. 3.4 and Fig. 3.8, the metrics are ranked by homogeneity and distinctiveness. TCC_2 ranks high in both properties, which makes it the *best candidate for domain characterization*. Models of real domains are entirely homogeneous in TCC_1 due to the dominance of zero values (99-100%) in their distributions. Therefore, it is not useful for distinguishing domains in general, even if some models have shown completely different values (e.g. Train Benchmark models because of their tightly connected structure). Some metrics, e.g. *Degree*, perform poorly in both properties.

[6] https://softwarerecs.stackexchange.com/questions/51330/sparse-matrix-library-for-java
[7] We omitted the *interdependence* metric due to its high computational complexity.
[8] http://docs.inf.mit.bme.hu/model-metrics/

Name	AutoFOCUS	BIM	Capella	JaMoPP	Train Benchmark	Yakindu
Typed local clustering coefficient 1	0.06	0.00	0.04	0.00	0.02	0.00
Typed local clustering coefficient 2	0.21	0.01	0.27	0.14	0.02	0.00
Clustering coefficient	0.21	0.28	0.19	0.14	0.02	0.00
Typed participation coefficient	0.55	0.78	0.30	0.42	0.01	0.29
Pairwise type activity	0.60	0.39	0.30	0.21	0.41	0.52
Node type connectivity	0.86	0.63	0.42	0.33	0.42	0.50
Node exclusive type connectivity	0.86	0.63	0.42	0.33	0.42	0.50
Node type activity	0.86	0.63	0.42	0.33	0.42	0.50
Edge type connectivity	0.82	0.59	0.40	0.33	0.42	0.76
Node activity	0.98	0.96	0.64	0.51	0.01	0.12
Degree list	0.99	0.99	0.92	0.99	0.99	0.99

Table 3.4: Summary of metric homogeneity (see Sec. 3.7.3).

Fig. 3.9 presents the empirical cumulative distribution functions of two extrema TCC_2 (left) and *Degree* (right). Except for AutoFOCUS models, the domains are mainly distinguishable using TCC_2. Inability of *Degree* for characterization is clearly noticeable in the figure: the distributions between different domains overlap significantly, making separation impossible.

3.7.2 How Do Domains Differ?

Some metrics turned out to be useful also for describing models by revealing structural characteristics or hidden properties. During the analysis, we made the following domain-specific observations.

Clusteredness The metrics indicating the number of triangles have significant differences across domains. Yakindu and Train Benchmark models represent the two extrema: while the former ones have almost exclusively zero *TCC* values resulting in an average value of 0.008, the latter ones have an average of 0.38. This is caused by the structural properties of the Train Benchmark (Chapter 5): railway segments and their sensors are tightly connected leading to many triangles.

Dominant types All domains contain a small set of dominant types, with at most four of them covering 80% of the graph. In particular, BIM and JaMoPP models contain a single type covering 40-50%. For BIM, these edges encode the layout of the buildings, while for JaMoPP, they form the containment hierarchy following the AST.

Dominance of containment edges We categorized types by splitting them in two groups indicating whether they represent a containment relation or not. *Containment edges* are the structural building blocks of models in software and systems engineering, while *non-containment edges* represent other semantic information between model elements.

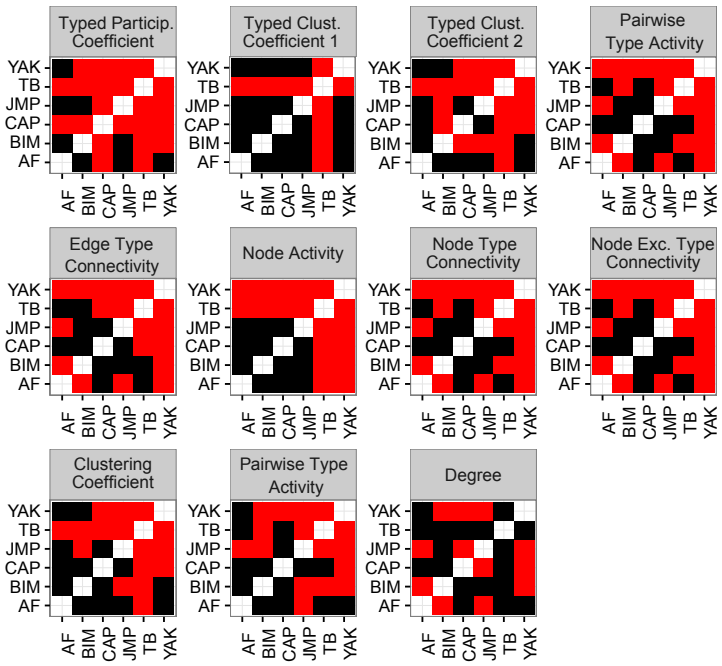


Figure 3.8: Summary of the *distinctiveness* of a given metric (Sec. 3.7.3). *Notation* – AF: AutoFOCUS, CAP: Capella, JMP: JaMoPP, TB: Train Benchmark, YAK: Yakindu.

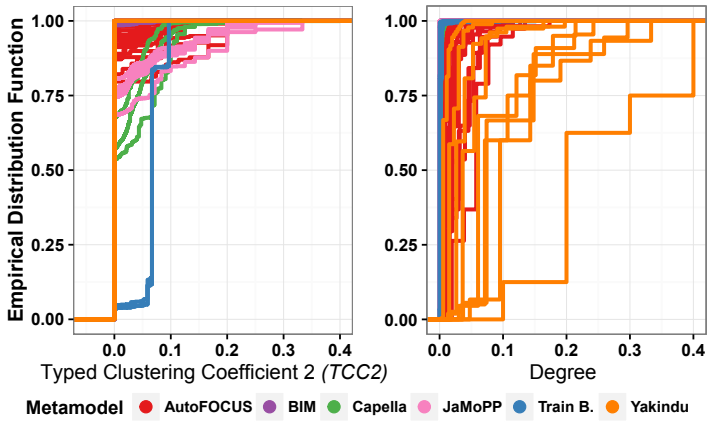


Figure 3.9: Distributions of TCC_2 and *Degree* values.

We found that the ratio of containment edges vary drastically across domains, e.g. it is approx. 45% in case of Capella and 80% in case of JaMoPP models. Moreover, there is no obvious relationship between the ratio of containment types in the metamodel and the containment edges in the instance models, thus metamodels in themselves are insufficient sources for characterizing realistic models.

Fig. 3.10 shows the *NTC* values for the containment and non-containment types. For real models, there are obvious differences in the non-containment subnetworks, only the synthetic Train Benchmark models provide identical characteristics. We observed significant differences even in the containment types for AutoFOCUS and Yakindu models. Although BIM models have no explicit containment edges, some of their types provide extreme similarity to containment subnetworks in other models, e.g. JaMoPP.

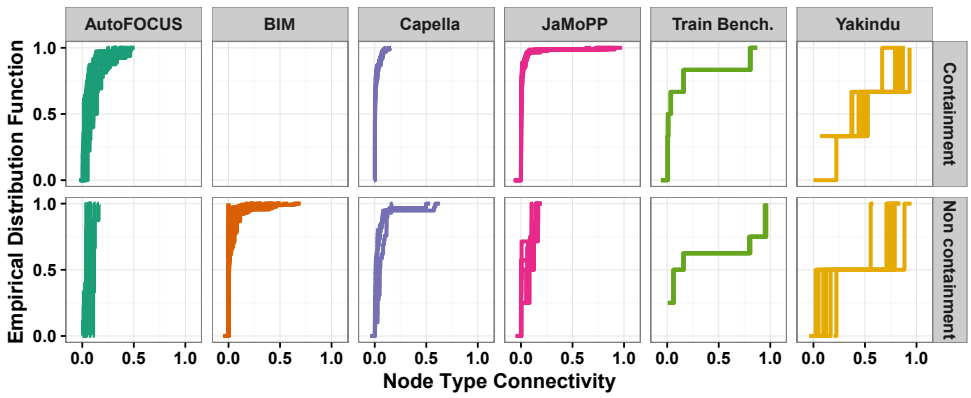


Figure 3.10: Distribution of *NTC* values in containment and non-containment subgraphs.

3.7.3 Statistical Methodology

In order to characterize homogeneity and distinctiveness, we needed to compare the distribution functions for each metric. We first used visual data analysis techniques, which falls under the umbrella of *exploratory data analysis*, to discover potential candidates. Then, we used *confirmatory data analysis* techniques to objectively characterize the similarity of distribution functions. Namely, we calculated the Kolmogorov–Smirnov statistic (*KS*) as a distance measure of graph models to judge how realistic a generated graph model is by comparing the whole distributions of values (and not only descriptive summary metrics like mean or variance) in different cases to the characteristics of real graph models.

The *KS* statistics quantifies the maximal difference between the distribution function lines at a given value. It is sensitive to both shape and location differences: it takes a 0 value only if the distributions are identical, while it is 1 if the values of models are in disjoint ranges (even if their shapes are identical). For comparing two set of models, M_1 and M_2, we took the average value of the *KS* statistics between each (m_1, m_2) pair of models that were generated by technique M_1 and M_2, respectively.

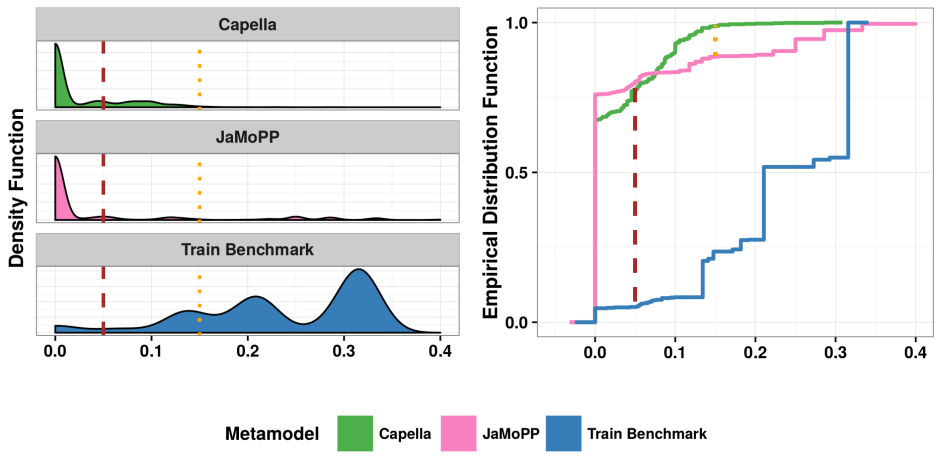


Figure 3.11: Comparing model pairs using the Kolmogorov–Smirnov distance in TCC_2.

Fig. 3.11 illustrates the *KS* distance of TCC_2 distributions of models originating from three different domains (density functions on the left, empirical cumulative distribution functions on the right). *KS* distance between JaMoPP and Train Benchmark is 0.8, reflecting that their lines are far from each other: while 74% of JaMoPP values are smaller than 0.05 (this is the value where the difference is the

largest, marked with the red dashed line), 90% of Train Benchmark values lay above this threshold. On the contrary, the maximal distance between Capella and JaMoPP models is only 0.1 (orange dotted line). Based on this distance function, we defined homogeneity and distinctiveness as follows.

- *Homogeneity of a domain* is calculated as the ratio of the maximal *KS* distance within the domain and the maximal distance across each model pair; it is 0 if every model of the domain has an identical distribution and 1 if this domain spans across the entire metric space.
- *Distinctiveness of domain pairs* is calculated as the average *membership confidence* of their models, which is the ratio of domain-identical instances (i.e. models from the same domain) in its *Degree*th neighbourhood, using the idea of *kNN* classification methods [Wu+08]. Distinctiveness is 1 if the minimal inter-cluster distance is larger than each *Degree*th intra-cluster distance and decreases with every pair of models, which, while belonging to different domains, produce a smaller distance to each other than to their domain-identical neighbours.

Tab. 3.4 contains the homogeneity values of the domains. Distinctiveness is computed with a *Degree* of 2, cells of Fig. 3.8 are coloured red if their distinctiveness is 1.

3.7.4 Threats to Validity

Metrics cannot capture semantics The metrics used in this work describe the structure of the models, thus they cannot explicitly express semantic content. However, the semantics of different domains may be captured in a significantly different way, which further hinders the characterization from different domains.

How real are our models? We used a variety of sources to gather models for analysis. BIM models are real models obtained from an industrial partner. JaMoPP models are generated from open-source code repositories, this way they are real large models. For Capella, AutoFOCUS and Yakindu, we used openly available tutorial models provided by the tool developers themselves who are experts in their domain. As an intentional exception, the Train Benchmark models are fully synthetic as they were created by the model generator of the benchmark.

3.7.5 Summary of Findings

Our analysis provides some insights that need to be considered in future generators to synthesize realistic models.

1. Relying only upon metamodel-level information is clearly insufficient, real instance models of human engineers are required to characterize the domain.
2. Containment edges frequently dominate distributions, which necessitates data preparation (Sec. 3.6). However, such edges can be exploited for generating realistic graph models.
3. Many edges follow the locality principle, i.e. they often lead to neighbouring nodes (and not to distant ones).
4. Characteristic metrics can be used as an objective function for a search-based graph model generator.

3.8 Optimization of Graph Metrics with Linear Algebra

The evaluation presented in this chapter targeted graph models with at most 10M nodes and less than 200 types (see Tab. 3.3). While this size range covers the majority of systems and software models, larger ones are increasingly common [Sch+12]. Additionally, property graphs in graph databases and

semantic graphs in triplestore often have 100M+ nodes and 1B+ edges, along with thousands of types. Our future analyses aims to incorporate such models, hence it is important to investigate potential optimizations. In this section, we will focus on the *untyped local clustering coefficient* (Def. 38), where we found that even for graphs with 10-100M nodes, naïve approaches fail to complete in days.

First, we discuss the *global triangle count* metric, which is typically used in similar application areas as the clustering coefficients: in social network analysis, recommendation systems, etc. This metric is already very challenging to calculate [Lat08]: numerous papers tackle the challenge of exact triangle count using linear algebraic techniques [ABG15; WBS15], while other works consider approximation techniques for improved performance [HD18]. An even more challenging variant is the *triangle enumeration* [ABG15] problem, also known as *triangle listing* [Zin+16], which requires the algorithm to explicitly enumerate each triangle.

To allow efficient computation of the *clustering coefficient*, we rewrote this metric in the language of linear algebra [KG11] and represented the *adjacency matrices* in the computation as *sparse matrices* to reduce memory requirements. We review related definitions in Sec. B.1. In the following, we present two algorithms: the naïve and an optimized versions. In both cases we will store the resulting *(untyped) local clustering coefficient* values in the $\overrightarrow{LCC}$ vector, which consists of n elements, and its ith element denotes the LCC value of node v_i.

Naïve algorithm The naïve algorithm to determine $\overrightarrow{LCC}$ is as follows. First, it calculates A^3 to get all three-length paths in the graph, then it keeps the ones that start and end in the same node. These are the ones with the same row and column index in the A^3 matrix, i.e. the *diagonal elements* (Def. 78). Next, calculate A^2 to get all two-length paths (*wedges*) and subtract the diagonal elements as they represent back and forth hops in the graph and not actual wedges. To determine the sum of each row, we multiply the matrix by a vector of ones [WBS15]:

$$\left(A^2 - \operatorname{diag}\left(A^2\right)\right) \cdot \vec{1}$$

Finally, to get the $\overrightarrow{LCC}$ vector, we perform an *element-wise* division (Def. 76) on the vectors containing the number of triangles and wedges.

$$\overrightarrow{LCC} = \operatorname{diag}^{-1}\left(A^3\right) \oslash \left(\left(A^2 - \operatorname{diag}\left(A^2\right)\right) \cdot \vec{1}\right).$$

It is easy to see that this approach introduces a large amount of redundancy: #1 only the elements in the diagonal of A^3 are used, and #2 the algorithm enumerates all wedges by calculating A^2, when only the number of wedges centered in a given node is of interest. Based on these observations, we present an optimized version of the algorithm.

Optimized algorithm To improve performance, we address redundancy #1 by only producing two-length paths (*wedges*) with A^2 and close them with an *element-wise multiplication* (Def. 75) similarly to Cohen's algorithm [Coh09; ABG15]. We also make use of the fact that it is not necessary to enumerate all triangles, but only to determine *node-wise triangle count*. Therefore, we can simply summarize each row in the resulting matrix by multiplying it with $\vec{1}$.

$$\operatorname{diag}^{-1}\left(A^3\right) = A^2 \odot A \cdot \vec{1}$$

For redundancy #2, it is sufficient to calculate the *degree* of each node with $A \cdot \vec{1}$ and determine the number of wedges as the number of 2-combinations of its degree ($\vec{d} = A \cdot \vec{1}$).

$$\left(A^2 - \operatorname{diag}(A)\right) \cdot \vec{1} = \begin{bmatrix} \binom{\vec{d}_1}{2} \\ \binom{\vec{d}_2}{2} \\ \vdots \\ \binom{\vec{d}_n}{2} \end{bmatrix}$$

Using the equality

$$\binom{k}{2} = \frac{k \cdot (k-1)}{2},$$

we can reformulate the expression as follows:

$$\begin{bmatrix} \binom{\vec{d}_1}{2} \\ \binom{\vec{d}_2}{2} \\ \vdots \\ \binom{\vec{d}_n}{2} \end{bmatrix} = \begin{bmatrix} \frac{\vec{d}_1 \cdot (\vec{d}_1 - 1)}{2} \\ \frac{\vec{d}_2 \cdot (\vec{d}_2 - 1)}{2} \\ \vdots \\ \frac{\vec{d}_n \cdot (\vec{d}_n - 1)}{2} \end{bmatrix} = \vec{d} \odot \left(\vec{d} + (-1)\right) \cdot \frac{1}{2} = \left(A \cdot \vec{1}\right) \odot \left(A \cdot \vec{1} + (-1)\right) \cdot \frac{1}{2}$$

Putting these together, we derive the following expression:

$$\overrightarrow{LCC} = \left(A^2 \odot A \cdot \vec{1}\right) \oslash \left(\left(A \cdot \vec{1}\right) \odot \left(A \cdot \vec{1} + (-1)\right) \cdot \frac{1}{2}\right)$$

Note that the element-wise division operator should be interpreted with care: for operations where the divisor is 0, the results should be 0. While this approach performs significantly less work than the naïve algorithm, it is still suboptimal as it requires the enumeration of all wedges (most of which typically cannot be completed to a triangle). The class of *worst-case optimal algorithms* eliminate this redundancy and might offer better performance. In fact, the authors of paper [Zin+16] used the worst-case optimal Leapfrog Triejoin algorithm for triangle enumeration, and created implementations on both CPUs and GPUs.[9] A deeper relationship between *matrix multiplication*, *triangle queries* and *worst-case optimal joins* is discussed in Sec. 10.6.

3.9 Related Work

3.9.1 Graph Analytical Engines

Graph analytical engines can be divided to two main categories, based on their programming model. The first one is the vertex-centric programming model which defines the computation as an iterative process between communicating vertices in the graph, defined in the Pregel model [Mal+10]. The second one is the linear algebra-based approach which defines the computation with matrix operations.

Vertex-centric approaches The vertex-centric model is used in the Apache Spark GraphX [Gon+14], Apache Flink Gelly [Car+15], and the Apache Giraph [Sak+16] frameworks. GPS (Graph Processing System) [SW13] is a graph analytical engine for scalable, fault-tolerant, and simple-to-program implementation of algorithms on large graphs. OpenG consists of hand-coded implementations for numerous graph algorithms. It is used by the *GraphBIG* [Nai+15] benchmark, an effort initiated by Georgia Tech and inspired by IBM's *System G*. A detailed survey of vertex-centric approaches is given in [MWM15], while an analysis of their scalability is discussed in [AÖ18].

[9] This algorithm is also relevant for incremental view maintenance on graphs, see Sec. 10.3.

Linear algebra-based approaches A notable recent initiative is the GraphBLAS[10] [Kep+15; Kep+16] specification, which defines standard building blocks that allow users to create graph algorithms with linear algebra. GraphBLAS is currently available as a C library, and as part of the Graphulo data processing system [Hut+16; Hut17]. Additionally, wrappers in high-level programming languages such as PyGB for Python [Cha+18] are under development. A distributed graph analytical engine was presented in [Ahm+18] under the name "LA3", which refers to the fact that the system is link-aware, locality-aware, and linear algebra-based.

Hybrid approaches Some advanced engines mix vertex-centric and linear algebra-based approaches. Gunrock [Wan+17] is a GPU-based graph analytical framework, which allows its users to define algorithms using high-level primitives (e.g. neighbourhood expansion, filtering, and frontier intersection), then translates these to low-level, linear algebra-based GPU operations. GraphMat [Sun+15b] is a graph analytical system developed by Intel. It uses a vertex-centric programming model, and translates analytical workloads to linear algebraic operations, primarily to SpMV (sparse matrix to vector multiplication).

Unified analytical and query engines Oracle PGX.D [Hon+15] is a distributed hybrid graph analytical and graph query engine. Users can specify graph analytical tasks in the *Green-Marl* domain-specific language [Hon+12] (Sec. B.3) and graph queries in PGQL (Sec. 2.6.2). Currently, the system is capable of supporting complex queries such as a large portion of the LDBC SNB's Business Intelligence workload (see Sec. 6.4). An interesting approach is taken by paper [Lin+16] which envisions a scenario where the data resides in a relational database, and a layer is built on top that utilizes a graph engine *to process SQL queries efficiently*.

Surveys Monograph [Yan+17] reviews various algorithms and techniques for graph analytics. A recent survey of distributed graph processing frameworks was presented in [KVH18], concluding that most current graph analytical systems use the vertex-centric programming model (as opposed to the linear algebra-based model).

3.9.2 Benchmarks for Graph Analytics

In the following, we provide a brief overview of the benchmarks that target *graph analytical workloads* (Sec. 2.7.2). It is interesting to observe that the number of benchmarks for this workload is noticeably lower than those for *graph query workloads* (presented in Chapter 7).

HPC SGAB Traditionally, many benchmarks for graph analytical workloads were conceived in the *high-performance computing* (HPC) community. A DARPA program titled *High Productivity Computing Systems* (HPCS) aimed to define a way to measure various metrics in the high-performance computing domain. Their target metrics include programmability, portability, robustness, productivity, and *performance*. As part of this programme, an initiative designed the Synthetic Compact Applications (SSCA) benchmark suite, whose SSCA#2 is a graph theoretical problem [KK05]. The authors of paper [BM05] presented a multi-processor implementation for this problem, and subsequently joined forces with the SSCA#2 team, resulting in updated versions of the specification [Bad+07], and the full *HPC Scalable Graph Analysis Benchmark* specification [Bad+09]. The benchmark requires its users to develop an application that has multiple analysis techniques, operating on a weighted, directed graph.

[10] http://graphblas.org/

		Metrics												
Benchmark	Reference	betweenness centrality	breadth-first search	community detection by LP	colouring	degree centrality	depth-first search	Gibbs inference	k-core decomposition	local clustering coefficient	PageRank	(single source) shortest path	triangle count	(weakly) connected components
HPC SGAB	[BM05]	⊗	○	○	○	○	○	○	○	○	○	○	○	○
"Big Data technologies"	[EM13]	○	○	○	○	○	○	○	⊗	○	○	○	○	○
GraphBIG	[Nai+15]	⊗	⊗	○	⊗	⊗	⊗	⊗	⊗	○	○	⊗	⊗	⊗
Graph500	–	○	⊗	○	○	○	○	○	○	○	○	⊗	○	○
GraphBench	[Suk+16]	⊗	⊗	○	○	○	○	○	○	○	⊗	⊗	⊗	○
LDBC Graphalytics	[Ios+16]	○	⊗	⊗	○	○	○	○	○	⊗	⊗	⊗	○	⊗

Table 3.5: Benchmarks for graph analytics.

The benchmark contains a graph generator based on the *Recursive MATrix* (R-MAT) scale-free graph generation algorithm [CZF04]. The benchmark tasks include classification, graph extraction, and calculating the *betweenness centrality* metric. Due to the high complexity of calculating this metric, the benchmark specification allows users to create both *exact* and *approximate* implementations.

Big Data technologies for graph analytics Paper [EM13] studied big data frameworks and their ability to implement graph algorithms. Namely, the authors selected the *k-core decomposition problem* [MPM11], which computes the centrality of each node by identifying the maximal induced subgraphs including that node. The problem takes untyped, undirected graphs as its input, and assigns a k value to each node in the graph. The paper presented implementations on multiple frameworks (Giraph [Sak+16], Hadoop [Whi15], Hama [Sid+16], etc.), and concluded that graph-specific frameworks clearly outperforms the ones designed for generic data processing tasks.

GraphBIG GraphBIG [Nai+15] is an extensive graph analytical benchmark suite inspired by the IBM System G project [Tan+14]. The benchmark was designed based on data structures, workloads, and data sets from 21 use cases in multiple real-world application domains. Its workload consists of 4 categories and 13 operation types in total:

- Graph traversal: breadth-first search, depth-first search;
- Graph update: graph construction, graph update, topology morphing;
- Graph analytics: shortest path (Def. 25), k-core decomposition, connected components (Def. 60), graph colouring, triangle count (Def. 35), Gibbs inference;
- Social analysis: degree centrality, betweenness centrality (Def. 64).

12 operations were implemented on CPUs and 8 were implemented on GPUs. Following an extensive evaluation on multiple systems, authors concluded that neither hardware architecture performs

well due to the highly irregular access patterns exhibited by graph data sets, as both CPU and GPU systems showcased significant inefficiencies in memory access and bandwidth utilization. The authors also observed that distributions of the input data has a profound impact on performance, but the impact is difficult to assess due to the diversity of the workload operations.

Graph500 Graph500[11] is a collaboration of HPC experts from industry and academia with the goal to establish a set of large-scale benchmarks for these applications. It provides a highly scalable Kronecker generator similar to the Recursive MATrix (R-MAT) graph generation algorithm [CZF04], which produces weighted, undirected graphs up to 1.1 PB in size (approx. 10^{14} edges). It covers two operations, *breadth-first search* (BFS) and *single source shortest paths* (SSSP), and continuously maintains a list of top performers.

GraphBench GraphBench[12] [Suk+16] is a community-driven graph benchmark suite. It contains a generator based on the Graph500 Kronecker generator and measures the performance of the *breadth-first search* (BFS) and *single source shortest paths* (SSSP), PageRank, triangle counting, and betweenness centrality operations.

LDBC Graphalytics Paper [Guo+14] presented a vision towards designing a comprehensive benchmarking suite for graph processing platforms. This vision was realized with the introduction of the Graphalytics benchmark suite [Cap+15; Ios+16], which defines a workload targeting scalable analysis of large graphs with weighted edges. It requires tools to calculate six popular graph metrics:

1. BFS (breadth-first search),
2. CDLP (community detection by label propagation),
3. LCC (local clustering coefficient, see Def. 38),
4. PR (PageRank),
5. SSSP (single-source shortest path),
6. WCC (weakly connected components, see Def. 61).

Graphalytics is an influential and popular benchmark with implementations ranging from graph databases to distributed graph processing frameworks, including GraphMat [Sun+15b], GraphX [Xin+13; Gon+14], OpenG [Nai+15], and PGX [Hon+15].

Lack of benchmarks for typed graphs None of the benchmarks surveyed here consider any type information in the graph (nodes and/or edges). Up to our best knowledge, currently there is no benchmark available for measuring the performance of calculating typed graph metrics. This indirectly confirms that *typed graph analysis* is still a new field and there is no common understanding on (1) which metrics are of significant importance and (2) how to perform such analyses efficiently.

3.9.3 Metrics for Typed Graphs in Other Fields

A collection of typed metrics is defined in [Ber+13; NL15; BNL14] where the authors study the expressiveness of their metrics on real-life networks from heterogeneous domains e.g. from a social network (Flickr), co-authorship data (DBLP), query log analysis, social, engineering and biological networks. A detailed discussion of paper [BNL14] is presented in dissertation [Bat17].

[11] https://graph500.org/
[12] https://github.com/uwsampa/graphbench

Network science Revealing essential structural similarities and differentiations among networks from different domains is a fundamental objective in network theory. Such studies [AB02; Cos+11] characterize a diverse set of models from a number of domains. However, these studies are carried out on untyped networks. So far, existing typed studies only focused on a single application domain, such as neighbourhood and centrality analysis of a Polish social network [Bró+12], relevance and correlation analysis of different types in Flickr [KMK11], community detection in the network of YouTube [TWL12], analysis of co-authorship in the DBLP network [BSK11] and characteristics of different transportation networks (European Air Network [Car+13], cargo ship movements [Kal+10]). Metrics for social network analysis are used in the *Corese semantic search engine* [Eré+09], including variants of well-known metrics that inspect a subgraph induced by edges of a given type. Such metrics include *degree*, *geodesic* (Def. 25), *betweenness centrality* (Def. 64), *diameter* (Def. 67), and *density* (Def. 68–69).

Network analysis in software engineering The authors of [Bha+12] use graph metrics to capture the structure and evolution of software products and processes in order to detect significant structural changes, help estimate bug severity, prioritize debugging efforts, and predict defect-prone releases in software engineering. Additionally, the principles of complex networks are used to measure the structural complexity of software systems [MHD05; Ma+06b] and to predict defects on dependency graphs [ZN08]. Our motivation is to find metrics that are able to characterize and distinguish models used in tools of software and systems engineering.

Metrics in model-driven engineering The analysis of domain-specific graph models has been studied in MDE. Fellow researchers of the Fault-Tolerant Systems Research Group [Izs+13b] investigated the correlation between model query performance and metrics describing the queries and the models. They introduced composite metrics such as the *absolute difficulty* (logarithm of the search space size), and the *relative difficulty*, which expresses how much worse a query engine does than the theoretical lower bound required by a certain query. The authors generated 12 graphs with different degree distributions along with 25+ queries of different shapes and measured their execution times. Then, they calculated the Kendall's τ rank correlation coefficient for $p < 0.001$ between each metric and the query execution times. The strongest correlation ($+0.38$) was exhibited by the *absolute difficulty* metric.

The authors of [Roc+14] use metrics to understand the main characteristics of domain-specific metamodels and to study model transformations with respect to the corresponding metamodels, and search correlations between them via analytical measures [Roc+15]. A generic σ-metric is proposed in [Mon+08] to assist in empirically validating various quality attributes. Finally, several approaches exist to define metrics using high-level constraint languages [Chi11; GDL08]. The main novel aspect of our work is to identify characteristic graph metrics for describing *real instance models* on a statistical basis to help develop future model generators.

3.10 Conclusion and Future Work

Conclusion In this chapter, we identified several graph metrics known from other disciplines and evaluated them on 83 instance models of 6 different tools dominantly from software and systems engineering domains in order to identify *characteristic* metrics using statistical and visual data analysis techniques. We consider a metric characteristic if it separates models of different domains from each other, while provides similar values for models within the same domain. We also discussed whether

some of these metrics can *distinguish real models from auto-generated synthetic ones*, which is the first investigation of graph models for such a purpose up to our best knowledge. Our initial finding is that different versions of clustering coefficients were particularly useful for such classifications. But, unsurprisingly, no single metric was able to sufficiently handle all the domains.

Future work The lack of triangles in our example statechart graph (Fig. 3.1c) clearly exemplifies why it is insufficient to only look for clusters of three nodes: many graphs only have circles of more than 3 elements. Hence, we plan to study the k-local clustering coefficient [JC04; Fro+02] and introduced the k-*TCC* metric, a common generalization of k-*LCC* and the *TCC* metric. *Labelled graphs* (Def. 6) have been studied as *heterogeneous information networks* over the last decade in database literature [Shi+17]. A central concept of this research was the *meta-path*, a path travelling over a given sequence of node and edge types. While this is slightly related to the *TCC* metric (as a three node meta-path which has the same start and end nodes is a typed triangle), it is also able to uncover different sort of structural properties of the graph. This makes the *meta-path* a promising candidate for our future analyses.

Chapter 4

Characterizing the Realistic Nature of Graphs

In Chapter 3, we presented a set of metrics to characterize typed graphs. These metrics allow us to gain greater insight into the structure of a given set of graphs, which unlocks further research directions:

1. It allows us to argue for/against the realism of a given set of synthetic graphs.
2. It aids generating synthetic graphs that are structurally similar to real ones.

In this chapter, we present our results for #1. Investigating techniques to generate realistic synthetic graphs (#2) is subject to future work. The work described here was presented in paper [c5] and in book chapter [b1].

4.1 Characterizing Randomized Graph Models

As one of our long-term research objectives is to generate realistic instance models, we attempted to identify a set of metrics which are able to capture the characteristics of real models.

4.1.1 Experimental Setup

We compared real and synthetic instance models from four domains presented in Sec. 3.6. We chose domains where real instance graphs were available and had a well-defined containment hierarchy (AutoFOCUS, Capella, JaMoPP, and Yakindu).[1] We created synthetic models with a random graph model generator, using the following approach:

1. We removed all edges from the graph model that are not part of the containment hierarchy, leaving only the containment edges.
2. For each removed edge, we inserted a new edge of the same type. The start and end nodes of the new edge were chosen using a pseudo-random generator with a uniform distribution from the nodes which fulfil the type constraints prescribed by the edge type.

Note that compared to a fully random model generator, our setup presents a more adverse situation for a metrics-based distinction since a significant part of the graph models (the containment hierarchy) remains real.

[1] The BIM models did not have a clear containment hierarchy.

4.1.2 Evaluation

We calculated the metrics on both real and synthetic models. Fig. 4.1 illustrates the influence of randomization on two metrics, *TCC* and *PTC* on Capella models. We found that while many metrics did not exhibit any particular difference (see e.g. *PTC* in the right part of the figure), clustering metrics such as *LCC* and TCC_2 showed significant changes both in their ranges (the maximum decreased) and distribution. This change may be explained by the fact that randomization decreased the clusteredness of the graph, as the randomly inserted edges are less likely to form a triangle (compared to a model designed by a domain expert). This phenomenon, well-kwown in network science [BNL14], could be observed in each domain with a strength depending on the number of removed edges. Thus, it was less drastic in Yakindu instances than in large models.

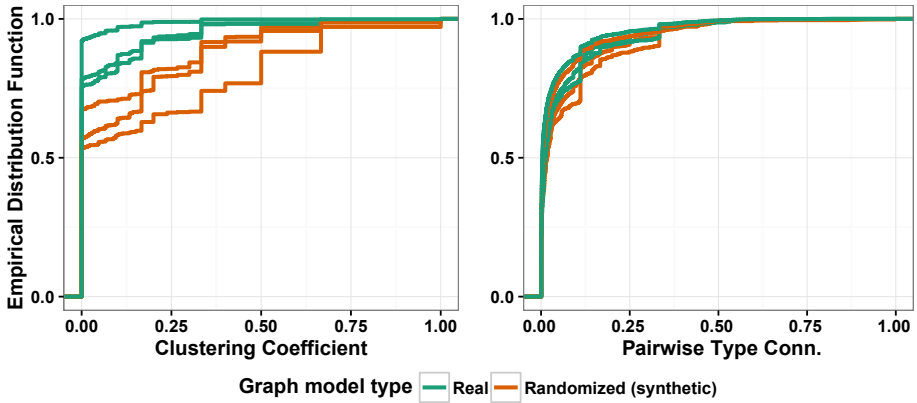


Figure 4.1: The distribution of *LCC* values is noticeably different between the real and randomized version of Capella instance models, while *PTC* values are similar for both versions.

4.2 Characterizing Graphs Synthesized by Solvers

In our next investigation, we studied graph models generated by the Alloy Analyzer [TJ07] and the Viatra Solver, designed by Oszkár Semeráth et al. [b1]. This work focused on studying the properties of a graph generator that targets the synthesis of graph models that are *consistent* w.r.t. a set of *well-formedness (WF) constraints* captured with graph patterns. To characterize the generators, we synthesized graph models describing statecharts over the Yakindu metamodel, similar to the one depicted in Fig. 3.1a. We then carried out an evaluation to address the following research questions:

RQ1 *Realism vs. diversity:* How realistic are the graph models which are synthesized by generators that promise diversity?

RQ2 *Realism vs. consistency:* How realistic are the graph models which are synthesized by generators that guarantee consistency?

Addressing these questions may help advancing future model generators by identifying some strength and weaknesses of different strategies.

4.2.1 Experimental Setup

Target domain We conducted measurements in the context of Yakindu statecharts.[2], with the statechart metamodel extracted directly from the original Yakindu metamodel. We formalized 10 WF constraints based on the validation rules of the Yakindu Statechart development environment.

Graph generator approaches We compared two different model generation approaches:

1. Kodkod, the popular *relational model finder*, which uses the Alloy Analyzer [TJ07], and
2. VIATRA Solver, a *graph-based generator*, which uses the refinement calculus presented in [b1].

We operated both solvers in two modes: in *well-formedd* mode (WF) all synthesized graph models need to be consistent, i.e. satisfy both the structural constraints of the metamodel along with the WF constraints, while in *metamodel-only* mode (MM), generated models only need to satisfy the structural constraints of the metamodel. With these settings, we generated the following four sets of models:

- "Alloy (MM)": 100 metamodel-compliant models with 50 objects using Alloy.
- "Alloy (WF)": 100 metamodel- and WF-compliant models with 50 objects using Alloy (which was unable to synthesize larger models within 1 minute).
- "VIATRA Solver (MM)": 100 metamodel-compliant instance models with 100 objects using VIATRA Solver.
- "VIATRA Solver (WF)": 100 metamodel- and WF-compliant instance models with 100 objects using VIATRA Solver.

To enforce diversity among the models in the same model set, we explicitly checked that generated graph models are non-isomorphic.

Real instance models To evaluate how realistic the synthetic graph model generators are, we took 1 253 statecharts as *real models* created by undergraduate students for a homework assignment. While the students had to solve the same modelling problem, the size of their models varied from 50 to 200 objects. Real models were filtered by removing inverse edges that introduce significant noise to the metrics (see Sec. 3.2 for details).

4.2.2 Analysis of Generated Graph Models

Graph metrics We selected two typed graph metrics from Chapter 3 to evaluate how realistic the models produced by a graph generator are:

1. *typed participation coefficient* (*TPC*, Def. 45) measures how the edges of nodes are distributed along the different edge types, and
2. *pairwise type connectivity* (*PTC*, Def. 56) captures how likely are two different types of edges to meet in a node.

Evaluation of measurement results We plot the distribution functions of the *typed participation coefficient* metric in Fig. 4.2, and the *pairwise type connectivity* metric in Fig. 4.3. Each line depicts a distribution function describing the characteristics of a single graph model. Graph model sets, such as "Alloy (MM)" and "VIATRA Solver (WF)", are grouped together in facets of the plot.

[2] https://github.com/ftsrg/publication-pages/wiki/Towards-the-Automated-Generation-of-Consistent,-Diverse,-Scalable,-and-Realistic-Graph-Models/ contains the detailed results.

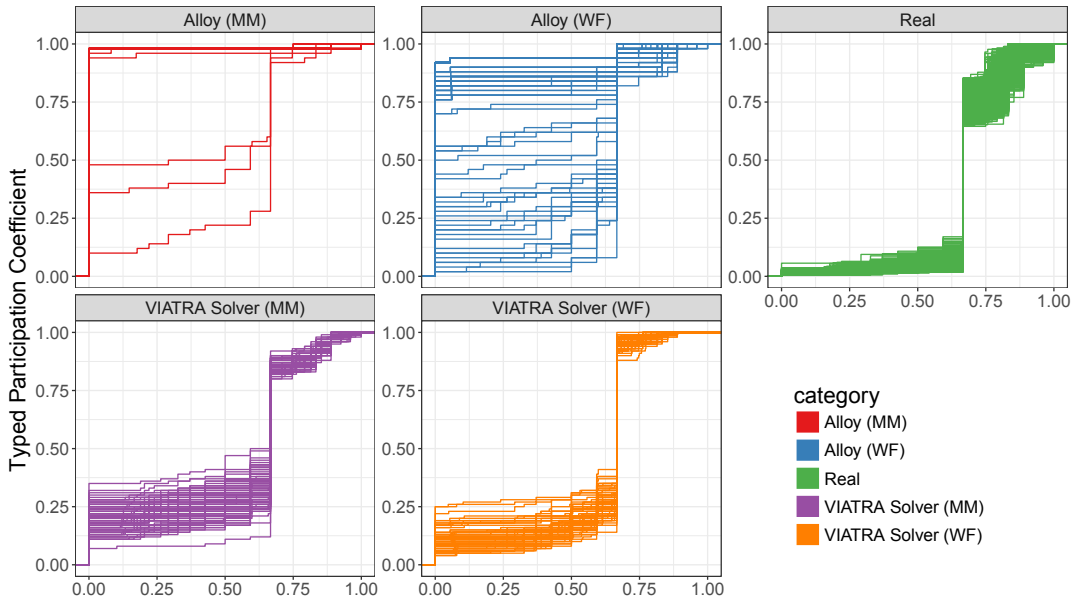


Figure 4.2: Empirical cumulative distribution functions of the *typed participation coefficient (TPC)* metric for real and synthetic graph models. The plot shows that approximately 2/3 of the nodes in "Real" statechart models have edges in only a few types, resulting in a *TPC* value under 0.2. The remaining 1/3 of the nodes have edges more evenly distributed among different types. The graph models produced by "Viatra Solver" (both in MM and WF variants) closely mirror this property, while the ones produced by "Alloy" deviate from it on multiple occasions.

Graph model set	*TPC*	*PTC*
Alloy (MM)	0.95	0.88
Alloy (WF)	0.74	0.60
Viatra Solver (MM)	0.27	0.37
Viatra Solver (WF)	0.24	0.30

Table 4.1: Average Kolmogorov–Smirnov statistics between the *TPC* and *PTC* distribution functions of real and generated graph model sets. Smaller values indicate more similarity between the distribution functions. The values show that compared to real graph models, "Alloy (MM)" produces the least similar ones, while "Viatra Solver (WF)" produces the most similar ones.

Comparison of distribution functions To characterize how realistic a generated graph model is, we used the Kolmogorov–Smirnov statistic (*KS*) as a distance measure of graph models, similarly to the approach presented in Sec. 3.7.3. The resulting average *KS* values are shown in Tab. 4.1, where a lower value denotes a graph model that is *more similar* to real ones, i.e. it is *more realistic*.[3] Our analysis revealed the following insights.

[3]Due to the excessive amount of homework models, we took a uniform random sample of 100 models.

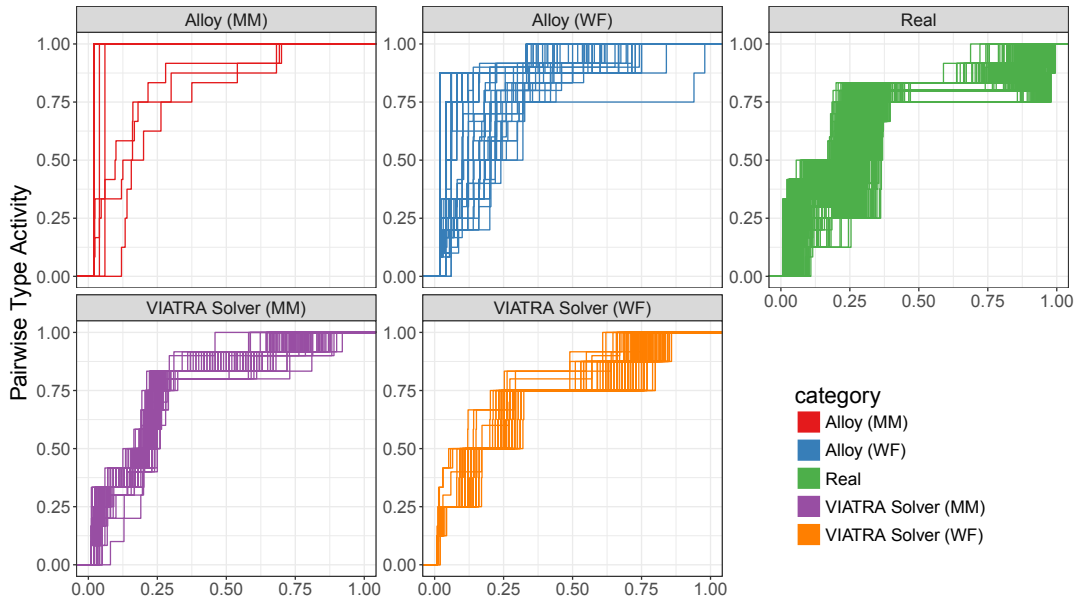


Figure 4.3: Empirical cumulative distribution functions of the *pairwise type connectivity (PTC)* metric for real and synthetic graph models.

Realism vs. diversity For the models that are only metamodel-compliant, the characteristics of the metrics for "Viatra Solver (MM)" are much closer to the "Real" model set than those of the "Alloy (MM)" model set for both metrics (*KS* values of 0.27 vs. 0.95 for *TPC*, and 0.38 vs. 0.88 for *PTC*). This implies that the "Viatra Solver (MM)" setup produced more realistic graph models. However, visual inspection also highlights that the set of generated metamodel-compliant models can be more easily distinguished from the set of real models as the plots of the latter show higher variability. Since the diversity of each model generation case is enforced (i.e. isomorphic models were dropped), we can draw as a conclusion that *a metamodel-compliant model generator does not provide any guarantees in itself on the realistic nature of the output model set.* In fact, graph generators that simultaneously ensure diversity and consistency always outperformed the random model generators (which only confirmed the metamodel) for both the Alloy and Viatra Solver cases. As such, we found that random, metamodel-compliant generators are diverse but *less realistic* if they do not enforce WF constraints.

Realism vs. consistency In case of models satisfying WF constraints, "Viatra Solver (WF)" generated more realistic graph models than "Alloy (WF)" according the *KS* statistics on both metrics. However, the plots show mixed results for differentiating between generated and realistic models. On the positive side, the shape of the plot for generated models is very close to that of real models in case of the *TPC* metric (Fig. 4.2), and have an average *KS* distance of 0.24. However, for the *PTC* metric (Fig. 4.3), real graph models are still substantially different from generated ones, as demonstrated visually on the distribution plots, and confirmed by the higher average *KS* distance of 0.3. Thus further research is needed to investigate how to make consistent models *more realistic.*

Takeaways for graph generators While many existing performance benchmarks claim that they generate realistic graph models, most of them ignore the WF constraints of the domain. According to

our measurements, it is a major drawback since real graph models dominantly satisfy WF constraints while randomly generated models inevitably violate some constraint. Thus, graphs synthesized by generators omitting WF constraints can hardly be considered fully realistic.

Threats to validity We carried out experiments solely in the domain of statecharts which limits the generalizability of our results. Additionally, statecharts are a *behavioural modelling language*, and the characteristics of such graph models (and thus their graph metrics) would likely differ from *structural modelling languages*, such as e.g. the BIM architectural DSL or the Train Benchmark's railway modelling language (Sec. 3.6.1). However, since many of our experimental results are negative for models as simple as Yakindu statecharts, it is unlikely that the Alloy generator would behave significantly better for other domains with more complex domains. In fact, while Alloy has been used as a back-end for mapping-based model generator approaches, we found that its use is not justified from a scalability perspective due to the lack of efficient evaluation for complex structural graph constraints. It is also unlikely that randomly generated metamodel-compliant models (the MM variants produced by the solvers) would be more realistic, or more consistent in any other (often more complex) domain.

Concerning our real graph models, we included all the statecharts created by students, which may be a bias since many of the students who are initially unfamiliar with a modelling tool might not produce good quality models. Thus, the variability of real statechart models created by engineers may actually be smaller. But this would actually increase the relative quality of models generated by Viatra Solver which currently differs from real models by providing a lower level of diversity (i.e. plots of *pairwise type connectivity* are thicker for real models).

4.3 Related Work

4.3.1 Generation of Graph Instances

Several recent works of the database research community targeted *generation* of graph instances.

gMark [Bag+17] is a schema-driven benchmark workload generator. Its approach relies on controlling the diversity of the generated graphs and the complexity of the generated instances, using a *selectivity estimation* algorithm. The graph generator of gMark was extended with the MonStaGen algorithm [Lee+17], which is capable of generating graph sequences ensuring two key properties: (1) "*monotonic containment* of graph instances as they grow in size and (2) *consistency* of structural properties across the sequence".

The Gscaler system [ZT16] aims to synthetically scale a given graph. As its authors state, they work "investigates the Graph Scaling Problem (GSP): Given a directed graph G and positive integers $\widetilde{n}$ and $\widetilde{m}$, generate a similar directed graph $\widetilde{G}$ with $\widetilde{n}$ nodes and $\widetilde{m}$ edges. [...] Analogous to DNA shotgun sequencing, Gscaler, decomposes G into small pieces, scales them, then uses the scaled pieces to construct $\widetilde{G}$."

Paper [Sat+17] investigates the problem of graph synthesis while taking into account the correlation between edges and *"labels"* (defined as attributes with enumeration-like values). More precisely, it aims to "preserve the node label and the edge connectivity distributions as well as their correlation, while also replicating the degree distribution." Its authors "model the edge connectivity by a joint distribution over pairs of label categories", which is somewhat similar to the *pairwise type connectivity* metric used in Sec. 3.5.4, and use Jensen–Shannon divergence statistic [Lin91] to compare distributions (as opposed to the Kolmogorov–Smirnov statistic used in our work Sec. 3.7.3).

The goal of the DataSynth system [Pra+17] "is to assist benchmark designers in generating graphs efficiently and at scale, saving from implementing their own generators. Additionally, DataSynth introduces novel features barely explored so far, such as modelling the correlation between properties and the structure of the graph. This is achieved by a novel property-to-node matching algorithm."

DataSynthesizer (not to be confused with DataSynth) is "a privacy-preserving synthetic data generator designed to facilitate collaborations between domain-expert data owners and external data scientists." [PSH17]. From the same research group, paper [AS18] tackled graph generation challenges, using an attribute-based preferential attachment model.

4.3.2 Domain-Agnostic Characterization of Realistic Graphs

Some works attempted to establish a *domain-agnostic, generic graph metric* to capture the realism of a given graph. Most notably, in their 2011 SIGMOD paper, Duan, Kementsietsidis, Srinivas, and Udrea compared real graphs to synthetic ones used in RDF benchmarks available at the time [Dua+11]. The real graphs in the paper included DBpedia [Biz+09], the Uniprot protein knowledge base [Apw+04], and the Barton library dataset [Aba+07], while the synthetic graphs were the data sets of the Berlin SPARQL Benchmark [BS09], the Lehigh University Benchmark [GPH05], and SP^{2}Bench [Sch+09]. To characterize graphs, the authors study the *structuredness* of datasets by calculating the *coverage* and *coherence* metrics. Based on their analysis, they found that "while real datasets cover the whole structuredness spectrum, benchmark datasets are very limited in their structuredness and are mostly relational-like".

Recently, the analysis presented in [c8] disputed the claim that benchmark datasets are limited in their structuredness. Namely, results on a more comprehensive set of real and synthetic graphs, our analysis shows that benchmark datasets are not necessarily limited in their structuredness, but cover the whole structuredness spectrum. This renders *structuredness* less useful for characterizing the realism of a given graph.[4] Whether there is a *generic metric* to capture the realism of graphs is still an open research problem.

4.4 Conclusion and Future Work

In this chapter, we demonstrated how typed graph metrics can be used to distinguish between real and generated graph instances. Besides *descriptive* purposes, these metrics can also be used for *prescriptive* goals. In particular, a major motivation for analyzing graphs with such metrics is to allow us to generate synthetic graph instances that are structurally similar to real graphs. To this end, we are currently experimenting with a *design-space exploration* (DSE) approach that grows graphs based on a set of *transformation rules*, against a set of *well-formedness constraints* and *metric values* as optimization goals. Such multi-objective optimization problems can be supported with genetic algorithms, an approach used by the Viatra-DSE system [Abd+14]. The performance of this system – in the context of a software optimization case – was demonstrated at the 2016 Transformation Tool Contest, where our submission achieved 1st place by simultaneously providing good performance and high-quality solutions [a25].

[4] This finding is a contribution of the first author of paper [c8]. The author of this dissertation is a co-author of the paper and made contributions towards the analysis of the Train Benchmark and LDBC Social Network Benchmark workloads (see Chapter 5 and Chapter 6, respectively).

Part II

Benchmarks for Global Queries over Evolving Property Graphs

Benchmarks

In the area of data management systems, benchmark designers typically design *macrobenchmarks* [BS97; Sel+99], which define complex workloads focusing on a certain application domain.[5] For example, the *Transaction Processing Performance Council* (TPC) [PF00] released multiple TPC benchmarks since its establishment in 1988, including ones that have influenced relational data processing systems for decades such as TPC-C, TPC-H, and TPC-DS [NP06; Pös17].

As discussed in Sec. 1.2, defining a *representative* workload is a key requirement for benchmark specifications. To resolve this challenge despite the lack of access to real workloads, benchmarks designers rely on implementing synthetic data generators and characterizing queries without sharing their exact specifications. Another challenge is the correct execution of benchmarks as users need to avoid numerous pitfalls to obtain meaningful results [Raa+18]. Potential issues involve setting up the execution environment (installing and tuning database systems), executing the benchmark carefully to mitigate factors in the execution environmental (e.g. allowing systems sufficient time to warm up, but minimizing the effects of OS- and system-level caching), validating the correctness of the results, ensuring *reproducibility* [Man+09; Jim+17], and so on. Therefore, it is a considerable amount of work for authors to *conduct meaningful performance experiments*. Jennifer Widom, the author of multiple seminal textbooks in database systems [WC96; GUW00; GUW09], stated in a presentation that *"It's easy to do «hokey» or meaningless experiments, and many papers do"* [Wid06]. It is therefore highly beneficial for members of the community to have access to standard benchmarks that provide frameworks and guidelines for running experiments.

In this part, we present two carefully designed and engineered benchmarks that define representative workloads. We believe both will be highly beneficial for researchers and practitioners of the MDE and graph processing communities.

[5] In contrast, *microbenchmarks* measure the performance of primitive operations supported by an underlying platform, often focusing on the performance of a single operation, such as the performance of a *join operator* (Def. 15).

Chapter 5

The Train Benchmark

5.1 Introduction

Context Model-driven engineering of critical systems, like automotive, avionics or train control systems, necessitates the use of different kinds of models on multiple levels of abstraction and in various phases of development. Advanced design and verification tools aim to *simultaneously improve quality and decrease costs by early validation* to highlight conceptual design flaws well before traditional testing phases in accordance with the correct-by-construction principle. Furthermore, they improve productivity of engineers by automatically synthesizing different design artifacts (source code, configuration tables, test cases, fault trees, etc.) required by certification standards.

Motivation A prime subproblem in many design tools is *the validation of well-formedness constraints* of the domain, similarly to the *integrity constraints* used in relational database systems. Industrial standard languages (e.g. UML, SysML) and platforms (e.g. AUTOSAR [AUT18], ARINC653 [Aer16]) frequently define a large number of such constraints as part of the standard. For instance, the AADL standard [SAE09] contains 75 constraints captured in the declarative Object Constraint Language (OCL) [Obj12] while AUTOSAR defines more than 500 design rules. These rules are often captured as *graph queries* over the instance model graph.

As it is much more expensive to fix design flaws in the later stages of development, it is essential to detect violations of well-formedness constraints as soon as possible, i.e. immediately after the violation is introduced by an engineer or some automated model manipulation steps. Therefore industrial design tools perform model validation by repeatedly checking constraints after certain model changes. In many ways, this is analogous to *continuous integration* used in source code repositories [DP07], and can be referred to as *continuous model validation.*

In practice, model validation is often addressed by using model query [Ujh+15a] or transformation engines [JT10]: error cases are defined by *graph queries on models*, the results of which can be automatically repaired by transformation steps. However, this is challenging due to two factors: (1) *instance model sizes* can grow very large as the complexity of systems-under-design is increasing [Sch+12], and (2) *validation constraints* get more and more sophisticated. As a consequence, validation of industrial models is challenging or may become infeasible.

To tackle increasingly large models, they are frequently split into multiple model fragments (as in open-source tools like ARTOP [Art18] or Papyrus [Gér+07]). This can be beneficial for *local constraints*

which can be checked in the close context of a single model element. However, there are *global well-formedness constraints* in practice, which necessitate to traverse and investigate many model elements situated in multiple model fragments, thus fragment-wise validation of models is insufficient.

As different underlying technologies are used in modelling tools for checking well-formedness constraints, assessing these technologies systematically on well-defined challenges and comparing their performance would be of high academic and industrial interest. In fact, similar scenarios occur when query techniques serve as a basis for calculating values of derived features [Heg+16], populating graphical views [Deb+14], or maintaining traceability links [Heg+16] frequently used in existing tools. Furthermore, runtime verification [LS09] of cyber-physical systems may also rely on incremental query systems or rule engines [Hav15].

While there are a number of existing benchmarks for *query performance* over relational databases [DeW91; TPC10] and triplestores [GPH05; BS09; Sch+09; Mor+11; Erl+15], workloads of modelling tools for validating well-formedness constraints are significantly different [Izs+13b]. Specifically, modelling tools use *more complex queries* than typical transactional systems [Kol+13] and the perceived performance is more affected by *response time* (i.e. execution time for a specific operation such as validation or transformation) rather than throughput (i.e. the number of parallel transactions). Moreover, it is the worst case performance of a query set which dominates practical usefulness rather than the average performance. Cases of *model transformation tool contests* (TTCs) [SNZ08; LRG09; RG10; MRV10; VMR11; VRK13; RKH14; RHK15; GKR16; GHK17] are also used as benchmarks. However, most case studies prior to 2015 – the publication of the Train Benchmark case [l12] – do not consider the performance of incremental model revalidation after model changes.

Contributions In this chapter, we define the Train Benchmark, a *cross-technology macrobenchmark* [BS97; Sel+99] that aims to measure the performance of continuous model validation with graph-based models and constraints captured as queries. The Train Benchmark defines a scenario that is specifically modelled after *model validation* in modelling tools: at first, an automatically generated model (of increasing sizes) is loaded and validated, then the model is changed by some transformations, which is immediately followed by the revalidation of constraints. The primary goal of the benchmark is to measure the execution time of each phase, while a secondary goal is a cross-technology assessment of existing modelling and query technologies that (could) drive the underlying implementation.

Railway applications often use MDE techniques [PFH12] and rule-based validation [LJS16]. This benchmark uses a domain-specific model of a railway system that originates from the MOGENTES EU FP7 [MOG11] project, where both the metamodel and the well-formedness rules were defined by railway domain experts. However, we introduced additional well-formedness constraints which are structurally similar to constraints from the AUTOSAR domain [Ber+10].

The Train Benchmark intends to answer the following research questions:

RQ1 How do existing query technologies scale for a continuous model validation scenario?
RQ2 What technologies or approaches are efficient for continuous model validation?
RQ3 What types of queries are the performance bottlenecks for different tools?

This chapter systematically documents the Train Benchmark [ISR14], which was used in multiple papers both in the Fault-Tolerant Systems Research Group [Izs+13b; Ujh+15a] [l9; c4; l10] and by other researchers [Mey+18]. A simplified version of the Train Benchmark (featuring only a single modelling language and scenario) was published in the 2015 Transformation Tool Contest [l12].

Design considerations We designed the Train Benchmark to comply with the four criteria defined in [Gra93] for domain-specific benchmarks.

1. *Relevance*: It must measure the peak performance and price/performance of systems when performing typical operations within that problem domain.
2. *Portability*: It should be easy to implement the benchmark on many different systems and architectures.
3. *Scalability*: The benchmark should apply to small and large computer systems.
4. *Simplicity*: The benchmark must be understandable, otherwise it lacks credibility.

Structure of the specification In Sec. 2.3, we presented the metamodel and instance models used in the benchmark. This chapter discusses the rest of the Train Benchmark and is structured as follows. Sec. 5.3 describes the workflow of the benchmark, and specifies the scenarios, queries, transformations and the instance model generator. Sec. 5.4 shows the benchmark setup and discusses the results. Sec. 5.5 concludes the paper and outlines future research directions. Appendix C contains a detailed specification of the queries and transformations used in the benchmark.[1] This chapter presents the first complete specification of the Train Benchmark as published in [j2].

5.2 Query Technologies

We implemented the benchmark for a wide range of open-source tools operating on graph models along with the traditional SQL data model (see Sec. 2.3 for the modelling technologies).

Format	Tool	Query lang.	Impl. lang.	Incr.	Mem.
EMF	Drools	DRL	Java	⊗	⊗
	Eclipse OCL	OCL	Java	○	⊗
	EMF API	–	Java	○	⊗
	Viatra Query	VQL	Java	⊗	⊗
graph	Neo4j	Cypher	Java	○	○
	TinkerGraph	–	Java	○	⊗
RDF	Jena	SPARQL	Java	○	⊗
	RDF4J	SPARQL	Java	○	⊗
SQL	SQLite	SQL	C	○	⊗
	MySQL	SQL	C++	○	○

Table 5.1: Tools used in the benchmark. Columns – *Query lang.:* query language, *Incr.:* supports incremental evaluation, *Mem.:* supports in-memory evaluation, *Impl. lang.:* implementation language of the tool, Notation – ⊗ feature supported, ○ feature not supported.

Tab. 5.1 shows the list of the implementations. We classify a tool *incremental* if it employs caching techniques and provides a dedicated incremental query evaluation algorithm that processes *changes* in the model and propagates these changes to query evaluation results in an incremental way (i.e. to avoid complete recalculations). Both Viatra Query and Drools are based on the

[1] The implementation and the detailed results are available online at http://docs.inf.mit.bme.hu/trainbenchmark.

Rete algorithm [Pro11]. Eclipse OCL also has an incremental extension called the *OCL Impact Analyzer* [UGH11], however, it is not actively developed, therefore it was excluded from the benchmark. In contrast, *non-incremental* tools use *search-based* algorithms. These algorithms evaluate queries with model traversal operations, which may be optimized using heuristics and/or caching mechanisms. The table also shows if a tool uses an *in-memory engine* and lists their *implementation languages*.

5.2.1 EMF Tools

We implemented the benchmark for multiple EMF-based tools.

- As a baseline, we have written a *local search-based* algorithm for each query in Java, using the EMF API. The implementations traverse the model without specific search plan optimizations, but they cut unnecessary search branches at the earliest possibility.
- The OCL [Obj12] language is commonly used for querying EMF model instances in validation frameworks. It is a standardized navigation-based query language, applicable over a range of modelling formalisms. Taking advantage of the expressive features and wide-spread adoption of this query language, the project Eclipse OCL [Ecl15a] provides a powerful query interface that evaluates such expressions over EMF models.
- Viatra Query [Var+16] is an Eclipse Modeling project where several designers of the Train Benchmark are involved. Viatra Query provides incremental query evaluation using the Rete algorithm [For82]. Queries are defined in a graph pattern-based query language [Ber+11], and evaluated over EMF models. Viatra Query is developed with a focus on incremental query evaluation, however, it is also capable of evaluating queries with a local search-based algorithm [Búr+15]. Its language, VQL is discussed in Sec. 2.6.5, while its incremental features are presented in Sec. 10.4.4.
- Incremental query evaluation is also supported by Drools [Pro11], a rule engine developed by Red Hat. Similarly to Viatra Query, Drools is based on ReteOO, an object-oriented version of the Rete algorithm [For82]. In particular, Drools 6 uses PHREAK, an improved version of ReteOO with support for lazy evaluation. Queries can be formalized using DRL, the Drools Rule Language. While Drools is not a dedicated EMF tool, the Drools implementation of the Train Benchmark works on EMF models. While using EMF objects instead of plain java objects (POJOs) induces some memory overhead, the effect is small [Ujh+15b], and EMF's built-in features, such as deserialization and notifications, make it well-suited for using with Drools.

5.2.2 RDF Tools

Triplestores are usually queried via SPARQL (recursive acronym for SPARQL Protocol and RDF Query Language) [SP08] which is capable of defining graph patterns.

- Jena[2] [McB02] is a Java framework for building Semantic Web and Linked Data applications. It provides an in-memory store and supports relational database backends.
- RDF4J[3] [BKH02] (formerly called Sesame) gives an API specification for many tools, and also provides its own implementation.

5.2.3 Property Graph Tools

We included two tools supporting the property graph data model:

[2] https://jena.apache.org/
[3] http://rdf4j.org/

- As of 2019, the most popular graph database is Neo4j [Web12] which provides multiple ways to query graphs: (1) a *low-level core API* for elementary graph operations, (2) the *Cypher language*, a declarative language focusing on graph pattern matching.
  While Cypher is very expressive and its optimization engine is being actively developed, it may be beneficial for some queries to manually implement the graph traversals using the core API [RWE15, Chapter 6: Graph Database Internals].
- TinkerGraph is an in-memory reference implementation of the property graph interfaces provided by the Apache TinkerPop framework.[4]

5.2.4 Relational Databases

We included two popular relational database management systems (RDBMSs) in the benchmark.

- MySQL[5] is a well-known and widely used open-source RDBMS, implemented in C and C++.
- SQLite[6] is a popular embedded RDBMS, implemented in C.

5.3 Benchmark Specification

This section presents the specification of the Train Benchmark including inputs and outputs (Sec. 5.3.1), benchmark phases (Sec. 5.3.2), use case scenarios (Sec. 5.3.3), queries (Sec. 5.3.4), transformations (Sec. 5.3.5), a selected query with its transformations (Sec. 5.3.6), and instance models (Sec. 5.3.7).

5.3.1 Inputs and Outputs

Inputs A *benchmark case* configuration in the Train Benchmark takes a *scenario*, an *instance model size* and a *set of queries* as input. The specific *characteristics* of the model (e.g. error percentages) are determined by the scenario, while the *transformation* is defined based on the scenario and a query. The *instance models* used in the Train Benchmark can be automatically generated using the generator module of the framework. The model generator uses a pseudorandom number generator with a fixed random seed to ensure the reproducibility of results.

Outputs Upon the successful run of a benchmark case, the *execution times* of each phase and the *number of invalid elements* are recorded. Moreover, the collection of the element identifiers in the result set must be returned to allow the framework to check the correctness of the solution (Sec. 5.3.8). Furthermore, this result set also serves as a basis for executing transformations in the Repair scenario.

5.3.2 Benchmark Phases

The authors of [Ber+10] analysed the performance of incremental graph query evaluation techniques, and defined four *phases* for model validation. We adapted and refined these for the Train Benchmark (depicted in Fig. 5.1):

1. During the read phase, the *instance model* is loaded from the disk to the memory and the *validation queries* are initialized (but not executed explicitly). The model has to be defined in one or more textual files (e.g. XMI, CSV, SQL dump, etc.), binary formats are disallowed. The read

[4] https://tinkerpop.apache.org/
[5] https://www.mysql.com/
[6] https://www.sqlite.org/

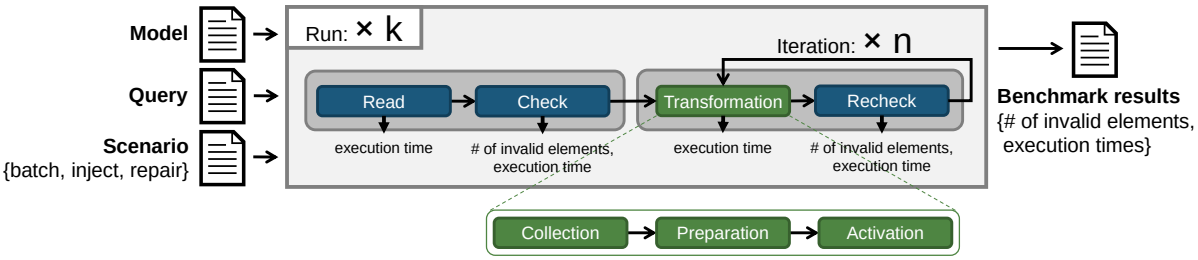


Figure 5.1: Phases of the benchmark and sub-phases of the *Transformation* phase.

phase includes the parsing of the input as well as the initialization of internal data structures of the tool.

2. In the check phase, the instance model is queried to identify invalid elements.
3. In the transformation phase, the model is changed to simulate the effects of model manipulations. This phase has three sub-phases: (1) collection, (2) preparation, and (3) activation. Based on the collection sub-phase, the transformations are either performed on a subgraph specified by a simple pattern (Inject scenario) or on a subset of the model elements returned by the check phase (Repair scenario); see Sec. 5.3.3 for details.
4. The revalidation of the model is carried out in the recheck phase similarly to the check phase. The transformations modify the model to induce a change in the match set, which implies that the recheck phase will return a different match set than the previous check/recheck phases did.

5.3.3 Use Case Scenarios

To increase the representativeness of the benchmark, we defined use case *scenarios* similar to typical workloads of real modelling tools, such as one-time validation (Batch scenario, used in [Izs+13b; Ujh+15b]), minor model changes introduced by an engineer (Inject scenario, used in [Ujh+15a]) or complex automated refactoring steps (Repair scenario, used in [c4; l12]). For space considerations, the results for the Batch scenario were omitted from this dissertation and are available in the Train Benchmark journal paper [j2].

Batch validation scenario (Batch) In this scenario, the instance model is loaded (read phase) from storage and a model validation is carried out by executing the queries in the check phase. This use case imitates a designer opening a model in an editor for the first time (e.g. after a checkout from a version control system) which includes an immediate validation of the model. In this scenario, the benchmark uses a model free of errors (i.e. no well-formedness constraints are violated), which is a common assumption for a model committed into a repository.

Fault injection scenario (Inject) After an initial validation, this scenario repeatedly performs transformation and recheck phases. After the first validation (check), the transformation first collects a set of model elements such as Routes (collection), then selects a subset of these elements (preparation), and executes the transformation on this subset (activation). The transformation is immediately followed by revalidation (recheck) to receive instantaneous feedback. The manipulation injects faults to the model, so the size of the match set always (monotonically) *increases*. Such scenario occurs in practice when engineers change the model in small increments using a domain-specific editor. These editors should detect design errors quickly and early in the development process to cut down verification costs according to the correct-by-construction principle.

Automated model repair scenario (Repair) In this scenario, an initial validation is also followed by transformation and recheck phases. However, the model is repaired in the transformation phase based on the violations identified during the previous validation step. On the execution level, this means that the collection sub-phase of the transformation takes the results of the previous check/recheck phase, and uses those for the subsequent preparation sub-phase, where it a subset of the results is selected. On this subset, the activation sub-phases executes transformation rules capturing quick fix-style operations [Heg+11]. In the recheck phase, the whole model is revalidated, and the remaining errors are reported. As the model manipulations fix errors in the model, the size of the match set (monotonically) *decreases*. Efficient execution of this workload profile is necessary in practice for refactoring, incremental code generation, and model transformations.

5.3.4 Specification of Queries

As discussed in Sec. 2.5.3, well-formedness constraints are often captured and checked by *queries*. Each query identifies *violations of a specific constraint* in the model [Ber+10]. These constraints can be formulated in constraint languages (such as OCL), graph patterns (e.g. VQL, Cypher) and in SQL.

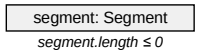
In the check and recheck phases of the benchmark, we perform a *query* to retrieve the elements violating the well-formedness constraint defined by the benchmark case. The complexity of queries ranges from simple property checks to complex path constraints consisting of several navigation operations. The graph patterns are defined with the following syntax and semantics.

- *Positive conditions* define the structure and type of the nodes and edges that must be satisfied.
- *Negative conditions* (also known as negative application conditions) define subpatterns which must not be satisfied. Negative conditions are displayed in a red rectangle with the NEG caption.
- *Filter conditions* are defined to check the value of node properties and are typeset in *italic*.

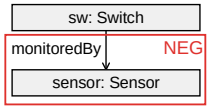
We define the following six constraints by graph patterns (see Fig. 5.2). Each corresponding query checks a specific constraint and covers some typical query language features.

- PosLength (Fig. 5.2a) requires that a segment must have a positive length. The corresponding query defines a simple property check, a common use case in validation.
- SwitchMonitored (Fig. 5.2b) requires every switch to have at least one sensor connected to it. The corresponding query checks whether a node is connected to another node. This pattern is also common in more complex queries, e.g. it is used in RouteSensor and SemaphoreNeighbor.
- RouteSensor (Fig. 5.2d) requires that all sensors associated with a switch that belongs to a route must also be associated directly with the same route. The corresponding query checks for the absence of circles, so the efficiency of evaluating negative conditions is tested.
- SwitchSet (Fig. 5.2e) requires that an entry semaphore of an active route may show GO only if all switches along the route are in the position prescribed by the route. The corresponding query tests the efficiency of navigation and filtering operations.
- ConnectedSegments (Fig. 5.2c) requires each sensor to have at most 5 segments. The corresponding query checks for "chains" similar to a reachability with an upper bound. This is a common use case in model validation.
- SemaphoreNeighbor (Fig. 5.2f) requires routes which are connected through a pair of sensors and a pair of track elements to belong to the same semaphore. The corresponding query checks for the absence of circles, so the efficiency of join (Def. 15) and antijoin (Def. 17) operations is tested. One-way navigable references are also present in the constraint, so the efficiency of their evaluation is also measured. *Subsumption inference* (see Sec. 2.3.6) is required, as the two track elements (te1, te2) can be switches or segments.

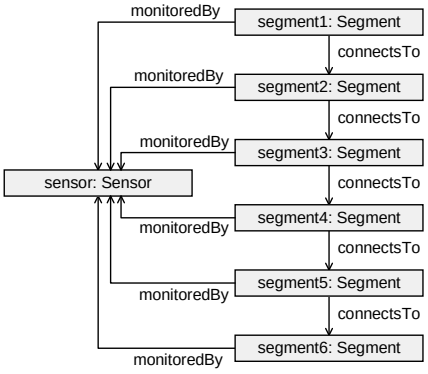
Structural similarity to AUTOSAR Several of the benchmark queries are adapted from constraints of the AUTOSAR standard [AUT18] and represent common validation tasks such as attribute and reference checks or cycle detection. In accordance with paper [Ber+10], the following graph patterns are inspired by AUTOSAR constraints, i.e. the matching subgraphs of corresponding graph queries are either isomorphic or structurally similar.



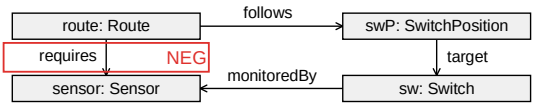
(a) The PosLength pattern.



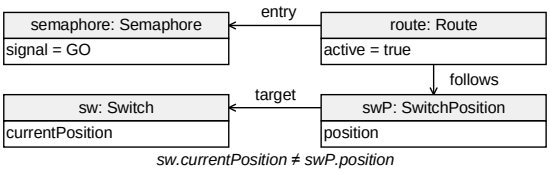
(b) The SwitchMonitored pattern.



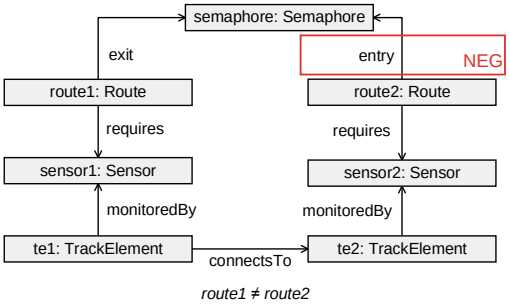
(c) The ConnectedSegments pattern.



(d) The RouteSensor pattern.



(e) The SwitchSet pattern.



(f) The SemaphoreNeighbor pattern.

Figure 5.2: The patterns of benchmark queries.

	parameters	variables	node types	attributes	attr./equality checks	edge constraints	negative conditions
ConnectedSegments	0	7	7	2	0	0	11
RouteSensor	1	4	4	4	0	0	3
PosLength	0	2	2	1	1	1	0
SemaphoreNeighbor	1	7	7	4	0	1	6
SwitchMonitored	1	1	2	2	0	0	0
SwitchSet	0	6	6	4	4	3	3

Table 5.2: Characterization of graph queries in the benchmark.

The query metrics adapted from [Izs+13b; Ujh+15b] are listed in Tab. 5.2. The metrics indicate that all relevant features of query languages are covered by our queries except for transitive closure and recursion. We decided to omit these features from the benchmark as they are supported by only a few query technologies.

5.3.5 Specification of Transformations

To capture complex operations in the scenarios, we use graph transformation rules [Roz97] which consist of (1) a precondition pattern captured as a graph query and (2) an action with a sequence of elementary graph manipulation operations. The transformations are defined with a syntax similar to tools such as GROOVE, FUJABA [NNZ00] and VIATRA2 [VB07]. For defining the patterns and transformations, we used a graphical syntax similar to GROOVE [Ren03]:

- *Inserting* new nodes and new edges between existing nodes (marked with «new»).
- *Deleting* existing nodes and edges (marked with «del»). The deletion of a node implies the deletion of all of its edges to eliminate dangling edges.
- *Updating* the properties of a node (noted as *property* ← *new value*).

Our transformations cover all elementary model manipulation operations, including the insertion and deletion of nodes and edges, as well as the update of attributes. A detailed specification of the queries and transformation is given in Appendix C. In this section, we only discuss the RouteSensor query and its transformations in detail.

5.3.6 Query and Transformations for Constraint RouteSensor

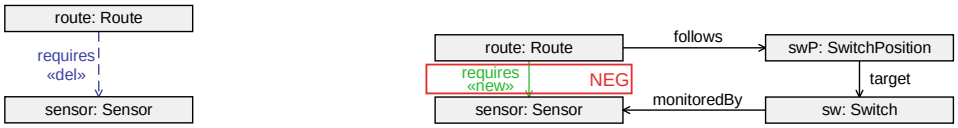
We present the specification of query RouteSensor and its related transformations.

Description To check if constraint RouteSensor (see Sec. 5.3.4) is violated, the corresponding query (Fig. 5.2d) looks for routes (route) that follow a switch position (swP) connected to a sensor (sensor) via a switch (sw), but without a requires edge from the route to the sensor.

Query specification The Cypher (Sec. 2.6.1) representation of the query is shown in Listing 5.1.

```
1 MATCH (route:Route)-[:follows]->(swP:SwitchPosition)
2   -[:target]->(sw:Switch)-[:monitoredBy]->(sensor:Sensor)
3 WHERE NOT (route)-[:requires]->(sensor)
4 RETURN route, sensor, swP, sw
```

Listing 5.1: The RouteSensor query.



(a) Inject transformation.

(b) Repair transformation.

Figure 5.3: The transformations for query RouteSensor.

Inject transformation Random requires edges are removed.

Constraint	Batch	Inject	Repair
PosLength	0%	2%	10%
SwitchMonitored	0%	2%	18%
RouteSensor	0%	4%	10%
SwitchSet	0%	8%	15%
ConnectedSegments	0%	5%	5%
SemaphoreNeighbor	0%	7%	25%

Table 5.3: Error probabilities in the generated instance model.

Repair transformation The missing requires edge is inserted from the route to the sensor in the match, which fixes the violation of the constraint.

5.3.7 Instance Model Generation and Fault Injection

To assess scalability, the benchmark uses instance models of growing sizes where each model contains twice as many model elements as the previous one. The sizes of instance models follow powers of two (1, 2, 4, …, 2 048): the smallest model contains about 5 000 triples, the largest one (in this work) contains about 19 million triples.

The instance models are systematically generated based on the metamodel: first, small instance model fragments are generated, then they are connected to each other. To avoid highly symmetric models, the exact number of elements and cardinalities are randomized to make it difficult for query tools to efficiently cache models.

The instance model generator is implemented in an imperative manner. The model is generated with nested loops, where each loops generates a specific element in an order driven by the containment hierarchy. The fault injection algorithm works as follows. For each well-formedness constraint, we select a model element which could introduce a violation of that constraint. For example, compared to a well-formed model, the violations are injected as follows.

- Constraint PosLength is violated by assigning an invalid value to the length attribute.
- Constraint SwitchMonitored is violated by deleting all monitoredBy edges of a Switch.
- Constraint RouteSensor is violated by deleting the requires edge from a Route to a Sensor.
- Constraint SwitchSet is violated by setting an invalid currentPosition attribute to a Switch (i.e. not the position of the corresponding SwitchPosition followed by the Route).
- Constraint SemaphoreNeighbor is violated by deleting an entry edge between a Route and a Semaphore.
- Constraint ConnectedSegments is violated by adding an additional (sixth) Segment to the same Sensor and connecting it to the last Segment.

The generator injects these faults with a certain probability (Tab. 5.3) using a random generator with a predefined random seed. These errors are found and reported in the check phase of the benchmark.

5.3.8 Ensuring Deterministic Results

During transformation phase of the Repair scenario, some invalid submodels (i.e. pattern matches) are selected and repaired. In order to ensure *deterministic, repeatable results*:

- The elements are always selected from a deterministically *sorted list.*

- The candidates for transformation are chosen using a pseudorandom generator with a fixed random seed.

The matches may be returned in any collection in any order, given that the collection is unique. The matches are interpreted as *tuples*, e.g. the RouteSensor query returns $\langle \mathsf{route}, \mathsf{sensor}, \mathsf{swP}, \mathsf{sw} \rangle$ tuples. The tuples are sorted using a lexicographical ordering.[7]

The ordered list is used to ensure that the transformations are performed on the same model elements, regardless of the return order of the match set. Neither sorting nor selecting candidates are included in the execution time measurements.

5.4 Evaluation

In this section, we discuss the benchmark methodology, present the environment for our experiments and analyse the results. The source code of the benchmark framework and tool-specific implementations are available online.[8]

5.4.1 Benchmark Setup

Benchmark Parameters A *measurement* is defined by a certain *tool* (with its optional *parameters*), *scenario*, *model size*, *queries*, and *transformations*. Tab. 5.4 shows the tools, parameters, scenarios, queries and sizes used in the benchmark. If a tool has no parameters, it is only executed once, otherwise it is executed with each optional parameter.

Parameter	Values	Sec.
Scenario	Batch	5.3.3
	Inject	5.3.3
	Repair	5.3.3
Queries	ConnectedSegments	C.1
	PosLength	C.2
	RouteSensor	5.3.6
	SemaphoreNeighbor	C.4
	SwitchMonitored	C.5
	SwitchSet	C.6
Size	1, 2, 4, …, 2 048	5.3.7

(a) Benchmark-specific parameters.

Tool	Version	Parameters
Drools	6.5.0	–
Eclipse OCL	3.3.0	–
EMF API	2.10.0	–
Jena	3.0.0	no inferencing
		inferencing
MySQL	5.7.16	–
Neo4j	3.0.4	core API
		Cypher
RDF4J	2.1	(no inferencing)
SQLite	3.8.11.2	–
TinkerGraph	3.2.3	–
VIATRA Query	1.4.0	local search
		incremental

(b) Tool-specific parameters.

Table 5.4: Configuration parameters.

[7] To *compare* matches $M_1 = \langle a_1, a_2, \dots, a_n \rangle$ and $M_2 = \langle b_1, b_2, \dots b_n \rangle$, we take the first elements in each match (a_1 and b_1) and compare their identifiers. If the first elements are equal, we compare the second elements (a_2 and b_2) and so on until we find two different model elements. This is guaranteed by the fact that matches are returned as a set, i.e. their each returned match is unique.

[8] http://docs.inf.mit.bme.hu/trainbenchmark

Benchmark environment The benchmark was performed on a virtual machine with an eight-core, 2.4 GHz Intel Xeon E5-2630L CPU with 16 GB of RAM, and an SSD hard drive. The machine was running a 64-bit Ubuntu 14.04 server operating system and the Oracle JDK 1.8.0 u111 runtime. The independence of performance measurements was guaranteed by running each sequentially and in a separate Java Virtual Machine (JVM). The heap memory limit for the JVM was set to 12 GB.

Benchmark scenarios We investigated three benchmark scenarios: Inject, Repair, and Batch. The latter is only covered in paper [j2], while the first two is detailed in this chapter.

The Inject scenario is structured as follows:

1. The benchmarks loads the model and evaluates the *queries* as *initial validation*, and we measure execution times for read, check, and their sum. The results are shown left in Fig. 5.4.
2. The benchmark iteratively performs the Inject transformations for each query 10 times (Fig. 5.1, $n = 10$) followed by an immediate recheck step in each iteration. The transformation modifies a *fixed* number of elements (10) in each iteration. We measure the mean execution time for continuous validation for each phase (transformation, recheck, and their sum). The results are shown in the right column of Fig. 5.4.

Meanwhile, the sequence of the Repair scenario is the following:

1. The benchmark performs the initial validation similarly to the Inject phase. The execution times for read, check, and their sum are listed in the left column of Fig. 5.5.
2. The benchmark iteratively performs the Repair transformation for each query 8 times (Fig. 5.1, $n = 8$) followed by an immediate recheck step in each iteration. The transformation modifies a *proportional* amount of the invalid elements (5%). We measure the mean execution time for continuous validation for each phase (transformation, recheck, and their sum). The results are shown in the right column of Fig. 5.5.

For each setup, a scenario was executed 5 times.

5.4.2 Measurement of Execution Times

If all runs are completed within a *timeout* of 15 minutes, the measurement is considered successful and the *measurement results* are saved. If the measurement does not finish within the time limit, its process is terminated and its results are discarded. The results were processed as follows.

1. The *mean* execution time was calculated for each phase. For example, in the Repair scenario, the execution times of the transformation and the recheck phases are determined by their *average* execution time. This is determined independently for all runs.
2. For each phase, the *median* value of the 5 runs was taken.

Using the mean value to the describe the execution time of repeated transformation and recheck phases is aligned with the recommendations of [FW86]. Moreover, from a statistical perspective, taking the median value of the sequential runs can better compensate for transients potentially perceived during a measurement.

5.4.3 How to Read the Charts?

Detailed plots The plots in Fig. 5.4 and 5.5 present the execution times of a certain workload with respect to the model size. Each plot can be directly interpreted as *an overall evaluation of execution time against increasing model sizes* dominated by the worst case behaviour of a tool.

On each plot, the horizontal axis (with base 2 logarithmic scale) shows the model size and the vertical axis (with base 10 logarithmic scale) shows the execution time of a certain operation. Note that as

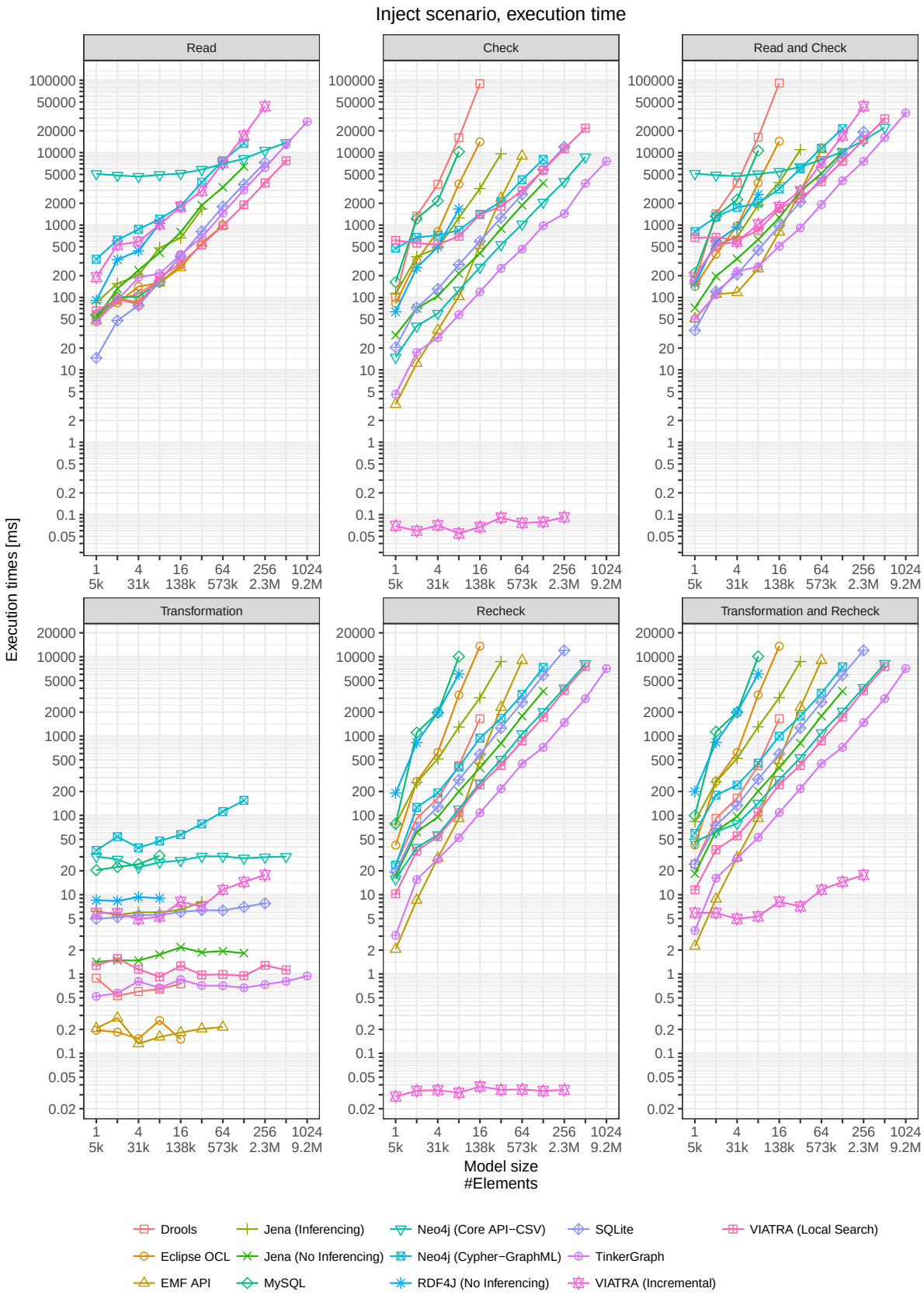


Figure 5.4: Execution times in the Inject scenario.

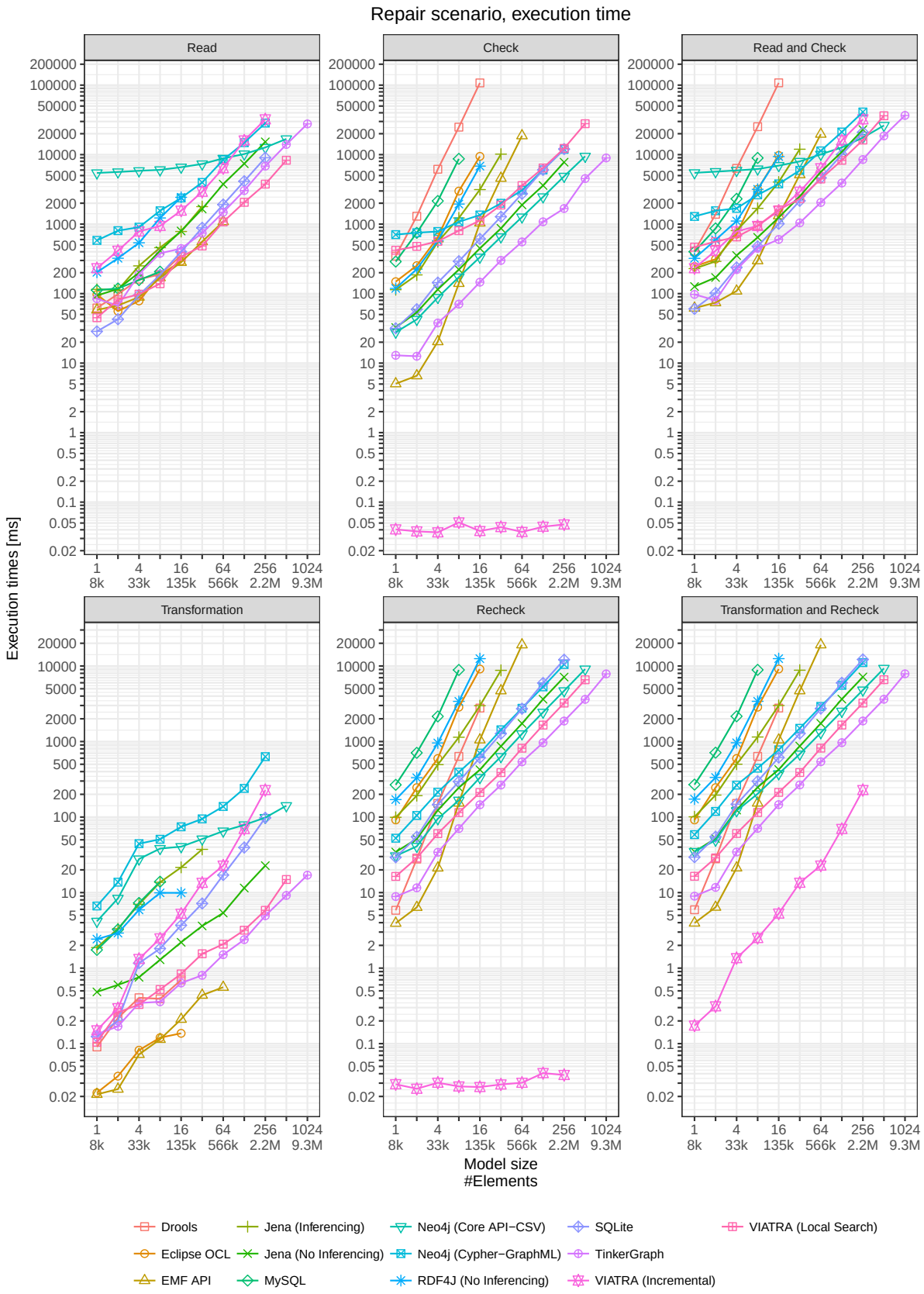


Figure 5.5: Execution times in the Repair scenario.

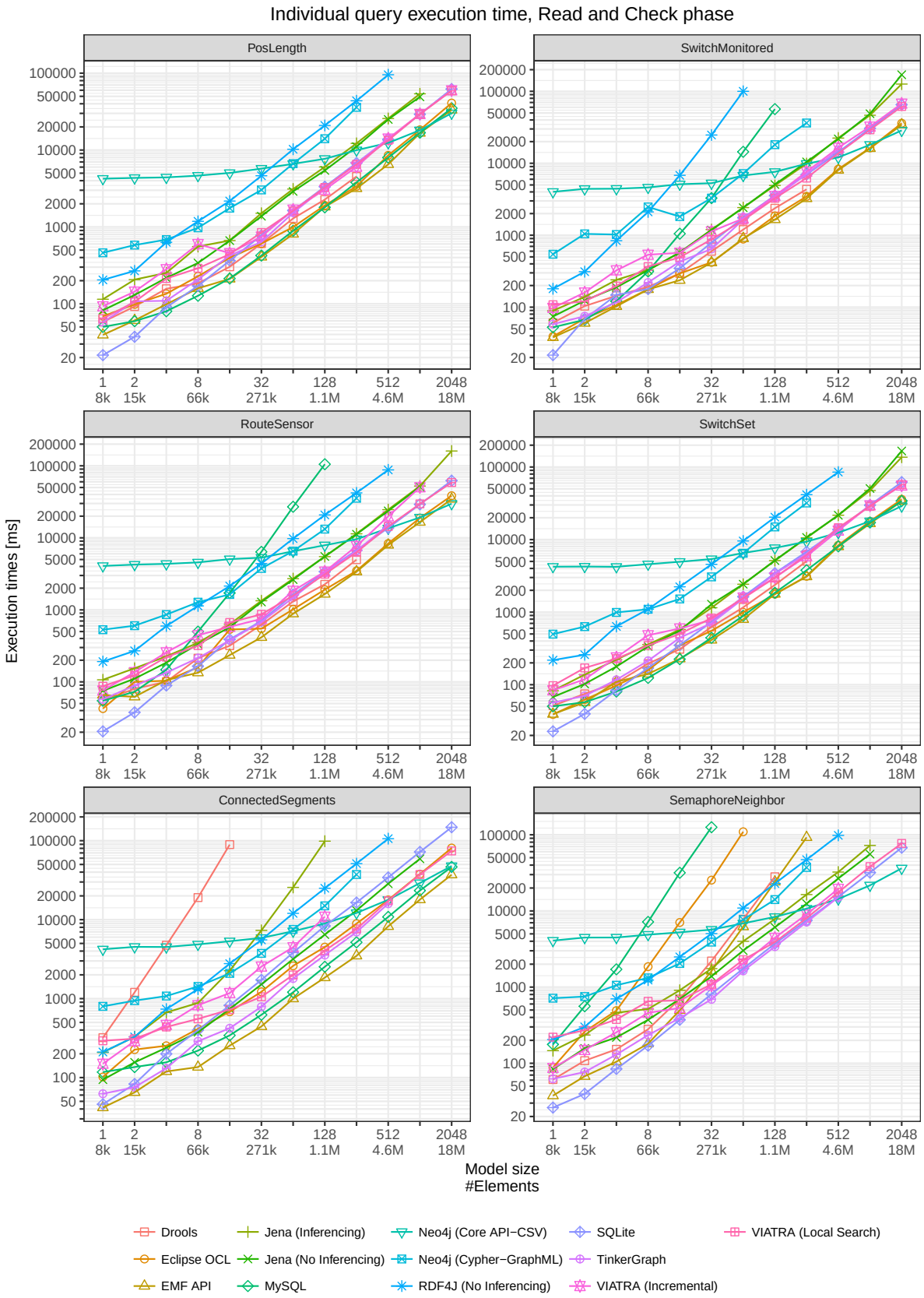


Figure 5.6: Execution times for individual queries (Read and Check).

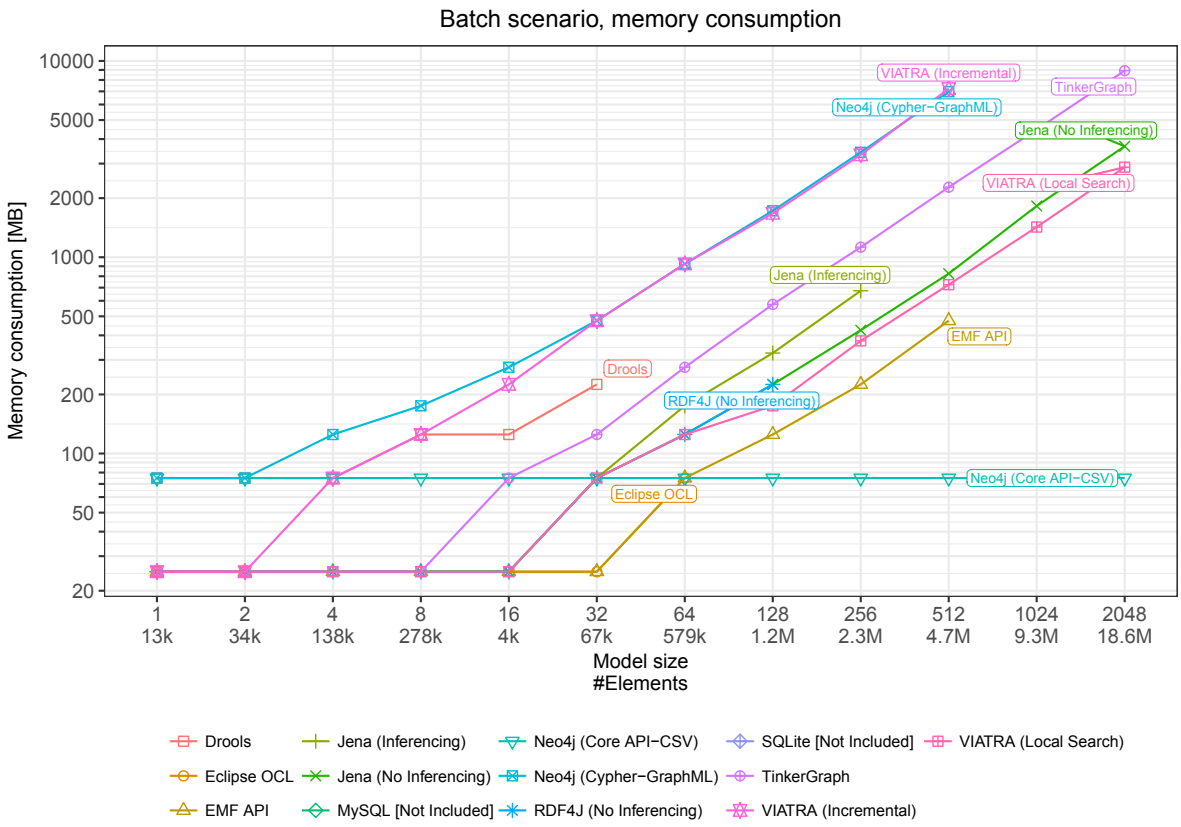


Figure 5.7: Estimated memory consumption for loading the model and evaluating all queries. (Memory consumption in this figure at a certain model size does not correspond to the execution times of Fig. 5.4–5.6.)

the execution time of phases vary greatly (e.g. the read phase takes longer than the check phase as it contains disk operations), the vertical axes on the plots *do not use the same scale*, i.e. the minimum and maximum values are adjusted to make the plots easier to read.
The logarithmic scales imply that a "linear" appearance of all measurement series correspond to a (low-order) polynomial $\mathcal{O}$ characteristic where the slope of a plot determines the dominant order (exponent). Moreover, a constant difference on a plot corresponds to a constant order-of-magnitude difference. Note that different plots are not directly comparable to each other visually due to the different scales.

Individual query plots The plots in Fig. 5.6 help us identify *specific strengths and weaknesses of different tools* and highlight which query turned out to be the performance bottleneck. This can explain why certain tools had a timeout even for medium-sized models in the detailed plots of Fig. 5.4–5.5.

5.4.4 Measurement of Memory Consumption

Determining the memory consumption of applications running in managed environments (such as the JVM) is a challenging task due to (1) the non-deterministic nature of the garbage collector and

(2) sophisticated optimizations in collection frameworks which often allocate memory in advance and only free memory when it is necessary [Boy08].

For a reliable estimation on memory consumption, we used the following approach.

1. We set a hard limit L to the available heap memory for the JVM and perform a *trial* of the run.
2. Based on the result of the trial, we either decrease or increase the limit.
   a) If the trial successfully executed within the specified timeout, we decrease the limit to $\frac{L}{2}$.
   b) If the execution failed (due to timeout/memory exhaustion), we increase the limit to $2L$.

This results in a binary search-like algorithm, which ensures a resolution of $\frac{L_{\text{initial}}}{2^{t-1}}$, given an initial limit L_{initial} and t trials. For example, with an initial limit of 6.4 GB of memory and 9 trials, this approach provides a resolution of $\frac{6400\ \text{MB}}{2^8}$ = 25 MB (as used in our measurements later).

The results are shown in Fig. 5.7.[9] Note that the measurements for execution time and memory consumptions were performed separately. The measurements in Fig. 5.4, 5.5, and 5.6 used a larger, fixed amount of memory. For instance, the low memory consumption of Neo4j in Fig. 5.7 corresponds to significantly larger execution time than reported in Fig. 5.6.

The results show that incremental tools (in particular, VIATRA Query in incremental mode) use more memory than non-incremental ones. This is expected as incremental tools utilize space-time tradeoff, i.e. they trade memory for execution speed by building caches of the interim query results and use it for efficient recalculations.

5.4.5 Analysis of Results

Following Tab. 5.5, we highlight some strengths and weaknesses identified during designing, implementing, executing, and evaluating the benchmark.

Findings	Area	Observations
Technology	EMF	⊕ EMF tools are suitable for model validation
		⊖ No built-in indexing support in EMF
	graph databases	⊕ Good storage performance for graph databases
	RDF databases	⊖ Underperforming RDF systems
		⊖ Slow inferencing in RDF4J
	relational databases	⊕ Fast model load and good scalability from SQLite
		⊖ MySQL slowdown for complex queries
Approach	incremental	⊕ Incremental tools prevail for continuous validation
		⊖ Scalability of incremental tools is limited by memory
	search-based	⊕ Search-based tools scale for large models on simple queries
		⊖ Search-based tools face problems for complex queries
	indexing	⊕ Substantial effect of indexing on performance
	language features	⊖ Long path expressions are hard to evaluate
	performance	⊖ Queries with many joins and negations are selective
		⊕ Huge differences in runtime across technologies
	size of updates	⊕ Noticeable differences between scenarios

Table 5.5: Summary of findings: strengths ⊕ and weaknesses ⊖.

[9] We excluded MySQL from this measurement as limiting its available memory only causes it to use the disk more extensively, so this method cannot give a good approximation on its memory consumption. We also excluded SQLite as it uses the native heap instead of the Java heap.

Technology-specific findings

EMF tools are suitable for model validation As expected, EMF tools perform well in model validation and transformation. EMF was designed to serve as a common basis for various MDE tools with in-memory model representation to improve performance. In principle, their in-memory nature may hinder scalability due to the memory limit of a single workstation, but despite this, some EMF solutions were among the most scalable ones.

No built-in indexing support in EMF EMF does not offer built-in indexing support which allows the system to quickly load the model, but may hinder efficient query evaluation. Indexing would significantly help local search approaches with adaptive model-specific search plans [Var+15; Gei+06].

Good storage performance for graph databases We benchmark Neo4j with both its *core API* and its *Cypher query language*. Both show similar performance characteristics, with the core API approach at least half an order of magnitude faster. For importing large datasets, the CSV import provides good performance and scalability (unlike the GraphML import), but it requires the user to manually map the graph to set of CSV files. However, query performance of the Cypher engine used in the benchmark has not yet reached the efficiency of other local search-based query engines (e.g. Viatra). As a workaround, complex queries can be optimized by hand-coded traversals implemented with the core API as recommended in [RWE15, Chapter 6: Graph Database Internals].

Underperforming RDF systems The in-memory SPARQL query engines (Jena, RDF4J) are in the slowest third of the tools, which is unexpected, considering their performance on benchmarks with different workloads (see Sec. 7.2). In our experiments between 2012 and 2015, openly available disk-based SPARQL engines were even slower, hence they were excluded from the benchmark.

Fast model load and good scalability from SQLite The SQLite implementation serves as a baseline for a comparison with more sophisticated tools. However, SQLite is surprisingly fast in several configurations. This may indicate that other technologies still have a lot of potential for performance enhancements.

MySQL slowdown for complex queries MySQL is not able to evaluate the more complex queries efficiently which prevents it from scaling for large models.

Approach-specific findings

Incremental tools prevail for continuous validation Incremental tools are very well-suited for performing continuous model validation due to their low runtime and robustness w.r.t. query complexity. The approach introduces an overhead during the read phase but enables the systems to perform quick transformation–recheck cycles.

Scalability of incremental tools is limited by memory Due to the memory overhead of incremental tools, they are unable to evaluate queries on the largest models used in the benchmark.

Search-based tools scale for large models on simple queries Non-incremental tools are able to scale well by evaluating simple and moderately difficult queries even for the largest models of the benchmark. However, revalidation takes well over 1 second for large models of 1M+ elements.

Search-based tools face problems for complex queries For the more complex queries, ConnectedSegments and SemaphoreNeighbor, most non-incremental tools are unable to scale for large models (Fig. 5.4) as they time out before completing evaluation.

Substantial effect of indexing on performance As observed for some tools, such as Neo4j and Viatra Query (local search), indexing has a substantial positive effect on performance. Using indexers allows Viatra Query to outperform the native EMF API solution, which lacks built-in indexing.

Long path expressions are hard to evaluate The ConnectedSegments query defines a long path expression: it looks for a sensor that has 6 segments (segment1, ..., segment6), connected by connectsTo edges. The results show that this query is quite difficult to evaluate Fig. 5.4. For RDF tools, queries using either *property paths* (Sec. 2.6.4) or metamodel-level *property chains* could lead to better performance. However, even though they are part of the SPARQL 1.1 [SP08] and the OWL 2 [Gro12] standard, respectively, these features are not supported by most of the tools.

Huge differences in runtime across technologies While the overall characteristics of all tools are similar (low-order polynomial with a constant component), there is a rather large variation in execution times (with differences up to 4 orders of magnitude in revalidation time). This confirms our expectation that the persistence format, model load performance, query evaluation strategy and transformation techniques can have a significant impact on overall performance and deficiencies in any of these areas likely have a negative effect.

Noticeable differences between scenarios As noted in Sec. 5.3.5, the main difference between the Inject and Repair scenarios is the number of model changes, which is significantly larger for the Repair scenario. The query result sets are also larger for the Repair scenario. By comparing corresponding plots, we observe that the overall evaluation time is affected linearly by this difference, meaning that all tools are capable of handling this efficiently.

5.4.6 Threats to Validity

Internal threats

Mitigating measurement risks To mitigate *internal validity threats*, we reduced the number of uncontrolled variables during the benchmark. Each measurement consisted of multiple runs to warm up the JVM and to mitigate the effect of transient faults such as noise caused by running our measurements in a public cloud environment.

Ensuring functional equivalence and correctness Queries are defined to be *semantically equivalent* across all query languages, i.e. for a given query on a given graph (defined by its scenario and size), the result set must be identical for all representations. To ensure the correctness of a solution, we designed and implemented tests for each query and transformation.

Code reviews To ensure comparable results, the query implementations were reviewed by experts of each technology.

Search plans The EMF API, the Neo4j Core API and the TinkerGraph implementations required a custom search plan. For each query, we used the same search plan in both implementations. As mentioned in Sec. 5.2.1, the search plans are not fully optimized, i.e. they are similar to what a developer would implement without fine-tuning performance. Our measurements exclude approaches with adaptive model-specific search plans [VFV06a; Var+15; Gei+06], which were reported to visit fewer nodes (thus achieve lower execution time) compared to local search approaches with fixed plans.

In-memory vs. disk-resident tools As shown in Tab. 5.1, some of the tools use in-memory engines while others persist data on the disk. Even with SSD drives, memory operations are more than an order of magnitude faster than disk operations, which favours the execution time of in-memory engines.

Memory overhead introduced by the framework To ensure deterministic results (see Sec. 5.3.8), the framework creates a copy of the match sets returned by the query engine. This introduces memory overhead by the framework itself. However, as the match sets are generally small compared to the size of the model (see Tab. 5.3), this overhead is negligible.

External threats

Considering *external validity*, the most important factor is the relevance to real use cases. Based on our past experience in developing tools for critical systems [Heg+16], we believe that the metamodel (Fig. 2.4), the queries (Sec. 5.3.4), and the transformations (Sec. 5.3.5) are representative to models and languages used for designing critical embedded systems. Furthermore, we believe the findings could also prove useful for other use cases with *similar workload profiles* that benefit from incremental query evaluation.

5.4.7 Summary

Finally, we revisit our research questions:

RQ1 *How do existing query technologies scale for a continuous model validation scenario?*
Most scalable techniques, have low memory consumption in order to load large models. However, few query technologies are able to evaluate the queries and transformations required for model validation on graphs with more than 5 million elements.

RQ2 *What technologies or approaches are efficient for continuous model validation?*
Incremental query engines (like Viatra Query) are well-suited to continuous validation workload by providing very low execution time, but their scalability for large models is limited by increased memory consumption.

RQ3 *What types of queries are the performance bottlenecks for different tools?*
Queries with many navigations and negative constraints are a serious challenge for most existing tools.

5.4.8 Comparison to Related Benchmarks

The Train Benchmark is a *cross-technology* macrobenchmark that aims to measure the performance of continuous model validation with graph-based models and constraints captured as queries. Earlier versions of the benchmark have been continuously used for performance measurements since 2012 (mostly related to the Viatra Query framework) in various papers [l9; c4; l10] [Izs+13b; Ujh+15a; Mey+18]. Compared to previous publications on the benchmark, this work has the following novel contributions:

- The benchmark features three distinct scenarios: Batch, Inject and Repair, each capturing a different aspect of real-world model validation scenarios. Previous publications only considered one or two scenarios.
- In this work, we investigated the performance of query sets. Previously, we only executed the individual queries separately.
- Previous publications only used tools from one or two technologies. In this chapter, we assessed 10 tools, taken from four substantially different technological spaces. This demonstrates that our benchmark is technology-independent, thus the results provide potentially useful feedback for different communities.

Compared to other benchmarks, the Train Benchmark has the following distinguishing features:

- The workload profile follows a *real-world model validation scenario* by updating the model with changes derived by simulated user edits or transformations.
- The benchmark measures the performance of both initial validation and (more importantly) incremental revalidation.
- This *cross-technology benchmark* can be adapted to different model representation formats and query technologies. This is demonstrated by 10 reference implementations over four different technological spaces (EMF, graph databases, RDF, and SQL) presented here.

The benchmark is also part of the benchmark suite used by the MONDO EU FP7 project, along with other query/transformation benchmarks, such as the ITM Factory Benchmark[10], the ATL Zoo Benchmark[11] and the OpenBIM Benchmark[12].

5.5 Conclusion

In this chapter, we presented the *Train Benchmark*, a framework for the definition and execution of benchmark scenarios for modelling tools. The framework supports the construction of benchmark test sets that specify the metamodel, instance model generation, queries and transformations, result collection and processing, and metric evaluation logic that are intended to provide an end-to-end solution. As a main added value, this chapter contains a comprehensive set of measurement results comparing 10 different tools from four technological domains (EMF, graph databases, RDF, and SQL). These results allow for both intra-domain and cross-technology tool comparison and detailed execution time characteristics analysis.

Criteria for domain-specific benchmarks We revisit how our benchmark addresses the criteria given in the *Benchmark Handbook* [Gra93].

1. *Relevance:* The Train Benchmark measures the runtime for the continuous revalidation of well-formedness constraints used in many industrial and academic design tools. It considers two separate practical scenarios: small model changes for manual user edits, and larger changes for automated refactorings.
2. *Portability:* We presented the results for 10 implementations from four different technological domains in this chapter. There are multiple other implementations available in the repository of the project.
3. *Scalability:* The size of underlying models ranges from 5000 to 19 million model elements (triples), while there are 6 queries of different complexity.

[10] https://github.com/atlanmod/mondo-itmfactory-benchmark
[11] https://github.com/atlanmod/mondo-atlzoo-benchmark
[12] https://github.com/atlanmod/mondo-openbim-benchmark

4. *Simplicity:* A simplified, EMF version of the Train Benchmark was used as part of the 2015 Transformation Tool Contest [l12] where experts of four other tools managed to come up with an implementation, which indirectly shows the relative simplicity of our benchmark. An earlier version of the Train Benchmark was also used in [Var+15] to assess the efficiency of various search plans.

Software engineering aspects From a software engineering perspective, the Train Benchmark has been continuously developed and maintained since 2012. The benchmark is available as an open-source project, implemented in Java 8. The benchmark workloads (scenario, models, queries, transformation, number of runs, etc.) are specified and executed in a Groovy script [Kni+15]. The project has end-to-end automation [l10] to perform the following tasks: (1) Set up configurations of benchmark runs. (2) Generate large model instances. (3) Execute benchmark measurements. (4) Analyse the results and synthesize diagrams using R scripts [R C18].

The project provides continuous integration using the Gradle build system [Mus14], and contains automated unit tests to check the correctness of the implementations. Parts of the visualization framework were used to analyse solution results in the Transformation Tool Contest, both for regular [Hin17] and live cases [Hin18b].

5.6 Future Work

Possible future extensions to the Train Benchmark include adding more implementations to the benchmark from all technological spaces, including Epsilon [Pai+09] (EMF), OrientDB[13] (property graphs), PostgreSQL [Mom00] (SQL), Instans [RNT12] and Virtuoso [EM09a; EM09b] (RDF). Based on our work on graph metrics Chapter 3, it would be interesting to further investigate the correlation between query performance and metrics (query metrics, graph metrics, and query on graph metrics), similarly to the approach presented in [Izs+13b] and recently in [c8].

[13] https://orientdb.com/

Chapter 6

The Business Intelligence Workload of the LDBC Social Network Benchmark

This chapter is based on the first peer reviewed publication of the LDBC Social Network Benchmark's Business Intelligence workload [l18] and its detailed technical report authored by the *Social Network Benchmark Task Force* [r22].

6.1 Introduction

Benchmarks with aggregation-heavy OLAP-like Business Intelligence (BI) workloads on graphs are still a rather unexplored area. While there are many existing graph benchmarks (see Chapter 7), existing proposals do not fully capture the complex nature of graph BI workloads. Currently, the only benchmark with global queries and aggregations on graph-like data is the Berlin SPARQL Benchmark's BI use case [BS09] (discussed in Tab. 7.2). However, while proposed on RDF, it is exactly equivalent to and exists in a SQL variation on flat tables in a star schema, i.e. its dataset lacks a true graph structure and its queries thus do not require graph functionality.

Graph BI workloads differ from other types of graph query workloads in that large portions of the graph are explored in search of occurrences of graph patterns. Compared to graph analytics workloads, the patterns under search combine both structural and attribute predicates of varying complexity [SEH12], from basic graph patterns [Ang+18] to more complex unbound patterns that rely on different reachability semantics, e.g. paths, trails (see Sec. 2.5.2). The identified patterns are typically grouped, aggregated, and sorted to provide succinct results, which are used to assist the user in critical decision making.

BI workloads on graphs are particularly challenging because they usually lead to large search spaces and consequently, to large intermediate results. Thus, systems that are not prepared to efficiently prune the search space – by finding good graph traversal orderings, leveraging reachability indexes, or taking advantage of top-k semantics to progressively reduce the size of candidate results – are heavily penalized. Moreover, the peculiar structure of real graphs induces difficult-to-predict, scattered memory access patterns, which can limit the memory bandwidth saturation by orders of magnitude if computations are not arranged correctly [Sha+17]. Finally, some complex graph patterns become difficult to express even with the most advanced query languages, leading to large and verbose queries, which are difficult to write and maintain.

In this chapter, we present the LDBC SNB Business Intelligence benchmark (LDBC SNB BI), an industry-graded graph BI benchmark for graph processing systems as a result of many interactions between industry, academia, and graph practitioners. LDBC SNB BI was designed along established guidelines for creating application-specific benchmarks [BDT83; Gra93; Hup09]. It is a *macrobenchmark* consisting of 25 queries on top of a synthetically generated social network graph with a rich data schema and correlated attributes. By following a choke point-based approach, LDBC SNB BI queries are carefully designed to reproduce the challenging aspects of real workloads while keeping the workload realistic so it can be expressed by existing graph systems.

The LDBC SNB Task Force has already created reference implementations using the Sparksee engine [Mar+07; MGE11], as well as declarative query languages such as openCypher [Fra+18], PGQL [Res+16], SPARQL [PAG09], and SQL.[1] Research and development on the BI workload is ongoing, with current work focusing on extending the queries and adding updates to the workload.

6.2 Benchmark Design

The LDBC SNB BI workload consists of 25 read queries that have been carefully designed around a set of *choke points* (CPs) [BNE13]. A choke point is *a well-chosen difficulty in the workload*. In other words, a CP is a particularly challenging aspect of evaluating certain types of queries on a given data set. CPs are determined so that (1) they are not commonly solved by systems at *benchmark design time*, i.e. a considerable amount of systems to not handle them well, (2) they may be tackled with different technical solution and/or algorithmic optimizations, and (3) they are likely to occur in actual or near-future database workloads. Choke points present specific optimization opportunities, which systems must identify to allow efficient processing of larger data volumes. Many of the choke points were adopted from the ones identified in [Erl+15] and extended the list with new graph and language-specific ones, detailed in Sec. 6.2.1. Similarly to the *Interactive workload* of the LDBC Social Network Benchmark, the *BI workload* uses query templates that contain parameters to be substituted with bindings from the corresponding domain of the data set (e.g. Persons) [Erl+15].

6.2.1 Choke Point-Based Query Design

To design the queries, the SNB task force has followed an iterative process where the connection between CPs and queries has been progressively updated. At each step, either a new query was proposed or an existing one was updated, so that each query fulfils at least three CPs, and each CP appears in at least one query. Additionally, the exercise of implementing the queries has helped us to reconsider some choke point assignments, either by adding some previously unforeseen ones or unassigning choke points from queries (as they turned out to be irrelevant in practice). This also revealed how dependent the impact of one choke point is on the scale factor and/or the input parameters, thus putting more pressure on the query optimizer and making the benchmark more challenging.

The task force carefully designed the queries to be expressible using state of the art query languages and represent realistic BI operations one would perform on a social network (e.g. *Popular topics in a Country*, *Tag evolution*, or *Trending Posts*). At the same time, we have tried to push the expressivity of existing query languages to their limits by formulating queries that are difficult to express. The detailed description of the choke points and final relations between queries and CPs can be found in Sec. D.1. In the following lines, we detail the new graph-specific and language choke points.

[1] https://github.com/ldbc/ldbc_snb_implementations

6.2.2 Graph-Specific Choke Points

CP-7.1 Incremental path computation This choke point tests the ability of the execution engine to reuse work across graph traversals. For example, when computing paths within a range of distances, it is often possible to incrementally compute longer paths by reusing paths of shorter distances that were already computed.

CP-7.2 Cardinality estimation of transitive paths This choke point tests the ability of the query optimizer to properly estimate the cardinality of intermediate results when executing transitive paths. A transitive path may cover a tree or a graph, e.g. descendants in a geographical hierarchy vs. graph neighbourhood or transitive closure in a many-to-many connected social network. In order to decide proper join order and type, the cardinality of the expansion of the transitive path needs to be correctly estimated. This could for example take the form of executing on a sample of the data in the cost model or of gathering special statistics, e.g. the depth and fan-out of a tree. In the case of hierarchical dimensions, e.g. geographic locations or other hierarchical classifications, detecting the cardinality of the transitive path will allow one to go to a star schema plan with scan of a fact table with a selective hash join. Such a plan will be on the other hand very bad for example if the hash table is much larger than the "fact table" being scanned.

CP-7.3 Efficient execution of a transitive step This choke point tests the ability of the query execution engine to efficiently execute transitive steps. Graph workloads may have transitive operations, for example finding a shortest path between nodes. This involves repeated execution of a short lookup, often on many values at the same time, while usually having an end condition, e.g. the target node being reached or having reached the border of a search going in the opposite direction. For the best efficiency, these operations can be merged or tightly coupled to the index operations themselves. Also parallelization may be possible but may need to deal with a global state, e.g. set of visited nodes. There are many possible tradeoffs between generality and performance.

CP-7.4 Efficient evaluation of termination criteria for transitive queries This choke pont tests the ability of a system to express termination criteria for transitive queries so that not the whole transitive relation has to be evaluated as well as efficient testing for termination.

6.2.3 Language Choke Points

CP-8.1 Complex patterns A natural requirement for graph query systems is to be able to express complex graph patterns. Here, we focus on two particularly challenging aspects:

- *Transitive navigation.* Transitive closure-style computations are common in graph query systems, both with fixed bounds (e.g. get nodes that can be reached through at least 3 and at most 5 `knows` edges), and without fixed bounds (e.g. get all `messages` that a `comment` replies to).
- *Negative edge conditions.* Some queries define *negative pattern conditions.* For example, the condition that a certain `message` does not have a certain `tag` is represented in the graph as the absence of a `hasTag` edge between the two nodes. Thus, queries looking for cases where this condition is satisfied check for negative patterns, also known as negative application conditions (NACs) in graph transformation literature [HHT96].

CP-8.2 Complex aggregations BI workloads are heavy on aggregation, including queries with *subsequent aggregations*, where the results of an aggregation serves as the input of another aggregation. Expressing such operations requires some sort of query composition or chaining (see also CP-8.4). It is also common to *filter on aggregation results* (similarly to the `HAVING` keyword of SQL).

CP-8.3 Windowing queries Additionally to aggregations, BI workloads often use *window functions*, which perform aggregations without grouping input tuples to a single output tuple. A common use case for windowing is *ranking*, i.e. selecting the top element with additional values in the tuple (nodes, edges or attributes).[2]

CP-8.4 Query composition Numerous use cases require *composition* of queries, including the reuse of query results (e.g. nodes, edges) or using scalar subqueries (e.g. selecting a threshold value with a subquery and using it for subsequent filtering operations).

CP-8.5 Dates and times Handling dates and times is a fundamental requirement for production-ready database systems. It is particularly important in the context of BI queries as these often calculate aggregations on certain periods of time (e.g. on a month).

CP-8.6 Handling paths To take full advantage of the graph data model, systems should be able to perform complex operations on paths in the graph [Ang+18]. Hence, additionally to reachability-style checks, a language should be able to express queries that operate on elements of a path, e.g. calculate a score on each edge of the path. Also, some use cases specify uniqueness constraints on paths [Ang+17]: *arbitrary path*, *shortest path*, *no-repeated-node semantics* (also known as *simple paths*, see Def. 23, and *no-repeated-edge semantics* (also known as *trails*, see Def. 22). Other variants are also used in rare cases, such as *maximal* (non-expandable) or *minimal* (non-contractable) paths, *longest trails*, and *longest simple paths*.

6.2.4 Benchmark Data Set

The Business Intelligence workload adopts the LDBC SNB data generator to generate synthetic social networks with realistic characteristics. We adopt the scale factor (SF) notion proposed for LDBC SNB, which is based on the accumulated file size of the CSV files on disk (e.g. SF1 is approx. 1 GB, SF3 is approx. 3 GB, etc.). We also adapted the *parameter curation* concept [GB14a] to generate parameter bindings for queries and extended the data generator with the new parameter generator component.[3]

The schema of the social network graph is shown in Fig. 6.1. For a detailed description of the graph schema and the generator, we refer the reader to [Erl+15] and the Social Network Benchmark technical report [r22]. In this work, we summarize the characteristics of the produced social network graph that make it an ideal data set for benchmarking a graph BI workload.

Complex schema Graphs contain a rich schema consisting of different entities. This allows the design of queries with rich and complex patterns requiring both small and large projections. These stress a number of choke points including:

[2] PostgreSQL defines the `OVER` keyword to use aggregation functions as window functions, and the `rank()` function to produce numerical ranks, see https://www.postgresql.org/docs/9.1/static/tutorial-window.html for details.

[3] https://github.com/ldbc/ldbc_snb_datagen

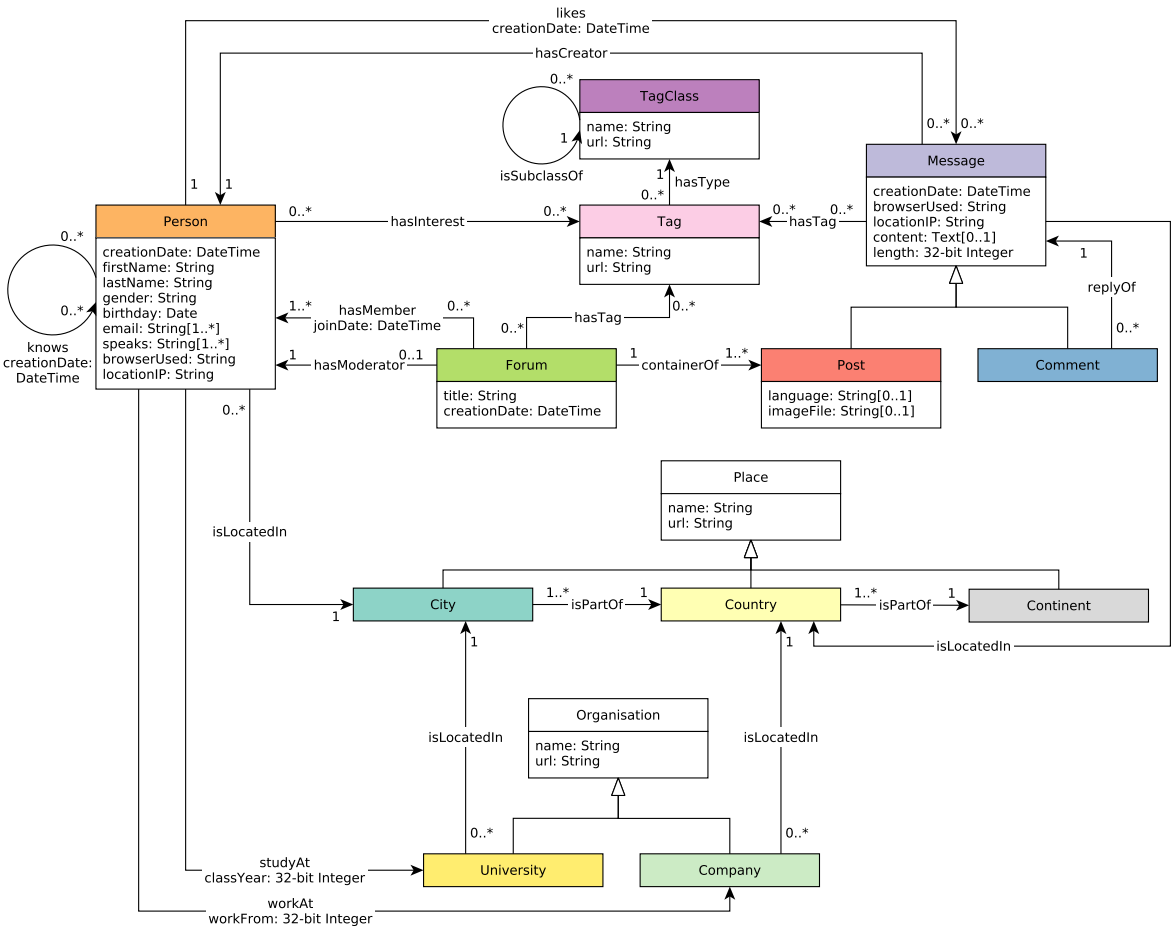


Figure 6.1: Graph schema of the LDBC Social Network Benchmark.

- *CP-1.2 High cardinality group-by performance*, which enables the engine to efficiently perform grouping operations when there are many different groups;
- *CP-1.4 Low cardinality group-by performance*, which enables specific optimizations when there are only few different groups;
- *CP-2.2 Late projection* which tests the ability of the optimizer to defer the projection of attributes not required until later phases of the evaluation.

The schema also has various datetime types (e.g. for storing a Person's birthday or the creation time of a Message), which stresses *CP-8.5 Dates and times*.

Correlated attributes Second, graphs are correlated, i.e. Persons with similar characteristics are more likely to be connected or where the values of the attributes of a given entity are correlated. Systems can exploit such correlations to leverage more compressed means of storing the graph or to improve data access locality via clustered indexes. Such optimization opportunities are specifically captured by choke points *CP-3.1 Detecting correlation* or *CP-3.2 Dimensional clustering*.

Realistic structure Finally, structural characteristics are also realistic, with the degree distributions of the knows edge type being Facebook-like [PD14] and the largest connected component containing a significant portion of the overall network. Such features in the graph allow us to stress choke points

CP-3.3 Scattered index access patterns, which tests the ability of the execution engine to efficiently access indexes using keys that are scattered, which is usually the case when performing traversals of more than one hop; and the new newly added *CP-7.2 Cardinality estimation of transitive paths* and *CP-7.3 Efficient execution of a transitive step* (Sec. 6.2.1).

6.3 Detailed Query Discussion

All queries present in the BI workload share common data access characteristics as they touch large portions of the social network graph and rely heavily on aggregation operations [Gra+97]. In contrast to *graph analytics* (Sec. 3.9.2), queries not only access the graph topology but use node/edge attributes. In the following, we discuss three example queries in detail, and show how properly dealing with different choke points can highly impact query evaluation time, revealing the relevance of choke point-based benchmark design and the proposed queries. The list of CPs is presented in Sec. D.1, while queries are listed in Sec. D.2.

To assess the complexity of each query and potential impact of optimization techniques, we ran multiple performance experiments. These were implemented in C++ on top of the Sparksee native graph database [Mar+07; MGE11], and evaluated on SF1 and SF10 data sets. Benchmarks were executed on a cloud VM with 8 Xeon E5-2673 CPU cores and 256 GB RAM, running Ubuntu 16.04. Detailed results for all 25 queries and multiple systems are available in Sec. 6.4.

6.3.1 Top Posters in a Country (Query 5)

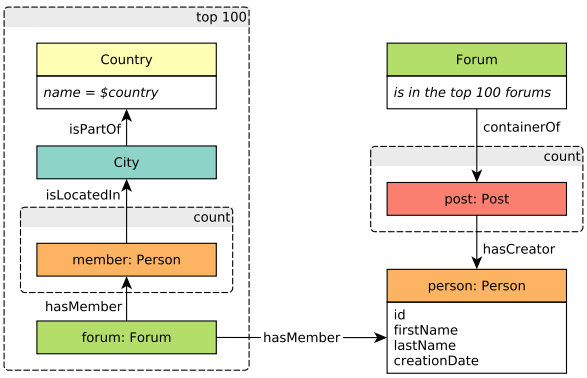


Figure 6.2: Graph pattern for BI Query 5.

Scale Factor	Forum to Country	Country to Forum
SF1	4 848	64
SF10	57 637	349

Table 6.1: Execution times of BI Q5 with different traversal directions.

Definition Find the most popular Forums for a given Country, where the popularity of a Forum is measured by the number of members that Forum has from the given Country. Calculate the top 100 most popular Forums. In case of a tie, the forum(s) with the smaller id value(s) should be selected.

For each member Person of the 100 most popular Forums, count the number of Posts (`postCount`) they made in any of those (most popular) Forums. Also include those member Persons who have not posted any messages (i.e. have a `postCount` of 0). The graph pattern of the query is shown in Fig. 6.2.

Performance CPs Q5 represents a simple graph pattern matching query with a top-k evaluation and a set of aggregation operations. Besides its simplicity, this query is highly relevant for several reasons: first, this is one of the queries that fulfils a number of choke points (10 in total), including those related to *CP-2.1 Rich join order optimization*, *CP-1.3 Top-k pushdown*, and *CP-2.2 Late projection*. Second, this query reveals that graph database systems must not only provide support for purely graph-specific operations but also non-graph operations, such as aggregations and top-k evaluation, to answer realistic graph BI queries efficiently.

To demonstrate the importance of the choke points fulfilled by this query, we focus on the first and most time-consuming part of the query, which looks for patterns connecting Forums to Persons living in a Country (the latter being provided as a query parameter). When looking for occurrences of such a pattern, a system has several alternatives to navigate the graph. For instance, one option is to navigate the graph from Forums to Persons, and then filter out those occurrences with Persons that are not locatedIn the Country in question. As an alternative, the system could also first obtain the Persons that belong to the Country, and then retrieve the Forum neighbors via the hasMember relationship.

Properly selecting the right strategy can heavily impact the query time, sometimes by orders of magnitude. Tab. 6.1 depicts the average query evaluation time for the two proposed traversal strategies. The execution times reveal that by following the second traversal evaluation strategy, the average execution time is two orders of magnitude lower on both scale factors. This particular example represents an instance of *CP-2.1 Rich join order optimization*, since traversing a graph can be interpreted as a sequence of joins. Other queries fulfilling this CP are Q2, Q4, Q9, Q10, Q11, Q19, Q20, Q21, Q22, Q24, and Q25. The large number of queries is caused by the fact that *navigation along edges* is a key operation in graph BI workloads.

Language CPs This query first performs an aggregation to determine the popularity of Forums, then sorts them, and selects the 100 most popular ones, and continues the computation with these. This covers two key language CPs: *CP-8.2 Complex aggregations* (to perform ordering on aggregations results) and *CP-8.4 Query composition* (to continue with results of the subquery). Due to the BI nature of the benchmark, complex aggregation (CP-8.2) is required by approx. 50% of the queries, 12 in total. Composition (CP-8.4) is also an important feature, required by Q10, Q15, Q18, Q21, Q22, and Q25.

6.3.2 Experts in Social Circle (Query 16)

Definition Given a Person, find all other Persons that live in a given Country and are connected to given Person by a transitive path *with no-repeated-edge semantics* (Sec. 2.5.2) of length in range `[minPathDistance, maxPathDistance]` through the knows relation. In the path, an edge can be only traversed once while nodes can be traversed multiple times. For each of these Persons, retrieve all of their Messages that contain at least one Tag belonging to a given TagClass (a direct relation is required, a transitive one is not sufficient). For each Message, retrieve all of its Tags. Group the results by Persons and Tags, then count the Messages by a certain Person having a certain Tag. The graph pattern of the query is shown in Fig. 6.3.

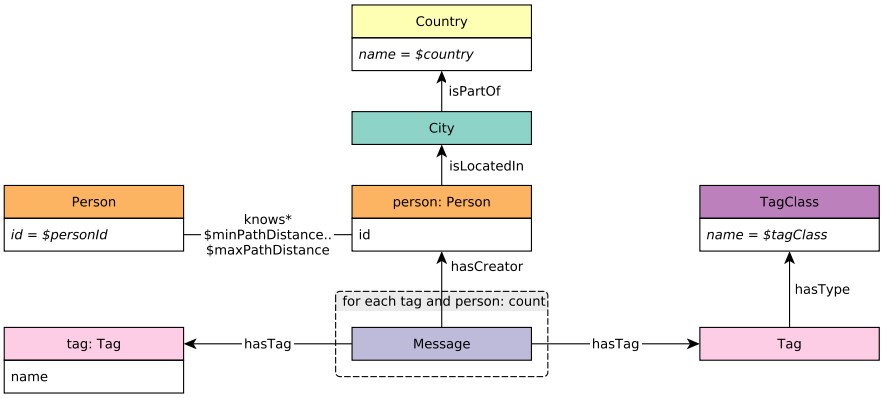


Figure 6.3: Graph pattern for BI Q16.

Scale Factor	Baseline	Top-k pushdown	Top-k pushdown with path pattern reuse
SF1	1 552	1 463	702
SF10	29 312	27 398	15 037

Table 6.2: Execution times of BI Q16 with different optimization strategies.

Performance CPs One way to evaluate this query is to first find all Persons reachable from the given Person and belonging to the input Country. Then, for each of these Persons, look for the Messages they created and their Tags to obtain the final result set and return the top-k elements.

However, a sophisticated query optimizer might be able to infer that the maximum number of Messages with a given Tag for a given Person can be at most the Person's total number of Messages. Thus, the system might first sort the reachable Persons by their Message count in descending order and start counting their Messages' Tags while maintaining a priority queue with the top-k results. Once the "total number of Messages of the next Person to evaluate" value is smaller than the last entry in the top-k (assuming this already contains k elements), the query evaluation can abort exploring more Persons, hence exploiting *CP-1.3 Top-k pushdown* [DR99]. In the first and second column of Tab. 6.2, we show the results of applying the optimization (for a top-100 case). We see that applying it decreases execution time by around 6%. Although this number might not seem significant, top-k pushdown is one of the most important choke points of the workload. The benefit of applying such optimization will depend on the SF and the size of the top. For some SFs, queries such as Q22 would become untractable if top-k pushdown was not exploited, since it would require comparing all Persons in one country to all Persons in another country. Also, other queries that rely on this choke point to be executed efficiently are Q2, Q4, Q5, Q9, and Q19.

Complementing the top-k pushdown optimization, a system could also try to compute the reachable Persons incrementally, instead of computing them at the beginning of the execution. As the Persons belonging to the input Country are explored (sorted in descending order by their Message count), a reachability set can be updated while checking whether the currently evaluated Person is reachable or not. Thus, before performing the expensive reachability test, we can check whether a certain Person has been already observed in an earlier Person reachability test. This optimization is an example of an exploitation of *CP-7.1 Incremental path computation*, which for this query would result in an improvement of approx. 1.8–2× over simple top-k pushdown, as shown in Tab. 6.2. Reachability

indexes are a form of reusing patterns and are an active research area [YCZ12]. This CP was partially designed to stimulate such research efforts. Other queries that fulfil this CP are Q19 and Q25.

Language CPs This query uses transitive paths with variable bounds, which is a key challenge in *CP-8.1 Complex patterns*. Other queries that stress this CP are Q8, Q11, and Q19 (negative edge conditions), and Q14, Q18, Q19, Q20, and Q25 (transitive paths). Due to the edge-uniqueness constraint, this query also relies on the language supporting *no-repeated-edge semantics* (Sec. 2.5.2), which is captured by *CP-8.6 Handling paths*.

6.3.3 Weighted Interaction Paths (Query 25)

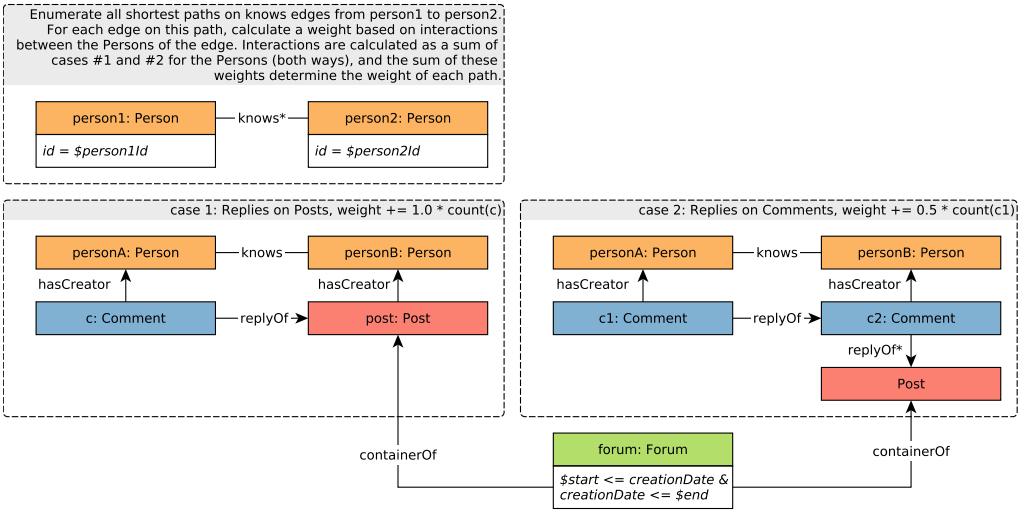


Figure 6.4: Graph pattern for BI Q25.

Scale Factor	Baseline	Result reuse
SF1	622	419
SF10	2 885	1 939

Table 6.3: Execution times of BI Q25: baseline and with result reuse.

Definition Given two Persons, find all (unweighted) shortest paths between these two Persons, in the subgraph induced by the knows relationship. Then, for each path calculate a weight. The nodes in the path are Persons, and the weight of a path is the sum of weights between every pair of consecutive Person nodes in the path. The weight for a pair of Persons is calculated based on their interactions:

- Every direct reply (by one of the Persons) to a Post (by the other Person) contributes 1.0.
- Every direct reply (by one of the Persons) to a Comment (by the other Person) contributes 0.5.

Only consider Messages that were created in a Forum that was created within the timeframe `[startDate, endDate]`. Note that for Comments, the containing Forum is that of the Post that the comment (transitively) replies to. Return all paths with the Person ids ordered by their weights descending. The graph pattern of the query is shown in Fig. 6.4.

Performance CPs This query looks first for all the shortest paths (Def. 25) between a given pair of Persons, and for each of them computes a score based on the interactions between each pair of consecutive Persons in the path. In a realistic graph – such as the one in the benchmark – it is likely that there exists a large overlap between such shortest paths, especially if the length of the shortest paths is relatively large. Such an overlap implies that many of the subqueries used to compute the path's score are essentially the same, and thus their results can be reused. This is an application of *CP-5.3 Intra-query result reuse*. Tab. 6.3 shows the average execution time on SF1 and SF10, with or without reusing the subquery results. We see that thanks to the optimization, we obtain a reduction in the execution time of approx. 30%. Moreover, as mentioned above the evaluation time of this query is highly sensitive to the length of the shortest path, since the longer these are, the more likely is to find more and more overlapping paths. Thus, we also compared the execution times between of the slowest instance of Q25, and found that for such instance the obtained speedup thanks to this optimization was approx. 2×. Other queries where this CP can be exploited are Q3, Q5, Q15, Q21, and Q22.

Language CPs This query stresses language features, covering all related CPs, including an important aspect of *CP-8.6 Handling paths*. In particular, it calculates the weight of a path based on interactions of consecutive nodes on the path, which is often difficult to express in existing languages. This has been recognized by recent language design efforts: the G-CORE language (Sec. 2.6.3) defines paths as part of its property graph data model, which defining queries on paths.

6.4 Benchmark Results

Systems We performed benchmarks on multiple database systems and analytical engines:

- the Oracle Labs PGX graph analytical system [Hon+15] (with queries formulated in the declarative PGQL language (Sec. 2.6.2),
- the PostgreSQL relational database management system (with queries formulated in PostgreSQL's SQL dialect [Mom00]), and
- the Sparksee native graph database [MGE11] (with queries formulated in imperative C++ code).

Environment Benchmarks for PostgreSQL and Sparksee were executed on a cloud VM with 8 Xeon E5-2673 CPU cores and 256 GB RAM, running Ubuntu 16.04. Benchmarks for Oracle Labs PGX were executed on 16 Xeon E5-2660 CPU cores and 256 GB RAM, running Oracle Linux Server 6.8.

Methodology We executed 100 queries for warmup, then executed 250 queries and measured their response time. Queries were selected randomly, following a uniform distribution and were executed one-by-one, i.e. with no interleave between them. For each scale factor/tool/query, we calculated the geometric mean of execution times (as recommended in [FW86]).

Results Fig. 6.5 shows preliminary benchmark results: namely, the execution times for all 25 queries specified in the Business Intelligence workload on 3 systems (Oracle Labs PGX, PostgreSQL, and Sparksee). Fig. 6.6 shows benchmark results for the Interactive workload [Erl+15]. While both result sets are preliminary (i.e. they are *validated*, but *not audited*), it is clear from that the Business Intelligence workload is significantly more complex than the Interactive workload. On PostgreSQL, many of the BI queries (e.g. Q7, Q11, Q12, Q14, Q18, Q19, Q22, Q24) take more than 30 seconds to complete for the SF10 data set, while all of the Interactive queries except Q14 take less than 10 seconds.

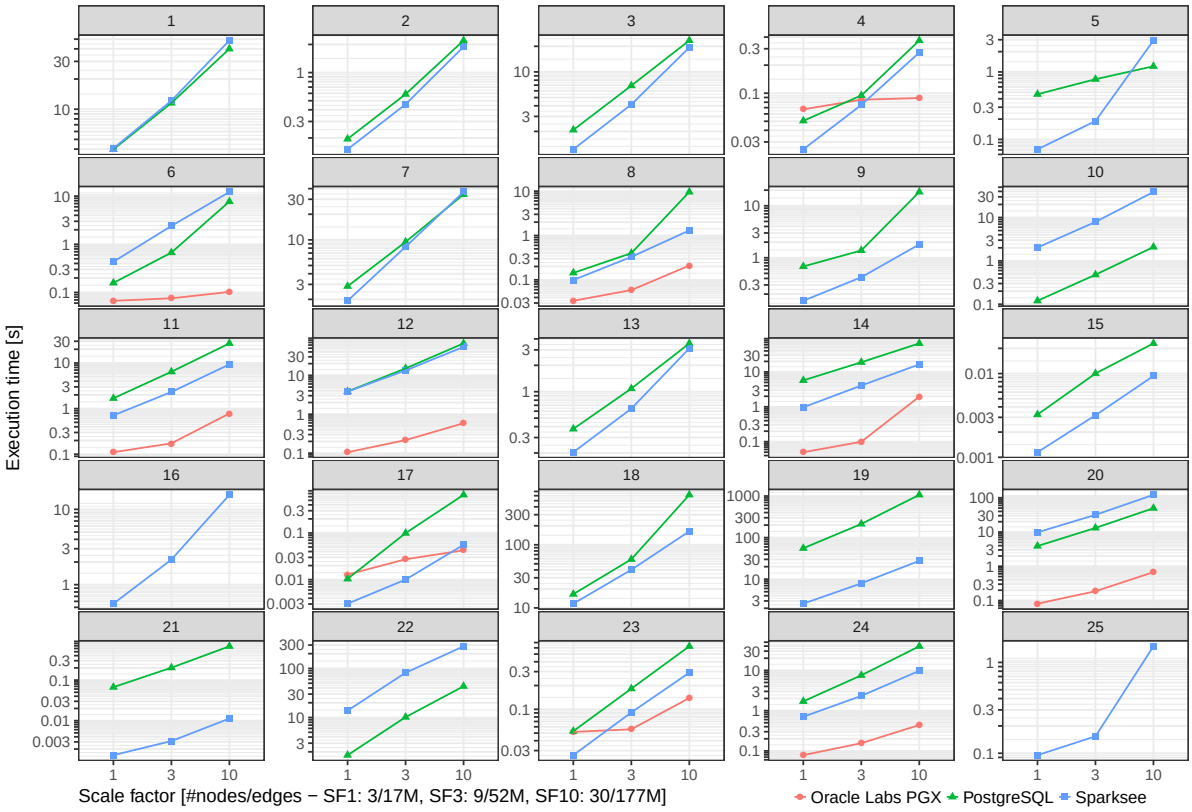


Figure 6.5: Execution time of the 25 BI queries in an in-memory hybrid graph query and analytical engine (Oracle PGX), a disk-based relational database (PostgreSQL), and an in-memory graph database (Sparksee).

Comparison to the Train Benchmark Compared to the Train Benchmark, the LDBC SNB strives less for simplicity but more towards systematically benchmarking relevant aspects of a given query engine. This is shown in their expected development times: for a given tool, a new initial implementation for the Train Benchmark can usually be created by an experienced developer in a few days. Implementing the LDBC SNB BI workload typically takes weeks to complete.

6.5 Conclusion and Future Work

In this chapter, we have presented our early work on LDBC SNB BI, a graph processing systems' benchmark for graph BI workloads. LDBC SNB BI combines a set of 25 carefully designed queries with a synthetic social network dataset to achieve a realistic yet challenging workload. We share our experiences on implementing the benchmark and showcase the benefits of the choke point-based design by means of a detailed discussion on three example queries. Moreover, we presented results for three different systems (Oracle Labs PGX, PostgreSQL, and Sparksee) and comparison to the Interactive workload on two systems.

Our experiences reveal how the designed queries capture the complexity of graph BI workloads by offering optimization opportunities that, if not taken into account, would make the evaluation of queries infeasible for large scale factors. We also studied the language aspects of the queries, making

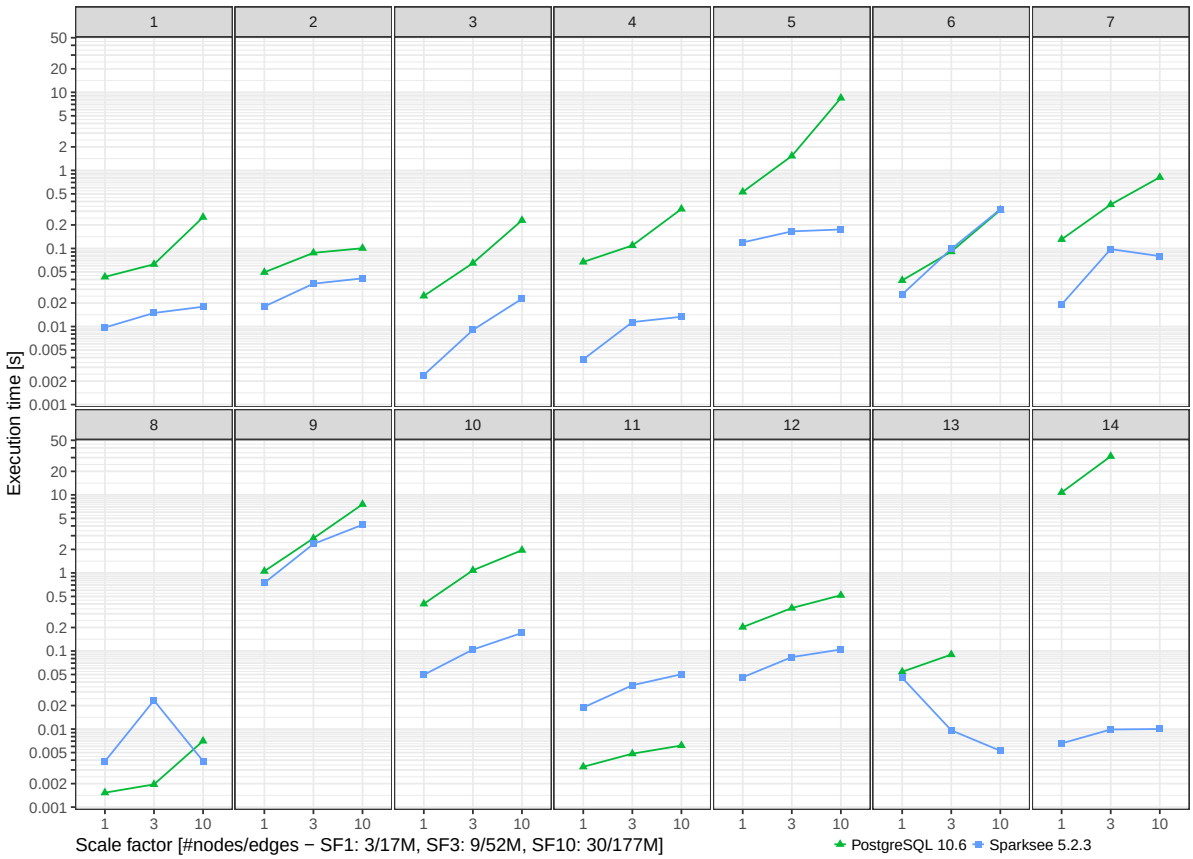


Figure 6.6: Execution time of the Interactive workload's 14 *complex read* queries. (*Short reads* and *updates* were omitted.) Results show that when executed on PostgreSQL with data sets up to SF10, only queries 5, 9, 13, and 14 require more than 2 seconds to execute.

them expressible using existing declarative query languages (openCypher, PGQL, SPARQL, and SQL), which also revealed some deficiencies of the languages.

As of 2019, the contours of LDBC SNB BI are clear, but the current specification is not yet complete. The next steps are adding more queries and introducing updates (including deletions) into the workload. We consider the scenario of systems that query a static snapshot of the graph, and from time to time receive batch updates that must be incorporated into the database. However, we also want the benchmark to match the capabilities of those systems that offer queryable snapshots while accepting update streams in parallel. Updates will tie into the overall performance metric function, which we will define in the complete benchmark. Such a function will need to consider two key aspects:

1. All queries are equally important regardless their data complexity, which favours *geometric mean* instead of *arithmetic mean* [FW86].
2. The function should accommodate different ways of accepting updates (stream vs. batch).

Finally, we plan to provide additional reference implementations in languages such as the Gremlin [Rod15], and G-CORE [Ang+18], a composable property graph query language, designed by the *Graph Query Language Task Force* of the Linked Data Benchmark Council.

Chapter 7

Related Graph Benchmarks

Numerous benchmarks have been proposed to measure and compare the performance of query and transformation engines in a specific technological space for a given use case. However, no openly available cross-technology benchmarks have been proposed for the continuous model validation and graph BI scenarios. Below, we overview the main existing benchmarks for model query and transformation (Sec. 7.1) as well as RDF technologies (Sec. 7.2).

7.1 Model Transformation and Graph Transformation Benchmarks

7.1.1 Benchmarks for Graph and Model Transformation

Up to our best knowledge, the first transformation benchmark was proposed in [VSV05], which gave an overview on typical application scenarios of graph transformations together with their characteristic features. The paper presents two cases: the *Petri net firing simulation* case and the *object-relational mapping by model synchronization* case. While both are capable of evaluating certain aspects of incremental query performance, they provide a different workload profile (e.g. model and query characteristics) than typical well-formedness validation scenarios. [GK07] suggested some improvements to the benchmarks of [VSV05] and reported measurement results for many graph transformation tools.

7.1.2 Tool Contests

Many transformation challenges have been proposed as cases for graph and model transformation contests. Most of them do not focus on query performance, instead, they measure the usability of the tools, the conciseness and readability of the query languages and tests various advanced features, including reflection, traceability, etc. The 2007 contest was organized as part of the AGTIVE conference [SNZ08], while the 2008 and 2009 contests were held during the GRaBaTS workshop [RG10; LRG09]. The contests in 2010, 2011, 2013, and onwards were organized as a separate event, the Transformation Tool Contest (TTC) [MRV10; VMR11; VRK13; RKH14; RHK15; GKR16; GHK17].

Year	Case	Goal	Scope	t2m	m2m	m2t	Upd.	Perf.
2018	Quality-based SS&HM	Optimize software variant selection and hardware mapping.	optimization	○	⊗	○	○	⊗
	Social Media (L)	Global queries and aggregations on a social network graph under updates.	queries under updates	○	⊗	○	⊗	⊗
2017	Smart Grid	Maintain incremental model views for a large cyber-physical system.	queries under updates	○	⊗	○	⊗	⊗
	Families to Persons	Run bidirectional transformation between families (mother, father, sons, and daughters) and persons.	bidirectional trf.	○	⊗	○	⊗	⊗
	State Elimination	Transform finite state automata to regular expression using state elimination.	model transformation	○	⊗	○	○	⊗
	Reuse with Redefinitions (L)	Transformation reuse in the presence of multiple inheritance and redefinitions.	scope	○	⊗	○	○	⊗
2016	Class Resp. Assignment	Given methods, attributes and their dependencies, find the optimal class diagrams	scope	○	⊗	○	○	⊗
	Dataflow Transformations (L)	Execute dataflow-based model transformation engine and run 4 transformations in batch/incremental mode: Families to Persons, Tree to Graph, Class to RDB, Flowchart to HTML.	incremental transformation	○	⊗	○	⊗	⊗
2015	The Train Benchmark	Perform well-formedness validations and quick fix-like repair transformations.	queries under updates	○	⊗	○	⊗	⊗
	The Model Execution Case	Define a transformation specifying the operational semantics of the UML activity diagram language.	model execution	○	⊗	○	○	⊘
	The Java Refactoring Case	Parse the source code, perform refactorings on the program graph and generate the source code.	refactoring	⊗	⊗	⊗	○	○
	Java Annotations (L)	Use annotations to extend existing Java code.	refactoring	⊗	⊗	⊗	○	○
2014	FIXML to Java, C#, and C++	Transform financial transaction data expressed in FIXML format into class definitions in Java/C#/C++.	deserialization	⊗	⊗	⊗	○	○
	Movie Database	Determine all actor couples who performed together in a set of at least three movies.	queries and update	○	⊗	○	○	⊗
	Soccer Worldcup (L)	Implement a soccer client, using model transformations.	AI	○	⊗	○	⊗	○
2013	Petri-Nets to Statecharts	Mapping from Petri-Nets to statecharts.	synthesis	○	⊗	○	○	⊗
	Class diagram restructuring	Perform refactoring operations: pull up, create superclass, create subclass.	program refactoring	○	⊗	○	○	○
	Flowgraphs	Analysis and transformations in compiler construction: data structures, control/data flow graphs.	synthesis, validation	○	⊗	⊗	○	○
2011	GMF Model Migration	Migrate models in response to metamodel adaptation.	model migration	○	⊗	○	○	○
	Compiler Optimization	Perform local optimizations on the intermediate code representation, and apply instruction selections.	compiler optimization	○	⊗	○	○	⊗
	Program Understanding	Create a state machine model out of a Java syntax graph.	synthesis	○	⊗	○	○	⊘
	Hello World	Several primitive tasks that can be solved straight away with most transformation tools.	various	○	⊗	⊗	⊗	○
2010	Model Migration	Define a transformation to migrate the activity diagrams from UML 1.4 to UML 2.2.	model migration	○	⊗	○	○	○
	Dynamic Comm. Systems	Compute topologies that may occur for the merge protocol, a communication protocol used in car platooning.	synthesis	○	⊗	○	○	⊗
	Ecore to GenModel	Use m2m transformation to synthesize the GenModel from the Ecore metamodel.	synthesis	○	⊗	○	○	○
2009	BPMN to BPEL	Define model transformations between BPMN and BPEL.	synthesis	○	⊗	○	○	○
	Program Comprehension	A simple filtering query on large models; a complex query on small models to produce code graphs.	synthesis	○	⊗	○	○	⊗
	Leader Election	Model and validate a simple leader election protocol using graph transformation rules and verification.	model verification	○	⊗	○	○	○
	Live Challenge Problem	Model a luggage system, define a transformation to a statechart and perform a simulation on it.	synthesis, simulation	○	⊗	○	○	○
2008	Program Refactoring	Import the models to GXL, allow for interactive transformations, export to GXL.	program refactoring	○	⊗	○	○	○
	AntWorld	Perform a simulation of ants searching for food based on a few simple rules.	model simulation	○	⊗	○	⊗	⊗
2007	Don't Get Angry; Ludo Game	Model and play a board game using graph transformation rules.	various	○	⊗	○	○	⊗
	UML to CSP Transformation	Perform a transformation from UML activity diagrams to Communicating Sequential Processes.	synthesis	○	⊗	○	○	○
	Sierpinski Triangle	Construct a Sierpinski triangle.	construction	○	⊗	○	⊗	⊗

Table 7.1: Cases presented at the transformation tool contest events between 2007 and 2018. Abbreviations – *Upd.:* updates, *Perf.:* performance-oriented. Notation for the *Case* column: (L) case for the live contest (which requires participants to solve to submit their solutions during the week-long conference). Notation for the *Perf.* column: ⊗ the case focuses on the performance of tools, ⊘ the case takes performance into consideration but it is not the main focus, ○ the performance of the tools is mostly irrelevant for solving the case.

Tab. 7.1 presents an overview of tool contest cases from 2007 to 2018. We shortly summarize their goal, scope and show whether solving them requires text-to-model (t2m), model-to-model (m2m) or model-to-text (m2t) transformations. We also denote whether the solution needs to perform updates on the model and whether the case explicitly measures the performance of the tools.

For the sake of conciseness, we only discuss cases that are potentially useful for measuring the performance of incremental model validation, meaning that (1) they are *performance-oriented*, i.e. include large models, complex patterns or both, (2) they measure the *incremental performance*, i.e. perform updates on the model and reevaluate the patterns.

AntWorld The *AntWorld* case study [RG10; GZ10] requires the solution to perform a simulation of ants searching for food based on a few simple rules. The environment, the ants, and the food are modelled as a graph, while the rules of the simulation are implemented with model transformation rules. Although this case study provides a complex queries and performs update operations on a large model, its workload profile is similar to a model simulation instead of a model validation scenario.

Sierpinski Triangle Generation The *Sierpinski Triangle Generation* [Tae+07] is another well-known transformation case, used in [KTG15]. The Sierpinski triangles are stored as a model and are generated using model transformations. The triangles can be modelled with a very simple metamodel and the characteristics of the instance models are very different from typical models used in MDE. While the required transformations are complex, the semantics of the transformation does not resemble any real-world applications.

Performance-oriented cases Other performance-oriented graph transformation cases include the *Movie Database* [HKT14], the *Petri-Nets to Statecharts* [VR13], and the GRaBaTS 2009 *Program Comprehension* [SJ09] cases. The latter case was used in papers [BK14; Sha+14] to benchmark the scalability of model persistence and query evaluation of graph and other NoSQL data stores. However, none of these perform update-and-reevaluation sequences.

The "Social Media" case A notable cross-over between the fields of *graph database benchmarking* and the *model transformations* was the live challenge of the 2018 Transformation Tool Contest (TTC 2018).[1] The challenge, titled the "Social Media" case requires participants to execute two queries with frequent updates in the data. The case uses a simplified schema of the LDBC SNB graph (the original is shown in Fig. 6.1), consisting of only Person, Message, Comment, and Post node types, as opposed to the 14 types used in the full SNB benchmark (presented in Chapter 6).[2] The case defines two global queries, which calculate scores for *controversial posts* and *influential comments*. The queries are to be evaluated continuously while the social network graph is updated with new nodes and edges (Posts, Comments, likes, etc.). While this case has very similar goals to the Train Benchmark and the Business Intelligence workload, it is less comprehensive then queries in other benchmarks.[3] However, it is worth pointing out that this benchmark poses a unique challenge not covered by the Train Benchmark or any current LDBC SNB query: it requires the enumeration of *strongly connected components*

[1] https://www.transformation-tool-contest.eu/2018/solutions_liveContest.html

[2] The social network graph used in the "Social Media" case is based on the data used in the DEBS 2016 Grand Challenge (http://debs.org/debs-2016-grand-challenge-social-networks/). This in turn is based on the SIGMOD 2014 Programming Contest (http://www.cs.albany.edu/~sigmod14contest/), where participants had to implement complex social network analysis queries, executed on a graph produced by an earlier version of the LDBC data generator.

[3] We recognize that this is partly due to the fact that the "Social Media" case was a live challenge, which participants have only a few days to solve.

(Def. 62) in the graph, which is difficult (and in some cases, impossible) to express in many popular query languages.

Participation in TTC events We participated in multiple TTC events: the 2014 *Movie Database* case [a23], the 2015 *Train Benchmark* case [l12], the 2015 *Java Refactoring* case [a24] (3rd prize), and the 2016 *Class Responsibility Assignment* [a25] (1st prize).

7.1.3 Assessment of Incremental Model Queries

In [Ber+08] and [Ber+10], authors aimed to design and evaluate model transformation benchmark cases corresponding to various usage patterns for the purpose of measuring the performance of incremental approaches on increasing model sizes. We assessed a hybrid model query approach (which combines local search and incremental evaluation) in [Hor+10] on the *AntWorld* case study [GZ10].

Queries are common means to implement *source code analysis*, which is traditionally a batch (and not continuous) validation scenario. Nevertheless, the performance of both local search-based and incremental model queries is assessed in [Ujh+15b] for detecting anti-patterns in source code transformed to EMF models. As model validation is an important use case for incremental model queries, several model query and/or validation tools have been assessed in incremental constraint validation measurements [RE12; Fal+14].

The Viatra CPS Benchmark[4] measures the performance of *model transformations* with a particular emphasis on repeated executions of the same transformation. It specifies a common workflow in model-driven engineering in the context of allocating components of a cyber physical systems. The workflow consists of three steps: First, (1) a system is described in a *source domain model*, then (2) it is transformed with model-to-model transformations to a *target domain model*. Finally, (3) *model-to-text transformation* is used to generate code from the domain model.

7.2 RDF Benchmarks

There are several well-defined performance benchmarks for assessing the performance of RDF technologies (listed in Tab. 7.2).

The *WatDiv* [Alu+14] benchmark defines a workload generator that allows users to fine-tune the *structuredness* of the synthesized graphs and the complexity of generated queries. *Stream WatDiv* [Gao+18] adds streaming updates to the workload, and equips SPARQL queries with time windows (using languages such as CQELS [Phu+11] and C-SPARQL [Bar+10]). The Linked Data Benchmark Council recently published the *Semantic Publishing Benchmark* (SPB) [Kot+16] based on a scenario of the BBC media organization. SPB combines a heavy query workload with a stream of update operations.

One of the first ontology benchmarks are the *Lehigh University Benchmark (LUBM)* [GPH05], and its improved version, the *UOBM Ontology Benchmark* [Ma+06a]. These are tailored to measure reasoning capabilities of ontology reasoners. Another early benchmark used the *Barton dataset* [Aba+07] for benchmarking RDF stores. The benchmark simulates a user browsing through the RDF Barton online catalog. Originally, the queries were formulated in SQL, but they can be adapted to SPARQL as well. However, the size of the graph is limited (50M elements) and there are no updates in the model.

SP²Bench [Sch+09] is a SPARQL benchmark that measures the execution time of various queries. The goal of this benchmark is to measure the query evaluation performance of different tools for a

[4] https://github.com/viatra/viatra-cps-benchmark

Benchmark	Reference	Graphs	Largest graph	#classes	#predicates	#queries	Multiple users	Workload profile	Focus metric	Updates
LUBM	[GPH05]	synthetic	6.9M	43	32	14	○	analyzing university data	inferencing performance	○
Barton	[Aba+07]	real	50M	11	28	7	○	library search	response time	○
SP^{2}Bench	[Sch+09]	synthetic	1B+	8	22	12	○	publication research	response time	○
BSBM	[BS09]	synthetic	150B	8	51	12	⊗	e-commerce	throughput	⊘
DBpedia	[Mor+11]	real	300M	8	1200	25	○	queries on DBpedia	throughput	○
LinkBench	[Arm+13]	synthetic	1B+	25+	100+	10+	○	social network	latency/thr.	⊗
WatDiv	[Alu+14]	synthetic	10B+	17	85	⊛	○	e-commerce	response time	○
SNB Interactive	[Erl+15]	synthetic	1B+	19	27	14+	○	local queries on a social network	response time	⊗
Semantic Pub.	[Kot+16]	synthetic	1B+	74	117	12	○	creative works	response time	⊗
Stream WatDiv	[Gao+18]	synthetic	10B+	17	85	⊛	○	user activity stream in e-commerce	response time	⊗
Train Benchmark	[j2]	synthetic	23M+	9	13	6	○	validation of engineering models	query/update response time	⊗
SNB BI	[l18]	synthetic	1B+	19	27	25	○	business intelligence on a social network	response time	○

Table 7.2: Benchmarks for semantic and graph databases. Notations – *Largest graph* column: "M" stands for million, "B" stands for billion; *#queries* column: ⊛ the benchmark allows users to generate an arbitrary number of queries; *Focus metric* column: *thr.* = throughput; *Updates* column: ⊗ measuring the performance of updates is an important aspect of the benchmark, ⊘ the benchmark uses updates, but the performance of reevaluation after updates is not an important aspect, ○ the benchmark does not consider updates.

single set of SPARQL queries that contain most language elements. The artificially generated data is based on the real-world DBLP bibliography; this way instance models of different sizes reflect the structure and complexity of the original real-world dataset. However, other model element distributions or queries were not considered, and the complexity of queries were not analysed.

The *Berlin SPARQL Benchmark (BSBM)* [BS09] measures SPARQL query evaluation throughput for an e-commerce case study modelled in RDF. The benchmark uses a single dataset, but recognizes several use cases with their own set of queries. The dataset scales in model size (10 million–150 billion), but does not vary in structure. BSBM defines multiple workloads, named *use cases*. The *Explore* use case follows the navigation of a consumer looking for a product, *Business Intelligence* runs complex analytical queries, while *Update* performs data addition and deletion operations.

In the *SPLODGE* [GTS12] benchmark, SPARQL queries are generated systematically, based on metrics for a predefined dataset. The method supports distributed SPARQL queries (via the `SERVICE` keyword), however the implementation scales only up to three steps of navigation, due to the resource consumption of the generator. The paper does not discuss the complexity of the instance model, and only demonstrates the adequacy of the approach demonstrated with the RDF3X engine.

The *DBpedia SPARQL benchmark* [Mor+11] presents a general SPARQL benchmark procedure, applied to the DBpedia knowledge base. The benchmark is based on query-log mining, clustering and SPARQL feature analysis. In contrast to other benchmarks, it performs measurements on real queries against existing RDF data.

The Linked Data Benchmark Council has recently developed the *Social Network Benchmark* defines two workloads: the *Interactive* and *Business Intelligence*. The *Interactive* workload [Erl+15] focuses on a transactional graph query system. The gist of the workload is defined by 14 complex queries, focusing on mostly local traversal operations which start from a specific node. The workload also contains 7 short queries, which define simple read operations, and 8 updates that add additional nodes and edges to the social network graph. The benchmark was reproduced in paper [Pac+17], extended with an Apache Kafka-based[5] mechanism to stream updates. It was also extended with additional implementations (PostgreSQL,[6] Neo4j,[7] Virtuoso SPARQL,[8] Titan,[9] and Sqlg[10]). The *Business Intelligence* workload is presented in this dissertation in Chapter 6 and was published in [l18].

LinkBench [Arm+13] was developed in collaboration with Facebook and is based on its social graph. It tests 10 operations, including reads and updates, such as object_get, assoc_multiget, and assoc_update. It measures both the *latency* and the *throughput* of the system under benchmark.

Benchmarks for other graph-like data models The OO7 (pronounced "double-o-seven") benchmark [CDN93; Car+94] targeted object-oriented database management systems in the early 1990s. Other notable OODBMS benchmarks include BUCKY (Benchmark of Universal or Complex Kwery Ynterfaces) [Car+97] and OCB (Object Clustering Benchmark) [DS00a]. XMark [Sch+02] is an influential a benchmark for XML databases. UniBench [Zha+18a] is a multi-model benchmark that builds on a modified version of the LDBC data generator to produce a mix of relational, graph, key-value, JSON, and XML data sets. The recent *Join Order Benchmark* targets join-heavy queries – which share many challenges with graph query workloads (Sec. 2.7.1) – in relational database systems [Lei+18].

[5] https://kafka.apache.org/
[6] https://www.postgresql.org/
[7] https://neo4j.com/
[8] https://virtuoso.openlinksw.com/
[9] http://titan.thinkaurelius.com/
[10] https://github.com/pietermartin/sqlg

Part III

Incremental View Maintenance on Schema-Optional Property Graphs

Chapter 8

Reducing Property Graph Queries to Relational Algebra for Incremental View Maintenance

Note on terminology In this chapter, we use the term *vertices* to refer to graph *nodes*, in order to differentiate graph vertices from nodes representing *relational operators* and *Rete nodes*.

8.1 Introduction

Graph processing problems are common in modern database systems, where the *property graph* (PG) data model [HG16; MSV17; Ang+18; Fra+18; Ang18] is gaining widespread adoption. Property graphs extend labelled graphs with properties for both vertices and edges. Compared to previous graph modelling approaches, such as the RDF data model (which represents properties as triples), PGs allow users to store their graphs in a more compact and comprehensible representation. Due to the novelty of the PG data model, no standard query language has emerged yet. The industry-driven *openCypher initiative* aims to standardize the Cypher language [Fra+18] of the Neo4j graph database. The openCypher language uses a SQL-like syntax and combines graph pattern matching with relational operators (aggregations, filtering, projection, etc.). In this chapter, we target queries specified in the openCypher language. This chapter is based on technical report [r21].

Motivation Numerous use cases of graph databases rely on complex queries and require low response time for repeated executions, including financial fraud detection, and recommendation engines. In addition, graph databases are increasingly used in software engineering context as a semantic knowledge base for model validation [Ber+10; Dan+17] [j2], source code analysis [HBC15], etc. Users of these scenarios could greatly benefit from an incremental query engine that allows them to *register views* on property graphs and *maintain their state* continuously upon changes.

The recent survey [Sah+17] also concluded that there is industrial need for incremental (and streaming) graph query engines. This investigation also revealed that currently no such systems exist, with the exception of DBToaster (a higher-order incremental query engine [Koc+14]) and Graphflow (an active graph database prototype [Kan+17]) which shares many authors with the survey. Our

own observation, which is based upon using these tools in an open property graph and RDF benchmark [l18], is that incremental evaluation is only supported for very basic language features (practically, only 5-6 queries out of a total 25 queries). This clearly indicates that *incremental query evaluation and view maintenance is still a major research challenge in the context of property graphs.*

Problem statement In relational database systems, *incremental view maintenance* (IVM) techniques have been used for decades for repeated evaluation of a predefined query set on continuously changing data [For82; BLT86; ML91; GMS93; GM95; Han96; GM99; HBC02; Koc+14; Ujh+15a]. However, these techniques typically build on assumptions that do not hold for PG queries. Incremental PG queries present numerous challenges:

1. *Schema-optional data model.* Existing IVM techniques assume that the database schema is a priori known. While this is a realistic assumption for relational databases, the data model of most property graph systems is schema-free or schema-optional at best [BTL11; Fra+18]. Hence, to use IVM, users are required to manually define the schema of the graph, which is a tedious and error-prone process.
2. *Nested data structures.* Most IVM techniques assume relational data model with 1NF relations. However, the property graph data model defines rich structures, including the properties on graph elements and paths. Collection types, such as sets, bags, lists, and maps are also allowed [Ang+18; Fra+18]. These can be represented as NF^2 (non-first normal form) data structures, but their mapping to 1NF relations is a complex challenge.
3. *Mix of instance- and meta-level data.* Queries access not only data fields from the instance graph (e.g. properties), but also metadata such as vertex labels/edge types [Res+16; Fra+18].
4. *Handling antijoins and outer joins.* Most PG languages allow negative and optional pattern conditions, similarly to antijoins and outer joins in relational databases. Most IVM works do not consider these operators, except [GK98], [GM06], and [LZ07a].
5. *Bag semantics and aggregations.* PG queries often include aggregation operations, which necessitate bag (multiset) semantics. Furthermore, languages usually do not restrict aggregations, e.g. they allow aggregations on aggregations [MQM97] and using non-distributive aggregation functions (e.g. min, max, standard deviation) which are difficult to calculate incrementally [Pal+02].
6. *Mix of queries and transformations.* Some PG query languages (e.g. openCypher) allow combining update operations with queries. Most traditional IVM techniques do not consider this challenge, and omit related issues such as conflict set resolution. *Discrimination networks* from rule-based expert systems are better suited to handle this issue [For82; ML91; HBC02].
7. *List handling.* Property graph data sets and queries make use of lists both as a way to store collection of primitive values and to represent paths in the graphs. Order-preserving techniques have only been studied in the context of IVM on XQuery expressions [DER03a], for trees but not for graphs.
8. *Reachability queries.* Unbounded reachability queries on graphs with few connected components need to calculate large transitive closures, which makes them inherently expensive [Ber+12]. Hence, the impact of the IVM on reachability is more limited compared to non-recursive queries and using space-time tradeoff techniques is more expensive.
9. *Skewed data distribution and cyclic queries.* Subgraph matching is often implemented as a series of binary joins. Recent work [Ngo+18] revealed that representing *cyclic queries* with binary (two-way) joins are inefficient on data sets with skewed distributions of certain edge types (displayed by graph instances in many fields, e.g. in social networks). Hence, new research

works propose n-ary (multiway) joins to achieve theoretically optimal complexity. Self-joins – join expressions that use the same relation more than once – are also common.

10. *Higher-order queries.* PG queries often employ *higher-order* expressions [Bun+95], e.g. processing the vertices/edges on a path (also known as *path unwinding* [Ang+17]). Incrementalization of higher-order expressions is not yet studied in depth [Cai+14], and currently there are no implementations using such techniques for query evaluation.

These problems are not only challenging in isolation, but require *IVM techniques that are composable.* According to our experience, tackling one problem often interferes with another. For example: (a) The IVM algorithm proposed in [LZ07a] for outer join operators does not allow any self-join operators. (b) Most results are presented on set semantics, but are not generalized for bag (e.g. [GK98]). While some are easy to adapt, determining whether certain IVM techniques can be adapted from sets to bag usually necessitates a detailed investigation. (c) Some approaches such as [GL95] support one level of aggregation but no group-by.

In this work, we address challenges 1–5 while leaving 6–10 for future work for the database community. Note that existing research tools with IVM support typically cover only 1 and 2 while industrial PG query tools do not support incremental evaluation. Thus, our partial coverage is a major advance in the state-of-the-art of IVM techniques for property graphs.

Contributions As the core contribution, we show how to reduce the challenge of incremental graph evaluation of openCypher queries over property graphs to traditional relational algebra to enable the use of existing IVM techniques. Instead of proposing a novel incremental algorithm directly for openCypher, our results aim to demonstrate how to reuse known IVM techniques in the context of a new challenge with high practical relevance. In particular,

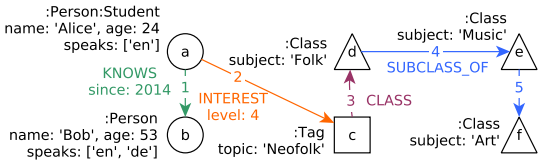
- We introduce extensions for relational algebra (RA) in order to handle graph-specific operators. We use the resulting *graph RA* (GRA) to capture the semantics of a large subset of the openCypher language.
- We define a mapping for PG data to nested relations, and transform the queries to *nested RA* (NRA). The data model can represent both the property graph and the resulting tables, while the NRA operators have sufficient expressive power to capture operations on the PG. This allows the algebra to be *composable and closed* [l17].
- We define a transformation chain to translate the nested algebraic query plans to *flat RA* (FRA) expressions.
- We present a *schema inferencing algorithm* that eliminates the need to define the graph schema in advance.
- We propose an architecture for IVM for PG queries based on the Rete algorithm [For82]. As a proof of concept prototype, we present the *ingraph* tool which is capable of evaluating openCypher graph queries incrementally.[1]
- We categorize and overview applicable IVM approaches from the literature in rule-based expert systems, integrity constraint checking, and materialized views.

vertex			
id	**labels**	**properties**	
	label	**key**	**value**
a	Student Person	name	Alice
		age	24
		speaks	[en]
b	Person	name	Bob
		age	53
		speaks	[en, de]
c	Tag	topic	Neofolk
d	Class	subject	Folk
e	Class	subject	Music
f	Class	subject	Art

(a) Nested vertex relation: V.

edge					
id	**src**	**trg**	**type**	**properties**	
				key	**value**
1	a	b	KNOWS	since	2014
2	a	c	INTEREST	level	4
3	c	d	CLASS	–	
4	d	e	SUBCLASS_OF	–	
5	e	f	SUBCLASS_OF	–	

(b) Nested edge relation: E.



(c) Example graph visualized.

$$L = \{\text{Person}, \text{Student}, \text{Class}, \text{Tag}\}$$
$$T = \{\text{KNOWS}, \text{INTEREST}, \text{SUBCLASS_OF}, \text{CLASS}\}$$
$$P_v = \{\text{name}, \text{speaks}, \text{topic}, \text{subject}, \text{age}\}$$
$$P_e = \{\text{since}, \text{level}\}$$

$$V = \{a, b, c, d, e, f\}; E = \{1, 2, 3, 4, 5\}$$
$$\mathit{src}: 1 \to a, 2 \to a, \ldots; \mathit{trg}: 1 \to b, 2 \to a, \ldots$$
$$\mathit{labels}: a \to \{\text{Person}, \text{Student}\}, b \to \{\text{Person}\}, \ldots$$
$$\mathit{type}: 1 \to \text{KNOWS}, 2 \to \text{INTEREST}, \ldots$$
$$\text{name}: a \to \text{“Alice”}, b \to \text{“Bob”}, \ldots; \quad \text{age}: a \to 24, \ldots$$
$$\text{speaks}: a \to \wr \text{“en”} \int, b \to \wr \text{“en”}, \text{“de”} \int, c \to \text{NULL}, \ldots$$
$$\text{topic}: a \to \text{NULL}, b \to \text{NULL}, c \to \text{“Neofolk”}, \ldots$$
$$\text{since}: 1 \to 2014, 2 \to \text{NULL}, \ldots \quad \text{level}: 2 \to 4, \ldots$$

(d) Example graph defined formally. $\wr \ldots \int$ denotes a bag of elements.

s			
id	**labels**	**properties**	
	label	**key**	**value**
a	Student Person	name	Alice
		age	24
		speaks	[en]

(e) Result of the $\left(\bigcirc_s^{\text{Student}}\right)$ *get-vertices* operator.

s				i						t			
id	**labels**	**properties**		**id**	**src**	**trg**	**type**	**properties**		**id**	**labels**	**properties**	
	label	**key**	**value**					**key**	**value**		**label**	**key**	**value**
a	Student Person	name	Alice	2	a	c	INTEREST	level	4	c	Tag	topic	Neofolk
		age	24										
		speaks	[en]										

(f) The result relation of the $\left[\text{Student}_s \bigcirc \xrightarrow[i]{\text{INTEREST}} \bigcirc_t \text{Tag}\right]$ *get-edges* operator.

Figure 8.1: Social graph represented graphically, formally, and as nested relations.

8.2 Data Models

8.2.1 The Property Graph Data Model

The concept of the *property graph* has been studied by only a few academic works, but it already has multiple flavours and definitions [HG16; MSV17; Ang+18; Fra+18; Ang18]. We previously introduced property graphs in Sec. 2.2. Here, we revisit this definition:

Definition 57 (property graph) A *property graph* is defined as

$$G = (V, E, \mathit{src}, \mathit{trg}, L, T, \mathit{labels}, \mathit{type}, P_v, P_e)$$

where V is a set of vertex identifiers, and E is a set of edge identifiers. Functions *src*, *trg* : $E \to V$ are the *source* and *target functions*, which are *total functions* assigning the source and target vertex to each edge, respectively.

To capture type information, the data model uses labels and types:

- The vertices of the graph are *labelled*: L is a set of vertex labels, and function *labels* : $V \to 2^L$ assigns a *set of labels* to each vertex.
- The edges of the graph are *typed*: T is a set of edge types, and function *type* : $E \to T$ assigns a *single type* to each edge.

To capture the *properties* in the graph, let S be a set of scalar literals, $\mathrm{FBAG}(S)$ denote the set of all finite bags of elements from S, and let $D = S \cup \mathrm{FBAG}(S)$ be the value domain for the PG. The properties of vertices and edges are defined as:

- P_v is the set of vertex properties. $p \in P_v$ is a function $p : V \to D$, which assigns a property value $d \in D$ to a vertex $v \in V$, if v has property p, otherwise returns NULL.
- P_e is the set of edge properties. $p \in P_e$ is a function $p : E \to D$, which assigns a property value $d \in D$ to an edge $e \in E$, if e has property p, otherwise returns NULL.

Example graph An example graph inspired by the LDBC Social Network Benchmark [l18] is shown formally in Fig. 8.1d and graphically in Fig. 8.1c. The graph contains a Tag, two Persons, and three Classes. Note that edges in the PG data model are always *directed*, hence the KNOWS relation is represented with a directed edge and its symmetric nature is captured by the queries.

8.2.2 Nested Relations

An openCypher query takes a property graph as its inputs and returns a *graph relation* [HG16; MSV17] as its output. To represent graphs and query results using the same algebraic constructs, we use *nested relations* [Col90], which allow data items of a relation to contain additional relations with an arbitrary level of nesting. The domain for the internal relations is $D \cup \{\mathsf{NULL}\}$. Relations on all levels of nesting follow bag semantics, i.e. duplicate tuples are allowed. We define the *schema* of a relation as a *list of (nested) attributes* and denote it with $\mathrm{sch}(r)$ for relation r. For the schemas of relations r and s, we use $\mathrm{sch}(r) - \mathrm{sch}(s)$ to denote *subtracting* the attributes of $\mathrm{sch}(s)$ from $\mathrm{sch}(r)$. Additionally, we use $\mathrm{sch}(r) \parallel \mathrm{sch}(s)$ to denote *unique concatenation*, which concatenates $\mathrm{sch}(r)$ with the attributes of $\mathrm{sch}(s) - \mathrm{sch}(r)$.

[1] ingraph is available as an open-source tool at http://github.com/ftsrg/ingraph.

To represent the vertices (V) and edges (E) of the property graph, we define two nested relations, V and E. Both relations have a single nested attribute with the following schemas:

$$\mathrm{sch}\,(\mathsf{V}) = \langle \mathsf{vertex}(\mathsf{id}, \mathsf{labels}(\mathsf{label}), \mathsf{properties}(\mathsf{key}, \mathsf{value})) \rangle$$

$$\mathrm{sch}\,(\mathsf{E}) = \langle \mathsf{edge}(\mathsf{id}, \mathsf{src}, \mathsf{trg}, \mathsf{type}, \mathsf{properties}(\mathsf{key}, \mathsf{value})) \rangle$$

For V.vertex, its "id" attribute corresponds to the elements in V. For a tuple representing a vertex, "labels" is the result of the *labels* function. Similarly, for E.edge, "id" corresponds to the elements in E. For a tuple representing an edge e, "type" corresponds *type*(e), "src" to *src*(e), and "trg" to *trg*(e). The "properties" attribute contains nested relations of "key"-"value" pairs representing the properties of the vertex/edge in the tuple.

The nested relations representing the example graph are shown in Figure 8.1b and 8.1a. These show that the set of vertex labels are stored as a nested relation "labels" with a single attribute "label", while edge types are simply stored as a string value. The properties of vertices/edges are stored as a nested relation properties with two attributes, "key" and "value". This representation is well-suited to the flexible schema of PG databases, as new labels, types, and property keys can be added without any changes to the schemas of the relations.

8.3 Graph Relational Algebra

Papers [HG16] and [c6] presented relational algebraic formalizations of the openCypher language. A more rigorous formalization was given in [Fra+18]. In this work, we follow the approach of our previous work [c6] as it is better suited to established IVM techniques. This approach uses *graph relational algebra* (GRA), which extends standard relational algebra operators with graph-specific navigation.

In the following, we define the *graph-specific operators* of GRA and show example queries specified in natural language and as an openCypher query, along with the equivalent GRA expression and the resulting output relation.

Shorthands For the sake of brevity, we allow two shorthands to access nested attributes and properties of graph elements.

S1. We use dot notation to directly access *nested attributes* of graph elements (such as id and type). For example, the expression $\sigma_{\mathsf{V.vertex.id}=42}\,(\mathsf{V})$ checks the nested attribute "id" of V.vertex.

S2. The *properties* of graph elements, stored as key-value pairs in the nested "properties" relation, can be accessed directly as if they were top-level attributes. For example, expression $\pi_{\mathsf{V.vertex.age}}\,(\mathsf{V})$ projects the "age" property of vertices. Unlike *S1*, this shorthand does not require each vertex to have an "age" property, it simply returns NULL in the absence of such a key.

Notations For mapping a property graph to relations, we use the nullary operators *get-vertices* and *get-edges*. We define these operators using the nested relations V and E introduced in Sec. 8.2.2. These operators are rather involved, hence we define some notational conventions:

N1. A vertex variable v is *free* w.r.t. the input relations r_i of a given operator iff $v \notin \mathrm{sch}\,(r_i)$ for all r_i and *bound* iff $v \in \mathrm{sch}\,(r_i)$ for any r_i. It follows from the definition that the variables of a nullary operator are always free. Notation: $\circ$ free vertex variable, $\odot$ bound vertex variable, $\circledcirc$ any (free or bound) vertex variable.

N2. Arrow symbols represent the direction of an edge variable. Notation: → outgoing, ← incoming, ↔ undirected edge.

N3. In the definitions, we use three example sets of labels for vertices (L, L1, and L2), along with an example set of types (T).

8.3.1 The Get-Vertices Operator

The *get-vertices* nullary operator $\left(\bigcirc_v^{\mathsf{L}}\right)$ returns a nested relation of a single attribute v with vertices which have *all* labels of L:

$$\left(\bigcirc_v^{\mathsf{L}}\right) \equiv \pi_{\mathsf{V.vertex}/v}\left(\sigma_{\mathsf{L}\subseteq\mathsf{V.vertex.labels}}\left(\mathsf{V}\right)\right)$$

The schema of the resulting relation is $\mathrm{sch}\left(\bigcirc_v^{\mathsf{L}}\right) \equiv \langle v \rangle$ (see Fig. 8.1e). The operator is illustrated with the following example:

Example 23 (Get the name of all Persons aged over 25)

```
MATCH (p:Person)
WHERE p.age > 25 RETURN p.name
```

p.name
Bob

$$\pi_{p.\mathsf{name}}\sigma_{p.\mathsf{age}>25}\left(\bigcirc_p^{\mathsf{Person}}\right)$$

Here, the get-vertices operator captures the `MATCH` clause, then the selection and projection operators capture the `WHERE` and `RETURN` clauses, respectively. Note that we used shorthand *S1* in the definition and *S2* in the example to access nested attributes.

8.3.2 The Get-Edges Operator

Next, we introduce the *get-edges* operator $[\bigcirc\!\!\rightarrow\!\!\bigcirc]$, which returns edges along with their source and target vertices. Using theta-joins (Def. 19) on *get-vertices* operators and relation E, the *directed get-edges* operator $\left[{}^{\mathsf{L1}}_{v}\bigcirc\!\xrightarrow[e]{\mathsf{T}}\!\bigcirc^{\mathsf{L2}}_{w}\right]$ has the schema $\langle v, e, w \rangle$ and is defined as:

$$\pi_{v,e,w}\left(\left(\bigcirc_v^{\mathsf{L1}}\right) \underset{v.\mathsf{id}=\mathsf{edge.src}}{\bowtie} \left(\sigma_{\mathsf{edge.type}\in\mathsf{T}}\left(\mathsf{E}\right)\right) \underset{\mathsf{edge.trg}=w.\mathsf{id}}{\bowtie} \left(\bigcirc_w^{\mathsf{L2}}\right)\right)$$

We also define the *undirected get-edges* operator $[\bigcirc\!\!\leftrightarrow\!\!\bigcirc]$, enumerating edges of both directions:

$$\left[{}^{\mathsf{L1}}_{v}\bigcirc\!\xleftrightarrow[e]{\mathsf{T}}\!\bigcirc^{\mathsf{L2}}_{w}\right] \equiv \left[{}^{\mathsf{L1}}_{v}\bigcirc\!\xrightarrow[e]{\mathsf{T}}\!\bigcirc^{\mathsf{L2}}_{w}\right] \bigcup \pi_{v,e,w}\left[{}^{\mathsf{L2}}_{w}\bigcirc\!\xrightarrow[e]{\mathsf{T}}\!\bigcirc^{\mathsf{L1}}_{v}\right]$$

8.3.3 The Expand Operators

To capture navigations, we define the unary *expand-out* operator $\odot\!\!\rightarrow\!\!\circledcirc$. The expression ${}_{v}\odot\!\xrightarrow[e]{\mathsf{T}}\!\circledcirc^{\mathsf{L1}}_{w}(r)$ takes tuples from relation r and returns a tuple for each possible navigation from a bound vertex

$v \in \mathrm{sch}(r)$ to vertex w through an edge e, while enforcing the label and type constraints (w is labelled with labels of L1 and e is typed with T). It can be defined using the *get-edges* operator:

$$ {}_{v}\odot\xrightarrow[e]{\mathsf{T}}\circledcirc^{\mathsf{L1}}_{w}(r) = r \bowtie \left[{}_{v}\bigcirc\xrightarrow[e]{\mathsf{T}}\bigcirc^{\mathsf{L1}}_{w} \right] $$

The schema of the resulting relation is $\mathrm{sch}\left({}_{v}\odot\xrightarrow[e]{\mathsf{T}}\circledcirc^{\mathsf{L1}}_{w}(r) \right) \equiv \mathrm{sch}(r) \parallel \langle e, w \rangle$. The operator is demonstrated as follows:

Example 24 (Get Persons and their interests)

```
MATCH
  (p:Person)-[i:INTEREST]->(t:Tag)
RETURN p.name, i.level, t.topic
```

p.name	i.level	t.topic
Alice	4	Neofolk

$$ \pi_{p.\mathsf{name},i.\mathsf{level},t.\mathsf{topic}} \left({}_{p}\odot\xrightarrow[i]{\mathsf{INTEREST}}\bigcirc^{\mathsf{Tag}}_{t} \left(\bigcirc^{\mathsf{Person}}_{p} \right) \right) $$

Edge directions We define two additional *expand* operators: the *expand-in* operator $\odot\!\leftarrow\!\circledcirc$ accepts incoming edges, while the *expand-both* operator $\odot\!\leftrightarrow\!\circledcirc$ accepts edges from both directions. Formally, they can be defined as follows:

$$ {}_{v}\odot\xleftarrow[e]{\mathsf{T}}\circledcirc^{\mathsf{L1}}_{w}(r) \equiv r \bowtie \left[{}^{\mathsf{L1}}_{w}\bigcirc\xrightarrow[e]{\mathsf{T}}\bigcirc_{v} \right] $$

$$ {}_{v}\odot\xleftrightarrow[e]{\mathsf{T}}\circledcirc^{\mathsf{L1}}_{w}(r) \equiv r \bowtie \left[{}_{v}\bigcirc\xleftrightarrow[e]{\mathsf{T}}\bigcirc^{\mathsf{L1}}_{w} \right] $$

Merging with get-vertices An *expand* operator following a *get-vertices* operator can be combined into a *get-edges* operator:

$$ {}_{v}\odot\xrightarrow[e]{\mathsf{T}}\bigcirc^{\mathsf{L2}}_{w}\left(\bigcirc^{\mathsf{L1}}_{v}\right) \equiv \left[{}^{\mathsf{L1}}_{v}\bigcirc\xrightarrow[e]{\mathsf{T}}\bigcirc^{\mathsf{L2}}_{w} \right] $$

$$ {}_{v}\odot\xleftrightarrow[e]{\mathsf{T}}\bigcirc^{\mathsf{L2}}_{w}\left(\bigcirc^{\mathsf{L1}}_{v}\right) \equiv \left[{}^{\mathsf{L1}}_{v}\bigcirc\xleftrightarrow[e]{\mathsf{T}}\bigcirc^{\mathsf{L2}}_{w} \right] $$

$$ {}_{v}\odot\xleftarrow[e]{\mathsf{T}}\bigcirc^{\mathsf{L2}}_{w}\left(\bigcirc^{\mathsf{L1}}_{v}\right) \equiv \pi_{v,e,w} \left[{}^{\mathsf{L2}}_{w}\bigcirc\xrightarrow[e]{\mathsf{T}}\bigcirc^{\mathsf{L1}}_{v} \right] $$

This allows us to rewrite the example in a more succinct form.

Example 25 (Get Persons and their interests – revisited)

$$ \pi_{p.\mathsf{name},i.\mathsf{level},t.\mathsf{topic}} \left[{}^{\mathsf{Person}}_{p}\bigcirc\xrightarrow[i]{\mathsf{INTEREST}}\bigcirc^{\mathsf{Tag}}_{t} \right] $$

8.3.4 Combining Pattern Matches

A single graph pattern is compiled to a GRA expression starting from *get-vertices* and *expand* operators. The *semijoin* operator $\ltimes$ can be used to express required pattern parts (that are not part of the result), and *antijoin* $\overline{\ltimes}$ for negative pattern parts.

Example 26 (Get every Class that has a subclass, but is not a subclass of any Class)

```
MATCH (c:Class)
WHERE (subC:Class)-[:SUBCLASS_OF]->(c)
  AND NOT (c)-[:SUBCLASS_OF]->(supC:Class)
RETURN c.name
```

c.name
Art

$$\pi_{c.\mathsf{name}}\left(\left(\bigcirc_{c}^{\mathsf{Class}}\right) \bowtie \left[{}^{\mathsf{Class}}_{subC}\bigcirc \xrightarrow{\mathsf{SUB...}} \bigcirc_{c}\right] \overline{\ltimes} \left[{}_{c}\bigcirc \xrightarrow{\mathsf{SUB...}} \bigcirc^{\mathsf{Class}}_{supC}\right]\right)$$

Multiple graph patterns can be combined together based on their common attributes using the *join* operator $\bowtie$, and optional pattern parts can be added with *left outer join* $⟕$.

Example 27 (Get Persons and their interests if they have any)

```
MATCH (p:Person)
OPTIONAL MATCH
  (p)-[i:INTEREST]->(t:Tag)
RETURN p.name, t
```

p.name	t			
	id	**labels**	**properties**	
		label	**key**	**value**
Alice	c	Tag	topic	Neofolk
Bob	NULL			

$$\pi_{p.\mathsf{name},t}\left(\left(\bigcirc_{p}^{\mathsf{Person}}\right) ⟕ \left[{}_{p}\bigcirc \xrightarrow[i]{\mathsf{INTEREST}} \bigcirc^{\mathsf{Tag}}_{t}\right]\right)$$

8.3.5 Unwinding and Aggregation

Unwinding It is often required to handle elements in nested collections one-by-one. To allow this, we introduce the *unwind* operator ω, a specialized version of the unnest operator μ of nested relational algebra [Bot+18]. In particular, $\omega_{xs \Rightarrow x}(r)$ takes the bag in attribute xs and creates a new tuple for each element of the bag by appending that element as an attribute x to $r_i \in r$.

Grouping and aggregation The *grouping* operator γ groups tuples according to their value in one or more attributes and aggregates the remaining attributes. We use the notation $\gamma^{c_1,\ldots,c_n}_{e_1/a_1,\ldots,e_k/a_k}$, where $c_1, \ldots, c_n$ form the *grouping criteria*, i.e. the list of expressions whose values partition the incoming tuples into groups. For every group this aggregation operator emits a single tuple of expressions $\langle e_1, \ldots, e_k \rangle$ with aliases $\langle a_1, \ldots, a_k \rangle$.

Example 28 (Get the number of speakers for each language)

```
MATCH (p:Person) WITH p
UNWIND p.speaks AS lang
RETURN lang, count(p) AS speakers
```

lang	*speakers*
en	2
de	1

$$\gamma^{lang}_{lang,\mathtt{count}(p)/speakers}\left(\omega_{p.\mathsf{speaks} \Rightarrow lang}\left(\bigcirc_{p}^{\mathsf{Person}}\right)\right)$$

openCypher language construct	GRA expression
`(«v»)`	$\bigcirc_v$
`(«v»:«l1»:⋯:«lk»)`	$\bigcirc_v^{\mathsf{l}_1,\ldots,\mathsf{l}_k}$
`(p)-[«e»:«t1»|⋯|«to»]->(«w»)`	$_v\odot\xrightarrow[e]{\mathsf{t}_1,\ldots,\mathsf{t}_o}\bigcirc_w(p)$
`(p)<-[«e»:«t1»|⋯|«to»]-(«w»)`	$_v\odot\xleftarrow[e]{\mathsf{t}_1,\ldots,\mathsf{t}_o}\bigcirc_w(p)$
`(p)<-[«e»:«t1»|⋯|«to»]->(«w»)`	$_v\odot\xleftrightarrow[e]{\mathsf{t}_1,\ldots,\mathsf{t}_o}\bigcirc_w(p)$
`MATCH (p1), (p2), ⋯`	$p1 \bowtie p_2 \bowtie \ldots$
`OPTIONAL MATCH (p)`	$\{\langle\rangle\} ⟕ p$
`{r} OPTIONAL MATCH (p) WHERE «θ»`	$r ⟕ (\sigma_\theta(r \bowtie p))$
`{r} OPTIONAL MATCH (p)`	$r ⟕ p$
`{r} WHERE «θ»`	$\sigma_\theta(r)$
`{r} WHERE («v»:«l1»:⋯:«lk»)`	$\sigma_{\{\mathsf{l}_1,\ldots,\mathsf{l}_k\}\subseteq labels(v)}(r)$
`{r} WHERE (p)`	$r \ltimes p$
`{r} WHERE NOT (p)`	$r \overline{\ltimes} p$
`{r} RETURN «x1» AS «y1», ⋯`	$\pi_{x_1/y_1,\ldots}(r)$
`{r} RETURN DISTINCT «x1» AS «y1», ⋯`	$\delta\left(\pi_{x_1/y_1,\ldots}(r)\right)$
`{r} RETURN «x1», «x2», aggr(«x3»)`	$\gamma^{x_1,x_2}_{x_1,x_2,\mathsf{aggr}(x_3)}(r)$
`{r} UNWIND «xs» AS «x»`	$\omega_{xs\Rightarrow x}(r)$

Table 8.1: openCypher patterns and clauses in GRA. Variables, labels, and types are typeset as «v». The notation (p) represents a pattern resulting in a relation p. In the example, we presume that relation p has an attribute v that represents a vertex. {r} stands for a relation r that is the result of the previous query parts. To avoid confusion with the "..." language construct (used for ranges), we use "⋯" to denote omitted query parts.

In this section, we defined the operators of GRA and gave an informal specification for compiling from openCypher queries. Tab. 8.1 shows a compact mapping of openCypher queries to GRA expressions. Note that the *get-edges* operator is not essential to capture the mapping – instead, only the *get-vertices* nullary operators are used and edges are retrieved by the *expand* operators.

8.4 Transforming Graph RA to Flat RA

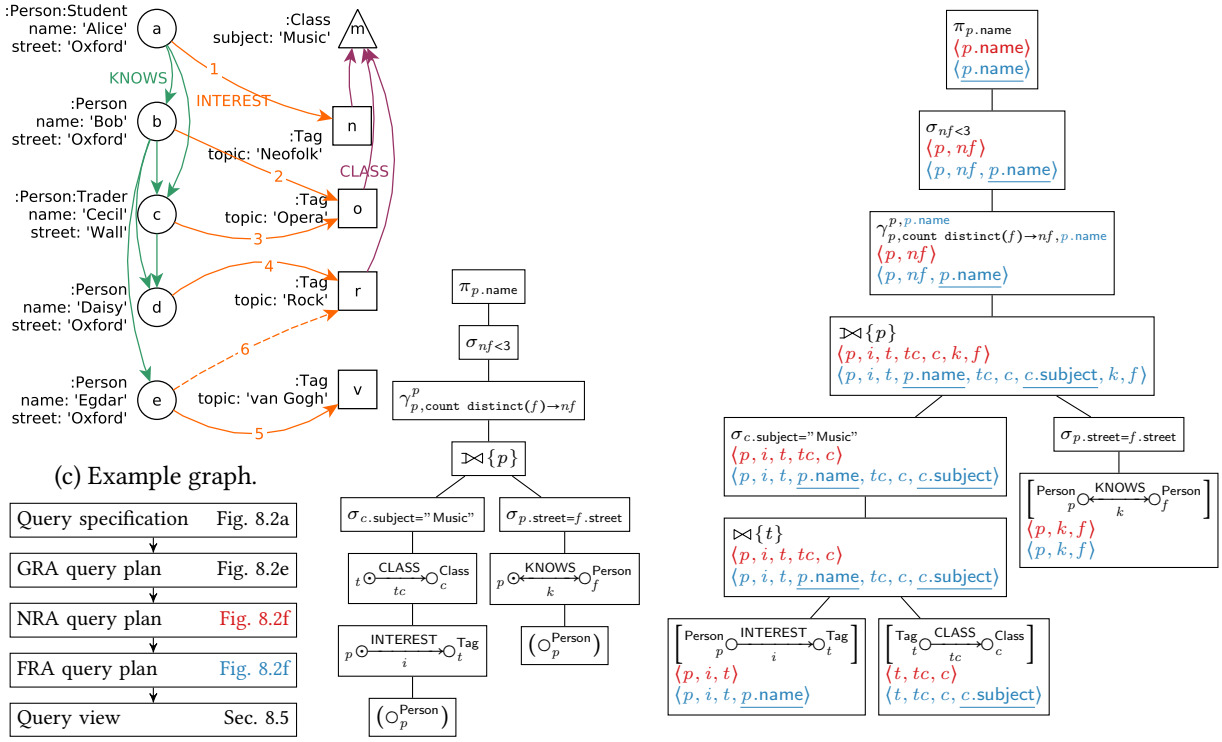
In Sec. 8.3, we presented the concepts of GRA and outlined how to compile openCypher queries to GRA, based on our previous work [c6], which provides a precise foundation for non-incremental openCypher queries by covering all queries in the public LDBC Social Network Benchmark (see Sec. 8.6) . However, despite the significant body of existing IVM literature for relational data models, existing incremental techniques for property graphs and openCypher queries only support very basic operations in GRA. In fact, only 6 out of the 25 queries in the LDBC benchmark are supported by

```
1 MATCH (p:Person)-[i:INTEREST]->(t:Tag)-[tc:CLASS]->(c:Class)
2 WHERE c.subject = 'Music'
3 OPTIONAL MATCH (p)-[k:KNOWS]-(f:Person)
4 WHERE p.street = f.street
5 WITH p, count(DISTINCT f) AS nf WHERE nf < 3
6 RETURN p.name
```

(a) Query specification in openCypher.

p.name
Alice
Cecil
Daisy
Edgar

(b) Output.



(c) Example graph. (d) Proposed workflow. (e) Query plan in graph RA. (f) Query plan in nested RA ■ and flat RA ■.

Figure 8.2: Example property graph, textual query specification, query plans, and output of the query on the given graph. The query finds persons p who are interested in *Music* and know less than 3 persons living in the same street.

DBToaster (see Tab. 8.2 for coverage). Such a low feature coverage in existing tools clearly demonstrates the complexity of supporting IVM for GRA which uses (1) graph-specific operators such as *expand* and (2) nested data structures.

A main contribution of our work is to significantly increase the coverage of GRA operators for incremental evaluation. But instead of proposing a novel algorithm for incremental evaluation, we show how to reduce the problem to traditional relational algebra where robust IVM techniques already exist. As such, the essence of our contribution is how to *address the major challenge of incremental evaluation of GRA by using known IVM techniques as the baseline* (without reinventing the wheel). For that purpose, we propose a transformation chain [Yie+12] which is common in compilers and software engineering applications to reduce the complexity and abstraction gap of the end-to-end mapping. We introduce two additional algebras: *nested relational algebra* (NRA), which uses joins instead of expand operators, and *flat relational algebra* (FRA), which uses flat relations instead of nested ones. We define a chain of steps which transform queries from GRA to NRA and from NRA to FRA (see Fig. 8.2d).

Tool	IVM	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Neo4j	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
DBToaster	⊗	○	⊗	○	⊗	○	○	○	○	⊗	○	○	⊗	○	○	○	○	○	○	○	○	○	○	⊗	○	○
ingraph	⊗	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗	⊗	⊘	○	⊗	○	○	⊗	○	⊘	○	○	○	⊗	⊗	⊗	⊗	○

Table 8.2: Support for IVM and the LDBC BI queries (Sec. D.2). *Notation:* ⊗ fully supported, ⊘ a slight variant is supported, ○ not supported.

8.4.1 Workflow Example

To demonstrate the workflow of our approach, we use the example graph in Fig. 8.2c, an extended and slightly altered version of the previous example graph in Fig. 8.1c. The example query in Fig. 8.2a finds Persons p interested in *Music*, who have less than 3 friends living in their street. Fig. 8.2e shows the GRA query plan for the example query. The first `MATCH` clause of the graph query and a filtering condition is compiled to a sequence of a *get-vertices*, two *expand-out*, and a *selection* operator as shown in the bottom left branch of the tree. The pattern in the `OPTIONAL MATCH` clause is compiled similarly, and combined with the other pattern using a *left outer join* on p. Finally, the result is produced by a sequence of *grouping*, *selection*, and *projection* operators.

8.4.2 From Graph RA to Nested RA

As a first transformation step, our workflow replaces *expand* operators with joins, resulting in an NRA query plan. Following the definitions in Sec. 8.3.3, we merge each pair of subsequent *get-vertices* and *expand* operators into a *get-edges* operator. Then, we replace each *expand* with a *join* on a *get-edges* operator.

> **Example 29** The NRA query plan of the example query is shown in Fig. 8.2f, with the corresponding schema definitions in red ■. The *expand* operators for the KNOWS / INTEREST edges and their child operators are combined to a *get-edges* operator, while the *expand* operator for CLASS is replaced with a join on a *get-edges* operator. Other nodes of the GRA plan are left unchanged in the NRA plan.

8.4.3 From Nested RA to Flat RA

Challenges Both GRA and NRA are nested algebras and represent vertex/edge properties as nested relations. As discussed in Sec. 8.3, we use shorthand *S2* to access properties with a convenient syntax, e.g. the projection operator in expression $\pi_{p.\mathsf{name}}$ is allowed to use the value of the name property of vertex p. However, due to the schema-free nature of property graphs, property keys of vertices/edges are not known in advance during compilation. The GRA and NRA formalizations work around this issue by treating the base relations of vertices and edges as nested (NF^2) relations. While this solves the problem in theory, it poses further challenges: nested relations are difficult to store efficiently and are not handled by most IVM algorithms. Hence, as the final step of the compilation, we transform the query plan to *flat relational algebra* (FRA), which is incrementally maintainable.

Schema inferencing We refer to the schema of NRA operators as the *nested schema*, as it describes nested relations. In contrast, an FRA operator has a *flat schema*, which contains all property keys required by the current operator and subsequent operators in the query plan. The flat schema is

```
Data: op: NRA operator
Data: props: properties required by later ops, initally ∅
Function InferRequiredProperties(op, props)
    props ← props ∪ ExtractProperties(op)
    switch op do
        case is a nullary operator do
            op.requiredProperties ← props
        case is a unary operator do
            if op.type ∈ {π, γ} then
                op.requiredProperties ← props
            op.child ← InferRequiredProperties(op.child, props)
        case is a binary operator do
            leftProps ← ∅; rightProps ← ∅
            foreach p ∈ props do
                if element of p ∈ op.left.nestedSchema then
                    leftProps ← leftProps ∪ {p}
                else
                    rightProps ← rightProps ∪ {p}
            op.leftChild ← InferRequiredProperties(op.leftChild, leftProps)
            op.rightChild ← InferRequiredProperties(op.rightChild, rightProps)
    return op
Function ExtractProperties(op)
    switch op do
        case is a π_A or a γ^C_A operator do
            ps ← ListPropertiesInExpression(A)
        case is a σ_θ operator do
            ps ← ListPropertiesInExpression(θ)
        case is an ω_{xs⇒x} operator do
            ps ← ListPropertiesInExpression(xs)
    return ps
Function ListPropertiesInExpression(exp)
    switch exp do
        case is a property, a labels(v) or a type(e) function do
            ps ← {exp}
        case is an arithmetic or logical operator alo do
            ps ← ListPropertiesInExpression(alo.operands)
        case is a function f do
            ps ← ListPropertiesInExpression(f.arguments)
        case is a collection do
            foreach item ∈ exp do
                ps ← ps ∪ ListPropertiesInExpression(item)
    return ps
```

Algorithm 1: Infer required properties for NRA operators.

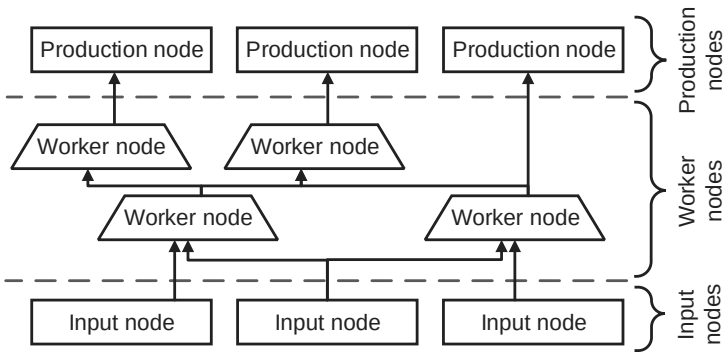


Figure 8.3: The structure of the Rete propagation network.

determined by a two-step *schema inferencing* algorithm, which unfolds the required properties from the nested schema:

1. Starting from the root, the *required properties* are calculated, accumulated and pushed down towards the leafs. The corresponding pre-order traversal in the `InferRequiredProperties` method in Alg. 1, which relies on two other methods: `ExtractProperties` to extract the properties from a given operator and `ListPropertiesInExpression` to list the properties in a given expression. In essence, this step determines the *required properties* for each operator, i.e. the list of properties required by subsequent operators.
2. Next, flat schemas of the FRA operators are calculated with a post-order traversal. For nullary operators, they are defined as a concatenation of the nested schema and the required properties; then, the schema of each subsequent operator is determined according to the conventions of relational algebra, except for the π and γ, operators, where flat schemas are again defined as a concatenation of the nested schema and the required properties.

In essence, the *schema inferencing algorithm* ensures that each relational algebraic operator receives and propagates the attributes required for their calculation (e.g. the attributes used in the selection condition) and those of subsequent operators.[2]

> **Example 30** The FRA plan of the example query is shown in Fig. 8.2f, with the corresponding schema definitions in blue ■. Note that the required properties were added to the schema of each operator. For example, the *get-edges* operator for INTEREST edges – the leaf in the left-most branch of the tree – produces $\langle p, i, t, \underline{p.\mathsf{name}} \rangle$ quadruples, which include the property $p.\mathsf{name}$ used by operator $\pi_{p.\mathsf{name}}$ in the root of the tree.

8.5 View Maintenance on Flat RA

In Sec. 8.4.3, we defined steps to translate queries to an FRA query plan to allow evaluation with existing relational IVM algorithms, e.g. [c4; j3] and [For82; ML91; Han96; HBC02; Ujh+15a]. However, the rich set of operators required by PG queries necessitates the combination of multiple techniques. In this section, we describe our flexible incremental query approach, used in our *ingraph* tool.

8.5.1 Query Evaluation by Rete Network

The Rete algorithm constructs a network of three types of operators. Following Fig. 8.3:

[2] *minimum schema*

1. *Input operators* are responsible for indexing the graph, i.e. they store the appropriate tuples for vertices and edges in the graph. They are also responsible for sending *change sets* as *update messages* to worker operators that are *subscribed* to them.
2. *Worker operators* perform a relational algebraic operation on their input and propagate the results to other worker operators or production operators. Some worker operators are *stateful*: they store partial query results in their memory to allow incremental reevaluation. Worker operators have two types: *unary nodes* have a single input slot, *binary operators* have two input slots.
3. *Production operators* are terminators that provide an interface for fetching the results.

Operation	U.	Update on p	Update on s
$t = p \bowtie s$	Δ	$\Delta t = \Delta p \bowtie s$	$\Delta t = p \bowtie \Delta s$
	∇	$\nabla t = \nabla p \bowtie s$	$\nabla t = p \bowtie \nabla s$
$t = p \overline{\ltimes} s$	Δ	$\Delta t = \Delta p \overline{\ltimes} s$	$\nabla t = p \ltimes (\Delta s \overline{\ltimes} s)$
	∇	$\nabla t = \nabla p \overline{\ltimes} s$	$\Delta t = (p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s)$
$t = p ⟕ s$	Δ	$\Delta t = \Delta p ⟕ s$	$\Delta t = p \bowtie \Delta s, \nabla t = [(p \ltimes (\Delta s \overline{\ltimes} s)) \times \langle \mathsf{NULL} \rangle]$
	∇	$\nabla t = \nabla p ⟕ s$	$\nabla t = p \bowtie \nabla s, \Delta t = [((p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s)) \times \langle \mathsf{NULL} \rangle]$

Table 8.3: Rules for incremental view maintenance. *U.:* update type

The Rete network works as follows. First, the network computes the set of pattern matches in the graph. Then upon a change in the graph, the network is incrementally maintained by propagating *update messages* (also known as *deltas*). Positive tuple sets are denoted with the Δ character, while negative ones are denoted with ∇. Adding new graph matches to the result set is expressed as *positive update messages*, while removing matches results in *negative update messages*. Rules of incremental view maintenance in the Rete network are presented in Tab. 8.3. For join-like operators, we show an example change set for each type of update operation (positive/negative updates on the primary/secondary slot) in Fig. 8.4. A more detailed discussion is available in Appendix E.

Our query engine is built on the *Rete algorithm* [For82; Ber+10; Ber13; Ujh+15a], which was originally developed to incrementally handle production rules in rule-based expert systems (Sec. 10.3). Unlike *algebraic* IVM techniques (e.g. [QW91; GL95]), which derive delta queries to maintain the results of the target query, the Rete algorithm follows a *procedural* approach that maintains each relational algebraic operator separately. This makes the algorithm simpler to implement for our many operators and allows a finer granularity when composing operators (e.g. multi-level aggregations).

The algorithm works as follows. First, it builds a *discrimination network* (referred to as the *Rete network*), which follows the topology of the FRA query plan. Each operator is subscribed to the output of its child operators and propagates the result to its parent operator. Calculations start from the leaf nodes which correspond to nullary operators *get-vertices* and *get-edges*. IVM in the Rete network is achieved by extensive caching: nodes in the Rete network store interim results which allows efficient computation for small updates. In other words, the Rete algorithm employs a *space-time tradeoff* [RSS96] to speed-up query processing evaluation.

> **Example 31** In the example query of Fig. 8.2, the $[\bigcirc\!\!\rightarrow\!\!\bigcirc]$ operator for INTEREST subscribes to the indexer and receives $\{\langle a, 1, n, \text{"Alice"}\rangle, \ldots, \langle e, 5, v, \text{"Edgar"}\rangle\}$ tuples. Other *get-edges* operators are populated similarly, and the results are propagated through the unary and binary relational algebraic operators, producing the initial query result (Fig. 8.2b).

p

A	B
x	1
y	2
z	3

s

B	C
1	0.4
1	0.5
2	0.6

Δp

A	B
m	2
n	4

∇p

A	B
y	2
z	3

Δs

B	C
2	0.7
3	0.8

∇s

B	C
1	0.5
2	0.6

$p \cup \Delta p$

A	B
x	1
y	2
z	3
m	2
n	4

$p - \nabla p$

A	B
x	1
~~y~~	~~2~~
~~z~~	~~3~~

$s \cup \Delta s$

B	C
1	0.4
1	0.5
2	0.6
2	0.7
3	0.8

$s - \nabla s$

B	C
1	0.4
~~1~~	~~0.5~~
~~2~~	~~0.6~~

$p \bowtie s$

A	B	C
x	1	0.4
x	1	0.5
y	2	0.6

$(p \cup \Delta p) \bowtie s$

A	B	C
x	1	0.4
x	1	0.5
y	2	0.6
m	2	0.6

$(p - \nabla p) \bowtie s$

A	B	C
x	1	0.4
x	1	0.5
~~y~~	~~2~~	~~0.6~~

$p \bowtie (s \cup \Delta s)$

A	B	C
x	1	0.4
x	1	0.5
y	2	0.6
y	2	0.7
z	3	0.8

$p \bowtie (s - \nabla s)$

A	B	C
x	1	0.4
~~x~~	~~1~~	~~0.5~~
~~y~~	~~2~~	~~0.6~~

$p \overline{\ltimes} s$

A	B
z	3

$(p \cup \Delta p) \overline{\ltimes} s$

A	B
z	3
n	4

$(p - \nabla p) \overline{\ltimes} s$

A	B
~~z~~	~~3~~

$p \overline{\ltimes} (s \cup \Delta s)$

A	B
~~z~~	~~3~~

$p \overline{\ltimes} (s - \nabla s)$

A	B
z	3
y	2

$p ⟕ s$

A	B	C
x	1	0.4
x	1	0.5
y	2	0.6
z	3	NULL

$(p \cup \Delta p) ⟕ s$

A	B	C
x	1	0.4
x	1	0.5
y	2	0.6
z	3	NULL
m	2	0.6
n	4	NULL

$(p - \nabla p) ⟕ s$

A	B	C
x	1	0.4
x	1	0.5
~~y~~	~~2~~	~~0.6~~
~~z~~	~~3~~	~~NULL~~

$p ⟕ (s \cup \Delta s)$

A	B	C
x	1	0.4
x	1	0.5
y	2	0.6
~~z~~	~~3~~	~~NULL~~
y	2	0.7
z	3	0.8

$p ⟕ (s - \nabla s)$

A	B	C
x	1	0.4
~~x~~	~~1~~	~~0.5~~
~~y~~	~~2~~	~~0.6~~
z	3	NULL
y	2	NULL

Figure 8.4: Example relations for primary and secondary slots (p and s) with positive and negative change sets ($\Delta p, \nabla p, \Delta s, \nabla s$) for demonstrating incremental maintenance operations for the join, antijoin, and left outer join operators.

8.5.2 Cache Maintenance in the Rete Network

Changes in the data, including the initial load phase, are represented logically as changes in nullary operators (○), [○→○], and [○↔○]. Changes are propagated through the network as *update messages* containing positive and negative change sets (representing insertions and deletions, respectively). For each unary and binary FRA operator, incremental maintenance operations are defined for both insertions and deletions in Appendix E.

> **Example 32** In Fig. 8.2, Person "Edgar" gains interest in "Rock" music. This change is represented as adding a tuple $\langle e, 6, r,$ "Edgar"$\rangle$ to the [○→○] operator for type INTEREST, which is propagated through the network, adding a new tuple $\langle$ "Edgar"$\rangle$ to the result set (Fig. 8.2b).

8.5.3 Data Representation and Indexing

The *ingraph* prototype is a memory-only engine with no permanent storage. To allow efficient lookup of vertices, edges, and their properties, it uses an indexer layer. The indexer is capable of performing lookups based on ids/labels, and sending *notifications* on updates of the data, similarly to *active databases* [WC96; PD99]. In general, lookups are cheaper when more constraints are provided, e.g. it is cheaper to get the set of edges when *the edge type and the source/target labels are specified* compared to when only the edge type is known. This is the key reason why our approach uses compound operators (such as *get-edges* which takes one type and two label constraints), instead of using primitive operators as building blocks.

8.5.4 Programming Model

As the Rete network follows the topology of the FRA query plan, the tuples in each network operator correspond to its *flat schema*. This ensures that the internal data representation of operators is compact and allows each operator to perform its computation based on local data without turning to the indexer. This allows us to implement each operator following the *actor programming model* [Agh90], i.e. as a local computation with an isolated mutable state, passing *asynchronous immutable messages* to its subscribers.

In practice, our engine builds an actor network based on the Rete network, i.e. it instantiates one actor for each operator. Nullary operators in the query plan are captured as subscriptions to the indexer (Sec. 8.5.3), which is responsible for performing efficient lookups (based on labels, types and identifiers), and provide *change notifications* upon updates. As actors have no shared state, they can be easily parallelized and deployed distributedly. We demonstrated this with the IncQuery-D engine that implements distributed IVM on top of RDF graphs, see Chapter 9 and paper [c4].

8.6 Evaluation

In order to evaluate the scalability of our approach, we used the Social Network Benchmark (SNB) published by the Linked Data Benchmark Council (LDBC) for property graphs and RDF graphs (Chapter 6). After assessing feature coverage for the benchmark queries, we compared the performance of initial query evaluation and query reevaluation with a non-incremental industrial PG query engine and an IVM research tool.

8.6.1 Benchmark Setup

Benchmark selection To evaluate the performance of our approach, we used the LDBC Social Network Benchmark (SNB), which defines two workloads: *Interactive* [Erl+15] (14 complex read queries, 7 short read queries, 8 updates) and *Business Intelligence* (BI) [l18] (25 complex read queries, updates not yet defined). The Interactive workload defines reads that access a given vertex and its neighbourhood (e.g. friends of friends), and the BI workload specifies queries that access a large portion of the graph. Therefore, the BI queries are significantly more complex to evaluate [l18] and are good candidates for defining views over a changing data set. Considering this, we selected the queries of the BI workload for our benchmark. However, due to the lack of updates, we complemented them with a set of updates from the Interactive workload, and introduced additional updates inspired by other public benchmarks [j2].

Queries We selected 11 queries from the SNB's BI workload: Q2, Q3, Q4, Q6, Q8, Q9, Q12, Q15, Q22, Q23, and Q24. Our selection aimed to maximize the number of queries supported by the IVM tools we compared. These cover a large set of language features such as negative/optional patterns, unwinding, ordering, and aggregations. The detailed specification of the queries can be found in Sec. D.2. For an example, we briefly review Q15:

> Given a *Country* `country`, determine the "social normal", i.e. the floor of average number of friends that *Persons* of `country` have in `country`. Then, find all *Persons* in `country`, whose number of friends in `country` equals the social normal value.

This query stresses many language features: it performs multiple levels of aggregations (count, average), transforms the aggregation result (floor), then uses it to filter on the results of another aggregation. It also uses optional pattern parts, employs ordering and limiting the number of results (see the full specification in [r22]).

```
1  MATCH (c:Country)<-[:PART]-(:City)<-[:IN]-(p1:Person)
2  WHERE c.name = $country
3  OPTIONAL MATCH
4    (c)<-[:PART]-(:City)<-[:IN]-(f1:Person)-[:KNOWS]-(p1)
5  WITH c, p1, count(f1) AS f1Count
6  WITH c, avg(f1Count) AS socialNormalFloat
7  WITH c, floor(socialNormalFloat) AS socialNormal
8  MATCH (c)<-[:PART]-(:City)<-[:IN]-(p2:Person)
9  MATCH (c)<-[:PART]-(:City)<-[:IN]-(f2:Person)-[:KNOWS]-(p2)
10 WITH c, p2, count(f2) AS f2Count, socialNormal
11 WHERE f2Count = socialNormal
12 RETURN p2.id, f2Count AS count
13 ORDER BY p2.id ASC LIMIT 100
```

Listing 8.1: openCypher specification of BI query 15.

Updates Since the BI workload currently lacks update operations, we used two sets of updates for our experiments.

- Append: we reused updates 2, 3, 5, and 8 from the *Interactive workload*. These queries are *append-only*, i.e. they only insert new vertices/edges to the graph.

- Delete: we defined three delete operations based on the Inject scenario of the *Train Benchmark* framework [j2]. While such delete workload is not frequently found in existing graph database benchmarks, we believe that it represents a crucial challenge of high practical relevance, e.g. the "right to be forgotten" regulation of GDPR. Each operation removes an element from the graph, namely a Post (D1), a Person (D2), and an INTEREST edge (D3).

```
1 MATCH (n:Post)
2 WHERE n.id = $postId
3 DETACH DELETE n
```

Listing 8.2: Specification of delete operation D1.

```
1 MATCH (n:Person)
2 WHERE n.id = $personId
3 DETACH DELETE n
```

Listing 8.3: Specification of delete operation D2.

```
1 MATCH (p:Person)-[i:HAS_INTEREST]->(t:Tag)
2 WHERE p.id = $personId AND t.id = $tagId
3 DELETE i
```

Listing 8.4: Specification of delete operation D3.

Benchmark goals In each benchmark run, our goal is to measure the aggregated performance of a *single read query* (e.g. BI Q15), alternated by *updates* that affect a small fragment of the data set. In particular, we measure and analyse the execution times of three benchmark phases:

1. *Initial evaluation:* load the graph and evaluate the query.
2. *Append and reevaluation:* execute the Append updates and reevaluate the query.
3. *Delete and reevaluation:* execute the Delete updates and reevaluate the query.

The execution times of these phases will provide insight into the incremental query evaluation performance of the tested tools.

Tools We executed the queries on our ingraph prototype tool, the DBToaster [Koc+14] engine, and the Neo4j graph database [Web12].

- The *ingraph* engine is a proof-of-concept implementation of the techniques presented in this work, written in Scala using the Akka actor programming library[3].
- *DBToaster* is an academic prototype providing state-of-the-art IVM for SQL queries. It uses *higher-order* IVM techniques, which recursively calculate algebraic deltas on queries and generates highly optimized Scala/C++ code.[4] For our experiments, we used the Scala code generator.
- *Neo4j* is a popular disk-resident graph database, implemented in Java and Scala. Internally, its query engine uses mostly standard relational operators extended with the graph-specific *Expand* and *VarExpand* operators [Fra+18].

Data sets We used 5 data sets of increasing scale factors generated by the SNB's DATAGEN component, ranging from SF0.1 (0.3M vertices, 1.5M edges) to SF10 (30M vertices, 177M edges). For all systems, the graphs were loaded from CSV files.

[3] https://akka.io

[4] Similarly to our approach, the generated Scala code also uses the Akka actor library, to evaluate queries concurrently.

Initial query evaluation For incremental tools, ingraph and DBToaster, we created an incremental view for the complex read query, loaded the data set to memory and measured the total time required to perform the view maintenance. As Neo4j is a disk-based, non-incremental database, we first loaded the data set in advance using its offline import tool[5] and measured its execution time. We then started the database, measured the execution time of the complex read query and used the sum of the two execution times to derive the initial query evaluation time.

Update and query reevaluation We performed the two sets of update operations as follows. For each Append update, we executed 20-20 instances of the operation, based on the update stream produced by the SNB DATAGEN. For each Delete operation, we randomly selected a single vertex/edge. For vertices, ingraph and Neo4j required only the ids of the elements to be deleted and removed all their incoming/outgoing edges. For DBToaster, we explicitly generated the set of records that need to be removed from the database. We used the same sequence of graph elements for each tool.

Environment The benchmark was executed on a cloud VM with Ubuntu 18.04, 240 GB RAM, 32 Xeon Platinum 8167M CPU threads, and Oracle JDK 8 with 224 GB heap memory. Execution times were taken as the median value of 3 runs. We used Neo4j Community Edition v3.3.5 with the *interpreted* runtime[6] and DBToaster v2.2.

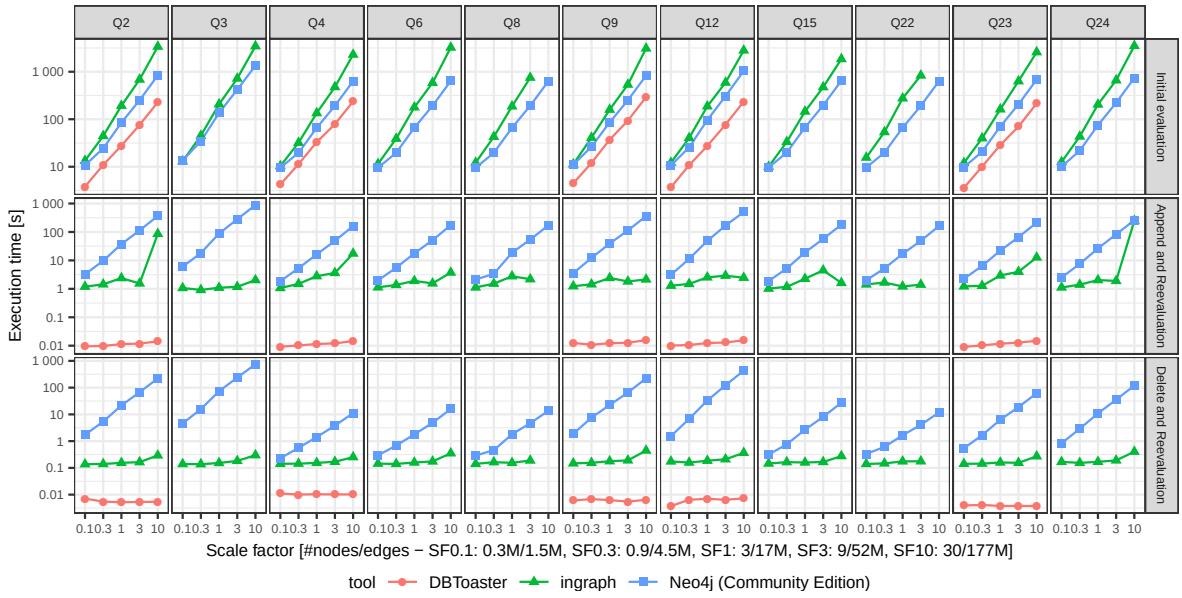


Figure 8.5: Execution time of the LDBC SNB BI benchmark queries – initial evaluation time and reevaluation times with updates. The results of Neo4j are obtained using the *interpreted* runtime of Community Edition 3.3.5. The correctness of the implementations has only been tested for small scale factor models and results are not audited by the LDBC.

[5] https://neo4j.com/docs/operations-manual/3.3/tools/import/

[6] More efficient approaches such as the *compiled* runtime are supported in Neo4j's Enterprise Edition.

8.6.2 Results and Analysis

The measured execution times for each benchmark phase, query, tool, and scale factor is shown in Fig. 8.5. The results clearly show both the benefits and tradeoffs of IVM techniques.

Initial evaluation The results show that ingraph is the slowest for initial evaluation, taking almost an hour for queries on SF10. This can be attributed to its extensive cache building mechanisms (which later allow efficient IVM) and, to some extent, its unoptimized CSV loader. On the contrary, DBToaster can only evaluate a few queries (Q2, Q4, Q9, Q12, and Q23), but provides the best performance on all scale factors even for initial evaluation. This confirms that DBToaster uses state-of-the-art techniques for IVM, which do not penalize the initial evaluation as much. Meanwhile, the results for Neo4j show that it has the best feature coverage and provides balanced performance.

Update and reevaluation For this phase, ingraph shows close to constant characteristics for all queries. The Append updates and subsequent reevaluations take noticeably longer, which can be explained with the larger number of updates needing more time just to compile. All Delete updates (each executed with a single Cypher query) and subsequent reevaluations are completed in less than 750 ms.

For its supported queries, DBToaster guarantees almost instant reevaluation with execution times in the range of 1…20 ms, regardless of the dataset size. These results confirm the justification for using relational IVM algorithms to evaluate PG queries, but they also highlight the requirement to provide sufficient feature coverage, which DBToaster lacks.

Neo4j does full recalculation after most updates and is therefore consistently outperformed by incremental engines for reevaluation. However, we can also observe that Neo4j provides some degree of incrementality: when performing Delete updates and reevaluating Q4, Q6, Q8, Q15, and Q22, it detects that the update do not influence the respective queries.

Analysis We compared our ingraph engine to two other tools: a popular industrial property graph database, Neo4j, and a state-of-the-art academic IVM tool prototype, DBToaster. The results show that a standard relational IVM approach can achieve significant speed-up for complex graph queries defined in a challenging graph benchmark w.r.t. non-incremental industry tools. However, the practical applicability of IVM techniques for property graphs not only depends on the performance of a technique, but also on its expressiveness and feature coverage. In that respect, ingraph clearly outperforms state-of-the-art research prototypes such as DBToaster, covering 11+ out of 25 queries in the BI workload. The results also highlight that the IVM algorithm of ingraph involves a costly initialization for the first execution, due to building caches for interim results. Hence, compared to non-incremental approaches, ingraph typically consumes more memory (which hinders its scalability) and performs the initial query evaluation slower.

8.6.3 Threats to Validity

Correctness and fairness To check the correctness of implementations, we tested the equivalence of the queries for all systems on small data sets. To ensure fairness, we used the same openCypher query specifications for Neo4j and ingraph.

A common threat to validity is the lack of representativeness of the benchmark workload. To ensure that the dataset is representative, LDBC SNB uses a data generator that generates graphs with realistic structure and attributes. To ensure this, it enforces degree distributions based on data released

by Facebook and uses a dictionary-based property generator [Erl+15]. LDBC designs its queries using a set of choke points (Sec. 6.2.1), which represent challenging aspects of query processing. Queries for a certain benchmark workload are designed to cover given choke points and are then refined in multiple iterations based on feedback from industry experts. While we did not benchmark all queries in the BI workload, the selected ones already cover all language-specific choke points defined in [l18], except *CP-8.6 Handling paths*, which would require us to tackle challenge 10 presented in Sec. 8.1. This way, we also avoided cherry-picking queries that would favour our approach.

8.7 Conclusion

We presented an approach for the incremental evaluation of property graph queries captured in the openCypher language. Our approach proposes a transformation chain that first compiles graph queries to graph relational algebra, then translates them to nested relational algebra and finally converts them to flat relational algebra. The expression obtained as a result can then be evaluated and maintained by using relational IVM techniques.

Our experimental evaluation carried out on an open benchmark (the LDBC SNB's Business Intelligence workload [l18]) clearly demonstrated that (1) the query reevaluation performance of our approach significantly outperforms the industrial (but non-incremental) Neo4j engine, and (2) the feature coverage of our ingraph tool significantly exceeds other modern IVM tools used for property graphs on the same benchmark.

Up to our best knowledge, this is the first work dedicated to systematic study of *incremental view maintenance on property graphs*. Our paper also present numerous unsolved challenges which can open up interesting research directions:

- It allows using recent advancements in incremental join algorithms such as [IUV17; Idr+18] and [Amm+18] for PG queries.
- It facilitates the development of cost-based optimization techniques for property graph queries [Gub15; VD13].
- The presented incremental evaluation techniques can be used to define *graph views* on top of RDBMSs [XSD17].
- It can be extended by adapting algorithms designed to perform graph-specific operations, e.g. *impact analysis techniques* [Egy06; RE12] and incremental transitive closure [KS02; Rod08a].

Chapter 9

Distributed Incremental View Maintenance for Scalability

In Chapter 8, we presented an approach to reduce property graph queries to the language of relational algebra. We performed the evaluation of the results on a single-node system with a many CPU cores and a lot of memory. It is easy to see that a single node system poses an inherent scalability bottleneck, although large machines with terabytes of memory are already available at big cloud providers. Still, arguments are being made both for [Lin18] and against [SÖ18] *scaling up* on a resourceful single machine for complex graph processing workloads.

In this chapter, we present an approach that allows horizontal scaling of graph queries, based on the well-known Rete algorithm and the actor programming model. We propose a *novel architecture for a distributed and incremental model query framework* by adapting incremental graph pattern matching techniques to a distributed cloud based infrastructure. A main contribution of our novel architecture is that the distributed storage of data is completely separated from the distributed handling of indexing and query evaluation. Therefore, caching the result sets of queries in a distributed fashion provides a way to *scale out* the memory intensive components of incremental query evaluation, while still providing instantaneous execution time for complex queries. To demonstrate the feasibility of the approach, we present IncQuery-D, a prototype tool based on a distributed Rete network that can scale up from a single workstation to a cluster to handle very large models and complex queries efficiently. We first briefly revisit the Train Benchmark (Sec. 9.1), then use its model and queries to demonstrate our approach (Sec. 9.2). For the performance experiments, we extend the benchmark it to a distributed setup and analyse the derived results (Sec. 9.3), which demonstrates that our distributed incremental query layer can be significantly more efficient than the native SPARQL query technology of an RDF triplestore.

This chapter mainly follows paper [c4].

9.1 Running Example

In this chapter, we use the Train Benchmark to present our core ideas and evaluate the feasibility of the proposed approach. The detailed specification of the benchmark is presented in Chapter 5. We use four queries of the benchmark: two simple ones involving at most 2 objects (PosLength and SwitchMonitored), and two more complex ones involving 4–8 objects and multiple join operations (RouteSensor and SemaphoreNeighbor). We use the Repair scenario, which defines a quick fix-like

model transformation for each query (Sec. 5.3.3). The transformation specification is given in Sec. 5.3.6 for query RouteSensor and in Appendix C for the rest of the queries.

9.2 A Distributed Incremental Graph Query Framework

The queries and transformations of the Train Benchmark represent a typical workload profile for state-of-the-art modelling tools [Izs+13b]. With current MDE technologies, such workloads can be acceptably executed for models up to several hundred thousand model elements [Ujh+15a], however when using larger models consisting of multiple million elements (a commonplace in complex domains such as AUTOSAR [Ber+10]), the performance of current tools is often not acceptable [Kol+13]. Incremental techniques can provide a solution, however they require additional (memory) resources.

The primary goal of our approach is to provide an architecture that can make use of the distributed cloud infrastructure to scale out memory-intensive incremental query evaluation techniques. As a core contribution, we propose a three-tiered architecture. To maximize the flexibility and performance of the system, model persistence, indexing and incremental query evaluation are delegated to three independently distributable asychronous components. Consistency is ensured by synchronized construction, change propagation and termination protocols.

9.2.1 Architecture

In the following, we introduce the architecture of IncQuery-D (see Fig. 9.1), a scalable distributed incremental graph pattern matcher. The *storage layer* is a distributed database which is responsible for persisting the model. The client application communicates with the *middleware* ①, which consists of two components: the *distributed indexer* and the *model access adapter*. Together, these provide a unified API for accessing the database ②, send change notifications ③ to the production network. The latter is implemented as a *distributed query evaluation network*, with the theoretical underpinning of the Rete algorithm, which provides incremental query evaluation. At the other end of the production network, the user transaction continuously receives the changes in the query results ④.

Storage For the storage layer, the most important issue from an incremental query evaluation perspective is that the *indexers* of the system should be filled as quickly as possible. This favours database technologies where model sharding can be performed appropriately (i.e. with balanced shards in terms of type-instance relationships), and elementary queries can be executed efficiently. Our framework can be adapted to fundamentally different storage back-ends, including triplestores, graph databases and relational database managements systems.

Model access adapter In contrast to a traditional setup where the distributed model repository is accessed on a per-node basis by a model manipulation transaction, IncQuery-D provides a model access adapter that offers three core services:

1. The primary task is to provide a *surrogate key mechanism* so that each model element in the entire distributed repository can be uniquely identified and located within storage shards.
2. The model access adapter provides a *graph-like data manipulation API* (① in Fig. 9.1) to the user. The model access adapter translates the operations issued by the user to the query language of the backend and forwards it to the underlying data storage.
3. *Change notifications* are required by incremental query evaluation, thus model changes are captured and their effects are propagated in the form of *notification objects* (③ in Fig. 9.1). The

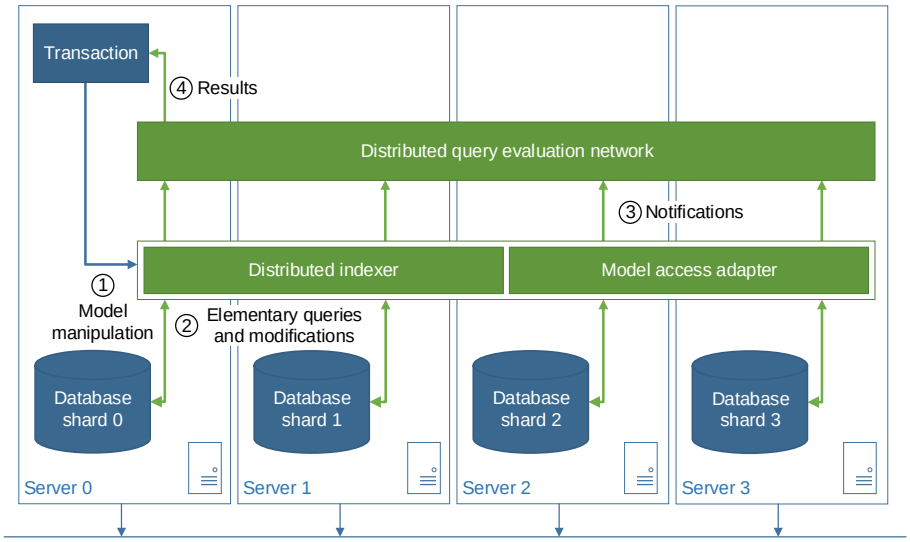


Figure 9.1: The architecture of the proposed incremental query framework, deployed in a sample 4-node cluster configuration.

notifications generate *update messages* that keep the state of the query evaluation network consistent with the model. While relational databases usually provide *triggers* for generating notifications, most triplestores and graph databases lack this feature. Due to the lack of general support, notifications are controlled by the model access adapter by providing a façade for all model manipulation operations.

Distributed indexer Indexing is a common technique for decreasing the execution time of database queries. In model-driven engineering (MDE), *model indexing* has a key role in high performance model queries. As MDE primarily uses a metamodelling infrastructure, all queries utilize some sort of type attribute. Typical elementary queries include retrieving all nodes of a certain type (e.g. get all nodes of the type Route), or retrieving all edges of a certain type/label (e.g. get all edges of label sensor).

To support efficient query processing, IncQuery-D maintains type-instance indexes so that all instances of a given type (both nodes and edges) can be enumerated quickly. These indexers form the bottom layer of the distributed query evaluation network. During initialization, these indexers are filled from the database backend (② in Fig. 9.1). The architecture of IncQuery-D facilitates the use of a *distributed indexer* which stores the index on multiple servers. A distributed indexer inherently provides some measures to mitigate the memory limits of a single node.

Distributed query evaluation network IncQuery-D constructs a distributed and asynchronous network of communicating nodes that are capable of producing the results set of the defined queries (④ in Fig. 9.1). Our prime candidate for this layer is the *Rete algorithm*, however, the architecture is capable of incorporating other incremental (e.g. TREAT [ML91]) and search-based query evaluation algorithms as well. In the upcoming section, we provide further details on this critical component of the architecture.

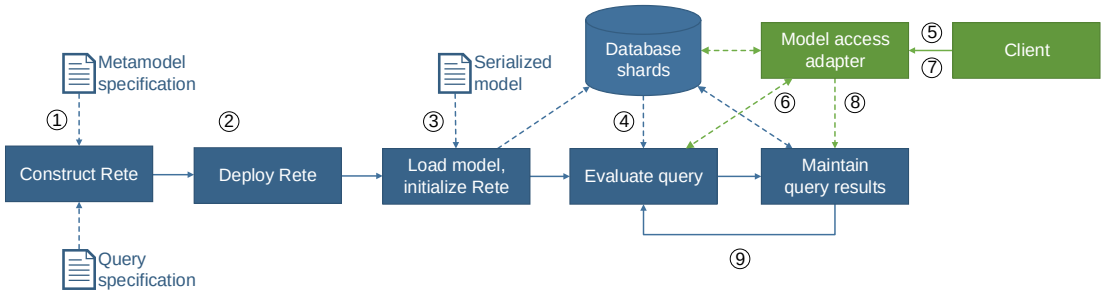


Figure 9.2: The operational workflow of the distributed Rete algorithm.

9.2.2 The Rete Algorithm in a Distributed Environment

Numerous algorithms were proposed for the purpose of incremental query evaluation. The Rete algorithm was originally defined for rule-based expert systems [For82], then later improved and adapted for EMF models in [Ber13]. This chapter discusses how to facilitate the Rete algorithm in a distributed environment.

Data representation and structure The Rete algorithm uses *tuples* to represent the nodes (along with their properties), edges and subgraphs in the graph. The algorithm defines an asynchronous network of communicating nodes (see Fig. 9.3).

The network consists of three types of nodes. *Input nodes* are responsible for indexing the model by type, i.e. they store the appropriate tuples for the nodes and edges. They are also responsible for producing the update messages and propagating them to the *worker nodes*. *Worker nodes* perform a transformation on the output of their parent node(s) and propagate the results. Partial query results are represented in tuples and stored in the memory of the worker node thus enabling incremental query reevaluation. *Production nodes* are terminators that provide an interface for fetching the results of the query and the changes introduced by the latest transformation.

Construction The system constructs the Rete network from the layout derived from the query specification. The construction algorithm may apply various optimization techniques, e.g. reusing existing Rete nodes, known as *node sharing* [Ber13]. An efficient Rete construction algorithm is discussed in detail in [VD13], but is out of scope for this work.

In a distributed environment, the construction of the Rete network introduces additional challenges. First, the system must keep track of the resources available in the server cluster and maintain the mapping between the Rete nodes and the servers accordingly. Second, the Rete nodes need to be aware of the current infrastructure mapping so they can send their messages to the appropriate servers. In our system, the Rete nodes are remotely instantiated by the coordinator node. The coordinator node then sends the infrastructure mapping of the Rete network to all nodes. This way, each node is capable of subscribing to the update messages of its parent node(s). The coordinator also starts the operations in the network, such as loading the model, initiating transformations and retrieving query results.

Operation The operational workflow of IncQuery-D is shown in Fig. 9.2. Based on the *metamodel* and the *query specification*, IncQuery-D first constructs a Rete network ① and deploys it ②. In the next step, it loads the model ③ and traverses it to initialize the indexers of the Rete network. The Rete

network evaluates the query by processing the incoming tuples ④. Because both the Rete indexers and the database shards are distributed across the cluster, loading the model and initializing the Rete network needs network communication. The client is able to retrieve the results ⑤–⑥, modify the model and reevaluate the query again ⑦–⑨.

The modifications are propagated in the form of *update messages* (also known as *deltas*). Creating new graph elements (vertices or edges) results in *positive update messages*, while removing graph elements results in *negative update messages*. The operation of the network is illustrated on the instance graph depicted in the lower left corner of Fig. 9.3. This graph violates the well-formedness constaint defined by the RouteSensor query, hence tuple $\langle 3, 4, 2, 1 \rangle$ appears in the result set of the query. The figure also shows the Rete network containing partial matches of the original graph.

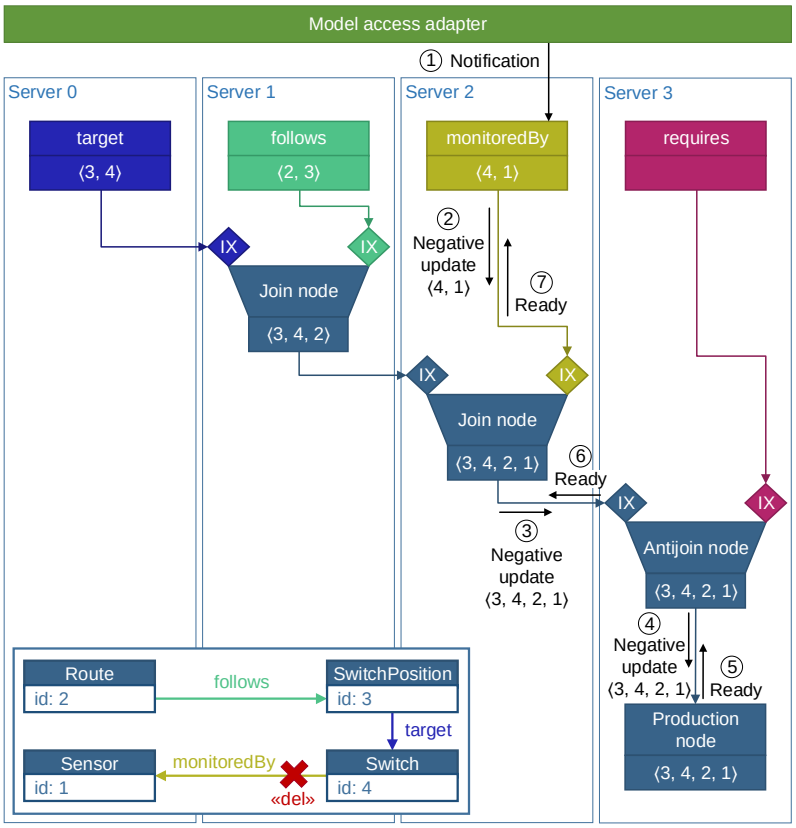


Figure 9.3: A transformation sequence on a distributed Rete network.

To resolve the violation, we apply the quick fix transformation defined in the Train Benchmark (Sec. 5.3.6) and delete the monitoredBy edge between nodes 4 and 1. When the edge is deleted, the monitoredBy type indexer (an input operator) receives a notification from the model access adapter ① and sends a *negative update* ② with the tuple $\langle 4, 1 \rangle$. The subsequent join node processes the update messages and propagates a negative update ③ with the tuple $\langle 3, 4, 2, 1 \rangle$. The *antijoin* operator (Def. 17) also propagates a negative update message with the same tuple ④. This is received by the *production* operator, which initiates the *termination protocol* ⑤–⑦. After the termination protocol finishes, the indexer signals the client about the successful update. The client is now able to retrieve the results from the production operator. The client may choose to retrieve only the *change set*, i.e. only the tuples that have been added or deleted since the last modification.

Termination protocol Due to the asynchronous propagation of changes in Rete, the system implements a *termination protocol* to ensure that the query results can be retrieved consistently with the model state after a given transaction (i.e. by signaling when the update propagation has been terminated). The protocol works by adding a *stack* data structure to the update message propagated through the network. The stack registers each Rete node the message passes through. After the message reaches a production operator, the termination protocol starts. Based on the content of the stack, acknowledgement messages (Ready) are propagated back along the network. When all relevant input operator (where the original update message(s) started from) receive the acknowledge messages, the termination protocol finishes. The operation of the termination protocol can be observed in Fig. 9.3 (messages ⑤–⑦).

Data representation Conceptually, the architecture of IncQuery-D allows the usage of a wide scale of model representation formats. Our prototype has been evaluated in the context of the *property graph* and the *RDF* data model, but other mainstream metamodelling and knowledge representation languages such as relational databases' SQL dumps and Ecore instance models [Ste+09] could be supported, as long as they can be mapped to an efficient and distributed storage backend.

The Rete network (Sec. 8.5.1) in IncQuery-D uses *tuples* for data representation. A tuple is an ordered list of elements, e.g. $\langle 15, 23, 81, 2\rangle, \langle 7, \text{“red”}, \text{true}\rangle$ are tuples. The Rete algorithm defined in [Ber13] uses tuples to represent the nodes, edges and matches of the (sub)patterns in the graph. The items in a tuple are referenced by their index. A *pattern mask* is an array of indexes, $\mu = (\mu_0, \mu_1, \dots, \mu_{n-1})$, which can be used to select certain elements in a tuple e.g. extracting the items defined by the pattern mask $\mu = (1, 3)$ from tuple $\langle 15, 23, 81, 2\rangle$ results in the tuple $\langle 23, 2\rangle$.

9.3 Evaluation

To evaluate the feasibility of the IncQuery-D approach, we created a distributed benchmark environment. We implemented a prototype of IncQuery-D and compared its performance to a state-of-the-art non-incremental SPARQL query engine of a (distributed) RDF store.

9.3.1 Benchmark Scenario

In order to measure the efficiency of model queries and manipulation operations over the distributed architecture, we adapted the Train Benchmark to a distributed environment. As discussed in Chapter 5, the main goal of the Train Benchmark is to measure the query reevaluation times in systems operating on a graph-like data set. The benchmark targets a “real-world” MDE workload by running a specific set of queries and transformations on the model. In this workload profile, the system runs either a single query or a single transformation at a time, as quickly as possible.

Instance models To assess scalability, the benchmark uses instance models of growing sizes, each model containing twice as many model elements as the previous one (Sec. 5.3.7). Scalability is also evaluated against queries of different complexity. For a successful run, the tested tool is expected to evaluate the query and return the *identifiers* of the model elements in the result set.

Transformations In the transformation phase, the benchmark runs quick fix transformations (Sec. 9.1) on 10% of the invalid elements (the result set of the check phase), except for the Semaphore-Neighbor query, where 1/3 of the invalid elements are modified. The transformations run in a single

SF	Triples	Nodes	Edges	PosLength (2)		RouteSensor (4)		SemN. (8)		SwitchM. (2)	
				RSS	MS	RSS	MS	RSS	MS	RSS	MS
1	23k	6k	17k	470	47	94	9	3	1	19	1
4	86k	23k	63k	1 769	176	348	31	6	2	91	9
16	334k	88k	245k	6 893	689	1 301	126	19	6	326	29
64	1M	361k	1M	28 239	2 823	5 324	511	69	19	1 287	119
256	5M	1M	3M	110 739	11 073	21 097	1 996	254	74	5 109	485
1 024	21M	5M	15M	443 458	44 345	84 107	8 024	983	287	20 716	1 977
4 096	85M	22M	63M	1 769 402	176 940	336 507	32 051	–	–	81 410	7 730

Table 9.1: Metrics of the instance models (of scale factors 4^n) and their metrics on selected Train Benchmark queries. *SF:* scale factor, *RSS:* result set size, *MS:* modification size.

logical transaction, implemented with multiple physical transactions, each affecting at most 500 graph elements.

Metrics To quantify the complexity of the benchmark test cases, we use a set of metrics that have been shown to correspond well to performance [Izs+13b]. The values for the test cases are shown in Tab. 9.1. The problem size numbers take the values of 2^n in the range from 1 to 4096. For space considerations, only every other problem size is listed. The complexity of an instance model is best described by the *number of its triples*, equal to the sum of its nodes and edges. The queries are quantified by the *number of their variables* (shown in parentheses) and their *result set size* (RSS). The transformations are characterized by the number of model elements modified (*modification size*, MS).

9.3.2 Benchmark Architecture

Benchmark executor The benchmark is controlled by a distinguished node of the system, called the *executor*. The executor delegates the operations (e.g. loading the model) to the distributed system. The queries and the model manipulation operations are handled by the underlying database management system which runs them distributedly and waits for the distributed operation to finish, effectively creating a synchronization point after each transaction.

Benchmark setups We defined two benchmark setups:

1. As a *non-incremental baseline*, we used an open-source distributed triplestore and SPARQL query system, 4store [HLS09].
2. We deployed IncQuery-D with 4store as a backend database.

Methodology It is important to mention that the benchmark is strongly centralized: the *coordinator* node of IncQuery-D runs on the same server as the benchmark *executor*. The benchmark executor software used the framework of the Train Benchmark to collect data about the results of the benchmark. These were not only used for performance benchmarking but also to ensure the functional equivalence of the systems under benchmark.

Phases The precise execution semantics for each phase are defined as follows:

1. The check phase includes loading the model from the disk (serialized as RDF/XML [GS14]), persisting it in the database backend, and, in the case of IncQuery-D, initializing the Rete network.

2. The execution time of the check phase is the time required for the first complete evaluation of the query.
3. The transformation phase starts with the selection of the invalid model elements and is finished after the modifications are persisted in the database backend. In the case of IncQuery-D, the transformation is only finished after the Rete network has processed the changes and is in a consistent state.
4. The recheck phase re-runs the query of the check phase, and retrieves the updated results.

The execution time includes the time required for the defined operation, the computation and I/O operations of the servers in the cluster and the network communication (to both directions). Both during the development and in runtime we ensured the *functional equivalence* of the measured tools by comparing their results to a reference result. During the development, we followed the Train Benchmark's specification as presented in [r19]. This precisely defines the steps for each phase, e.g. the number of elements to modify in each transformation. For testing, we checked the result set for correctness against the reference implementation.

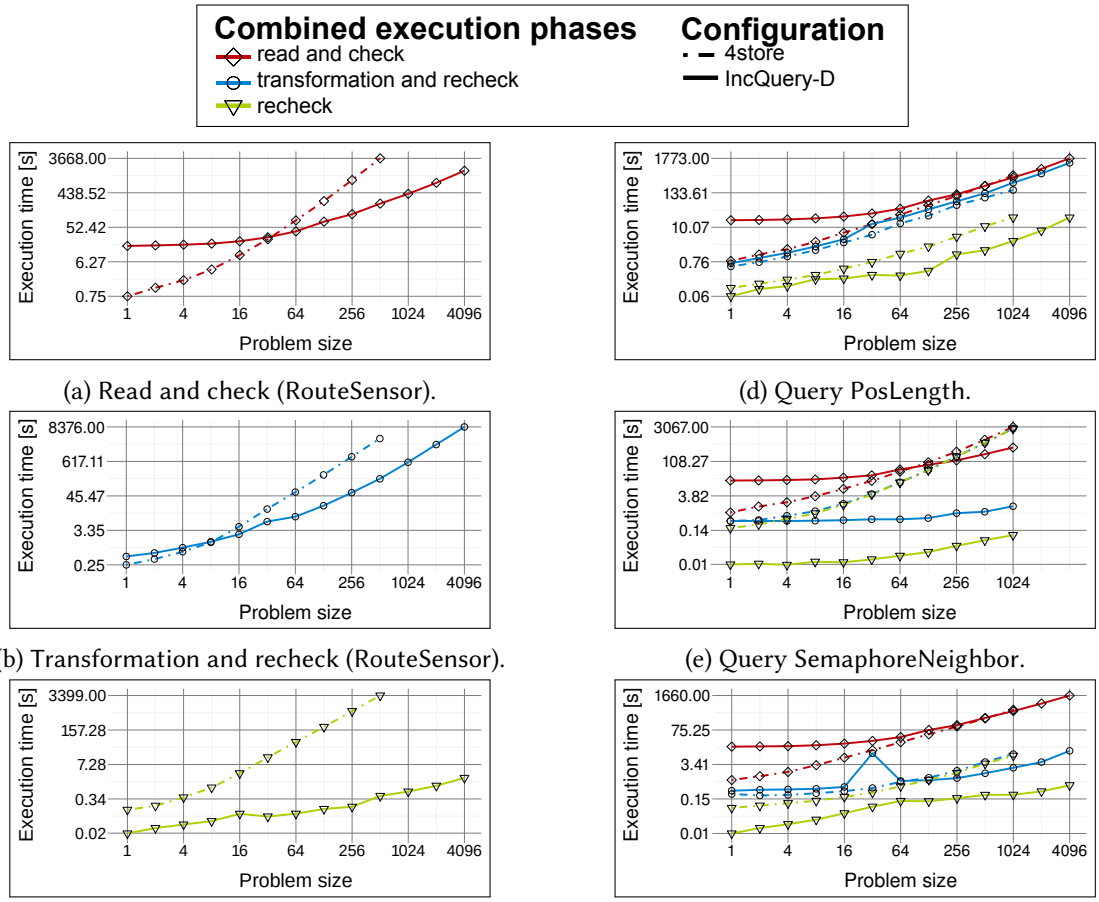
Environment We used 4store [HLS09] (version 1.1.5) as our storage backend. The benchmark ran on the Ubuntu 12.10 64-bit operating system with Oracle JDK 7. For the implementation of the distributed Rete network, we used Akka [Lig18] (version 2.1.4), a distributed, asynchronous messaging system. The system was deployed on the private cloud that runs on the Apache VCL (Virtual Computing Lab) platform. We reserved four virtual machines on separate host machines, with each using a quad-core Intel Xeon L5420 CPU running at 2.5 GHz and having 16 GB of RAM. The host machines were connected to a dedicated gigabit Ethernet network.

9.3.3 Results

The benchmark results of our experiments are shown in Fig. 9.4. On each plot, the x axis shows the problem size, i.e. the size of the instance model, while the y axis shows the execution time of a certain phase, measured in seconds. Both axes use logarithmic scale.

First, we discuss the results for RouteSensor, a query of medium complexity. Fig. 9.4a presents the combined execution time for the load and initial validation phases. The execution time is a low order polynomial of the model size for both the standalone 4store and the IncQuery-D system. The results show that despite the initial overhead of the Rete network initialization, IncQuery-D has a significant advantage starting from medium-sized models (with approximately 1 million triples). Fig. 9.4b shows the execution time for the sum of the transformation and recheck phases. The results show that the Rete maintenance overhead imposed by IncQuery-D on model manipulation operations is low, and overall the model transformation phase when using IncQuery-D is considerably faster for models larger than a few hundred thousand triples. Fig. 9.4c focuses on the recheck phase. The performance of IncQuery-D is characteristically different from that of the SPARQL engine of 4store. Even for models with tens of millions of tuples, IncQuery-D provides close to instantaneous query re-evaluation.

Fig. 9.4d, 9.4e, and 9.4f show detailed results for the PosLength, the SemaphoreNeighbor and the SwitchMonitored queries, respectively. The PosLength query uses only a few variables but has a large result set, which is a challenge for incremental query evaluation systems. Still, IncQuery-D still provides reasonably fast load, transformation and query evaluation times, while outperforming 4store on the recheck time. The SemaphoreNeighbor query includes many variables but has a small match set. Its results show that IncQuery-D has a characteristic advantage on both the transformation and the recheck phases. The SwitchMonitored query uses a few variables and has a medium-sized result set, also showing a clear advantage of IncQuery-D for transformation and recheck.



(a) Read and check (RouteSensor).

(d) Query PosLength.

(b) Transformation and recheck (RouteSensor).

(e) Query SemaphoreNeighbor.

(c) Standalone recheck phase (RouteSensor).

(f) Query SwitchMonitored.

Figure 9.4: Execution times of queries and transformations in different benchmark phases.

Summary of observations Based on the results, we can conclude the following observations. As expected, due to the overhead of constructing the Rete network, the *non-incremental approach* is often faster for small models. However, even for medium-sized models (with a couple of million triples), the Rete construction overhead already pays off for the initial query evaluation (check phase). After the Rete network is initialized, IncQuery-D provides significantly improved transformation and recheck times, with the recheck times being consistently orders of magnitude faster due to the different characteristics of their execution time. In summary, these observations show that IncQuery-D is not just capable of processing models with over 10 million elements, but also that it provides close to instantaneous query evaluation times even for very complex queries.

Threats to validity To minimize *internal threats to validity*, we turned off the caching mechanisms of the operating system to force rereading the serialized model from the disk. Additionally, to avoid the propagation of the warmup effect of the Java Virtual Machine between the runs, each test case was started independently in separate JVM.

As our cloud infrastructure was subject to minimal concurrent load during the measurements, we aimed to minize the distortion due to load transients by running the benchmark three times and taking the *minimum value* for each phase into consideration. We did experience a certain deviation of

execution times for smaller models (Fig. 9.4f). However, for larger models (our most important target), the transient effects have no significant influence on the validity of the benchmark results.

Regarding *external validity*, we used a benchmark that is a faithful representation of a workload profile of a modelling tool for large-scale models (Chapter 5). For both systems (4store and IncQuery-D), the query implementations were validated by domain experts. We aimed to minimize the potential bias introduced by the additional degrees of freedom inherent in distributed systems, e.g. by a randomized manual allocation of the processing nodes of Rete network in the cloud.

9.4 Conclusion

In this chapter, we presented IncQuery-D, an approach to extend incremental systems for distributed processing, which allows processing large graphs with complex query workloads. Similarly to ingraph (presented in Chapter 8, our proposal is based on a distributed Rete network. In this setup, the network is decoupled from a sharded graph database by a distributed model indexer and model access adapter. We presented a detailed performance evaluation in the context of quick-fix software design model transformations combined with on-the-fly well-formedness validation, using the Train Benchmark's Repair scenario. The results show nearly instantaneous complex query re-evaluation well beyond 10^7 model elements.

Chapter 10

Comparison to Related Graph Query and IVM Techniques

In this chapter, we discuss why grap queries are inherently difficult (Sec. 10.1), review the literature on worst-case optimal join algorithms (Sec. 10.2), incremental view maintenance (IVM) techniques (Sec. 10.3), graph query languages (Sec. 10.4), and mapping between graph and relational queries (Sec. 10.5). We finish the chapter with remarks on the relationship between joins, matrix multiplications, and graph queries (Sec. 10.6).

10.1 The Complex Structure of Graphs and Its Implications

There are numerous works on query optimization and evaluation for relational databases. However, the challenges running graph queries efficiently often receives significantly less attention, despite the fact that it is often considered *more difficult than relational query optimization and evaluation* [Che+08; Fan+10; Erl+15]. The key reasons behind this are the following.

1. The *curse of connectedness*: graph queries exhibit scattered, random access patterns on data. Common latency hiding and program optimization techniques employed in *contemporary computer architectures* – such as *CPU-level caching* and *branch prediction* [HP12] – are designed to operate on linear and simple hierarchical data structures (lists, stacks, or trees). However, they are considerable less efficient for graph processing: graph workloads can expect orders of magnitude more cache misses and failed predictions [Sha+17].
2. Due to the difficulty of sharding the graph into separate chunks, it is very difficult to process a graph in parallel [Sha+17]. This is true both for single machine (multi-core) system and presents even more challenges in multi-server distributed setups.
3. Access of properties (or attributes) on graph elements makes query evaluation more expensive as it requires extra memory for building indices over the graph attributes in addition to the structural indices. Additionally, it makes optimization significantly more complex [SEH12].

All of these reasons affect tools both in MDE tools and database engines equivalently.

10.2 Worst-Case Optimal Join Algorithms for Subgraph Queries

Relational and graph database managements systems rely heavily on join operations, and often spend the majority of query execution time evaluating joins (see the choke points considering *join perfor-*

mance in Sec. D.1.2). Acknowledging the importance of their role, improving the efficiency of join operators is a well-researched aspect of database systems. That said, industrial implementations carry a lot of inertia and often stick with proven solutions instead of adapting new algorithms. In particular, current industry-grade DBMSs use almost exclusively a sequence of *binary joins* (also known as *pairwise* or *two-way* joins) to express the joins on multiple relations and apply optimization techniques that have their roots in IBM's System R experimental DBMS [Sel+79].

While the challenge of evaluating *cyclic queries* (Def. 27) has been known om the context of object-oriented databases [KKD89], it has only been shown recently that binary joins are asymptotically suboptimal for such queries [NRR13; Ngu+15a] (including ones common in subgraph matching). The most basic example is the so-called *triangle query*:

$$Q_{\triangle} \equiv {}_{c}\odot\!\!\leftrightarrow\!\!\odot_{a}\left({}_{b}\odot\!\!\leftrightarrow\!\!\bigcirc_{c}\left({}_{a}\odot\!\!\leftrightarrow\!\!\bigcirc_{b}\left(\bigcirc_{a}\right)\right)\right),$$

This query can be rewritten into NRA/FRA as two binary joins:

$$Q_{\triangle} \equiv \left[{}_{a}\bigcirc\!\!\leftrightarrow\!\!\bigcirc_{b}\right] \bowtie \left[{}_{b}\bigcirc\!\!\leftrightarrow\!\!\bigcirc_{c}\right] \bowtie \left[{}_{c}\bigcirc\!\!\leftrightarrow\!\!\bigcirc_{a}\right]$$

Paper [NRR13] proved that for this query, all evaluation plans that only use binary joins yield suboptimal results as they need to enumerate all wedges (Def. 33), requiring at least $\Omega\left(n^2\right)$ steps for a graph with n edges. In many realistic graphs, only a small fraction of these wedges will close into a triangle (Def. 34). The family of *worst-case optimal join algorithms* (WCOJs) tackles this issue by using a set of *intersection operations* to reduce the size of interim datasets, reducing the complexity to $\mathcal{O}\left(n^{3/2}\right)$, an considerable asymptotic speedup of $\sqrt{n}$.

WCOJ algorithms Non-incremental WCOJ algorithms include the *Leapfrog Triejoin algorithm* (LFTJ) of the Datalog-based LogicBlox system [Are+15], which was implemented before the limitations of binary joins were identified and the term WCOJ was coined. Hence, it can be considered as an "accidentally worst-case optimal" algorithm.

Paper [Ngu+15a] and its accompanying technical report [Ngu+15b] found that "mathematically, (hyper)graph pattern matching is equivalent to join processing", and stated that RDBMSs provide subpar performance for graph queries partly due to the inherent suboptimality of using query plans with exclusively binary joins. To investigate whether WCOJ algorithms are able to speedup *subgraph matching* queries, they test two algorithms: LFTJ and the *Minesweeper* algorithm [Ngo+14].

Experiments for applying WCOJ algorithms were carried out in the context of RDF data management [Abe+16].[1] WCOJ algorithms are implemented in two research prototypes: the graph-specific *EmptyHeaded* [Abe+17] system and its generalization for linear algebra, BI, and graph queries, *LevelHeaded* [Abe+18]. LevelHeaded is discussed later in Sec. 10.6. Groundwork for applying WCOJs on relational and XML data was laid down in [Che18].

Applications in graph analytics It is interesting to note that *counting* and *enumerating triangles* are well-known challenges in graph analytics. In fact, LFTJ has been used for graph analytics in [Zin+16] which implemented an efficient *triangle enumeration* algorithm on both CPU and GPU computing platforms. In Sec. 3.8, we discussed this related result along with our linear algebra-based (but still suboptimal) approach to calculate the *local clustering coefficient*. These challenges and concepts have a profound relationship as we will discuss in Sec. 10.6.

[1] Note the similarity between the title of the papers – "Join Processing for Graph Patterns: An Old Dog with New Tricks" [Ngu+15a] and "Old Techniques for New Join Algorithms: A Case Study in RDF Processing" [Abe+16].

Ref.	Contributions	A/P	Bag	NF^2	Null	Agg.	Ord.	Gra.
[BLT86]	determining irrelevant updates, maintenance of select–project–join views	A+P	○	○	○	○	○	○
[QW91]	change propagation equations for relational algebra	A	○	○	○	○	○	○
[GMS93]	counting algorithm (non-recursive views), DRed algorithm (recursive views)	P	⊗	○	○	○	○	○
[Col+96]	change propagation equations for bag alg., incl. aggregation but no group-by	A	⊗	○	○	⊘	○	○
[Qua96]	group-by aggregations	P	⊗	○	⊗	⊗	○	○
[Kaw+97]	extending IVM techniques to maintain views defined over a nested data model	A	⊗	⊗	○	○	○	○
[MQM97]	group-by-aggregation, summary-deltas for representing changes	P	⊗	○	○	⊗	○	○
[GLT97]	improved change propagation equations for relational algebra, fixes [QW91]	A	○	○	○	○	○	○
[GK98]	change propagation equations for semijoins, antijoins and outer joins	A	○	○	⊗	○	○	○
[Don+99]	maintenance of transitive closure on directed graphs using a SQL-like language	A	⊗	⊗	○	⊗	○	○
[Pal+02]	maintenance of non-distributive aggregate functions	P	⊗	○	⊗	⊗	○	○
[LVM03]	incremental equations for the operators of the nested model	A	○	⊗	○	○	○	○
[DER03a]	order-preserving maintenance of XQuery views	A	⊗	⊗	○	⊘	⊗	○
[Yi+03]	IVM on top-k views	P	○	○	○	○	⊘	○
[GM06]	generalized summary-deltas, group-by-aggregations, outer joins	A	⊗	○	⊗	⊗	○	○
[LZ07a]	outer joins and aggregation, fixes [GM06]	P	⊗	○	⊗	⊗	○	○
[Vel13]	IVM for Leapfrog Triejoin, a worst-case optimal join algorithm	P	○	○	○	○	○	⊗
[Koc+14]	higher-order IVM, viewlet transformations, the DBToaster system	A	⊗	○	⊘	⊘	⊘	○
[IUV17]	Dynamic Yannakakis Algorithm for incremental evaluation of acyclic queries	A	⊗	○	○	⊗	○	○
[NO18]	factorized IVM	A	⊗	○	○	○	○	○
[Idr+18]	generalized Dynamic Yannakakis Algorithm	A	⊗	○	○	○	○	⊘
[Amm+18]	delta-generic join for subgraph matching, using WCOJs for cyclic queries	A	○	○	○	○	○	⊗
[Sza+18]	IVM for recursive aggregations	A+P	○	○	○	⊗	○	○

Table 10.1: Overview of related literature on IVM techniques, presented in order of appearance. Notation – *A/P*: algebraic/procedural; *NF^2:* non-first normal form relations, *Agg.:* aggregation, *Ord.:* ordering, *Gra.:* subgraph matching queries; ⊗ fully supported, ⊘ supported to some extent, ○ not supported.

10.3 Incremental View Maintenance Techniques

Due to the rich set of features required by property graph queries (as discussed in Sec. 8.1), an incremental query system on these queries needs to combine multiple incremental algorithms. Here, we give a brief overview of the literature on incremental view maintenance. Extensive surveys on IVM approaches were presented in paper [GM95], book [GM99], monograph [CY12], and tutorial [Elg+18].[2] According to the taxonomy of [CY12], our requirements given in Chapter 8 can be categorized as:

1. The *language* and *data model* are openCypher and *the property graph data model*, respectively.
2. The *information* used by the maintenance procedure is *not restricted*, i.e. base tables are accessible at any time.
3. The *timing* of the maintenance is *immediate*.

View maintenance techniques are also categorized according to how they calculate the differences between execution results. On the one hand, *algebraic* approaches take query Q and derive *delta expressions* (or *delta tables*) ΔQ and ∇Q that hold the positive and negative change sets, respectively. They evaluate these with the same query engine that calculate Q and apply their results (e.g. the maintained version of the query can be determined as $Q^m = Q - \nabla Q \cup \Delta Q$. On the other hand, *procedural* approaches can rely on custom code (such as stored procedures or other imperative methods) and auxiliary data structures to calculate the changes in the result set. Monograph [CY12] gives an excellent overview of the trade-offs of these approaches (emphasis ours):

> "The algebraic approach is attractive for several reasons. Besides being conceptually simple, it is modular and composable: change propagation equations are defined separately

[2] It is worth pointing out that even such comprehensive surveys do not cover challenges 1–3 and 8–10 raised in Sec. 8.1.

> for each operator in the view definition language, and together they can handle arbitrarily complex query expressions. In its purest form, this approach expresses all maintenance tasks using only the standard query operators, and hence can leverage existing query execution and optimization engines. However, in reality [...], *efficient implementation sometimes requires giving up the conceptual simplicity of the algebraic approach and specifying aspects of the maintenance in a more procedural manner. Therefore, the division between algebraic and procedural approaches toward specifying incremental view maintenance has been increasingly blurred.*"

The remark stating that *the division between algebraic and procedural techniques is not always clear* is confirmed by the lack of consistency demonstrated by papers categorizing IVM approaches. For example, the approach of [BLT86] is considered as algebraic in some works [CY12] and procedural in others [DER03a].

Procedural approaches Papers [Kel85; BLT86; GMS93] laid groundwork in the area of IVM and introduced *procedural approaches* for select–project–join (SPJ) views on a set-based relational algebra with some limitations. For example, the approach of paper [Kel85] only supports relations in Boyce-Codd Normal Form (BCNF). Incremental maintenance for non-distributive aggregate functions were discussed in [Pal+02]. Maintenance of top-k queries were discussed in [Yi+03].

Algebraic approaches A preliminary work on *algebraic recomputation* was presented in [QW91]. Its algorithm was fixed in [GLT97], co-authored by Griffin and Libkin, who also produced one of the seminal papers in the field that described IVM for multisets [GL95]. Later works added extensions to support additional operators:

- aggregations with group-by attributes [Qua96],
- semijoins/outer joins [GK98],
- ordering [LD00; DER03a; DER03b], and
- outer joins with aggregations [LZ07a] (using a *procedural* approach).

Semijoins, antijoins and outer joins The evaluation of *semijoin, antijoin, and outer join* expressions is particularly important in the context of graph queries. For example, if we would like to issue a query to "count all first- and second-degree friends of each Person", given relations $\mathsf{p}(\mathsf{id})$ and $\mathsf{kA} = \mathsf{kB} = \mathsf{knows}(\mathsf{p1}, \mathsf{p2})$, we would get the following complex expression:

$$\gamma^{\mathsf{p.id}}_{\mathsf{p.id},\texttt{count distinct}(\mathsf{kA.p2})\rightarrow\mathsf{firstDeg},\texttt{count distinct}(\mathsf{kB.p2})\rightarrow\mathsf{secondDeg}}\sigma_{\mathsf{p.id}\neq\mathsf{kB.p2}}\left(\mathsf{p} \underset{\mathsf{p.id=kA.p1}}{⟕} \mathsf{kA} \underset{\mathsf{kA.p2=kB.p1}}{⟕} \mathsf{kB}\right)$$

(Note that the selection operation is required to filter out the start person from second degree friends.) If person p returned by $\mathsf{kA.p1}$ does not have any first degree friends, the query produces tuple $\langle p, 0, 0\rangle$. If p has a friend, but the friend has no other friends, the query produces $\langle p, 1, 0\rangle$. Allowing NULL values is not strictly necessary (it is possible to formulate this query without the left outer join operator and therefore avoiding interim NULL values during the computation) such expressions are more cumbersome as they consist of the union of multiple expressions using antijoins. Therefore, the challenge of maintaining the results of antijoin-like operator is unavoidable and handling left outer join– or providing a way to immediately return a nested (possibly empty) collection for later aggregation – is a highly desirable feature in graph query processing.

There are a number of works on IVM for outer join expressions, but none were conceived in the context of graph queries. Looking at the history of the problem, the first *delta expressions* for outer joins were presented in [GK98]. Paper [GM06] proposed a more efficient approach, but unfortunately it turned out to be incorrect in some cases. Later, paper [LZ07a] defined a complicated algorithm that (1) rewrites the query expressions into *join-disjunctive normal form* [Gal94; LZ05; LZ07b], (2) creates a DAG of the relations involved in the expression, (3) computes the primary delta that should be added to the previous results, and (4) secondary the secondary delta (a set of expressions based on the primary delta) that should be removed from the results. This results in efficient and correct view maintenance, but also places some limitations on its expressions, e.g. it does not allow self joins in the expression (which is often required by graph queries) and also requires certain primary key–foreign key constraints to be held between relations. Paper [Nic12] lifted some of these limitations based on the implementation of the Sybase SQL Anywhere RDBMS. Combining outer joins with multiway, WOCJ algorithms, especially in the context of IVM is an open research area.

Papers [LZ05] and [LZ07b] target the loosely related *view matching* problem, which requires to select certain materialized views to speed up query processing. This introduces a complex problem as the query optimizer needs to determine whether a query or part of a query can be computed from existing materialized views."

Relationship of IVM and integrity constraint checking Paper [RSS96] established that *view maintenance* and *integrity constraint checking* are closely related issues, and that both can be performed more efficiently by using *space-time tradeoff*.

IVM on non-first normal form data Papers [Kaw+97] and [LVM99] discuss IVM techniques for nested relational algebra. An approach for incremental calculation of XQuery expressions is presented in [DER03a] and its accompanying technical report [DER03b]. These works present an *order-preserving* technique to perform IVM over XML data structures, which can prove useful when representing ordered data sets (such as paths in graph queries).

Schema inferencing algorithm The most similar approach to our *schema inferencing* algorithm (Alg. 1 in Sec. 8.4.3) is the *schema cleanup* algorithm [ZPR02], which is defined in the context of evaluating XQuery expressions on XML documents. It assumes a-priori knowledge of the schema and aims to remove as many attributes as possible from the inputs of the query. Another loosely related work is the *schema merging* algorithm of [Li+11], defined for consolidating multiple schemas into a mediated one. Neither of these algorithms aim to determine the schema of relational algebraic operators based exclusively on the query specifications. Papers on *schema-free XQuery* [LYJ08] and *schema-free SQL* [LPJ14] present approaches that allow users to define queries using a partial XML or SQL database schema.

DBToaster The DBToaster project [Koc+14] introduced the *viewlet transforms* technique which allows efficient incremental maintenance. Instead of using separate (multi)sets for positive and negative changesets, DBToaster employs *generalized multiset relations* (GMRs), which allow negative multiplicities and even rational (i.e. non-integer) numbers. This allows compact representation of the delta queries, which gives way to using *higher-order derivation* that repeatedly calculates deltas on queries. DBToaster is available as an open-source project[3] and supports a considerable subset of the SQL language, including nested subqueries. Some of its authors also contributed towards IVM on analytical,

[3] https://dbtoaster.github.io/

linear algebra-based queries [NEK14], optimization of incremental queries [TK15], and IVM on collections [KLT16]. Another recent followup of the DBToaster project is F-IVM, "a unified incremental maintenance approach for analytics over normalized databases" [NO18].

IVM on object-oriented data The problem of defining views for OODBMSs was studied in [Ber92], while the semantics of update operations were discussed in [And+92]. An active OODBMS, *REACH*, built on top of Texas Instruments' Open OODB Toolkit was introduced in [Buc+95]. REACH used a rule-based approach, but did not support a declarative query language and the paper does not discuss transitive reachability or path queries. An approach for continuously checking integrity constraints defined in the PROGRES environment [SWZ99] and supported by the underlying GRAS OODBMSs was presented in [MSW98].

An important feature of OODBMSs is their focus on *object identity* [AK89], which requires each object to have an existence which is independent of its value [Atk+89]. This feature has profound consequences when defining views: approaches that define the view as an object graph – instead of a relation – need to overcome the problem of providing *identities* to the created object nodes. This challenge currently does not affect openCypher-based implementations, but will be relevant in the context of later versions and other PG query languages such as G-CORE (see Sec. 2.6.3). The authors of [KR98] presented IVM techniques on top of their *MultiView* OODBMS [KR96]. Papers [LVM00; LVM03] demonstrated methods for IVM on object-oriented databases, based on the previous work of the authors in IVM on nested data structures [LVM99].

IVM on non-relational data The dominance of relational databases in the last decades significantly influenced the development of IVM techniques. Consequently, most research papers on IVM considered the relational data model. Still, there are some important works that target IVM on non-relational data. IVM techniques for semistructured data were studied as early as 1998 in paper [Abi+98], which conluded that "view maintenance in a graph-based data model such as OEM [Object Exchange Model] is fundamentally more difficult than in the relational model". GraphIVM, a system for IVM on graphs that uses non-relational caching techniques, was presented in a Master's thesis work [Sax15]. Paper [Zha+17] targeted IVM on array data for scientific computations.

Incrementalized WCOJ algorithms An incremental version of LFTJ, implemented in the LogicBlox system [Are+15], was described in preprint [Vel13]. The *delta generic join* algorithm with stronger theoretical guarantees was presented in paper [Amm+18], along with an implementation on top of *timely dataflow* [Mur+13; Mur+16], a framework for defining iterative and incremental computations. Another recent result in the *Dynamic Yannakakis Algorithm* (Dyn) [IUV17], an incremental version of the well-known *Yannakakis algorithm* [Yan81]. Dyn was generalized by extending the notion of acyclicity and allowing θ-joins in [Idr+18].

Recursive aggregations Paper [Sza+18] introduced a novel algorithm that performs IVM for Datalog queries defining recursive aggregation. The paper used a *static code analysis* use case as a motivating example for evaluating the algorithm.

Rule-based expert systems IVM has been used extensively in the context of *rule-based expert systems* (also known as *production systems*), which support similar features with their *discrimination networks* [BG15]. Notable approaches include Rete [For82] and TREAT [ML91]. In expert systems, users formulate *rules* (or *productions*), which have a *left-hand side* (LHS) and a *right-hand side* (RHS).

As described in [ML91], a *rule engine* (or *production system interpreter*) repeatedly executes a cycle of three operations:

1. *Match:* enumerate working memory element subsets (*instantiations*) that satisfy an LHS and collect them to a *conflict set*.
2. *Conflict set resolution:* select an element from the instantiations.
3. *Act:* execute the actions in the RHS of the selected instantiation.

Note that the "match" step is very similar to performing IVM for the LHS on the working memory elements. However, most IVM use cases define read-only queries for continuous evaluation and do not require the system to execute actions based on the set of matches. Hence, production system algorithm accomplish a more complex task than regular IVM. For this reason, we do not discuss these algorithms in detail, but include a few reference for completeness. The lazy evaluation-based LEAPS algorithm was presented in [MB90; Bat94]. A performance comparison of the Rete and TREAT algorithms was given in [WH92]. The Rete, TREAT and LEAPS algorithms were compared in [Bra+91], concluding that TREAT is favourable in many cases, but noting that it is more difficult to implement. Gator networks (**g**ener**a**lized **T**REAT **o**r **R**ete) were discussed in [HBC02]. Another bridge between the world of expert and database systems was laid down in [BW94], which presented an *algebraic approach* to rule analysis in expert database systems. A heavily modified version of the Rete algorithm, named PHREAK,[4] is used in the Drools[5] [Pro11] rule-based expert system. A GPU-based parallel rule-based reasoner based on the Rete algorithm was presented in [Pet+14], while paper [JO18] adapted the Rete algorithm to the Apache Spark [Zah+10] distributed data processing framework for reasoning on RDFS [GB14b] data structures.

Summary of approaches Tab. 10.1 shows an overview of IVM techniques and their applicability to bags, NF^2 data, null values, complex aggregations, ordering, and support for subgraph matching-style queries, along with their categorization to algebraic/procedural.

Further reading For further reading, we collected PhD dissertations dedicated to the problem of improving query performance. The Rete and TREAT algorithms were published in the dissertations of Charles Forgy [For79] and Daniel Miranker [Mir90], respectively. Robert B. Doorenbos presented optimization techniques for the Rete algorithm [Doo95]. Jose-Luis Ambite showed an approach to optimize queries with graph rewriting techniques [Amb98]. Gergely Varró designed efficient search-based query algorithms on graph models [Var08], while Gábor Bergmann adapted the Rete algorithm to EMF models [Ber13]. Andrey Gubichev's work investigates both graph query optimization and efficient evaluation techniques [Gub15]. Recent results in *higher-order incremental view maintenance* on SQL queries were presented by Milos Nikolic in [Nik16].

10.4 Graph Query Languages for IVM

A recent survey [Ang+17] presents an overview of modern graph query languages. It discusses popular data models, defines two categories of query functionalities (graph patterns and navigational expressions) and presents important concepts such as *matching semantics*. According to this categorization, our work focuses on *graph patterns*. In the following, we discuss languages for graph pattern matching and mainly focus on implementations that provide (some degree of) incremental view maintenance for queries in the language. For more details on graph queries, see also Sec. 2.6.

[4] https://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html_single/#PHREAK
[5] http://drools.org/

10.4.1 Cypher and openCypher

Early attemps to formalize the Cypher language were presented in [HG16] and [c6], which use an extended relational graph algebra to capture the semantics of the language. In [Jun+17], graph queries were defined in a Cypher-like language and evaluated over Apache Flink-based Gradoop framework using relational operators. However, the formalization of the language was not discussed in detail. A formal semantic definition of the core read-query features of Cypher was presented in [Fra+18], co-authored by several members of Neo4j's openCypher working group. A summary on openCypher's current state and future roadmap was published in a tutorial [Gre+18].

Non-incremental implementations The Cypher language is originated in the Neo4j graph database, which introduced the language in 2013.[6] Cypher is also supported by the AgensGraph[7] and SAP HANA Graph [Rud+13] systems, although neither has full feature coverage. The Cypher for Apache Spark project[8] maps openCypher queries to the Spark DataFrame API and the Catalyst optimizer (see also Sec. 10.5). The Cypher for Gremlin project[9] translates openCypher queries to the Gremlin language [Rod15].

Incremental implementations *Graphflow*[10] [Kan+17] is an incremental (or *active*) openCypher database. As such, it bears the closest similarity to our approach. Its Cypher++ language extends Cypher with *user-defined functions* that trigger on new matches. It uses the delta generic join algorithm [Amm+18] (Sec. 10.3), an incrementalized version of generic join, a worst-case optimal join algorithm. However, Graphflow lacks support for some language features such as negative/optional edges and transitive closures.

10.4.2 G-CORE

G-CORE [Ang+18] is a *design language* created by the Linked Data Benchmark Council's (LDBC) Graph Query Language task force. According to its author, G-CORE does not strive to be a standard, and instead aims to "guide the evolution of both existing and future graph query languages". The language was designed to fulfil two key characteristics, lacking from popular graph query languages available at the time: (1) allow composition of queries and (2) treat paths as first class citizens. To these ends, (1) G-CORE queries return graphs as their results (which allows composability), and (2) they are defined over the *path property graph data model*, which handles paths as part of the graph with their own labels and properties.

Implementations G-CORE currently has a proof-of-concept reference implementation[11] [Cio18]. This transpiles G-CORE queries to Apache Spark[12] [Zah+10], using Spark SQL [Arm+15] and GraphX [Xin+13] to evaluate the queries (see also Sec. 10.5). The goal of this implementation is not to provide high-performance, instead it aims to discover ambiguities in the specification and discover possible points for improving the language. Additionally to the reference implementation, the

[6] https://neo4j.com/
[7] http://bitnine.net/agensgraph/
[8] https://github.com/opencypher/cypher-for-apache-spark
[9] https://github.com/opencypher/cypher-for-gremlin
[10] http://graphflow.io/
[11] https://github.com/ldbc/gcore-spark
[12] https://spark.apache.org/

G-CORE grammar specification and a parser for the language are also available as open-source software.[13]

10.4.3 SPARQL

Of the existing practical graph query languages, SPARQL is best understood in terms of semantics and complexity [PAG09], hence it is worth studying its implementations for further research on IVM support for existing graph query languages.

Implementations *Trinity* is a graph engine operating on top of a *memory cloud*. The latest publication on Trinity is [Sha+17], which provides a high-level overview and details some use cases. The detailed architecture of the system and some performance experiments for graph algorithms are presented in [SWL13], while the Trinity.RDF system, *graph exploration algorithms* and their performance for evaluating SPARQL queries are discussed in [Zen+13]. The core of Trinity is available open-source as the *Microsoft Graph Engine*.[14]

Diamond [Mir+12] uses the Rete algorithm to evaluate SPARQL queries on *distributed RDF data*. During the evaluation of a query, it identifies additional pieces of data by dereferencing URLs and turning to remote servers. Newly introduced data elements are fed into the Rete network of the algorithm as (positive) updates. The implementation of the approach was not published.

Instans[15] [RNT12] uses the Rete algorithm to perform *complex event processing* on streaming RDF data. Instans is a research prototype implemented in LISP. *Strider*[16] [Ren+17] is a recently developed research prototype, which supporting continuous SPARQL queries on top of Apache Spark.

G-SPARQL G-SPARQL [SEH12] adds *attribute handling* to the SPARQL language, resulting in a language that is has an expressive power very similar to property graph query languages. While the G-SPARQL paper contains some experimental results, suggesting that the language had a working prototype, the implementation was not released and work on the language seems discontinued.

Streaming SPARQL Semantics of the *Continuous Query Language* (CQL) were studied in [ABW06]. Between 2010 and 2011, numerous languages were proposed to define streaming queries on SPARQL, including C-SPARQL [Bar+10], $\text{SPARQL}_{\text{Stream}}$ [CCG10], and CQUELS [Phu+11].

10.4.4 VIATRA Query Language

Graph pattern matching has been used extensively in the domain of model-driven engineering. We introduced the Viatra Query Language (VQL) in Sec. 2.6.5.

Implementations The Viatra Query project fully implements VQL and supports both one-time [Búr+15] and incremental query processing, primarily targeting EMF models (Sec. 2.3.1). The IncQuery-D[17] project [c4] (Chapter 9) is a *distributed* incremental graph query engine, which uses a

[13] https://github.com/ldbc/ldbc_gcore_parser
[14] https://github.com/Microsoft/GraphEngine
[15] https://github.com/aaltodsg/instans/
[16] https://github.com/renxiangnan/strider
[17] https://github.com/viatra/incqueryd

query language based on the Viatra Query Language and operates on RDF graphs. Viatra also supports *complex event processing* (CEP), i.e. defining temporal conditions for queries on continuously changing graphs [DRV18].

10.4.5 Other Approaches for Graph Querying

To conclude the discussion on related graph languages, we list some notable languages which take a different approach for formulating graph queries. As neither of these languages are closely related to the challenges proposed in this dissertation, we only give a brief summary for them.

GraphQL *GraphQL* is a language for querying property graphs, originally proposed by Facebook.[18] The first in-depth study on its semantics and complexity was published in [HP18]. Despite its name, GraphQL is not a fully-featured graph query language as it lacks many basic constructs such as transitive paths.

Gremlin *Gremlin* [Rod15] is a high-level imperative domain-specific language (DSL) based on the dynamically typed Groovy language [Kni+15]. It allows users to formulate graph queries and algorithms using *traversal steps*, relying on *function composition* and *lazy evaluation* for performing the computations. Recent versions of Gremlin facilitate rewriting rules for optimization and support basic *pattern matching steps*, which can evaluate graph patterns formulated in a declarative style. The *Gremlin language* should not be confused with the *Gremlin Structure API* of the *TinkerPop* v3 framework,[19] which is a low-level programming interface, formerly known as *Blueprints* in *TinkerPop* v2.[20]

Triple Graph Grammars *Triple Graph Grammars* (TGGs) [SK08] define transformations from *source model* to a *target model* through a *correspondence model*. In a forward transformation scenario, TGGs implement IVM as changes in the source model are incrementally reflected in the target model. However, TGG implementations [JKS06; SK08; Anj+14] focus on *bidirectional transformations* and therefore offer a limited set of rule, only supporting restricted *negative application conditions* (i.e. antijoins) and no aggregations.

10.5 Mapping Between Graph and Relational Queries

A line of research closely related to our work considers *query transpilers*. These translate from a given query language to another one, often simultaneously bridging the gap between different data models. In most approaches, they transpile from a convenient and expressive language to a more mature language with stable and efficient implementations. In this section, we investigate approaches that translate between a graph query language (Cypher or SPARQL) and SQL.

Query transpilers consist of two key components:

1. the *schema mapping* that defines how to interpret the relational schema in terms of graph concepts (e.g. how to interpret tables as nodes and edges) and
2. the *query mapping*, which defines the translation between the *source language* and the *target language*.

[18] https://facebook.github.io/graphql/October2016/
[19] https://tinkerpop.apache.org/docs/3.3.4/reference/
[20] https://github.com/tinkerpop/blueprints/

Tool	Schema mapping	Query mapping	
		Source language	Target language
Entity Framework [Bla+06]	Entity Data Model	OO operations	SQL
Hibernate [ONe08]	Java annotations	OO operations	SQL
VIATRA prototype [BHH12]	manual	VQL	MySQL
SQLGraph [Sun+15a]	generic schema	Gremlin	SQL
Cypher for Apache Spark	SQL Graph DDL	Cypher	Spark DataFrames
Cytosm [Ste+17]	gTop	Cypher	Vertica SQL
RDB-RDF views [Vid+17]	manual	manual	PostgreSQL
GraphGen [XSD17]	GraphGenDL	GraphGenQL	PostgreSQL or SQLite
G-CORE interpreter [Cio18]	graph schema	G-CORE	Spark SQL and GraphX
R2D [Ram+09]	RDF-to-Rel. mapping	SQL	SPARQL
R2G [VMT14]	Data Mapper	PostgreSQL	Gremlin

Table 10.2: Tools for mapping between graph/relational schema and queries. The first group of tools maps from a graph model to the relational model, while the seconds group does maps in the opposite direction.

The tools with *SQL as the target language* aim to exploit the maturity and performance of relational databases, along with the large amount of legacy data stored in such systems. Meanwhile, tools using *SQL as the source language* rely on the existing tools that produce SQL queries (e.g. visualization frameworks that generate SQL queries for importing data), and execute them on graph-based systems. A summary of query transpilers is shown in Tab. 10.2.

10.5.1 Mapping from Graph Queries to Relational Queries

Object-relational mapping *Entity Framework* [Bla+06] and *Hibernate* [ONe08] are widely used frameworks for object-relational mapping (ORM). Both provide a mapping from an object-oriented data model (.NET or Java objects, respectively) to relational tables. Navigating between objects can be interpreted as graph traversal and objects themselves are similar to the nodes of the property graph data model. Hence, they share many common challenges with the *graph-to-relational mapping* problem, but also lack some, e.g. calculating shortest paths.

VIATRA SQL mapping prototype Paper [BHH12] presented an approach in the context of model-driven engineering with the goal of continuously evaluating graph patterns in a relational database. The patterns, formulated in the VIATRA Query Language (Sec. 2.6.5), were transpiled to SQL queries. These were then maintained using database triggers.

SQLGraph SQLGraph [Sun+15a][21] compiles Gremlin queries to SQL queries. In their approach, the *adjacency information* (nodes and edges) is stored in a relational database, while properties are stored in a JSON store.

Cypher for Apache Spark The *Cypher for Apache Spark* project is discussed in Sec. 10.4.1.

[21] SQLGraph should not be confused with GraphSQL, the predecessor of the TigerGraph database, see http://www.zdnet.com/article/tigergraph-a-graph-database-born-to-roar/

Cytosm *Cytosm* [Ste+17] is an approach for **Cy**pher **to S**QL **m**apping, targeting the Vertica system. Cytosm defines *gTop*, a JSON-based language to define the mapping between the **g**raph **top**ology and the relational schema. It preprocesses queries using the *Pathfinder* module, which uses the gTop schema to narrow the search scope (e.g. if a node in the query references a property firstName, it can infer that the node is of type Person if no other node labels are associated with this property) and expands variable-length paths into the union of fixed length paths. For the actual transpilation, it uses the *OpenCypher to SQL converter* module, which builds an abstract syntax tree (AST) from the Cypher query. It then applies transformations on the tree to get the AST of the SQL query and generates the textual query specification. Preliminary benchmark results – based on two queries from the LDBC Social Network Benchmark's Interactive workload [Erl+15] – have demonstrated good performance compared to the graph available at the time of the experiments. However, the Cytosm project has been abandoned since its publication.

RDB-RDF views The maintenance of *RDF views* on top of RDBMSs is presented in [Vid+17].

GraphGen GraphGen is a tool for compiling graph queries to relational ones. Their authors define two Datalog-based languages: GraphGenDL for defining *graph views* and GraphGenQL for querying those views. Their paper [XSD17] also presents an overview of possible approaches of combining graph and relational systems. Another recent paper by the authors of GrapGen discusses how to extract hidden graphs from relational databases [XD17], similarly to GraphMapper [KPT16] which also targets JSON, XML, and CSV datasets.

G-CORE interpreter The *G-CORE interpreter* project is discussed in Sec. 2.6.3.

10.5.2 Mapping from Relational Queries to Graph Queries

Some works consider mapping from relational queries to graph queries, targeting either the RDF or the property graph data model.

R2D R2D (RDF-to-Database) [Ram+09] converts *RDF to relational structures* by extracting relational structures from RDF stores and presenting them as views. The schema mapping is defined by *RDF-to-Relational mapping files*, while the queries are transpiled from SQL to SPARQL.

R2G R2G [VMT13; VMT14] is a tool for migrating *relations to graphs*, including the mapping of SQL queries to graph queries. It has four key components and works as follows: (1) the Metadata Analyzer inspects the relational schema and tries to find graph structures. Based on the extracted structures, (2) the Data Mapper maps the relational schema to a graph schema (formulated using Tinkerpop Blueprints), and (3) the Query Mapper maps graph queries from SQL to the Tinkerpop Gremlin v2 language. Finally, (4) the Graph Manager executes queries over the target graph using the mappings generated by (2) and (3).

10.6 Connecting Joins, Matrix Multiplications, and Graph Queries

While linear algebra is often used for data analytics, query processing and linear algebra have been traditionally treated as different aspects of data processing. However, it is easy to see that there are important similarities between the two: joins are analogous to matrix multiplication operations, while

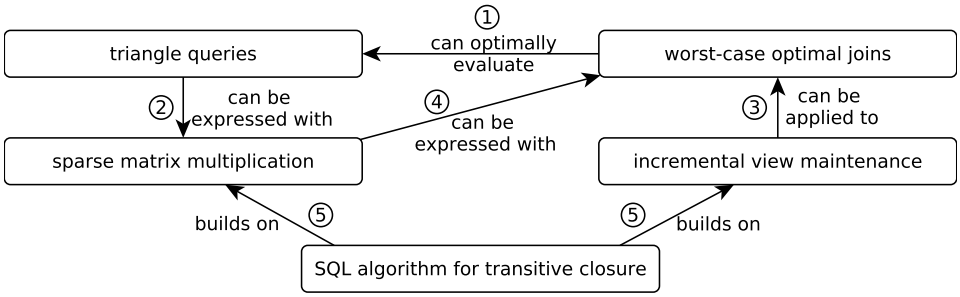


Figure 10.1: Relationships between concepts in linear algebra, graph analytics, and query processing.

global aggregations can be expressed as row- or column-wise summarizations. A well-studied example to highlight this similarity is the *matrix multiplication query*, which calculates the product of matrices A and B represented in sparse format with their non-zero elements represented in tables `A(i, j, v)` and `B(j, k, v)`:

```
1 SELECT A.i, B.k, SUM(A.v*B.v)
2 FROM A, B
3 WHERE A.j = B.j
4 GROUP BY A.i, B.k
```

Theoretically, this query presents an important bridge between the fields of query processing and linear algebraic (LA) computations. Its complexity and implications have been discussed in [KSS18]. Very recently, paper [Abe+18] presented the *LevelHeaded* system, which can be viewed as a generalization of the authors' previous *EmptyHeaded* system [Abe+17]. Compared to its predecessor, *LevelHeaded* does not focus solely on graph processing, but instead unifies the evaluation of complex BI-style database queries and LA computations using a *worst-case optimal join algorithm*. The paper uses the *matrix multiplication query* extensively in its examples.

While the matrix multiplication query might seem very academic, it has important practical applications, e.g. it is one of the recommended approaches in book "SQL Design Patterns" for expressing *reachability queries* in graphs [TB07, Chapter 6: Graphs]. In this book, the authors use adjacency matrices (stored in the sparse representation presented above) and adapt the incremental evaluation techniques of papers [Don+99; DS00b] to determine the transitive closure on the input graph.

Relationships between concepts Fig. 10.1 shows the relationships between concepts in linear algebra, graph analytics, and query processing. In Sec. 3.8, we showed that *triangle queries* can be succinctly expressed with *sparse matrix multiplication* ① and cited an approach [Zin+16] that uses the *worst-case optimal* Leapfrog Triejoin algorithm to enumerate triangles efficiently ②. We gave a short overview of *worst-case optimal joins* in Sec. 10.2 and presented ongoing work to apply them for IVM in Sec. 10.3 ③. Finally, in this section, we presented recent findings that apply *worst-case optimal joins* to express linear algebraic operations such as *sparse matrix multiplication* [Abe+18] ④, along with an algorithm that uses *sparse matrix multiplication* and IVM to calculate transitive closure in relational databases [TB07, Chapter 6: Graphs] ⑤.

Chapter 11

Conclusion and Future Work

In this chapter, I summarize the key contributions and highlight potential future research directions.

11.1 Summary of Contributions

11.1.1 Structural Analysis of Typed Graphs

How does a network emerge and what patterns does its graph exhibit? These are central questions of network science that have been studied to great depth during the last few decades. However, these works almost exclusively studied homogeneous networks, i.e. networks of a single node label and a single edge type, which can be modelled with an untyped graph. While this approach yielded groundbreaking results in many areas (such as the analysis of biological and social networks), it is difficult to apply in fields which use heterogeneous graphs with edge types (also known as *multiplex networks*, see Sec. 2.2). Types introduce even more variety in the *structural interplay* between the nodes and edges of the graph. On a fine-grained level, this could mean investigating how likely it is that two friends of a given person engage in common activities. At a higher granularity, it would be interesting to observe how communities are formed based on such heterogeneous triangles. This example shows how in-depth structural analyses that take the types into account could lead to a better understanding of *the emergence of both the microscopic and the macroscopic structure of typed graphs.*

Creating synthetic graph instances is an important research challenge that has a variety of use cases, ranging from producing customer data to self-driving autonomous vehicles. To implement such *graph generators*, we need a basic understanding of how the target graphs are structured. This allows us to guide the generator so that the synthesized graph satisfies the desired structural properties.

Group of contributions 1 I proposed various graph metrics and statistical analysis techniques to characterize domain-specific engineering models and graph generators.

1.1 **Multidisciplinary graph metrics.** I adapted graph metrics originally proposed in network science to describe typed graphs of systems engineering models. [c5]

1.2 **Characterization of engineering models.** I proposed characteristic graph metrics to distinguish graph models from different domains using statistical analysis. [c5]

1.3 **Characterization of synthetic graphs.** I identified graph metrics to characterize synthetic graphs derived by various graph generators in order to distinguish them from real graphs. [b1]

Added value In this work, I proposed a domain-agnostic extendable method to characterize graph models, which can be applied on any graph model. These metrics allow users to estimate the workload of real application scenarios by providing a toolkit that characterizes the complex structure of graphs without revealing detailed and sensitive information, thus respecting intellectual property rights (IPR). Finally, I showed that these metrics can be used to differentiate between *real* and *synthetically generated* graph instances, thereby highlighting the limitations of existing state-of-the-art graph generation approaches.

Previous and related results in the research group Related graph and query metrics were used in [Izs+13b] to characterize and predict performance of graph queries to assess how much worse a query engine performs compared to a theoretical lower bound. A key difference of my work is to characterize domain-specific graph models and model generators (and not the queries themselves), which is an orthogonal use of graph metrics. Related graph metrics have been evaluated for generic graph models in order to study their effect on query performance by Zsolt Kővári in [Kőv15], which was co-supervised by István Ráth and myself. Zsolt Kővári and Ágnes Salánki contributed the statistical analysis of graph models in our joint paper [c5]. Oszkár Semeráth implemented the model generator to produce synthetic statechart models used in our joint paper [b1].

Open-source software The framework for analyzing typed graphs is available as an open-source project[1]. The project consists of approx. 5 000 lines of Java code for loading and calculating graph metrics, along with 500 lines of R code for visualization and data analysis. Additionally, a less scalable but more visual demo implementation, using the Neo4j graph database [Web12] and its Cypher query language [Fra+18], is available online.[2]

11.1.2 Benchmarks for Global Queries over Evolving Property Graphs

Standard benchmarks specify a workload and inspect the behaviour of various implementations executing it, measuring a set of metrics (response time, energy consumption, correctness, etc.). It is established that benchmarks have the power to shape a field [Pat12] and accelerate its progress, particularly for new and emerging areas, which are yet to establish a common understanding of what the focal points of the field should be. In such cases, a de-facto benchmark accepted by members of the community allows competing tools to be compared using a standard and precisely specified workload. Benchmarks which sufficiently cover important features also free authors from the burden of designing and implementing their own ad-hoc benchmarks (which might be biased towards their tools), and aid the reproducibility of their performance experiments.

Group of contributions 2 I contributed to the design of two benchmark frameworks for global property graph queries. My specific contributions include categorization of query language features, numerous enhancements to the benchmarks as well as experimental evaluation.

2.1 **Expressivity requirements for query languages.** I identified abstract and qualitative language features (choke points) to systematically assess the expressivity requirements for graph queries used in performance benchmarks. [l18]

2.2 **Adaptation of benchmarks for the property graph data model.** I adapted two open benchmarks, the Train Benchmark and the LDBC Social Network Benchmark, for property

[1] https://github.com/ftsrg/graph-analyzer
[2] https://neo4j.com/graphgist/multidimensional-graph-metrics-with-neo4j-and-cypher-2

graphs, including the implementation of scalable data generators, novel global queries, and deterministic update transformations. [j2; l18]

2.3 **Extensive experiments with benchmarks.** I conducted an experimental evaluation and exploratory analysis of different query technologies over static RDF and evolving property graphs under various workloads. [j2; c8; l18]

Added value Up to our best knowledge, both benchmarks target a unique workload. The Train Benchmark is first cross-technology benchmark that measures the performance of repeated model validation operations, considering representation formats, query languages and query engines from multiple technological spaces (EMF, RDF, property graph, and SQL). The LDBC SNB's Business Intelligence workload is the first benchmark for *decision support-style, aggregation-dominated global graph queries*, which, through systematic choke point-driven design process, covers challenging features both language- and performance-wise.

Previous and related results in the research group Benchmarking of graph-based query and transformation tools has been in the focus of the research group and investigated in a series of papers such as [VSV05; Hor+10; Ujh+15a; Ujh+15b].

The design of the first version of the Train Benchmark was led by István Ráth with major contributions Benedek Izsó, Balázs Polgár, Zoltán Szatmári, Gábor Bergmann, and Ákos Horváth [Izs+13b; Ujh+15a; Sza17]. Follow-up versions of the benchmark [l10; c4; l12; j2] were joint work with Benedek Izsó, István Ráth, Oszkár Semeráth, Gábor Bergmann, and Dénes Harmath. Since 2012, I contributed to all major components of the benchmark. My contribution statements presented above exclude inseparable joint contributions and claim results in the context of property graphs where my own work was predominant. In case of the LDBC Social Network Benchmark, I extended existing update operations to include removal of elements, which is a major challenge for incremental query evaluation engines, and also resolved a number of ambiguities in the benchmark specification.

Open-source software The Train Benchmark is available as a single open-source project.[3] It contains approx. 18 000 lines of Java code, 800 lines of Groovy code and 600 lines of R code. The LDBC Social Network Benchmark consists of multiple projects with contributions from dozens of authors. Most of my work manifested in the *specification*,[4] the *workload driver*,[5] and the *reference implementations* project[6], where I performed extensive refactoring and added a number of new query implementations. My contributions in the latter account for approx. 4 000 lines of Java code and 800 lines of Cypher code. I also performed various maintenance tasks in the data generator.[7]

11.1.3 Incremental View Maintenance on Schema-Optional Property Graphs

Efficient query evaluation – incorporating many techniques from indexing through query optimization to join processing algorithms – is one of the holy grails of database research. In many workloads, especially in OLAP-style data processing, queries are known in advance and are evaluated against a continuously changing data set. In these cases, query processing can be sped up by defining *materialized views* and using *incremental view maintenance* (IVM) to keep their content in sync with the

[3] https://github.com/ftsrg/trainbenchmark
[4] https://github.com/ldbc/ldbc_snb_docs
[5] https://github.com/ldbc/ldbc_snb_driver
[6] https://github.com/ldbc/ldbc_snb_implementations/
[7] https://github.com/ldbc/ldbc_snb_datagen

changes in the data. Today, IVM is supported by many (mostly commercial) relational database systems, and would certainly be of interest in the graph processing space. However, while IVM has more than three decades of literature, most works only considered the relational data model, and only a few proposed IVM algorithms for the nested and graphs data models. Additionally, the *schema-optional* approach of property graph databases and the bottleneck of single node systems make the application of IVM even more challenging.

Group of contributions 3 I developed scalable incremental view maintenance techniques for evolving schema-optional property graphs.

3.1 **Schema inferencing from property graph queries.** I designed a schema inferencing algorithm from nested to flat relational algebra that deduces the relevant minimal schema of the property graph from query specifications. [l17; r21]

3.2 **Mapping from property graph queries to nested relational algebra.** I defined a chain of consecutive mappings from high-level property graph query languages to nested and flat relational algebra to enable the use of traditional relational query evaluation and optimization techniques. [c6; r21]

3.3 **Parallel query processing for incremental view maintenance.** I proposed an asynchronous graph query technique for incremental view maintenance that uses Rete-based query evaluation of flat relational algebra over schema-optional property graphs. [l17; r21]

3.4 **Scalable distributed query evaluation.** I proposed an asynchronous execution strategy and a termination protocol by combining the actor model with Rete-based query evaluation for scalable incremental view maintenance on RDF graphs. [l9; c4; l14]

3.5 **Experimental evaluation of incremental graph query processing.** I evaluated the performance and scalability of a prototype implementation of for parallel execution (in-graph) using the LDBC Social Network Benchmark. I adapted the Train Benchmark for a cloud-based execution environment to carry out scalability evaluation of a prototype implementation for the distributed approach (IncQuery-D). [c4; r21]

Added value The compilation and transformations steps presented in this thesis can be used to reduce a large set of property graph queries to flat relational algebra and thus allows query engine developers to apply existing IVM techniques for these queries. As IVM incurs a significant memory overhead, I also studied the possibility of using a distributed setup to improve the scalability of the approach for large graphs. I believe one of the most future-proof contributions of this thesis is the identification of the required challenges to adapt IVM techniques for property graph queries. While some of these challenges were already addressed in this dissertation, and some can be tackled by adapting existing solutions, many of them are open research problems.

Previous and related results in the research group Gergely Varró designed efficient *search-based graph query algorithms* on graph models [Var08] which was further extended by Ákos Horváth for hybrid search plans [Hor+10] and to EMF models by the development team of the eMoflon and Viatra tools in [Var+15; Búr+15]. Gábor Bergmann's work discussed *incremental view maintenance on EMF models* [Ber13] and proposed an initial version for a custom parallelization of Rete network [BRV09], which is generalized in my work to a distributed actor-based environment. István Ráth presented *event-driven and incremental model transformation techniques* in his dissertation [Rát11].

Furthermore, he was the main contributor of the IncQuery-D architecture [c4]. József Makai contributed allocation and reconfiguration strategies for IncQuery-D [l14]. József Marton defined a formalization of the openCypher language [c6]. János Maginecz made major contributions to the ingraph prototype, including various optimization techniques and significant implementation efforts [l15; j3; r21]. Csaba Debreceni proposed novel incremental synchronization for secure *view models* to support collaborative development [Deb19].

Open-source software IncQuery-D[8] [l9; c4; l14; l15] was implemented in Java and Scala, using the compiler of Viatra Query. Its *distributed and sharded* extension, IncQuery-DS (implemented purely in Scala) is available as a separate project[9] [l15]. ingraph [j3; c6; c7; l17; r21] is a mix of Scala and Java code, where I contributed to all components, including the compiler, the data loader and the incremental query engine.[10]

Related software projects The *IncQuery Server for Teamwork Cloud* [Heg+18] is a server-side query middleware service for collaborative modelling. Like IncQuery-D, it supports distributed processing, but on a different (microservice) level of granularity. Furthermore, it is also based on the reactive programming paradigm, but uses the Eclipse Vert.x library[11] instead of Akka actors used in IncQuery-D.

11.2 Open Challenges and Future Work

We believe that the work presented so far opens up a number of interesting research directions. Here, we highlight key directions for future research.

Meta-paths We plan to adapt state-of-the-art graph analysis approaches such as *meta paths* [Shi+17], and to apply proven metrics such as the *k-local clustering coefficient* to typed graphs [JC04; Fro+02]. To improve the scalability of analysis workloads, we already reformulated multiple challenging metrics in the language of linear algebra [Vár18] and plan to experiment with them on large data sets. We are also looking into other means to improve scalability, including recent advancements in *elastic graph processing* [Uta+18].

Graph synthesis We plan to combine the *characteristic graph metrics* identified in Part I with the work on diverse and consistent graph generation by Oszkár Semeráth [Sem19] to *synthesize consistent, realistic, diverse, and scalable graph models.*

Modification workload for LDBC We are currently working on extending the LDBC SNB's Business Intelligence workload with a set of modifications that contain *insertion*, *deletion*, and *property update* operations. These will not only test the performance of the modification operations themselves but also stress *incremental view maintenance* approaches (which are more difficult to adapt for modifications involving delete operations).

[8] https://github.com/viatra/incqueryd
[9] https://github.com/viatra/incquery-ds
[10] https://github.com/ftsrg/ingraph
[11] https://vertx.io/

Support for new languages As new graph languages such as the Linked Data Benchmark Council's G-CORE [Ang+18], Cypher 10 [Fra+18], and the work-in-progress GQL (Graph Query Language)[12] emerge, we will continuously investigate how *incremental view maintenance* techniques can support to them. We also plan to experiment with adapting techniques to design a more efficient *incremental transitive closure* algorithm [Rod08a].

Graph-specific IVM techniques While we addressed some challenges regarding incremental view maintenance, there are a number of complex open research questions. We believe the major challenges in incremental view maintenance for property graphs as follows (in increasing order of difficulty):

1. Outer join queries (including antijoins) on graph data structures.
2. Cyclic queries on graphs with skewed data distribution.
3. Lists and ordering.
4. Traversals (also in increasing order of difficulty):
    a) Transitive reachability queries.
    b) Recursive queries.
    c) Path unwinding and higher-order queries.

We expect many of these problems to receive significant interest from researchers in the next decade and beyond.

[12] https://www.gqlstandards.org/

Publications

Publication List

Number of publications:	25
Number of peer-reviewed journal papers (written in English):	2
Number of articles in journals indexed by WoS or Scopus:	1
Number of publications (in English) with at least 50% contribution of the author:	4
Number of peer-reviewed publications:	22
Number of independent citations:	58

Publications Linked to the Contributions

	Journal papers	International conference and workshop papers	Local events	Technical reports	Book chapters
Thesis 1	—	[c5]	—	—	[b1]
Thesis 2	[j2]	[c8]	[l10], [l12], [l13], [l18]	— [r19], [r22]	—
Thesis 3	[j3]	[c4], [c6], [c7],	[l9], [l11], [l14], [l15], [l17]	[r20], [r21]	—

This classification follows the faculty's Ph.D. publication score system.

Book Chapters

[b1] Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In: *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, pp. 285–312. Springer, 2018. DOI: 10.1007/978-3-319-75396-6_16.

Journal Papers

[j2] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The Train Benchmark: Cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.* 17(4), 2018, pp. 1365–1393. DOI: 10.1007/s10270-016-0571-8.

[j3] Gábor Szárnyas, János Maginecz, and Dániel Varró. Evaluation of optimization strategies for incremental graph queries. *Period. Polytech. Elec. Eng. Comp. Sci.* 61(2), 2017, pp. 175–192. DOI: 10.3311/PPee.9769.

International Conference Papers

[c4] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. IncQuery-D: A distributed incremental model query framework in the cloud. In: *ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS)*, pp. 653–669. Springer, 2014. DOI: 10.1007/978-3-319-11653-2_40. Core rank: A. Acceptance rate: 26%.

[c5] Gábor Szárnyas, Zsolt Kővári, Ágnes Salánki, and Dániel Varró. Towards the characterization of realistic models: Evaluation of multidisciplinary graph metrics. In: *ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS)*, pp. 87–94. ACM, 2016. DOI: 10.1145/2976767.2976786. Core rank: A. Acceptance rate: 24%.

[c6] József Marton, Gábor Szárnyas, and Dániel Varró. Formalising openCypher graph queries in relational algebra. In: *Advances in Databases and Information Systems (ADBIS)*, Lecture Notes in Computer Science, pp. 182–196. Springer, 2017. DOI: 10.1007/978-3-319-66917-5_13. Core rank: B. Acceptance rate: 24%.

[c7] József Marton, Gábor Szárnyas, and Márton Búr. Model-driven engineering of an openCypher engine: Using graph queries to compile graph queries. In: *System Design Languages Forum (SDL): Model-Driven Engineering for Future Internet*, pp. 80–98. Springer, 2017. DOI: 10.1007/978-3-319-68015-6_6.

[c8] Muhammad Saleem, Gábor Szárnyas, Felix Conrads, Syed Ahmad Chan Bukhari, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. How representative is a SPARQL benchmark? An analysis of RDF triplestore benchmarks. In: *The Web Conference (WWW)*, pp. 1–11. ACM, 2019. Core rank: A*. Acceptance rate: 18%.

Workshop, Symposium, and Contest Papers

[l9] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. IncQuery-D: Incremental graph search in the cloud. In: *Workshop on Scalability in Model Driven Engineering (BigMDE) part of the Software Technologies: Applications and Foundations (STAF) federation of conferences*, 4:1–4:4. ACM, 2013. DOI: 10.1145/2487766.2487772.

[l10] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. MONDO-SAM: A framework to systematically assess MDE scalability. In: *Workshop on Scalability in Model Driven Engineering (BigMDE) part of the Software Technologies: Applications and Foundations (STAF) federation of conferences*, pp. 40–43. 2014. URL: http://ceur-ws.org/Vol-1206/paper_12.pdf.

[l11] Gábor Szárnyas, István Ráth, and Dániel Varró. Scalable query evaluation in the cloud. In: *Doctoral Symposium part of the Software Technologies: Applications and Foundations (STAF) federation of conferences*, 2014.

[l12] Gábor Szárnyas, Oszkár Semeráth, István Ráth, and Dániel Varró. The TTC 2015 Train Benchmark case for incremental model validation. In: *Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF) federation of conferences*, pp. 129–141. 2015. URL: http://ceur-ws.org/Vol-1524/paper2.pdf.

[l13] Gábor Szárnyas, Márton Búr, and István Ráth. Train Benchmark case: An EMF-IncQuery solution. In: *Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF) federation of conferences*, pp. 157–166. 2015. URL: http://ceur-ws.org/Vol-1524/paper19.pdf.

[l14] József Makai, Gábor Szárnyas, István Ráth, Ákos Horváth, and Dániel Varró. Optimization of incremental queries in the cloud. In: *International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE) at the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pp. 1–6. 2015. URL: http://ceur-ws.org/Vol-1563/paper1.pdf.

[l15] János Maginecz and Gábor Szárnyas. Sharded joins for scalable incremental graph queries. In: *Proceedings of the 23rd PhD Mini-Symposium*, pp. 26–29. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2016.

[l16] Gábor Szárnyas. Scalable graph query evaluation and benchmarking with realistic models. In: *Student Research Competition (SRC) at the International Conference on Model-Driven Engineering Languages and Systems (MODELS), sponsored by Microsoft Research*, ACM, 2016. URL: http://ceur-ws.org/Vol-1775/MODELS2016-SRC_paper_5.pdf.

[l17] Gábor Szárnyas. Incremental view maintenance for property graph queries. In: *Student Research Competition (SRC) at the International Conference on Management of Data (SIGMOD), sponsored by Microsoft*, pp. 1843–1845. ACM, 2018. DOI: 10.1145/3183713.3183724.

[l18] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. An early look at the LDBC Social Network Benchmark's Business Intelligence workload. In: *Joint International Workshop on Graph Data Management Experiences & Systems and Network Data Analytics (GRADES-NDA) at the International Conference on Management of Data (SIGMOD)*, 9:1–9:11. ACM, 2018. DOI: 10.1145/3210259.3210268.

Technical Reports

[r19] Benedek Izsó, Gábor Szárnyas, and István Ráth. *Train Benchmark*. Tech. rep. Budapest University of Technology and Economics, 2014. URL: https://inf.mit.bme.hu/research/publications/train-benchmark-technical-report.

[r20] József Marton and Gábor Szárnyas. *Formalisation of openCypher Queries in Relational Algebra (Extended Version)*. Tech. rep. Budapest University of Technology and Economics, 2017. URL: http://hdl.handle.net/10890/5395.

[r21] Gábor Szárnyas, József Marton, János Maginecz, and Dániel Varró. Reducing property graph queries to relational algebra for incremental view maintenance. *CoRR* arXiv:abs/1806.07344, 2018. URL: http://arxiv.org/abs/1806.07344.

[r22] LDBC Social Network Benchmark task force. *The LDBC Social Network Benchmark (version 0.3.2)*. Tech. rep. Linked Data Benchmark Council, 2019. URL: https://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf.

Other Publications (Not Linked to Contributions)

Workshop, Symposium, and Contest Papers

[a23] Gábor Szárnyas, Oszkár Semeráth, Benedek Izsó, Csaba Debreceni, Ábel Hegedüs, Zoltán Ujhelyi, and Gábor Bergmann. Movie Database case: An EMF-IncQuery solution. In: *Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF) federation of conferences*, pp. 103–115. 2014. URL: http://ceur-ws.org/Vol-1305/paper14.pdf.

[a24] Dániel Stein, Gábor Szárnyas, and István Ráth. Java Refactoring case: A VIATRA solution. In: *Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF) federation of conferences*, pp. 100–117. 2015. URL: http://ceur-ws.org/Vol-1524/paper21.pdf.

[a25] András Szabolcs Nagy and Gábor Szárnyas. Class Responsiblity Assignment case: A VIATRA-DSE solution. In: *Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF) federation of conferences*, pp. 39–44. 2016. URL: http://ceur-ws.org/Vol-1758/paper7.pdf.

Bibliography

[AB02] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.* 74(1), 2002, pp. 47–97. DOI: 10.1103/revmodphys.74.47.

[Aba+07] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. *Using the Barton Libraries Dataset as an RDF Benchmark*. Tech. rep. MIT-CSAIL-TR-2007-036. Massachusetts Institute of Technology, 2007.

[Abd+14] Hani Abdeen, Dániel Varró, Houari A. Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-objective optimization in rule-based design space exploration. In: *ASE*, pp. 289–300. 2014. DOI: 10.1145/2642937.2643005.

[Abe+16] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Old techniques for new join algorithms: A case study in RDF processing. In: *DESWeb at ICDE*, pp. 97–102. IEEE Computer Society, 2016. DOI: 10.1109/ICDEW.2016.7495625.

[Abe+17] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.* 42(4), 2017, 20:1–20:44. DOI: 10.1145/3129246.

[Abe+18] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. LevelHeaded: A unified engine for business intelligence and linear algebra querying. In: *ICDE*, pp. 449–460. IEEE, 2018. DOI: 10.1109/ICDE.2018.00048.

[ABG15] Ariful Azad, Aydin Buluç, and John R. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In: *GABB at IPDPS*, pp. 804–811. IEEE Computer Society, 2015. DOI: 10.1109/IPDPSW.2015.75.

[Abi+98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In: *VLDB*, pp. 38–49. 1998. URL: http://www.vldb.org/conf/1998/p038.pdf.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB J.* 15(2), 2006, pp. 121–142. DOI: 10.1007/s00778-004-0147-z.

[Aer16] Aeronautical Radio, Incorporated. A653 – Avionics Application Software Standard Interface. 2016.

[AG08] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.* 40(1), 2008, 1:1–1:39. DOI: 10.1145/1322432.1322433.

[Agh90] Gul A. Agha. *ACTORS - A model of concurrent computation in distributed systems*. MIT Press Series in Artificial Intelligence. MIT Press, 1990.

[Agr88] Rakesh Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. Software Eng.* 14(7), 1988, pp. 879–885. DOI: 10.1109/32.42731.

[Ahm+18] Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mucahid Kutlu, Mostafa Shehata, and Tamer Elsayed. LA3: A scalable link- and locality-aware linear algebra-based graph analytics system. *VLDB* 11(8), 2018, pp. 920–933. DOI: 10.14778/3204028.3204035.

[AJB99] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Diameter of the World-Wide Web. *Nature* 401(6749), 1999, pp. 130–131. DOI: 10.1038/43601.

[AK02] David H. Akehurst and Stuart Kent. A relational approach to defining transformations in a metamodel. In: *UML*, Lecture Notes in Computer Science, vol. 2460, pp. 243–258. Springer, 2002. DOI: 10.1007/3-540-45800-X_20.

[AK89] Serge Abiteboul and Paris C. Kanellakis. Object identity as a query language primitive. In: *SIGMOD*, pp. 159–173. ACM Press, 1989. DOI: 10.1145/67544.66941.

[Ake+17] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In: *SIGMOD*, pp. 1009–1024. ACM, 2017. DOI: 10.1145/3035918.3064029.

[Alu+14] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In: *ISWC*, pp. 197–212. 2014. DOI: 10.1007/978-3-319-11964-9_13.

[Amb98] José Luis Ambite. Planning by Rewriting. Ph.D. dissertation. University of Southern California, 1998.

[Amm+18] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *VLDB* 11(6), 2018, pp. 691–704. URL: http://www.vldb.org/pvldb/vol11/p691-ammar.pdf.

[And+92] Marc Andries, Marc Gemis, Jan Paredaens, Inge Thyssens, and Jan Van den Bussche. Concepts for graph-oriented object manipulation. In: *EDBT*, Lecture Notes in Computer Science, vol. 580, pp. 21–38. Springer, 1992. DOI: 10.1007/BFb0032421.

[Ang+17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.* 50(5), 2017, 68:1–68:40. DOI: 10.1145/3104031.

[Ang+18] Renzo Angles et al. G-CORE: A core for future graph query languages. In: *SIGMOD*, pp. 1421–1432. ACM, 2018. DOI: 10.1145/3183713.3190654.

[Ang12] Renzo Angles. 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*, pp. 171–177. IEEE Computer Society, 2012. DOI: 10.1109/ICDEW.2012.31.

[Ang18] Renzo Angles. The property graph database model. In: *AMW*, CEUR Workshop Proceedings, vol. 2100, CEUR-WS.org, 2018. URL: http://ceur-ws.org/Vol-2100/paper26.pdf.

[Anj+14] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. Efficient model synchronization with View Triple Graph Grammars. In: *ECMFA*, Lecture Notes in Computer Science, vol. 8569, pp. 1–17. Springer, 2014. DOI: 10.1007/978-3-319-09195-2_1.

[AÖ18] Khaled Ammar and M. Tamer Özsu. Experimental analysis of distributed graph systems. *PVLDB* 11(10), 2018, pp. 1151–1164. DOI: 10.14778/3231751.3231764.

[Apw+04] Rolf Apweiler et al. UniProt: The Universal Protein knowledgebase. *Nucleic Acids Research* 32, 2004, pp. 115–119. DOI: 10.1093/nar/gkh131.

[Ara+15a] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. Towards an automation of the mutation analysis dedicated to model transformation. *Softw. Test., Verif. Reliab.* 25(5-7), 2015, pp. 653–683. DOI: 10.1002/stvr.1532.

[Ara+15b] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. In: *ACES-MB & WUCOR at MODELS*, CEUR Workshop Proceedings, vol. 1508, pp. 19–26. CEUR-WS.org, 2015. URL: http://ceur-ws.org/Vol-1508/paper4.pdf.

[Are+15] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In: *SIGMOD*, pp. 1371–1382. ACM, 2015. DOI: 10.1145/2723372.2742796.

[Arm+13] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A database benchmark based on the Facebook social graph. In: *SIGMOD*, pp. 1185–1196. 2013. DOI: 10.1145/2463676.2465296.

[Arm+15] Michael Armbrust et al. Spark SQL: Relational data processing in Spark. In: *SIGMOD*, pp. 1383–1394. ACM, 2015. DOI: 10.1145/2723372.2742797.

[Art18] Artop User Group. *Artop: The AUTOSAR Tool Platform*. https://www.artop.org/. 2018.

[AS18] Amir Aghasadeghi and Julia Stoyanovich. Generating evolving property graphs with attribute-aware preferential attachment. In: *DBTest at SIGMOD*, 7:1–7:6. ACM, 2018. DOI: 10.1145/3209950.3209954.

[Ast+76] Morton M. Astrahan et al. System R: Relational approach to database management. *ACM Trans. Database Syst.* 1(2), 1976, pp. 97–137. DOI: 10.1145/320455.320457.

[Atk+89] Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, and Stanley B. Zdonik. The object-oriented database system manifesto. In: *DOOD*, pp. 223–240. 1989.

[AUT18] AUTOSAR Consortium. *The AUTOSAR Standard, R4.3*. 2018.

[Bac13] Michal Bachman. GraphAware: Towards Online Analytical Processing in Graph Databases. Master's thesis. Imperial College London, 2013. URL: https://graphaware.com/assets/bachman-msc-thesis.pdf.

[Bad+07] David A. Bader, John Feo, John Gilbert, Jeremy Kepner, David Koester, Eugene Loh, Kamesh Madduri, Bill Mann, and Theresa Meuse. HPCS SSCA# 2 Graph Analysis Benchmark Specification, v2.2. http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.2.pdf. 2007.

[Bad+09] David A. Bader, John Feo, John Gilbert, Jeremy Kepner, David Koester, Eugene Loh, Kamesh Madduri, Bill Mann, Theresa Meuse, and Eric Robinson. HPC Scalable Graph Analysis Benchmark. http://www.graphanalysis.org/benchmark/GraphAnalysisBenchmark-v1.0.pdf. 2009.

[Bag+17] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. gMark: Schema-Driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.* 29(4), 2017, pp. 856–869. DOI: 10.1109/TKDE.2016.2633993.

[Bar+10] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: A continuous query language for RDF data streams. *Int. J. Semantic Computing* 4(1), 2010, pp. 3–25. DOI: 10.1142/S1793351X10000936.

[Bat+15] Omar Batarfi, Radwa El Shawi, Ayman G. Fayoumi, Reza Nouri, Seyed-Mehdi-Reza Beheshti, Ahmed Barnawi, and Sherif Sakr. Large scale graph processing systems: Survey and an experimental evaluation. *Cluster Computing* 18(3), 2015, pp. 1189–1213. DOI: 10.1007/s10586-015-0472-6.

[Bat+18] Federico Battiston, Jeremy Guillon, Mario Chavez, Vito Latora, and Fabrizio De Vico Fallani. Multiplex core–periphery organization of the human connectome. *J. Royal Soc. Interface* 15(146), 2018. DOI: 10.1098/rsif.2018.0514.

[Bat17] Federico Battiston. The structure and dynamics of multiplex networks. Ph.D. dissertation. Queen Mary University of London, 2017. URL: https://qmro.qmul.ac.uk/xmlui/handle/123456789/30631.

[Bat94] Don Batory. *The LEAPS Algorithm.* Tech. rep. 1994.

[BB08] Robert Battle and Edward Benson. Bridging the Semantic Web and Web 2.0 with Representational State Transfer (REST). *J. Web Semant.* 6(1), 2008, pp. 61–69. DOI: 10.1016/j.websem.2007.11.002.

[BBE15] Gaël Blondelle, Francis Bordeleau, and Daniel Exertier. PolarSys: A new collaborative ecosystem for open source solutions for systems engineering driven by major industry players. *INSIGHT* 18(2), 2015, pp. 35–38.

[BDK92] François Bancilhon, Claude Delobel, and Paris C. Kanellakis, eds. *Building an Object-Oriented Database System, The Story of O2.* Morgan Kaufmann, 1992.

[BDT83] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems a systematic approach. In: *VLDB*, pp. 8–19. 1983. URL: http://www.vldb.org/conf/1983/P008.PDF.

[BEC12] Fabian Büttner, Marina Egea, and Jordi Cabot. On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: *MODELS*, Lecture Notes in Computer Science, vol. 7590, pp. 432–448. Springer, 2012. DOI: 10.1007/978-3-642-33666-9_28.

[Ber+08] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In: *ICGT*, pp. 396–410. 2008. DOI: 10.1007/978-3-540-87405-8_27.

[Ber+10] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltan Balogh, and András Ökrös. Incremental evaluation of model queries over EMF models. In: *MODELS*, pp. 76–90. 2010. DOI: 10.1007/978-3-642-16145-2_6.

[Ber+11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for EMF models. In: *ICMT*, pp. 167–182. 2011. DOI: 10.1007/978-3-642-21732-6_12.

[Ber+12] Gábor Bergmann, István Ráth, Tamás Szabó, Paolo Torrini, and Dániel Varró. Incremental pattern matching for the efficient computation of transitive closure. In: *ICGT*, Lecture Notes in Computer Science, vol. 7562, pp. 386–400. Springer, 2012. DOI: 10.1007/978-3-642-33654-6_26.

[Ber+13] Michele Berlingerio, Michele Coscia, Fosca Giannotti, Anna Monreale, and Dino Pedreschi. Multidimensional networks: Foundations of structural analysis. *World Wide Web* 16(5-6), 2013, pp. 567–593. DOI: 10.1007/s11280-012-0190-4.

[Ber13] Gábor Bergmann. Incremental Model Queries in Model-Driven Design. Ph.D. dissertation. Budapest University of Technology and Economics, 2013. URL: http://home.mit.bme.hu/~bergmann/download/phd-thesis-bergmann.pdf.

[Ber92] Elisa Bertino. A view mechanism for object-oriented databases. In: *EDBT*, Lecture Notes in Computer Science, vol. 580, pp. 136–151. Springer, 1992. DOI: 10.1007/BFb0032428.

[BG15] Thomas Beyhl and Holger Giese. *Efficient and scalable graph view maintenance for deductive graph databases based on generalized discrimination networks*. Tech. rep. 99. Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam. Universität Postdam, 2015, pp. 1–148.

[BGL12] Matthias Biehl, Wenqing Gu, and Frédéric Loiret. Model-based service discovery and orchestration for OSLC services in tool chains. In: *ICWE*, Lecture Notes in Computer Science, vol. 7387, pp. 283–290. Springer, 2012. DOI: 10.1007/978-3-642-31753-8_21.

[Bha+12] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In: *ICSE*, pp. 419–429. 2012. DOI: 10.1109/ICSE.2012.6227173.

[BHB09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The story so far. *Int. J. Semantic Web Inf. Syst.* 5(3), 2009, pp. 1–22. DOI: 10.4018/jswis.2009081901.

[BHH12] Gábor Bergmann, Dóra Horváth, and Ákos Horváth. Applying incremental graph transformation to existing models in relational databases. In: *ICGT*, pp. 371–385. 2012. DOI: 10.1007/978-3-642-33654-6_25.

[Biz+09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A crystallization point for the Web of Data. *J. Web Semant.* 7(3), 2009, pp. 154–165. DOI: 10.1016/j.websem.2009.07.002.

[BK13] Konstantinos Barmpis and Dimitrios S. Kolovos. Hawk: Towards a scalable model indexing architecture. In: *BigMDE at STAF*, p. 6. ACM, 2013. DOI: 10.1145/2487766.2487771.

[BK14] Konstantinos Barmpis and Dimitrios S. Kolovos. Evaluation of contemporary graph databases for efficient persistence of large-scale models. *J. Object Technol.* 13(3), 2014, pp. 1–26. DOI: 10.5381/jot.2014.13.3.a3.

[BKH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In: *ISWC*, pp. 54–68. 2002. DOI: 10.1007/3-540-48005-6_7.

[Bla+06] José A. Blakeley, David Campbell, S. Muralidhar, and Anil Nori. The ADO.NET Entity Framework: Making the conceptual level real. *SIGMOD Record* 35(4), 2006, pp. 32–39. DOI: 10.1145/1228268.1228275.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13(7), 1970, pp. 422–426. DOI: 10.1145/362686.362692.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In: *SIGMOD*, pp. 61–71. 1986. DOI: 10.1145/16894.16861.

[BM05] David A. Bader and Kamesh Madduri. Design and implementation of the HPCS Graph Analysis Benchmark on symmetric multiprocessors. In: *HiPC*, pp. 465–476. 2005. DOI: 10.1007/11602569_48.

[BNE13] Peter A. Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In: *TPCTC*, pp. 61–76. 2013. DOI: 10.1007/978-3-319-04936-6_5.

[BNL14] Federico Battiston, Vincenzo Nicosia, and Vito Latora. Structural measures for multiplex networks. *Phys. Rev. E* 89, 3 2014, p. 032804. DOI: 10.1103/PhysRevE.89.032804.

[Bot+18] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and complexity of MongoDB queries. In: *ICDT*, LIPIcs, vol. 98, 9:1–9:23. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. DOI: 10.4230/LIPIcs.ICDT.2018.9.

[Boy08] Brent Boyer. Robust Java benchmarking. *IBM developerWorks*, 2008. URL: https://www.ibm.com/developerworks/library/j-benchmark1/index.html.

[BP16] Albert-László Barabási and Márton Pósfai. *Network Science*. Cambridge University Press, 2016. URL: http://networksciencebook.com/.

[Bra+91] David A. Brant, Timothy Grose, Bernie J. Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In: *VLDB*, pp. 287–296. 1991. URL: http://www.vldb.org/conf/1991/P287.PDF.

[Bró+12] Piotr Bródka, Przemyslaw Kazienko, Katarzyna Musial, and Krzysztof Skibicki. Analysis of neighbourhoods in multi-layered dynamic social networks. *Int. J. Comput. Intell. Syst.* 5(3), 2012, pp. 582–596. DOI: 10.1080/18756891.2012.696922.

[Bru+18] Hugo Brunelière, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. In: *ACM/IEEE MODELS*, p. 211. ACM, 2018. DOI: 10.1145/3239372.3242895.

[BRV09] Gábor Bergmann, István Ráth, and Dániel Varró. Parallelization of graph transformation based on incremental pattern matching. *ECEASST* 18, 2009. DOI: 10.14279/tuj.eceasst.18.265.

[BS09] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.* 5(2), 2009, pp. 1–24. DOI: 10.4018/jswis.2009040101.

[BS97] Aaron B. Brown and Margo I. Seltzer. Operating system benchmarking in the wake of Lmbench: A case study of the performance of NetBSD on Intel x86 architecture. In: *SIGMETRICS*, pp. 214–224. 1997. DOI: 10.1145/258612.258690.

[BSK11] Piotr Bródka, Pawel Stawiak, and Przemyslaw Kazienko. Shortest path discovery in the multi-layered social network. In: *ASONAM*, pp. 497–501. 2011. DOI: 10.1109/ASONAM.2011.67.

[BTL11] Katrin Braunschweig, Maik Thiele, and Wolfgang Lehner. A flexible graph-based data model supporting incremental schema design and evolution. In: *ICWE Doctoral Symposium*, pp. 302–306. 2011. DOI: 10.1007/978-3-642-27997-3_29.

[Buc+95] Alejandro P. Buchmann, Jürgen Zimmermann, José A. Blakeley, and David L. Wells. Building an integrated active OODBMS: Requirements, architecture, and design decisions. In: *ICDE*, pp. 117–128. IEEE Computer Society, 1995. DOI: 10.1109/ICDE.1995.380401.

[Bun+95] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.* 149(1), 1995, pp. 3–48. DOI: 10.1016/0304-3975(95)00024-Q.

[Búr+15] Márton Búr, Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Local search-based pattern matching features in EMF-IncQuery. In: *ICGT at STAF*, pp. 275–282. 2015. DOI: 10.1007/978-3-319-21145-9_18.

[BW94] Elena Baralis and Jennifer Widom. An algebraic approach to rule analysis in expert database systems. In: *VLDB*, pp. 475–486. 1994. URL: http://www.vldb.org/conf/1994/P475.PDF.

[Cai+14] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing lambda-calculi by static differentiation. In: *PLDI*, pp. 145–155. ACM, 2014. DOI: 10.1145/2594291.2594304.

[Cap+15] Mihai Capota, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter A. Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In: *GRADES at SIGMOD*, 7:1–7:6. ACM, 2015. DOI: 10.1145/2764947.2764954.

[Car+13] Alessio Cardillo, Massimiliano Zanin, Jesús Gómez-Gardeñes, Miguel Romance, Alejandro J. García del Amo, and Stefano Boccaletti. Modeling the multi-layer nature of the European Air Transport Network: Resilience and passengers re-scheduling under random failures. *The European Physical Journal Special Topics* 215(1), 2013, pp. 23–33. DOI: 10.1140/epjst/e2013-01712-8.

[Car+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.* 38(4), 2015, pp. 28–38. URL: http://sites.computer.org/debull/A15dec/p28.pdf.

[Car+94] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In: *OOPSLA*, pp. 414–426. ACM, 1994. DOI: 10.1145/191080.191147.

[Car+97] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes Gehrke, and Dhaval Shah. The BUCKY object-relational benchmark (experience paper). In: *SIGMOD*, pp. 135–146. ACM Press, 1997. DOI: 10.1145/253260.253283.

[CB00] R. G. G. Cattell and Douglas K. Barry. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[CCG10] Jean-Paul Calbimonte, Óscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In: *ISWC*, pp. 96–111. 2010. DOI: 10.1007/978-3-642-17746-0_7.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In: *SIGMOD*, pp. 12–21. ACM Press, 1993. DOI: 10.1145/170035.170041.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.* 1(1), 1989, pp. 146–166. DOI: 10.1109/69.43410.

[Cha+18] Jesse Chamberlin, Marcin Zalewski, Scott McMillan, and Andrew Lumsdaine. PyGB: GraphBLAS DSL in Python with dynamic compilation into efficient C++. In: *GABB at IPDPS*, pp. 310–319. IEEE Computer Society, 2018. DOI: 10.1109/IPDPSW.2018.00059.

[Che+08] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. Fast graph pattern matching. In: *ICDE*, pp. 913–922. IEEE Computer Society, 2008. DOI: 10.1109/ICDE.2008.4497500.

[Che18] Yuxing Chen. Worst case optimal joins on relational and XML data. In: *SIGMOD*, pp. 1833–1835. ACM, 2018. DOI: 10.1145/3183713.3183721.

[Chi11] Joanna Chimiak-Opoka. Measuring UML models using metrics defined in OCL within the SQUAM framework. In: *MODELS*, pp. 47–61. 2011. DOI: 10.1007/978-3-642-24485-8_5.

[Cho+05] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In: *VLDB*, pp. 1216–1227. ACM, 2005. URL: http://www.vldb.org/archives/website/2005/program/paper/fri/p1216-chong.pdf.

[Chu14] Fan Chung. A brief survey of PageRank algorithms. *IEEE Trans. Network Science and Engineering* 1(1), 2014, pp. 38–42. DOI: 10.1109/TNSE.2014.2380315.

[Cio18] Georgiana Diana Ciocirdel. A G-CORE (Graph Query Language) Interpreter. Master's thesis. Vrije Universiteit Amsterdam, 2018. URL: https://homepages.cwi.nl/~boncz/msc/2018-GeorgianaCiocirdel.pdf.

[CL14] Arnaud Castelltort and Anne Laurent. NoSQL graph-based OLAP analysis. In: *KDIR*, pp. 217–224. SciTePress, 2014. DOI: 10.5220/0005072902170224.

[CM84] George P. Copeland and David Maier. Making smalltalk a database system. In: *SIGMOD*, pp. 316–325. ACM Press, 1984. DOI: 10.1145/602259.602300.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM* 13(6), 1970, pp. 377–387. DOI: 10.1145/362384.362685.

[Coh09] Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engineering* 11(4), 2009, pp. 29–41. DOI: 10.1109/MCSE.2009.120.

[Col+96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In: *SIGMOD*, ACM Press, 1996. DOI: 10.1145/233269.233364.

[Col90] Latha S. Colby. A recursive algebra for nested relations. *Inf. Syst.* 15(5), 1990, pp. 567–582. DOI: 10.1016/0306-4379(90)90029-O.

[Cor+04] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26(10), 2004, pp. 1367–1372. DOI: 10.1109/TPAMI.2004.75.

[Cos+11] Luciano da Fontoura Costa, Osvaldo N. Oliveira, Gonzalo Travieso, Francisco Aparecido Rodrigues, Paulino Ribeiro Villas Boas, Lucas Antiqueira, Matheus Palhares Viana, and Luis Enrique Correa Rocha. Analyzing and modeling real-world phenomena with complex networks: A survey of applications. *Adv. Phys.* 60(3), 2011, pp. 329–412. DOI: 10.1080/00018732.2011.572452.

[CY12] Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases* 4(4), 2012, pp. 295–405. DOI: 10.1561/1900000020.

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In: *SIAM International Conference on Data Mining (SDM)*, pp. 442–446. 2004. DOI: 10.1137/1.9781611972740.43.

[Dan+17] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. NeoEMF: A multi-database model persistence framework for very large models. *Sci. Comput. Program.* 149, 2017, pp. 9–14. DOI: 10.1016/j.scico.2017.08.002.

[Dat70] Data Base Task Group. Data Base Task Group report to the CODASYL programming language committee. *DATA BASE* 2(2), 1970, pp. 11–18. DOI: 10.1145/2579329.2579336.

[Dat78] Data Base Task Group. CODASYL: Reports of the Data Description Language Committee. *Inf. Syst.* 3(4), 1978, pp. 247–320.

[Dav18] Timothy A. Davis. Graph algorithms via SuiteSparse:GraphBLAS: Triangle counting and K-truss. In: *HPEC*, pp. 1–6. IEEE, 2018. DOI: 10.1109/HPEC.2018.8547538.

[DC74] C. J. Date and E. F. Codd. The relational and network approaches: Comparison of the application programming interfaces. In: *SIGMOD*, pp. 83–113. ACM, 1974. DOI: 10.1145/800297.811532.

[DCL18] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on NoSQL stores. *ACM Comput. Surv.* 51(2), 2018, 40:1–40:43. DOI: 10.1145/3158661.

[Deb+14] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. Query-driven incremental synchronization of view models. In: *VAO at STAF*, pp. 31–38. 2014. DOI: 10.1145/2631675.2631677.

[Deb+17] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Property-based locking in collaborative modeling. In: *MODELS*, pp. 199–209. IEEE Computer Society, 2017. DOI: 10.1109/MODELS.2017.33.

[Deb19] Csaba Debreceni. Advanced Techniques and Tools for Secure Collaborative Modeling. Ph.D. dissertation. Budapest University of Technology and Economics, 2019.

[DER03a] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. Order-sensitive view maintenance of materialized XQuery views. In: *ER*, pp. 144–157. 2003. DOI: 10.1007/978-3-540-39648-2_14.

[DER03b] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. *Order-Sensitive View Maintenance of Materialized XQuery Views*. Tech. rep. WPI-CS-TR-03-17. Computer Science Department, Worcester Polytechnic Institute, 2003. URL: ftp://ftp.cs.wpi.edu/pub/techreports/pdf/03-17.pdf.

[DeW91] David J. DeWitt. The Wisconsin Benchmark: Past, present, and future. In: *The Benchmark Handbook*, pp. 119–165. 1991. URL: https://jimgray.azurewebsites.net/BenchmarkHandbook/chapter4.pdf.

[Die12] Reinhard Diestel. *Graph Theory*. 4th. Graduate texts in mathematics. Springer, 2012.

[DM02] S. N. Dorogovtsev and J. F. F. Mendes. Evolution of networks. *Adv. Phys.* 51(4), 2002, pp. 1079–1187. DOI: 10.1080/00018730110112519.

[Don+99] Guozhu Dong, Leonid Libkin, Jianwen Su, and Limsoon Wong. Maintaining transitive closure of graphs in SQL. *J. Inf. Tech.* 51(1), 1999, p. 46. URL: https://corescholar.libraries.wright.edu/knoesis/399/.

[Doo95] Robert B. Doorenbos. Production matching for large learning systems. Ph.D. dissertation. University of Southern California, 1995.

[DP07] P.M. Duvall and D. Paul. *Continuous Integration*. Pearson Education, 2007. URL: https://books.google.com/books?id=YPlgL1aILqIC.

[DR99] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top N queries. In: *VLDB*, pp. 411–422. 1999. URL: http://www.vldb.org/conf/1999/P40.pdf.

[DRV18] István Dávid, István Ráth, and Dániel Varró. Foundations for streaming model transformations by complex event processing. *Softw. Syst. Model.* 17(1), 2018, pp. 135–162. DOI: 10.1007/s10270-016-0533-1.

[DS00a] Jérôme Darmont and Michel Schnieder. Benchmarking OODBs with a generic tool. *J. Database Manag.* 11(3), 2000, pp. 16–27. DOI: 10.4018/jdm.2000070102.

[DS00b] Guozhu Dong and Jianwen Su. Incremental maintenance of recursive views using relational calculus/SQL. *SIGMOD Record* 29(1), 2000, pp. 44–51. DOI: 10.1145/344788.344808.

[DSC18] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Scalable queries and model transformations with the mogwai tool. In: *ICMT*, Lecture Notes in Computer Science, vol. 10888, pp. 175–183. Springer, 2018. DOI: 10.1007/978-3-319-93317-7_9.

[DTG00] Klaus R. Dittrich, Dimitris Tombros, and Andreas Geppert. Databases in software engineering: A roadmap. In: *ICSE*, pp. 293–302. ACM, 2000. DOI: 10.1145/336512.336580.

[Dua+11] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. In: *SIGMOD*, pp. 145–156. 2011. DOI: 10.1145/1989323.1989340.

[Eas+08] Chuck Eastman, Paul Teicholz, Rafael Sacks, and Kathleen Liston. *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*. Wiley Publishing, 2008.

[Ecl15a] Eclipse Foundation. Model Development Tools (MDT) - Eclipse OCL. http://www.eclipse.org/modeling/mdt/?project=ocl. 2015.

[Ecl15b] Eclipse Foundation. Teneo. https://wiki.eclipse.org/Teneo. 2015.

[Ecl19] Eclipse Foundation. CDO - The Model Repository. https://www.eclipse.org/cdo/. 2019.

[ECM11] Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, and Jesús Garcia Molina. Morsa: A scalable approach for persisting and accessing large models. In: *MODELS*, pp. 77–92. 2011. DOI: 10.1007/978-3-642-24485-8_7.

[Egy06] Alexander Egyed. Instant consistency checking for the UML. In: *ICSE*, pp. 381–390. 2006. DOI: 10.1145/1134339.

[EKS93] Wolfgang Emmerich, Petr Kroha, and Wilhelm Schäfer. Object-oriented database management systems for construction of CASE environments. In: *DEXA (Database and Expert Systems Applications)*, Lecture Notes in Computer Science, vol. 720, pp. 631–642. Springer, 1993. DOI: 10.1007/3-540-57234-1_65.

[Elg+18] Iman Elghandour, Ahmet Kara, Dan Olteanu, and Stijn Vansummeren. Incremental techniques for large-scale dynamic query processing. In: *CIKM*, pp. 2297–2298. ACM, 2018. DOI: 10.1145/3269206.3274271.

[EM09a] Orri Erling and Ivan Mikhailov. RDF support in the Virtuoso DBMS. In: *Networked Knowledge - Networked Media - Integrating Knowledge Management, New Media Technologies and Semantic Systems*, pp. 7–24. 2009. DOI: 10.1007/978-3-642-02184-8_2.

[EM09b] Orri Erling and Ivan Mikhailov. Virtuoso: RDF support in a native RDBMS. In: Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca (eds.), *Semantic Web Information Management - A Model-Based Perspective*, pp. 501–519. Springer, 2009. DOI: 10.1007/978-3-642-04329-1_21.

[EM13] Benedikt Elser and Alberto Montresor. An evaluation study of Big Data frameworks for graph processing. In: *International Conference on Big Data*, pp. 60–67. 2013. DOI: 10.1109/BigData.2013.6691555.

[EN00] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. 3rd. Addison-Wesley-Longman, 2000.

[EN79] José L. Encarnação and Thomas Neumann. A survey of DB requirements for graphical applications in engineering. In: *Data Base Techniques for Pictorial Applications*, Lecture Notes in Computer Science, vol. 81, pp. 285–297. Springer, 1979. DOI: 10.1007/3-540-09763-5_15.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In: *Annual Symposium on Switching and Automata Theory (SWAT)*, pp. 167–180. IEEE Computer Society, 1973. DOI: 10.1109/SWAT.1973.11.

[Eré+09] Guillaume Erétéo, Michel Buffa, Fabien Gandon, and Olivier Corby. Analysis of a real online social network using semantic web frameworks. In: *ISWC*, pp. 180–195. 2009. DOI: 10.1007/978-3-642-04930-9_12.

[Erl+15] Orri Erling, Alex Averbuch, Josep-Lluis Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC Social Network Benchmark: Interactive workload. In: *SIGMOD*, pp. 619–630. 2015. DOI: 10.1145/2723372.2742786.

[ESW93] Wolfgang Emmerich, Wilhelm Schäfer, and Jim Welsh. Databases for software engineering environments - The goal has not yet been attained. In: *ESEC (European Software Engineering Conference)*, Lecture Notes in Computer Science, vol. 717, pp. 145–162. Springer, 1993. DOI: 10.1007/3-540-57209-0_11.

[Fal+14] Jean-Rémy Falleri, Xavier Blanc, Reda Bendraou, Marcos Aurélio Almeida da Silva, and Cédric Teyton. Incremental inconsistency detection with low memory overhead. *Softw., Pract. Exper.* 44(5), 2014, pp. 621–641. DOI: 10.1002/spe.2171.

[Fan+10] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB* 3(1), 2010, pp. 264–275. DOI: 10.14778/1920841.1920878.

[FFF99] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In: *SIGCOMM*, pp. 251–262. 1999. DOI: 10.1145/316188.316229.

[FHT17] Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In: *SIGMOD*, pp. 155–169. ACM, 2017. DOI: 10.1145/3035918.3035944.

[FM95] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In: *SIGMOD*, pp. 47–58. ACM Press, 1995. DOI: 10.1145/223784.223789.

[For79] Charles Forgy. On the efficient implementation of production systems. Ph.D. dissertation. Carnegie-Mellon University, 1979.

[For82] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.* 19(1), 1982, pp. 17–37. DOI: 10.1016/0004-3702(82)90020-0.

[Fra+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In: *SIGMOD*, pp. 1433–1445. ACM, 2018. DOI: 10.1145/3183713.3190657.

[Fre78] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks* 1(3), 1978, pp. 215–239.

[Fro+02] Agata Fronczak, Janusz A Hołyst, Maciej Jedynak, and Julian Sienkiewicz. Higher order clustering coefficients in Barabási–Albert networks. *Physica A: Statistical Mechanics and its Applications* 316(1), 2002, pp. 688–694. DOI: 10.1016/S0378-4371(02)01336-5.

[FRP15] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The case against specialized graph analytics engines. In: *CIDR*, 2015. URL: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper20.pdf.

[FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM* 29(3), 1986, pp. 218–221. DOI: 10.1145/5666.5673.

[Gal94] César A. Galindo-Legaria. Outerjoins as disjunctions. In: *SIGMOD*, pp. 348–358. ACM Press, 1994. DOI: 10.1145/191839.191908.

[Gao+18] Libo Gao, Lukasz Golab, M. Tamer Özsu, and Günes Aluç. Stream WatDiv: A streaming RDF benchmark. In: *SBD at SIGMOD*, 3:1–3:6. 2018. DOI: 10.1145/3208352.3208355.

[GB14a] Andrey Gubichev and Peter A. Boncz. Parameter curation for benchmark queries. In: *TPCTC*, Lecture Notes in Computer Science, vol. 8904, pp. 113–129. Springer, 2014.

[GB14b] Ramanathan Guha and Dan Brickley. *RDF Schema 1.1*. W3C Recommendation. W3C, 2014. URL: http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[GDL08] Esther Guerra, Paloma Diaz, and Juan de Lara. Visual specification of metrics for domain specific visual languages. *Electr. Notes Theor. Comput. Sci.* 211, 2008, pp. 99–110. DOI: 10.1016/j.entcs.2008.04.033.

[Gei+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In: *ICGT*, pp. 383–397. 2006. DOI: 10.1007/11841883_27.

[Gen+03] John H. Gennari, Mark A. Musen, Ray W. Fergerson, William E. Grosso, Monica Crubézy, Henrik Eriksson, Natalya Fridman Noy, and Samson W. Tu. The evolution of Protégé: An environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud.* 58(1), 2003, pp. 89–123. DOI: 10.1016/S1071-5819(02)00127-1.

[Gér+07] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. Papyrus: A UML2 tool for domain-specific language modeling. In: *Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop*, pp. 361–368. 2007. DOI: 10.1007/978-3-642-16277-0_19.

[GHK17] Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, eds. *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017.* Vol. 2026. CEUR Workshop Proceedings. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-2026.

[GHM98] John C. Grundy, John G. Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Software Eng.* 24(11), 1998, pp. 960–981. DOI: 10.1109/32.730545.

[GK07] Rubino Geiß and Moritz Kroll. *On Improvements of the Varro Benchmark for Graph Transformation Tools.* Tech. rep. 2007-7. ISSN 1432-7864. Universität Karlsruhe, IPD Goos, 2007. URL: http://www.info.uni-karlsruhe.de/papers/TR_2007_7.pdf.

[GK98] Timothy Griffin and Bharat Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record* 27(3), 1998, pp. 22–27. DOI: 10.1145/290593.290597.

[GKR16] Antonio Garcia-Dominguez, Filip Krikava, and Louis M. Rose, eds. *Proceedings of the 9th Transformation Tool Contest, co-located with the 2016 Software Technologies: Applications and Foundations (STAF 2016), Vienna, Austria, July 8, 2016.* Vol. 1758. CEUR Workshop Proceedings. CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1758.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In: *SIGMOD*, pp. 328–339. 1995. DOI: 10.1145/223784.223849.

[GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Trans. Knowl. Data Eng.* 9(3), 1997, pp. 508–511. DOI: 10.1109/69.599937.

[GM06] Himanshu Gupta and Inderpal Singh Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Inf. Syst.* 31(6), 2006, pp. 435–464. DOI: 10.1016/j.is.2004.11.011.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18(2), 1995, pp. 3–18. URL: http://sites.computer.org/debull/95JUN-CD.pdf.

[GM99] Ashish Gupta and Iderpal Singh Mumick, eds. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, 1999.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In: *SIGMOD*, pp. 157–166. 1993. DOI: 10.1145/170035.170066.

[Góm+15] Abel Gómez, Massimo Tisi, Gerson Sunyé, and Jordi Cabot. Map-based transparent persistence for very large models. In: *FASE*, Lecture Notes in Computer Science, vol. 9033, pp. 19–34. Springer, 2015. DOI: 10.1007/978-3-662-46675-9_2.

[Gon+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In: *OSDI*, pp. 599–613. USENIX Association, 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.* 3(2-3), 2005, pp. 158–182. DOI: 10.1016/j.websem.2005.06.005.

[Gra+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.* 1(1), 1997, pp. 29–53. DOI: 10.1023/A:1009726021843.

[Gra73] Mark S. Granovetter. The strength of weak ties. *Am. J. Sociol.* 78(6), 1973, pp. 1360–1380. DOI: 10.1086/225469.

[Gra93] Jim Gray, ed. *The Benchmark Handbook for Database and Transaction Systems*. 2nd. Morgan Kaufmann, 1993.

[Gre+18] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. openCypher: New directions in property graph querying. In: *EDBT*, pp. 520–523. 2018. DOI: 10.5441/002/edbt.2018.62.

[Gro12] OWL Working Group. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. W3C, 2012. URL: http://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

[GS14] Fabien Gandon and Guus Schreiber. *RDF 1.1 XML Syntax*. W3C Recommendation. W3C, 2014. URL: http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

[GS18] Jana Giceva and Mohammad Sadoghi. Hybrid OLTP and OLAP. In: *Encyclopedia of Big Data Technologies*. Springer International Publishing, 2018, pp. 1–8. DOI: 10.1007/978-3-319-63962-8_179-1.

[GTS12] Olaf Görlitz, Matthias Thimm, and Steffen Staab. SPLODGE: Systematic generation of SPARQL benchmark queries for Linked Open Data. In: *ISWC*, pp. 116–132. 2012. DOI: 10.1007/978-3-642-35176-1_8.

[Gub15] Andrey Gubichev. Query Processing and Optimization in Graph Databases. Ph.D. dissertation. Technical University Munich, 2015. URL: http://nbn-resolving.de/urn:nbn:de:bvb:91-diss-20150625-1238730-1-7.

[Guo+14] Yong Guo, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke. Benchmarking graph-processing platforms: A vision. In: *ICPE*, pp. 289–292. 2014. DOI: 10.1145/2568088.2576761.

[GUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice-Hall, 2000.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems – The complete book*. 2nd. Pearson Education, 2009.

[GZ10] Leif Geiger and Albert Zündorf. Fujaba case studies for GraBaTs 2008: Lessons learned. *Softw. Tools Technol. Transfer* 12(3-4), 2010, pp. 287–304. DOI: 10.1007/s10009-010-0152-z.

[Hae+19] Martin Haeusler, Thomas Trojer, Johannes Kessler, Matthias Farwick, Emmanuel Nowakowski, and Ruth Breu. ChronoSphere: A graph-based EMF model repository for IT landscape models. *Softw. Syst. Model.*, 2019. DOI: 10.1007/s10270-019-00725-0.

[Han96] Eric N. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Trans. Knowl. Data Eng.* 8(1), 1996, pp. 157–172. DOI: 10.1109/69.485644.

[Hav15] Klaus Havelund. Rule-based runtime verification revisited. *Softw. Tools Technol. Transfer* 17(2), 2015, pp. 143–170. DOI: 10.1007/s10009-014-0309-2.

[HB15] Daniel S. Himmelstein and Sergio E. Baranzini. Heterogeneous network edge prediction: A data integration approach to prioritize disease-associated genes. *PLoS Computational Biology* 11(7), 2015. DOI: 10.1371/journal.pcbi.1004259.

[HBC02] Eric N. Hanson, Sreenath Bodagala, and Ullas Chadaga. Trigger condition testing and view maintenance using optimized discrimination networks. *IEEE Trans. Knowl. Data Eng.* 14(2), 2002, pp. 261–280. DOI: 10.1109/69.991716.

[HBC15] Nathan Hawes, Ben Barham, and Cristina Cifuentes. Frappé: Querying the Linux kernel dependency graph. In: *GRADES at SIGMOD*, 4:1–4:6. ACM, 2015. DOI: 10.1145/2764947.2764951.

[HD18] Mohammad Al Hasan and Vachik S. Dave. Triangle counting in large networks: A review. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 8(2), 2018. DOI: 10.1002/widm.1226.

[Heg+11] Ábel Hegedüs, Ákos Horváth, István Ráth, Moisés Castelo Branco, and Dániel Varró. Quick fix generation for DSMLs. In: *VL/HCC*, pp. 17–24. 2011. DOI: 10.1109/VLHCC.2011.6070373.

[Heg+16] Ábel Hegedüs, Ákos Horváth, István Ráth, Rodrigo Rizzi Starr, and Dániel Varró. Query-driven soft traceability links for models. *Softw. Syst. Model.* 15(3), 2016, pp. 733–756. DOI: 10.1007/s10270-014-0436-y.

[Heg+18] Ábel Hegedüs, Gábor Bergmann, Csaba Debreceni, Ákos Horváth, Péter Lunk, Ákos Menyhért, István Papp, Dániel Varró, Tomas Vileiniskis, and István Ráth. IncQuery Server for Teamwork Cloud: Scalable query evaluation over collaborative model repositories. In: *MODELS*, pp. 27–31. ACM, 2018. DOI: 10.1145/3270112.3270125.

[Hei+11] Florian Heidenreich, Jendrik Johannes, Jan Reimann, Mirko Seifert, Christian Wende, Christian Werner, Claas Wilke, and Uwe Assmann. Model-driven modernisation of Java programs with JaMoPP. In: *MDSM at CSMR*, pp. 8–11. 2011.

[Hei+18] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Comput. Surv.* 51(3), 2018, 60:1–60:53. DOI: 10.1145/3199523.

[HG16] Jürgen Hölsch and Michael Grossniklaus. An algebra and equivalences to transform graph patterns in Neo4j. In: *GraphQ at EDBT/ICDT*, CEUR Workshop Proceedings, vol. 1558, CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1558/paper24.pdf.

[HHL14] Johannes Härtel, Lukas Härtel, and Ralf Lämmel. Test-data generation for Xtext - Tool paper. In: *SLE*, Lecture Notes in Computer Science, vol. 8706, pp. 342–351. Springer, 2014. DOI: 10.1007/978-3-319-11245-9_19.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.* 26(3/4), 1996, pp. 287–313. DOI: 10.3233/FI-1996-263404.

[Hin17] Georg Hinkel. The TTC 2017 Outage System case for incremental model views. In: *TTC at STAF*, CEUR Workshop Proceedings, vol. 2026, pp. 3–12. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-2026/paper1.pdf.

[Hin18a] Georg Hinkel. NMF: A multi-platform modeling framework. In: *ICMT*, pp. 184–194. 2018. DOI: 10.1007/978-3-319-93317-7_10.

[Hin18b] Georg Hinkel. The TTC 2018 Social Media case. In: *TTC at STAF*, CEUR Workshop Proceedings, vol. To appear, CEUR-WS.org, 2018.

[HKT14] Tassilo Horn, Christian Krause, and Matthias Tichy. The TTC 2014 Movie Database case. In: *TTC at STAF*, pp. 93–97. 2014. URL: http://ceur-ws.org/Vol-1305/paper2.pdf.

[HLS09] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In: *SSWS (International Workshop on Scalable Semantic Web Knowledge Base Systems)*, CEUR Workshop Proceedings, vol. 517, CEUR-WS.org, 2009. URL: http://ceur-ws.org/Vol-517/ssws09-paper7.pdf.

[Hon+12] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In: *ASPLOS*, pp. 349–362. ACM, 2012. DOI: 10.1145/2150976.2151013.

[Hon+15] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A fast distributed graph processing engine. In: *SC*, 58:1–58:12. 2015. DOI: 10.1145/2807591.2807620.

[Hor+10] Ákos Horváth, Gábor Bergmann, István Ráth, and Dániel Varró. Experimental assessment of combining pattern matching strategies with VIATRA2. *Softw. Tools Technol. Transfer* 12(3-4), 2010, pp. 211–230. DOI: 10.1007/s10009-010-0149-7.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. 5th. Morgan Kaufmann, 2012.

[HP18] Olaf Hartig and Jorge Pérez. Semantics and complexity of GraphQL. In: *WWW*, pp. 1155–1164. ACM, 2018. DOI: 10.1145/3178876.3186014.

[HS13] Steven Harris and Andy Seaborne. *SPARQL 1.1 Query Language*. W3C Recommendation. W3C, 2013. URL: http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[Hup09] Karl Huppler. The art of building a good benchmark. In: *TPCTC*, pp. 18–30. 2009. DOI: 10.1007/978-3-642-10424-4_3.

[Hut+16] Dylan Hutchison, Jeremy Kepner, Vijay Gadepally, and Bill Howe. From NoSQL Accumulo to NewSQL Graphulo: Design and utility of graph algorithms inside a BigTable database. In: *HPEC*, pp. 1–9. IEEE, 2016. DOI: 10.1109/HPEC.2016.7761577.

[Hut17] Dylan Hutchison. Distributed triangle counting in the Graphulo matrix math library. In: *HPEC*, pp. 1–7. IEEE, 2017. DOI: 10.1109/HPEC.2017.8091041.

[Idr+18] Muhammad Idris, Martin Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with inequalities under updates. *VLDB* 11(7), 2018, pp. 733–745. URL: http://www.vldb.org/pvldb/vol11/p733-idris.pdf.

[Ios+16] Alexandru Iosup et al. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *VLDB* 9(13), 2016, pp. 1317–1328. DOI: 10.14778/3007263.3007270.

[Ire+09] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In: *DBKDS*, pp. 36–43. IEEE Computer Society, 2009. DOI: 10.1109/DBKDA.2009.11.

[ISO99] ISO. *Information technology – Database languages – SQL – Part 2: Foundation (SQL/-Foundation)*. Standard. International Organization for Standardization, 1999. URL: https://www.iso.org/standard/26197.html.

[Itz+09] Ben-Gan Itzik, Lubor Kollár, Dejan Sarka, and Steve Kass. *Inside Microsoft SQL Server 2008 - T-SQL Querying*. Microsoft Press, 2009.

[IUV17] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and efficient query processing under updates. In: *SIGMOD*, pp. 1259–1274. 2017. DOI: 10.1145/3035918.3064027.

[Izs+13b] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, and István Ráth. Towards precise metrics for predicting graph query performance. In: *ASE*, pp. 421–431. 2013. DOI: 10.1109/ASE.2013.6693100.

[JC04] Bin Jiang and Christophe Claramunt. Topological analysis of urban street networks. *Environment and Planning B: Planning and Design* 31(1), 2004, pp. 151–162. DOI: 10.1068/b306.

[Jim+17] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay F. Lofstead, Kathryn Mohror, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Popper convention: Making reproducible systems evaluation practical. In: *REPPAR at IPDPS*, pp. 1561–1570. IEEE Computer Society, 2017. DOI: 10.1109/IPDPSW.2017.157.

[Jin+15] Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. Graph analytics using Vertica relational database. In: *International Conference on Big Data*, pp. 1191–1200. IEEE, 2015. DOI: 10.1109/BigData.2015.7363873.

[JKS06] Johannes Jakob, Alexander Königs, and Andy Schürr. Non-materialized model view specification with Triple Graph Grammars. In: *ICGT*, Lecture Notes in Computer Science, vol. 4178, pp. 321–335. Springer, 2006. DOI: 10.1007/11841883_23.

[JO18] Hyunsu Ju and Sangyoon Oh. Enabling RETE algorithm for RDFS reasoning on Apache Spark. In: *(International Symposium on Cloud and Service Computing*, pp. 135–138. IEEE, 2018. DOI: 10.1109/SC2.2018.00028.

[JT10] Frédéric Jouault and Massimo Tisi. Towards incremental execution of ATL transformations. In: Lecture Notes in Computer Science, vol. 6142, pp. 123–137. Springer, 2010. DOI: 10.1007/978-3-642-13688-7_9.

[Jun+17] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. Cypher-based graph pattern matching in Gradoop. In: *GRADES at SIGMOD*, 3:1–3:8. ACM, 2017. DOI: 10.1145/3078447.3078450.

[Kah+18] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Softw. Syst. Model.*, 2018. DOI: 10.1007/s10270-018-0665-6.

[Kal+10] P. Kaluza, A. Kölzsch, M.T. Gastner, and B. Blasius. The complex network of global cargo ship movements. *J. Royal Soc. Interface* 7(48), 2010, pp. 1093–1103. DOI: http://doi.org/10.1098/rsif.2009.0495.

[Kan+17] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In: *SIGMOD*, pp. 1695–1698. 2017. DOI: 10.1145/3035918.3056445.

[Kaw+97] Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Implementing incremental view maintenance in nested data models. In: *DBPL*, pp. 202–221. 1997. DOI: 10.1007/3-540-64823-2_12.

[KD96] Udo Kelter and Dirk Däberitz. An assessment of non-standard DBMSs for CASE environments. In: *EDBT*, Lecture Notes in Computer Science, vol. 1057, pp. 96–113. Springer, 1996. DOI: 10.1007/BFb0014145.

[Kel85] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In: *PODS*, pp. 154–163. ACM, 1985. DOI: 10.1145/325405.325423.

[Kel92] Udo Kelter. H-PCTE - A high performance object management system for system development environments. In: *International Computer Software and Applications Conference (COMPSAC)*, pp. 45–50. IEEE, 1992. DOI: 10.1109/CMPSAC.1992.217605.

[Kep+15] Jeremy Kepner, David A. Bader, Aydin Buluç, John R. Gilbert, Timothy G. Mattson, and Henning Meyerhenke. Graphs, matrices, and the GraphBLAS: Seven good reasons. In: *ICCS*, Procedia Computer Science, vol. 51, pp. 2453–2462. Elsevier, 2015. DOI: 10.1016/j.procs.2015.05.353.

[Kep+16] Jeremy Kepner et al. Mathematical foundations of the GraphBLAS. In: *HPEC*, IEEE, 2016. DOI: 10.1109/HPEC.2016.7761646.

[KG11] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/1.9780898719918.

[KG89] Simon M. Kaplan and Steven K. Goering. Priority controlled incremental attribute evaluation in attributed graph grammars. In: *CAAP at TAPSOFT*, Lecture Notes in Computer Science, vol. 351, pp. 306–336. Springer, 1989. DOI: 10.1007/3-540-50939-9_140.

[KH10] Maximilian Koegel and Jonas Helming. EMFStore: A model repository for EMF models. In: *ICSE*, pp. 307–308. ACM, 2010. DOI: 10.1145/1810295.1810364.

[Kin94] Roger King. Workshop on the intersection between databases and software engineering. In: *ICSE*, p. 355. IEEE Computer Society / ACM Press, 1994. URL: http://portal.acm.org/citation.cfm?id=257734.257800.

[Kiv+14] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P. Gleeson, Yamir Moreno, and Mason A. Porter. Multilayer networks. *Journal of Complex Networks*, 2014. DOI: 10.1093/comnet/cnu016.

[KK05] Jeremy Kepner and David Koester. HPCS SSCA# 2 Graph Analysis Benchmark Specification, v1.0. 2005.

[KKD89] Kyung-Chang Kim, Won Kim, and Alfred G. Dale. Cyclic query processing in object-oriented databases. In: *ICDE*, pp. 564–571. IEEE Computer Society, 1989. DOI: 10.1109/ICDE.1989.47263.

[KLT16] Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In: *PODS*, pp. 75–90. 2016. DOI: 10.1145/2902251.2902286.

[KMK11] Przemyslaw Kazienko, Katarzyna Musial, and Tomasz Kajdanowicz. Multidimensional social network in the social recommender system. *IEEE Trans. Systems, Man, and Cybernetics* 41(4), 2011, pp. 746–759. DOI: 10.1109/TSMCA.2011.2132707.

[Kni+15] Dierk Knig, Paul King, Guillaume Laforge, Hamlet D'Arcy, Cdric Champeau, Erik Pragt, and Jon Skeet. *Groovy in Action*. 2nd. Manning Publications Co., 2015.

[Koc+14] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23(2), 2014, pp. 253–278. DOI: 10.1007/s00778-013-0348-4.

[Kol+13] Dimitrios S. Kolovos et al. A research roadmap towards achieving scalability in model driven engineering. In: *BigMDE at STAF*, p. 2. 2013. DOI: 10.1145/2487766.2487768.

[Kőn31] Dénes Kőnig. Graphen und matrizen (graphs and matrices). *Mat. Lapok (Math Ann.)* 38, 1931, pp. 116–119.

[Kőn36] Denes Kőnig. *Theorie der endlichen und unendlichen Graphen (Theory of finite and infinite graphs).* Akademische Verlagsgesellschaft, Leipzig, 1936.

[Kot+16] Venelin Kotsev, Nikos Minadakis, Vassilis Papakonstantinou, Orri Erling, Irini Fundulaki, and Atanas Kiryakov. Benchmarking RDF query engines: The LDBC Semantic Publishing Benchmark. In: *BLINK at ISWC*, CEUR Workshop Proceedings, vol. 1700, CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1700/paper-01.pdf.

[Kőv15] Zsolt Kővári. Performance Analysis of Graph Queries. Scientific Students' Association report, Budapest University of Technology and Economics. 2015. URL: https://tdk.bme.hu/VIK/Intelligens1/Graflekerdezesek-Teljesitmenyelemzese.

[KPT16] Udayan Khurana, Srinivasan Parthasarathy, and Deepak S. Turaga. Graph-based exploration of non-graph datasets. *VLDB* 9(13), 2016, pp. 1557–1560. URL: http://www.vldb.org/pvldb/vol9/p1557-khurana.pdf.

[KR96] Harumi A. Kuno and Elke A. Rundensteiner. The MultiView OODB view system: Design and implementation. *Theory Pract. Object Syst.* 2(3), 1996, pp. 202–225.

[KR98] Harumi A. Kuno and Elke A. Rundensteiner. Incremental maintenance of materialized object-oriented views in MultiView: Strategies and performance evaluation. *IEEE Trans. Knowl. Data Eng.* 10(5), 1998, pp. 768–792. DOI: 10.1109/69.729731.

[KS02] Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. Syst. Sci.* 65(1), 2002, pp. 150–167. DOI: 10.1006/jcss.2002.1883.

[KSS18] Paraschos Koutris, Semih Salihoglu, and Dan Suciu. Algorithmic aspects of parallel data processing. *Foundations and Trends in Databases* 8(4), 2018, pp. 239–370. DOI: 10.1561/1900000055.

[KSW95] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Inf. Syst.* 20(1), 1995, pp. 21–51. DOI: 10.1016/0306-4379(95)00002-L.

[KTG15] Christian Krause, Matthias Tichy, and Holger Giese. Implementing graph transformations in the bulk synchronous parallel model. In: *Software Engineering & Management*, Lecture Notes in Informatics, vol. 239, pp. 99–100. 2015.

[KVH18] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-level programming abstractions for distributed graph processing. *IEEE Trans. Knowl. Data Eng.* 30(2), 2018, pp. 305–324.

[Lat08] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.* 407(1-3), 2008, pp. 458–473. DOI: 10.1016/j.tcs.2008.07.017.

[Lau+12] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Bidirectional model transformation with precedence triple graph grammars. In: *ECMFA*, Lecture Notes in Computer Science, vol. 7349, pp. 287–302. Springer, 2012. DOI: 10.1007/978-3-642-31491-9_22.

[Law+79] C. L. Lawson, Richard J. Hanson, D. R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.* 5(3), 1979, pp. 308–323. DOI: 10.1145/355841.355847.

[LBV18] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. Beyond macrobenchmarks: Microbenchmark-based graph database evaluation. *PVLDB* 12(4), 2018, pp. 390–403. URL: http://www.vldb.org/pvldb/vol12/p390-lissandrini.pdf.

[LD00] Hartmut Liefke and Susan B. Davidson. View maintenance for hierarchical semistructured data. In: *DaWaK*, Lecture Notes in Computer Science, vol. 1874, pp. 114–125. Springer, 2000. DOI: 10.1007/3-540-44466-1_12.

[Lee+12] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB* 6(2), 2012, pp. 133–144. DOI: 10.14778/2535568.2448946.

[Lee+17] Wilco van Leeuwen, Angela Bonifati, George H. L. Fletcher, and Nikolay Yakovets. Stability notions in synthetic graph generation: A preliminary study. In: *EDBT*, pp. 486–489. 2017. DOI: 10.5441/002/edbt.2017.51.

[Lei+18] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27(5), 2018, pp. 643–668. DOI: 10.1007/s00778-017-0480-7.

[Li+11] Xiang Li, Christoph Quix, David Kensche, Sandra Geisler, and Lisong Guo. Automatic generation of mediated schemas through reasoning over data dependencies. In: *ICDE*, pp. 1280–1283. 2011. DOI: 10.1109/ICDE.2011.5767913.

[Lig18] Lightbend, Inc. Akka Documentation. http://akka.io/. 2018.

[Lin+16] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast in-memory SQL analytics on typed graphs. *VLDB* 10(3), 2016, pp. 265–276. URL: http://www.vldb.org/pvldb/vol10/p265-lin.pdf.

[Lin18] Jimmy Lin. Scale up or scale out for graph processing? *IEEE Internet Computing* 22(3), 2018, pp. 72–78. DOI: 10.1109/MIC.2018.032501520.

[Lin91] Jianhua Lin. Divergence measures based on the Shannon entropy. *IEEE Trans. Information Theory* 37(1), 1991, pp. 145–151. DOI: 10.1109/18.61115.

[LJS16] Bjørnar Luteberget, Christian Johansen, and Martin Steffen. Rule-based consistency checking of railway infrastructure designs. In: *IFM*, pp. 491–507. 2016. DOI: 10.1007/978-3-319-33693-0_31.

[Low+17] Tze Meng Low, Varun Nagaraj Rao, Matthew Lee, Doru-Thom Popovici, Franz Franchetti, and Scott McMillan. First look: Linear algebra-based triangle counting without matrix multiplication. In: *HPEC*, pp. 1–6. IEEE, 2017. DOI: 10.1109/HPEC.2017.8091046.

[LP91] Mark Levene and Alexandra Poulovassilis. An object-oriented data model formalised through hypergraphs. *Data Knowl. Eng.* 6, 1991, pp. 205–234. DOI: 10.1016/0169-023X(91)90005-I.

[LPJ14] Fei Li, Tianyin Pan, and Hosagrahar Visvesvaraya Jagadish. Schema-free SQL. In: *SIGMOD*, pp. 1051–1062. ACM, 2014. DOI: 10.1145/2588555.2588571.

[LRG09] Tihamer Levendovszky, Arend Rensink, and Pieter Van Gorp. International Workshop on Graph-Based Tools (GraBaTs). http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/. 2009.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.* 78(5), 2009, pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004.

[Lu+14] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB* 8(3), 2014, pp. 281–292. DOI: 10.14778/2735508.2735517.

[LV13] Yunkai Liu and Theresa M. Vitolo. Graph data warehouse: Steps to integrating graph databases into the traditional conceptual structure of a data warehouse. In: *BigData Congress*, pp. 433–434. IEEE Computer Society, 2013. DOI: 10.1109/BigData.Congress.2013.72.

[LVM00] Jixue Liu, Millist W. Vincent, and Mukesh K. Mohania. Maintaining views in object-relational databases. In: *CIKM*, pp. 102–109. ACM, 2000. DOI: 10.1145/354756.354807.

[LVM03] Jixue Liu, Millist W. Vincent, and Mukesh K. Mohania. Maintaining views in object-relational databases. *Knowl. Inf. Syst.* 5(1), 2003, pp. 50–82. DOI: 10.1007/s10115-002-0067-z.

[LVM99] Jixue Liu, Millist W. Vincent, and Mukesh K. Mohania. Incremental maintenance of nested relational views. In: *IDEAS*, pp. 197–205. 1999. DOI: 10.1109/IDEAS.1999.787268.

[LYJ08] Yunyao Li, Cong Yu, and H. V. Jagadish. Enabling schema-free XQuery with meaningful query focus. *VLDB J.* 17(3), 2008, pp. 355–377. DOI: 10.1007/s00778-006-0003-4.

[LZ05] Per-Åke Larson and Jingren Zhou. View matching for outer-join views. In: *VLDB*, pp. 445–456. ACM, 2005. URL: http://www.vldb.org/archives/website/2005/program/paper/wed/p445-larson.pdf.

[LZ07a] Per-Åke Larson and Jingren Zhou. Efficient maintenance of materialized outer-join views. In: *ICDE*, pp. 56–65. 2007. DOI: 10.1109/ICDE.2007.367851.

[LZ07b] Per-Åke Larson and Jingren Zhou. View matching for outer-join views. *VLDB J.* 16(1), 2007, pp. 29–53. DOI: 10.1007/s00778-006-0027-9.

[Ma+06a] Li Ma, Yang Yang, Zhaoming Qiu, Guo Tong Xie, Yue Pan, and Shengping Liu. Towards a complete OWL ontology benchmark. In: *ESWC*, pp. 125–139. 2006. DOI: 10.1007/11762256_12.

[Ma+06b] Yutao Ma, Keqing He, Dehui Du, Jing Liu, and Yulan Yan. A complexity metrics set for large-scale object-oriented software systems. In: *CIT*, p. 189. 2006. DOI: 10.1109/CIT.2006.3.

[Ma+16] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. G-SQL: Fast query processing via graph exploration. *VLDB* 9(12), 2016, pp. 900–911. URL: http://www.vldb.org/pvldb/vol9/p900-ma.pdf.

[Mai83] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983. URL: http://web.cecs.pdx.edu/~maier/TheoryBook/TRD.html.

[Mal+10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In: *SIGMOD*, pp. 135–146. ACM, 2010. DOI: 10.1145/1807167.1807184.

[Man+09] Stefan Manegold et al. Repeatability & workability evaluation of SIGMOD 2009. *SIGMOD Record* 38(3), 2009, pp. 40–43. DOI: 10.1145/1815933.1815944.

[Mar+07] Norbert Martinez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, Jordi Nin, Mario-A. Sanchez-Martinez, and Josep-Lluis Larriba-Pey. DEX: High-performance exploration on large graphs for information retrieval. In: *CIKM*, pp. 573–582. 2007. DOI: 10.1145/1321440.1321521.

[MB90] Daniel P. Miranker and David A. Brant. An algorithmic basis for integrating production systems and large databases. In: *ICDE*, pp. 353–360. 1990. DOI: 10.1109/ICDE.1990.113488.

[McB02] Brian McBride. Jena: A semantic web toolkit. *IEEE Internet Computing* 6(6), 2002, pp. 55–59. DOI: 10.1109/MIC.2002.1067737.

[Mey+18] Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. Continuous model validation using reference attribute grammars. In: *SLE*, pp. 70–82. ACM, 2018. DOI: 10.1145/3276604.3276616.

[MGE11] Norbert Martinez-Bazan, Sergio Gómez-Villamor, and Francesc Escale-Claveras. DEX: A high-performance graph database management system. In: *GDM at ICDE*, pp. 124–127. 2011. DOI: 10.1109/ICDEW.2011.5767616.

[MHD05] Yutao Ma, Keqing He, and Dehui Du. A qualitative method for measuring the structural complexity of software systems based on complex networks. In: *APSEC*, pp. 257–263. 2005. DOI: 10.1109/APSEC.2005.14.

[Mir+12] Daniel P. Miranker, Rodolfo K. Depena, Hyunjoon Jung, Juan F. Sequeda, and Carlos Reyna. Diamond: A SPARQL query engine, for Linked Data based on the Rete match. In: *AImWD at ECAI*, 2012.

[Mir90] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. Morgan Kaufmann Publishers Inc., 1990.

[ML91] Daniel P. Miranker and Bernie J. Lofaso. The organization and performance of a TREAT-based production system compiler. *IEEE Trans. Knowl. Data Eng.* 3(1), 1991, pp. 3–10. DOI: 10.1109/69.75882.

[MOG11] MOGENTES project. *Model-based Generation of Tests for Dependable Embedded Systems, 7th EU Framework Programme*. http://mogentes.eu/. 2011.

[Mom00] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, 2000.

[Mon+08] Martin Monperrus, Jean-Marc Jézéquel, Joël Champeau, and Brigitte Hoeltzener. Measuring models. In: *Model-Driven Software Development: Integrating Quality Assurance*, IDEA Group, 2008. DOI: 10.4018/978-1-60566-006-6.ch007.

[MON16] MONDO Project. *Scalable Modeling and Model Management on the Cloud Project, 7th EU Framework Programme*. http://www.mondo-project.org/. 2016.

[Mor+11] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. DBpedia SPARQL benchmark - Performance assessment with real queries on real data. In: *ISWC*, pp. 454–469. 2011. DOI: 10.1007/978-3-642-25073-6_29.

[MPM11] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. In: *PODC*, pp. 207–208. ACM, 2011.

[MQM97] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In: *SIGMOD*, pp. 100–111. ACM Press, 1997. DOI: 10.1145/253260.253277.

[MRV10] Steffen Mazanek, Arend Rensink, and Pieter Van Gorp. 4th Transformation Tool Contest, 2010. URL: https://ris.utwente.nl/ws/files/5096115/wp10-03.pdf.

[MSW98] Manfred Münch, Andy Schürr, and Andreas J. Winter. Integrity constraints in the multi-paradigm language PROGRES. In: *TAGT (Theory and Application of Graph Transformations)*, Lecture Notes in Computer Science, vol. 1764, pp. 338–351. Springer, 1998. DOI: 10.1007/978-3-540-46464-8_24.

[MTA18] MTA-BME. *Lendület Cyber-Physical Systems Research Group.* http://lendulet.inf.mit.bme.hu/. 2018.

[Mur+13] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In: *SOSP*, pp. 439–455. 2013. DOI: 10.1145/2517349.2522738.

[Mur+16] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM* 59(10), 2016, pp. 75–83. DOI: 10.1145/2983551.

[Mus14] Benjamin Muschko. *Gradle in Action.* 1st. Manning Publications Co., 2014.

[MW95] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.* 24(6), 1995, pp. 1235–1258. DOI: 10.1137/S009753979122370X.

[MWM15] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.* 48(2), 2015, 25:1–25:39. DOI: 10.1145/2818185.

[Nai+15] Lifeng Nai, Yinglong Xia, Ilie Gabriel Tanase, Hyesoon Kim, and Ching-Yung Lin. GraphBIG: Understanding graph computing in the context of industrial solutions. In: *SC*, 69:1–69:12. 2015. DOI: 10.1145/2807591.2807626.

[NEK14] Milos Nikolic, Mohammed Elseidy, and Christoph Koch. LINVIEW: Incremental view maintenance for complex analytical queries. In: *SIGMOD*, pp. 253–264. 2014. DOI: 10.1145/2588555.2610519.

[New03] Mark E. J. Newman. The structure and function of complex networks. *SIAM Review* 45(2), 2003, pp. 167–256. DOI: 10.1137/S003614450342480.

[Ngo+14] Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra. Beyond worst-case analysis for joins with Minesweeper. In: *PODS*, pp. 234–245. ACM, 2014. DOI: 10.1145/2594538.2594547.

[Ngo+18] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM* 65(3), 2018, 16:1–16:40. DOI: 10.1145/3180143.

[Ngu+15a] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In: *GRADES at SIGMOD*, 2:1–2:8. ACM, 2015. DOI: 10.1145/2764947.2764948.

[Ngu+15b] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. *CoRR* abs/1503.04169, 2015. URL: http://arxiv.org/abs/1503.04169.

[Nic+13] Vincenzo Nicosia, Ginestra Bianconi, Vito Latora, and Marc Barthelemy. Growing multiplex networks. *Phys. Rev. Lett.* 111, 5 2013, p. 058701. DOI: 10.1103/PhysRevLett.111.058701.

[Nic12] Anisoara Nica. Incremental maintenance of materialized views with outerjoins. *Inf. Syst.* 37(5), 2012, pp. 430–442. DOI: 10.1016/j.is.2011.06.001.

[Nik16] Milos Nikolic. Efficient Incremental Data Analysis. Ph.D. dissertation. École Polytechnique Fédérale de Lausanne, 2016. URL: https://infoscience.epfl.ch/record/220881/files/EPFL_TH7183.pdf.

[NL15] Vincenzo Nicosia and Vito Latora. Measuring and modeling correlations in multiplex networks. *Phys. Rev. E* 92, 3 2015, p. 032805. DOI: 10.1103/PhysRevE.92.032805.

[NLP13] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In: *SIGOPS*, pp. 456–471. ACM, 2013. DOI: 10.1145/2517349.2522739.

[NM09] Thomas Neumann and Guido Moerkotte. A framework for reasoning about share equivalence and its integration into a plan generator. In: *BTW*, pp. 7–26. 2009. URL: http://subs.emis.de/LNI/Proceedings/Proceedings144/article5220.html.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In: *ICSE*, pp. 742–745. 2000. DOI: 10.1145/337180.337620.

[NO18] Milos Nikolic and Dan Olteanu. Incremental view maintenance with triple lock factorization benefits. In: *SIGMOD*, pp. 365–380. 2018. DOI: 10.1145/3183713.3183758.

[NP06] Raghunath Othayoth Nambiar and Meikel Pöss. The making of TPC-DS. In: *VLDB*, pp. 1049–1058. 2006. URL: http://dl.acm.org/citation.cfm?id=1164217.

[NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Record* 42(4), 2013, pp. 5–16. DOI: 10.1145/2590989.2590991.

[Obj12] Object Management Group. *Object Constraint Language Specification (Version 2.3.1)*. http://www.omg.org/spec/OCL/2.3.1/. 2012.

[ONe08] Elizabeth J. O'Neil. Object/relational mapping 2008: Hibernate and the Entity Data Model (EDM). In: *SIGMOD*, pp. 1351–1356. ACM, 2008. DOI: 10.1145/1376616.1376773.

[Özs16] M. Tamer Özsu. A survey of RDF data management systems. *Frontiers Comput. Sci.* 10(3), 2016, pp. 418–432. DOI: 10.1007/s11704-016-5554-y.

[Pac+17] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. Do we need specialized graph databases? Benchmarking real-time social networking applications. In: *GRADES at SIGMOD*, 12:1–12:7. 2017. DOI: 10.1145/3078447.3078459.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34(3), 2009, 16:1–16:45. DOI: 10.1145/1567274.1567278.

[Pai+09] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nikolaos Drivalos, and Fiona A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In: *ICECCS*, pp. 162–171. IEEE, 2009. DOI: 10.1109/ICECCS.2009.14.

[Pal+02] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental maintenance for non-distributive aggregate functions. In: *VLDB*, pp. 802–813. 2002. URL: http://www.vldb.org/conf/2002/S22P04.pdf.

[Par+17] Marcus Paradies, Cornelia Kinder, Jan Bross, Thomas Fischer, Romans Kasperovics, and Hinnerk Gildhoff. GraphScript: Implementing complex graph algorithms in SAP HANA. In: *DBPL*, 13:1–13:4. 2017. DOI: 10.1145/3122831.3122841.

[Pat12] David A. Patterson. For better or worse, benchmarks shape a field: Technical perspective. *Commun. ACM* 55(7), 2012, p. 104. DOI: 10.1145/2209249.2209271.

[PD14] Arnau Prat-Pérez and David Dominguez-Sal. How community-like is the structure of synthetically generated graphs? In: *GRADES at SIGMOD*, 7:1–7:9. 2014. DOI: 10.1145/2621934.2621942.

[PD99] Norman W. Paton and Oscar Diaz. Active database systems. *ACM Comput. Surv.* 31(1), 1999, pp. 63–103. DOI: 10.1145/311531.311623.

[Pet+14] Martin Peters, Christopher Brink, Sabine Sachweh, and Albert Zündorf. Scaling parallel rule-based reasoning. In: *ESWC*, Lecture Notes in Computer Science, vol. 8465, pp. 270–285. Springer, 2014. DOI: 10.1007/978-3-319-07443-6_19.

[PF00] Meikel Pöss and Chris Floyd. New TPC benchmarks for decision support and web commerce. *SIGMOD Record* 29(4), 2000, pp. 64–71. DOI: 10.1145/369275.369291.

[PFH12] Jan Peleska, Johannes Feuser, and Anne Elisabeth Haxthausen. The model-driven openETCS paradigm for secure, safe and certifiable train control systems. In: *Railway Safety, Reliability and Security: Technologies and System Engineering*. IGI global, 2012, pp. 22–52. DOI: 10.4018/978-1-4666-1643-1.ch002.

[Phu+11] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and Linked Data. In: *ISWC*, pp. 370–388. 2011. DOI: 10.1007/978-3-642-25073-6_24.

[Pös17] Meikel Pöss. Methodologies for a Comprehensive Approach to Measuring the Performance of Decision Support Systems. Ph.D. dissertation. Technical University Munich, 2017. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20171222-1401796-1-3.

[Pra+17] Arnau Prat-Pérez, Joan Guisado-Gámez, Xavier Fernández Salas, Petr Koupy, Siegfried Depner, and Davide Basilio Bartolini. Towards a property graph generator for benchmarking. In: *GRADES at SIGMOD*, 6:1–6:6. ACM, 2017.

[Pro11] Mark Proctor. Drools: A rule engine for complex event processing. In: *AGTIVE*, 2011. DOI: 10.1007/978-3-642-34176-2_2.

[PSH17] Haoyue Ping, Julia Stoyanovich, and Bill Howe. DataSynthesizer: Privacy-Preserving synthetic datasets. In: *SSDBM*, 42:1–42:5. ACM, 2017.

[Qi+13] Zichao Qi, Yanghua Xiao, Bin Shao, and Haixun Wang. Toward a distance oracle for billion-node graphs. *PVLDB* 7(1), 2013, pp. 61–72. DOI: 10.14778/2732219.2732225.

[Qua96] Dallan Quass. Maintenance expressions for views with aggregation. In: *VIEWS at SIGMOD*, pp. 110–118. 1996.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. Knowl. Data Eng.* 3(3), 1991, pp. 337–341. DOI: 10.1109/69.91063.

[R C18] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. 2018. URL: https://www.R-project.org.

[Raa+18] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. Fair benchmarking considered difficult: Common pitfalls in database performance testing. In: *DBTest at SIGMOD*, 2:1–2:6. ACM, 2018. DOI: 10.1145/3209950.3209955.

[Rah+01] J. Wenny Rahayu, Elizabeth Chang, Tharam S. Dillon, and David Taniar. Performance evaluation of the object-relational transformation methodology. *Data Knowl. Eng.* 38(3), 2001, pp. 265–300. DOI: 10.1016/S0169-023X(01)00026-X.

[Ram+09] Sunitha Ramanujam, Anubha Gupta, Latifur Khan, Bhavani M. Thuraisingham, and Steven Seida. R2D: A framework for the relational transformation of RDF data. *Int. J. Semantic Computing* 3(4), 2009, pp. 471–498. DOI: 10.1142/S1793351X09000884.

[Rát11] István Ráth. Event-driven model transformations in domain-specific modeling languages. Ph.D. dissertation. Budapest University of Technology and Economics, 2011. URL: http://mit.bme.hu/~rath/pub/phd/phd-thesis-rath.pdf.

[Ray99] Eric S. Raymond. *The cathedral and the bazaar - Musings on Linux and Open Source by an accidental revolutionary*. O'Reilly, 1999.

[RE12] Alexander Reder and Alexander Egyed. Incremental consistency checking for complex design rules and larger model changes. In: *MODELS*, pp. 202–218. 2012. DOI: 10.1007/978-3-642-33666-9_14.

[Ren+17] Xiangnan Ren, Olivier Curé, Li Ke, Jérémy Lhez, Badre Belabbess, Tendry Randriamalala, Yufan Zheng, and Gabriel Képéklian. Strider: An adaptive, inference-enabled distributed RDF stream processing engine. *VLDB* 10(12), 2017, pp. 1905–1908. URL: http://www.vldb.org/pvldb/vol10/p1905-ren.pdf.

[Ren03] Arend Rensink. The GROOVE simulator: A tool for state space generation. In: *AGTIVE*, pp. 479–485. 2003. DOI: 10.1007/978-3-540-25959-6_40.

[Res+16] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A property graph query language. In: *GRADES at SIGMOD*, 2016. DOI: 10.1145/2960414.2960421.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. 3rd. McGraw-Hill, 2003.

[RG10] Arend Rensink and Pieter Van Gorp. Graph transformation tool contest 2008. *Softw. Tools Technol. Transfer* 12(3-4), 2010, pp. 171–181. DOI: 10.1007/s10009-010-0157-7.

[RG98] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In: *ER*, Lecture Notes in Computer Science, vol. 1507, pp. 449–464. Springer, 1998. DOI: 10.1007/978-3-540-49524-6_35.

[RHK15] Louis M. Rose, Tassilo Horn, and Filip Krikava, eds. *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015*. Vol. 1524. CEUR Workshop Proceedings. CEUR-WS.org, 2015. URL: http://ceur-ws.org/Vol-1524.

[RKH14] Louis M. Rose, Christian Krause, and Tassilo Horn, eds. *Transformation Tool Contest*. Vol. 1305. CEUR Workshop Proceedings. CEUR-WS.org, 2014. URL: http://ceur-ws.org/Vol-1305.

[RNT12] Mikko Rinne, Esko Nuutila, and Seppo Törmä. INSTANS: High-performance event processing with standard RDF and SPARQL. In: *ISWC*, CEUR Workshop Proceedings, vol. 914, CEUR-WS.org, 2012. URL: http://ceur-ws.org/Vol-914/paper_22.pdf.

[Roc+14] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Mining metrics for understanding metamodel characteristics. In: *MiSE at ICSE*, pp. 55–60. 2014. DOI: 10.1145/2593770.2593774.

[Roc+15] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Mining correlations of ATL model transformation and metamodel metrics. In: *MiSE at ICSE*, pp. 54–59. 2015. DOI: 10.1109/MiSE.2015.17.

[Rod08a] Liam Roditty. A faster and simpler fully dynamic transitive closure. *ACM Trans. Algorithms* 4(1), 2008, 6:1–6:16.

[Rod08b] Marko A. Rodriguez. A collectively generated model of the world. In: Hassan Masum and Yochai Benkler (eds.), *Collective intelligence: Creating a prosperous world at peace*, pp. 261–264. Oakton: Earth Intelligence Network, 2008.

[Rod15] Marko A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In: *DBPL*, pp. 1–10. 2015. DOI: 10.1145/2815072.2815073.

[Ros+86] Arnon Rosenthal, Sandra Heiler, Umeshwar Dayal, and Frank Manola. Traversal recursion: A practical approach to supporting recursive applications. In: *SIGMOD*, pp. 166–176. ACM Press, 1986. DOI: 10.1145/16894.16871.

[Roz97] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[RS14] Yves Raimond and Guus Schreiber. *RDF 1.1 Primer*. W3C Note. W3C, 2014. URL: http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In: *SIGMOD*, pp. 447–458. 1996. DOI: 10.1145/233269.233361.

[Rud+13] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the SAP HANA database. In: *BTW*, pp. 403–420. 2013. URL: http://www.btw-2013.de/proceedings/The%20Graph%20Story%20of%20the%20SAP%20HANA%20Database.pdf.

[Rum+91] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William E. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[RVV08] István Ráth, David Vago, and Dániel Varró. Design-time simulation of domain-specific models by incremental pattern matching. In: *VL/HCC*, pp. 219–222. IEEE Computer Society, 2008. DOI: 10.1109/VLHCC.2008.4639089.

[RWE15] Ian Robinson, Jim Webber, and Emil Eifrém. *Graph Databases*. 2nd. O'Reilly Media, 2015.

[Saa03] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003. DOI: 10.1137/1.9780898718003.

[SAE09] SAE - Radio Technical Commission for Aeronautic. Architecture Analysis & Design Language (AADL) v2, AS-5506A, SAE International. 2009.

[Sah+17] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *VLDB* 11(4), 2017, pp. 420–431. URL: http://www.vldb.org/pvldb/vol11/p420-sahu.pdf.

[Sak+16] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. *Large-Scale Graph Processing Using Apache Giraph*. Springer, 2016. DOI: 10.1007/978-3-319-47431-1.

[Sat+17] Arun V. Sathanur, Sutanay Choudhury, Cliff Joslyn, and Sumit Purohit. When labels fall short: Property graph simulation via blending of network structure and vertex attributes. In: *CIKM*, pp. 2287–2290. 2017. DOI: 10.1145/3132847.3133065.

[Sax15] Gaurav Saxena. GraphIVM: Accelerating Incremental View Maintenance through Non-relational Caching. Master's thesis. University of California, San Diego, 2015. URL: https://www.slideshare.net/gsaxena81/graphivm-accelerating-ivmthrough-nonrelational-caching-49866935.

[Sch+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In: *VLDB*, pp. 974–985. Morgan Kaufmann, 2002. URL: http://www.vldb.org/conf/2002/S30P01.pdf.

[Sch+09] Michael Schmidt, Thomas Hornung, Michael Meier, Christoph Pinkel, and Georg Lausen. SP^{2}Bench: A SPARQL performance benchmark. In: Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca (eds.), *Semantic Web Information Management - A Model-Based Perspective*, pp. 371–393. 2009. DOI: 10.1007/978-3-642-04329-1_16.

[Sch+12] Markus Scheidgen, Anatolij Zubow, Joachim Fischer, and Thomas H. Kolbe. Automated and transparent model fragmentation for persisting large models. In: *MODELS*, pp. 102–118. 2012. DOI: 10.1007/978-3-642-33666-9_8.

[SEH12] Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-SPARQL: A hybrid engine for querying large attributed graphs. In: *CIKM*, pp. 335–344. 2012. DOI: 10.1145/2396761.2396806.

[Sel+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In: *SIGMOD*, pp. 23–34. ACM, 1979. DOI: 10.1145/582095.582099.

[Sel+99] Margo I. Seltzer, David Krinsky, Keith A. Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In: *Proceedings of The 7th Workshop on Hot Topics in Operating Systems*, pp. 102–109. 1999. DOI: 10.1109/HOTOS.1999.798385.

[Sem19] Oszkár Semeráth. Formal Validation and Model Generation for Domain-Specic Languages by Logic Solvers. Ph.D. dissertation. Budapest University of Technology and Economics, 2019. URL: http://hdl.handle.net/10890/13134.

[Sey+16] Daniel Seybold, Jörg Domaschka, Alessandro Rossini, Christopher B. Hauser, Frank Griesinger, and Athanasios Tsitsipas. Experiences of models@run-time with EMF and CDO. In: *SLE*, pp. 46–56. ACM, 2016. URL: http://dl.acm.org/citation.cfm?id=2997380.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 1st. Addison-Wesley Professional, 2012.

[Sha+14] Seyyed M. Shah, Ran Wei, Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Konstantinos Barmpis. A framework to benchmark NoSQL data stores for large-scale model persistence. In: *MODELS*, pp. 586–601. 2014. DOI: 10.1007/978-3-319-11653-2_36.

[Sha+17] Bin Shao, Yatao Li, Haixun Wang, and Huanhuan Xia. Trinity graph engine and its applications. *IEEE Data Eng. Bull.* 40(3), 2017, pp. 18–29. URL: http://sites.computer.org/debull/A17sept/p18.pdf.

[Shi+17] Chuan Shi, Yitong Li, Jiawei Zhang an Yizhou Sun, and Philip S. Yu. A survey of heterogeneous information network analysis. *IEEE Trans. Knowl. Data Eng.* 29(1), 2017, pp. 17–37. DOI: 10.1109/TKDE.2016.2598561.

[Shi+18] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on GPUs: A survey. *ACM Comput. Surv.* 50(6), 2018, 81:1–81:35. DOI: 10.1145/3128571.

[Sid+16] Kamran Siddique, Zahid Akhtar, Edward J. Yoon, Young-Sik Jeong, Dipankar Dasgupta, and Yangwoo Kim. Apache Hama: An emerging bulk synchronous parallel computing framework for big data applications. *IEEE Access* 4, 2016, pp. 8879–8887. DOI: 10.1109/ACCESS.2016.2631549.

[SJ09] Jean-Sébastien Sottet and Frédéric Jouault. Program comprehension. In: *International Workshop on Graph-Based Tools (GraBaTs)*, 2009. URL: https://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009reverseengineering.pdf.

[SK08] Andy Schürr and Felix Klar. 15 years of Triple Graph Grammars. In: *ICGT*, Lecture Notes in Computer Science, vol. 5214, pp. 411–425. Springer, 2008. DOI: 10.1007/978-3-540-87405-8_28.

[SKS05] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 5th. McGraw-Hill Book Company, 2005.

[SLO18] Sven Schneider, Leen Lambers, and Fernando Orejas. Automated reasoning for attributed graph properties. *Softw. Tools Technol. Transfer* 20(6), 2018, pp. 705–737. DOI: 10.1007/s10009-018-0496-3.

[SM96] Michael Stonebraker and Dorothy Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, 1996.

[Sni12] Tom A.B. Snijders. Transitivity and Triads. http://www.stats.ox.ac.uk/~snijders/Trans_Triads_ha.pdf. Lecture at the University of Oxford. 2012.

[SNZ08] Andy Schürr, Manfred Nagl, and Albert Zündorf, eds. *Applications of Graph Transformations with Industrial Relevance*. Vol. 5088. LNCS. Springer, 2008.

[SÖ18] Semih Salihoglu and M. Tamer Özsu. Response to "scale up or scale out for graph processing". *IEEE Internet Computing* 22(5), 2018, pp. 18–24. DOI: 10.1109/MIC.2018.053681359.

[SP08] Andy Seaborne and Eric Prud'hommeaux. *SPARQL Query Language for RDF*. W3C Recommendation. W3C, 2008. URL: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[Ste+09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009.

[Ste+17] Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Félix Cuadrado, Luis M. Vaquero, and Joan Varvenne. Cytosm: Declarative property graph queries without data migration. In: *GRADES at SIGMOD*, 4:1–4:6. 2017. DOI: 10.1145/3078447.3078451.

[Sto75] Michael Stonebraker. Implementation of integrity constraints and views by query modification. In: *SIGMOD*, pp. 65–78. ACM, 1975. DOI: 10.1145/500080.500091.

[Suk+16] Sreenivas R. Sukumar, Seokyong Hong, Sangkeun Lee, and Seung-Hwan Lim. GraphBench. 2016. URL: https://www.osti.gov/biblio/1324313.

[Sun+15a] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. SQLGraph: An efficient relational-based property graph store. In: *SIGMOD*, pp. 1887–1901. 2015. DOI: 10.1145/2723372.2723732.

[Sun+15b] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *PVLDB* 8(11), 2015, pp. 1214–1225. DOI: 10.14778/2809974.2809983.

[SW13] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. In: *SSDBM*, 22:1–22:12. 2013. DOI: 10.1145/2484838.2484843.

[SWL13] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In: *SIGMOD*, pp. 505–516. 2013. DOI: 10.1145/2463676.2467799.

[SWZ99] A. Schürr, A. J. Winter, and A. Zündorf. Handbook of graph grammars and computing by graph transformation. In: H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), pp. 487–550. World Scientific Publishing Co., Inc., 1999. URL: http://dl.acm.org/citation.cfm?id=328523.328617.

[Syr00] Apostolos Syropoulos. Mathematics of multisets. In: *Workshop on Multiset Processing (WMP)*, Lecture Notes in Computer Science, vol. 2235, pp. 347–358. Springer, 2000. DOI: 10.1007/3-540-45523-X_17.

[Sza+18] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in Datalog. *PACMPL/OOPSLA* 2, 2018, 139:1–139:29. DOI: 10.1145/3276509.

[Sza17] Zoltán Szatmári. Metamodel-Based Model Generation and Validation Techniques with Applications. Ph.D. dissertation. Budapest University of Technology and Economics, 2017. URL: https://repozitorium.omikk.bme.hu/handle/10890/5374.

[Szu+12] Aneta Szumowska, Marta Burzanska, Piotr Wisniewski, and Krzysztof Stencel. Extending HQL with plain recursive facilities. In: *ADBIS*, Advances in Intelligent Systems and Computing, vol. 186, pp. 265–272. Springer, 2012. DOI: 10.1007/978-3-642-32741-4_24.

[Tae+07] Gabriele Taentzer et al. Generation of Sierpinski triangles: A case study for graph transformation tools. In: *AGTIVE*, pp. 514–539. 2007. DOI: 10.1007/978-3-540-89020-1_35.

[Tan+14] Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, Yanbin Liu, Wei Tan, Jason Crawford, and Ching-Yung Lin. In: *GRADES at SIGMOD*, 10:1–10:6. CWI/ACM, 2014. DOI: 10.1145/2621934.2621945.

[TB07] Vadim Tropashko and Donald Burleson. *SQL Design Patterns: Expert Guide to SQL Programming*. Rampant TechPress, 2007.

[Tet+18] Frank Tetzel, Hannes Voigt, Marcus Paradies, Romans Kasperovics, and Wolfgang Lehner. Analysis of data structures involved in RPQ evaluation. In: *DATA*, pp. 334–343. SciTePress, 2018. DOI: 10.5220/0006860303340343.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In: *TACAS*, pp. 632–647. 2007. DOI: 10.1007/978-3-540-71209-1_49.

[TK15] Immanuel Trummer and Christoph Koch. An incremental anytime algorithm for multi-objective query optimization. In: *SIGMOD*, pp. 1941–1953. 2015. DOI: 10.1145/2723372.2746484.

[TPC10] TPC (Transaction Processing Performance Council). TPC Benchmark C, revision 5.11, 2010, pp. 1–132. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[TPC17] TPC (Transaction Processing Performance Council). TPC Benchmark H, revision 2.17.3, 2017, pp. 1–137. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf.

[TPC18] TPC (Transaction Processing Performance Council). TPC Benchmark DS, revision 2.10.0, 2018, pp. 1–137. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.10.0.pdf.

[TWL12] Lei Tang, Xufei Wang, and Huan Liu. Community detection via heterogeneous interaction analysis. *Data Min. Knowl. Discov.* 25(1), 2012, pp. 1–33. DOI: 10.1007/s10618-011-0231-0.

[UGH11] Axel Uhl, Thomas Goldschmidt, and Manuel Holzleitner. Using an OCL impact analysis algorithm for view-based textual modelling. In: *OCL and Textual Modelling (OCL)*, ECEASST, vol. 44, 2011.

[Ujh+15a] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98, 2015, pp. 80–99. DOI: 10.1016/j.scico.2014.01.004.

[Ujh+15b] Zoltán Ujhelyi, Gábor Szőke, Ákos Horváth, Norbert István Csiszár, László Vidács, Dániel Varró, and Rudolf Ferenc. Performance comparison of query-based techniques for anti-pattern detection. *Inf. Softw. Technol.* 65, 2015, pp. 147–165. DOI: 10.1016/j.infsof.2015.01.003.

[Ull76] Julian R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM* 23(1), 1976, pp. 31–42. DOI: 10.1145/321921.321925.

[URI01] URI Planning Interest Group. *URIs, URLs, and URNs: Clarifications and Recommendations 1.0.* W3C Note. W3C, 2001. URL: http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/.

[Uta+18] Alexandru Uta, Sietse Au, Alexey Ilyushkin, and Alexandru Iosup. Elasticity in graph analytics? A benchmarking framework for elastic graph processing. In: *IEEE CLUSTER*, pp. 381–391. IEEE Computer Society, 2018. DOI: 10.1109/CLUSTER.2018.00056.

[Var+15] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Softw. Syst. Model.* 14(2), 2015, pp. 597–621. DOI: 10.1007/s10270-013-0372-2.

[Var+16] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: Three generations of the VIATRA framework. *Softw. Syst. Model.* 15(3), 2016, pp. 609–629. DOI: 10.1007/s10270-016-0530-4.

[Var08] Gergely Varró. Advanced Techniques for the Implementation of Model Transformation Systems. Ph.D. dissertation. Budapest University of Technology and Economics, 2008. URL: http://www.cs.bme.hu/~gervarro/thesis/Gergely_Varro-PhDThesis.pdf.

[Vár18] Petra Várhegyi. Matrix-Based Analysis of Multiplex Graphs. Scientific Students' Association report, Budapest University of Technology and Economics. 2018. URL: http://tdk.bme.hu/VIK/information/Tipusos-grafok-hatekony-elemzese.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.* 68(3), 2007, pp. 214–234. DOI: 10.1016/j.scico.2007.05.004.

[VD13] Gergely Varró and Frederik Deckwerth. A Rete network construction algorithm for incremental pattern matching. In: *ICMT*, pp. 125–140. 2013. DOI: 10.1007/978-3-642-38883-5_13.

[Vel13] Todd L. Veldhuizen. Incremental maintenance for Leapfrog Triejoin. *CoRR* abs/1303.5313, 2013. URL: http://arxiv.org/abs/1303.5313.

[VFV06a] Gergely Varró, Katalin Friedl, and Dániel Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electr. Notes Theor. Comput. Sci.* 152, 2006, pp. 191–205. DOI: 10.1016/j.entcs.2005.10.025.

[VFV06b] Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a graph transformation engine in relational databases. *Software and System Modeling* 5(3), 2006, pp. 313–341. DOI: 10.1007/s10270-006-0015-y.

[Vid+17] Vânia Maria P. Vidal, Narciso Moura Arruda Jr., Matheus Cruz, Marco Antonio Casanova, Carlos Eduardo Brito, and Valéria Magalhães Pequeno. Computing changesets for RDF views of relational data. In: *MEPDaW/LDQ at ESWC*, CEUR Workshop Proceedings, vol. 1824, pp. 43–58. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-1824/mepdaw_paper_4.pdf.

[VMR11] Pieter Van Gorp, Steffen Mazanek, and Louis M. Rose, eds. *Transformation Tool Contest.* Vol. 74. EPTCS. 2011. DOI: 10.4204/EPTCS.74.

[VMT13] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Converting relational to graph databases. In: *GRADES at SIGMOD*, 2013. DOI: 10.1145/2484425.2484426.

[VMT14] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. R2G: A tool for migrating relations to graphs. In: *EDBT*, pp. 640–643. 2014. DOI: 10.5441/002/edbt.2014.63.

[VP03] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The mathematics of metamodeling is metamodeling mathematics). *Softw. Syst. Model.* 2(3), 2003, pp. 187–210. DOI: 10.1007/s10270-003-0028-8.

[VR13] Pieter Van Gorp and Louis M. Rose. The Petri-nets to statecharts transformation case. In: *Transformation Tool Contest*, pp. 16–31. 2013. DOI: 10.4204/EPTCS.135.3.

[VRK13] Pieter Van Gorp, Louis M. Rose, and Christian Krause, eds. *Transformation Tool Contest.* Vol. 135. EPTCS. 2013. DOI: 10.4204/EPTCS.135.

[VSV05] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In: *VL/HCC*, pp. 79–88. 2005. DOI: 10.1109/VLHCC.2005.23.

[Wan+17] Yangzihao Wang et al. Gunrock: GPU graph analytics. *TOPC* 4(1), 2017, 3:1–3:49. DOI: 10.1145/3108140.

[WBS15] Michael M. Wolf, Jonathan W. Berry, and Dylan T. Stark. A task-based linear algebra building blocks approach for scalable graph analytics. In: *HPEC*, pp. 1–6. 2015. DOI: 10.1109/HPEC.2015.7322450.

[WC96] Jennifer Widom and Stefano Ceri, eds. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

[Web12] Jim Webber. A programmatic introduction to Neo4j. In: *SPLASH*, pp. 217–218. 2012. DOI: 10.1145/2384716.2384777.

[WF94] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*. Cambridge University Press, 1994.

[WH92] Yu-Wang Wang and Eric N. Hanson. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In: *ICDE*, pp. 88–97. 1992. DOI: 10.1109/ICDE.1992.213202.

[Whi15] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*. 4th. O'Reilly, 2015. URL: http://www.oreilly.de/catalog/9781491901632/index.html.

[WHR14] Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software* 31(3), 2014, pp. 79–85. DOI: 10.1109/MS.2013.65.

[Wid06] Jennifer Widom. Tips for Writing Technical Papers. Presented at Stanford InfoLab. 2006. URL: https://cs.stanford.edu/people/widom/paper-writing.html.

[WS98] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature* (393), 1998, pp. 440–442.

[Wu+08] Xindong Wu et al. Top 10 algorithms in data mining. *Knowl. Inf. Syst.* 14(1), 2008, pp. 1–37. DOI: 10.1007/s10115-007-0114-2.

[Wyl+18] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. RDF data storage and query processing schemes: A survey. *ACM Comput. Surv.* 51(4), 2018, 84:1–84:36. DOI: 10.1145/3177850.

[XD17] Konstantinos Xirogiannopoulos and Amol Deshpande. Extracting and analyzing hidden graphs from relational databases. In: *SIGMOD*, pp. 897–912. 2017. DOI: 10.1145/3035918.3035949.

[Xin+13] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on Spark. In: *GRADES at SIGMOD*, 2013. DOI: 10.1145/2484425.2484427.

[XSD17] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. GraphGen: Adaptive graph processing using relational databases. In: *GRADES at SIGMOD*, 2017. DOI: 10.1145/3078447.3078456.

[Yak18] Yakindu. *Statechart Tools*. http://statecharts.org/. 2018.

[Yan+17] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases* 7(1-2), 2017, pp. 1–195. DOI: 10.1561/1900000056.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In: *VLDB*, pp. 82–94. 1981.

[YCZ12] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: A scalable index for reachability queries in very large graphs. *VLDB J.* 21(4), 2012, pp. 509–534. DOI: 10.1007/s00778-011-0256-4.

[Yi+03] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient maintenance of materialized top-k views. In: *ICDE*, pp. 189–200. 2003. DOI: 10.1109/ICDE.2003.1260792.

[Yie+12] Andrés Yie, Rubby Casallas, Dirk Deridder, and Dennis Wagelaar. Realizing model transformation chain interoperability. *Softw. Syst. Model.* 11(1), 2012, pp. 55–75. DOI: 10.1007/s10270-010-0179-3.

[You+18] Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. GraphRNN: Generating realistic graphs with deep auto-regressive models. In: *ICML*, JMLR Workshop and Conference Proceedings, vol. 80, pp. 5694–5703. 2018. URL: http://proceedings.mlr.press/v80/you18a.html.

[Zah+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In: *HotCloud at USENIX*, 2010. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

[ZCC95] Mansour Zand, Val Collins, and Dale Caviness. A survey of current object-oriented databases. *DATA BASE Advances* 26(1), 1995, pp. 14–29. DOI: 10.1145/206476.206480.

[Zen+13] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *VLDB* 6(4), 2013, pp. 265–276. URL: http://www.vldb.org/pvldb/vol6/p265-zeng.pdf.

[Zha+11] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: On warehousing and OLAP multidimensional networks. In: *SIGMOD*, pp. 853–864. ACM, 2011. DOI: 10.1145/1989323.1989413.

[Zha+15] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Trans. Knowl. Data Eng.* 27(7), 2015, pp. 1920–1948. DOI: 10.1109/TKDE.2015.2427795.

[Zha+17] Weijie Zhao, Florin Rusu, Bin Dong, Kesheng Wu, and Peter Nugent. Incremental view maintenance over array data. In: *SIGMOD*, pp. 139–154. 2017. DOI: 10.1145/3035918.3064041.

[Zha+18a] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. UniBench: A benchmark for multi-model database management systems. In: *TPCTC*, Lecture Notes in Computer Science, vol. 11135, pp. 7–23. Springer, 2018. DOI: 10.1007/978-3-030-11404-6_2.

[Zha+18b] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. GraphIt: A high-performance graph DSL. *PACMPL/OOPSLA* 2, 2018, 121:1–121:30. DOI: 10.1145/3276491.

[Zin+16] Daniel Zinn, Haicheng Wu, Jin Wang, Molham Aref, and Sudhakar Yalamanchili. General-purpose join algorithms for large graph triangle listing on heterogeneous systems. In: *GPGPU at PPoPP*, pp. 12–21. ACM, 2016. DOI: 10.1145/2884045.2884054.

[ZN08] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In: *ICSE*, pp. 531–540. 2008. DOI: 10.1145/1368088.1368161.

[ZPR02] Xin Zhang, Bradford Pielech, and Elke A. Rundensteiner. Honey, I shrunk the XQuery! – An XML algebra optimization approach. In: *WIDM at CIKM*, pp. 15–22. 2002. DOI: 10.1145/584931.584936.

[ZT16] J. W. Zhang and Y. C. Tay. GSCALER: Synthetically scaling a given graph. In: *EDBT*, pp. 53–64. 2016. DOI: 10.5441/002/edbt.2016.08.

[Zün08] Albert Zündorf. AntWorld benchmark specification. In: *International Workshop on Graph-Based Tools (GraBaTs)*, 2008. URL: http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2008performancecase.pdf.

[ZY17] Kangfei Zhao and Jeffrey Xu Yu. Graph processing in RDBMSs. *IEEE Data Eng. Bull.* 40(3), 2017, pp. 6–17. URL: http://sites.computer.org/debull/A17sept/p6.pdf.

Appendices

Appendix A

Untyped Graph Metrics

In this chapter, we review common metrics used to characterize untyped graphs. These metrics complement the ones listed in Sec. 3.4, and while they are not in the focus of this dissertation, we included them for reference and completeness.

A.1 Connectivity Metrics

We first introduce the concept of *reachability* (sometimes called *transitive reachability*) that allows the description of connectedness between nodes.

Definition 58 (reachability or connectedness of nodes) Node u is *reachable* from node v in the graph if there is a path (Def. 23) between the two nodes. In other words, nodes u and v are *connected*.

In directed graphs, *reachability* is interpreted by default with taking the direction of the edges into account. In cases when edge directions are omitted (and directed edges can be traversed both ways), it is important to note the distinction. Based on the reachability between their nodes, graphs can be divided into components.

Definition 59 (connected component) A *connected component* is a maximal subgraph in which any two nodes are connected to each other. The *component* of a given node is the component that contains that node.

Definition 60 (connected components (CC)) The *connected components* metric returns every *connected component* in the graph.

The definition has two variants based on whether edge directions are enforced:

Definition 61 (weakly connected components (WCC)) The *weakly connected components* metric returns every *connected component* determined without taking edge directions into account (i.e. edges can be traversed both ways).

Definition 62 (strongly connected components (SCC)) The *strongly connected components* metric returns every *connected component* with edge directions enforced.

In typical graph analytical workloads, the expected result of a CC algorithm is to assign the index of its *component* to each node. In some cases (e.g. in the LDBC Graphalytics benchmark [Ios+16], discussed in Sec. 3.9.2) it is expected to assign the *component size* to each node.

Definition 63 (connectedness of graphs) A graph is *connected* if it consist of a single connected component.

A.2 Shortest Path-Based Metrics

Numerous metrics are defined for *connected graphs* (Def. 63) to characterize the distribution of *shortest paths* (Def. 25). Depending on the goal of the analysis, directed graphs are sometimes treated as undirected, in which case paths can lead through edges in any directions.

The *betweenness centrality* metric [Fre78] measures the *centrality* of a given node in the graph by determining the ratio of shortest paths passing through a single node to all shortest paths in the graph. Formally:

Definition 64 (betweenness centrality)

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}},$$

where σ_{st} is the total number of shortest paths (Def. 25) from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through node v.

Definition 65 (eccentricity) The *eccentricity* of a node $\epsilon(v)$ is the longest *distance* (Def. 26) between v and any other node.

Intuitively, *eccentricity* shows the distance from node v the most distant node in the graph.

Definition 66 (radius) The *radius* r of a graph is its minimum *eccentricity* value:

$$r(G) = \min_{v \in V} \epsilon(v).$$

Definition 67 (diameter) The *diameter* d of a graph is its maximum *eccentricity* value:

$$d(G) = \max_{v \in V} \epsilon(v).$$

A possible approach to determine the *diameter* of a given graph is to determine the *shortest path* between each pair of nodes. The longest of these paths gives the diameter of the graph. This method is often prohibitively expensive for large graphs, therefore approximate algorithms are used instead [NLP13]. Computations targeting the *betweenness centrality* metric also face scalability challenges and are often evaluated using approximation techniques [Qi+13].

A.3 Density Metrics

The *density* describes how tightly connected the nodes in the graph are. For simple directed graphs it is defined as follows.

Definition 68 (density for directed graphs)

$$D(G) = \frac{|E|}{|V|\,(|V| - 1)}.$$

For simple undirected graphs, it is defined as:

Definition 69 (density for undirected graphs)

$$D(G) = \frac{2|E|}{|V|\,(|V| - 1)},$$

where the constant 2 in the numerator accounts for the fact that each potential edge is counted twice in the denominator.

Further metrics Other popular untyped metrics include *PageRank*, which has been studied extensively over the last two decades [Chu14]. However, it is difficult to apply for edge-typed and more expressive graph data models, therefore we do not discuss it in more detail.

Appendix B

Techniques for Graph Analytics

B.1 Foundations of Linear Algebra

Many algorithms use matrices to represent graphs for performance reasons, especially in the field of graph analytics. To allow us to express graph operations in the language of linear algebra, we review the definitions of basic matrix operations.

B.1.1 Data Structures

First, we briefly review the definitions of matrices and vectors.

Definition 70 (real matrix) A *real matrix* M that has n rows and m columns, and stores real numbers is denoted as:

$$M \in \mathbb{R}^{n \times m}$$

$M_{i,j}$ denotes the element in the ith row and jth column of the matrix.

In the following, we always to real matrices.

Definition 71 (square matrix) A *square matrix* is a matrix $M \in \mathbb{R}^{n \times n}$, i.e. one that the same number of rows and columns.

Definition 72 (vector) A *vector* is a matrix $M \in \mathbb{R}^{n \times 1}$, i.e. one with a single column.

Definition 73 (symmetric matrix) A matrix M is *symmetric* iff $M_{i,j} = M_{j,i}$.

B.1.2 Matrix Operations

Next, we define basic matrix operations.

Definition 74 (matrix multiplication) For matrices Q of $n \times m$ elements and R of $m \times o$ elements, the *matrix product* operation (also known as the *matrix multiplication*) is denoted

with $P = Q \cdot R$ and is defined as:

$$P_{i,j} = \sum_{k=1}^{m} Q_{i,k} \cdot R_{k,j},$$

for $i = 1, \dots, n$ and $j = 1, \dots, o$.

As a special case, an n-by-m matrix M can be multiplied with a vector of m elements. Notably, a matrix M multiplied by a vector of ones ($\vec{1}$) returns a vector containing the sum of the values for each row in M:

$$\vec{v} = M \cdot \vec{1}$$
$$\vec{v}_i = \sum_{k=1}^{m} M_{i,k}$$

Definition 75 (element-wise multiplication) For matrices Q and R, both containing $m \times n$ elements, the *element-wise multiplication* operation (also known as the *Hadamard-product*) is denoted with $P = Q \odot R$ and is defined as:

$$P_{i,j} = Q_{i,j} \cdot R_{i,j},$$

Definition 76 (element-wise division) For matrices Q and R, both containing $m \times n$ elements, the *element-wise divison* operation is denoted with $Q \oslash R$ and defined as:

$$P_{i,j} = \frac{Q_{i,j}}{R_{i,j}}$$

Complexity For *sparse matrices* (Sec. B.1.3), the complexity of matrix operations is difficult to estimate as it depends mostly on the *sparsity* of the matrix (Def. 80). For dense matrices, the complexity of multiplication operations is well-studied. For square matrices of $n \times n$ elements, the *computation complexity* of the naïve algorithms are $\mathcal{O}\left(n^3\right)$ for the *matrix multiplication*, and $\mathcal{O}\left(n^2\right)$ for the *element-wise multiplication*.[1] Similarly:

Definition 77 (matrix power) The power M^k of a matrix M is defined as a matrix product of k copies of M, e.g. $M^3 = M \cdot M \cdot M$. Note that the power operation can only be applied to square matrices, i.e. matrices of $n \times n$ elements.

Definition 78 (diagonal elements) Given a square matrix M of $n \times n$ elements, function $\mathrm{diag}^{-1}(M)$ returns a vector of n elements, containing the values of the main diagonal of M, i.e. elements $M_{i,i}$.

[1] The complexity of the matrix multiplication problem has been studied for decades and there are many sophisticated algorithms available. The most well-known ones are the *Strassen algorithm* and the *Coppersmith–Winograd algorithm*, providing an upper time bound of $\mathcal{O}\left(n^{2.807355}\right)$ and $\mathcal{O}\left(n^{2.375477}\right)$, respectively.

B.1.3 Sparse Matrices

Many matrices in practical applications mostly consist of zeroes and have very few non-zero elements. Such matrices are called *sparse*, while matrices with many non-zero elements are called *dense*. While there is no commonly accepted threshold for calling a matrix *sparse*, the notion of *density* can be used to characterize a given matrix:

Definition 79 (matrix density) For a matrix $M \in \mathbb{R}^{n \times m}$, its *density* is defined as:

$$\text{density of matrix } M = \frac{\text{number of non-zero elements in } M}{n \cdot m}$$

In some cases, it is preferable to use the inverse of density:

Definition 80 (matrix sparsity) For a matrix $M \in \mathbb{R}^{n \times m}$, its *sparsity* is defined as

$$\text{sparsity of matrix } M = \frac{\text{number of zero elements in } M}{n \cdot m} = 1 - \text{density of matrix } M$$

It is worth pointing out that for adjacency matrices describing graphs, the *density of the adjacency matrix* is $\frac{|E|}{|N|^2}$, which is almost identical to the *density of directed graphs* in Def. 68 and the *density of undirected graphs* in Def. 69 (note that the constant 2 in the numerator takes care of the fact that adjacency matrices of undirected graphs are symmetric). It is also worth pointing out that for graphs that have an out-degree close to 1 (which is a realistic assumption for many graphs including engineering models), the density values can be estimated as $\frac{n}{n^2} = \frac{1}{n}$. Consequently, for such a graph with 10^6 nodes, the density value is approx. 10^{-6}, i.e. 1 in a million elements in the matrix take a non-zero value.

Compressed representation High-performance sparse matrix libraries such as Eigen[2] and SuiteSparse[3] [Dav18] use a compressed matrix representation, e.g. the *Compressed Row Storage* (CRS) or *Compressed Column Storage* (CCS) formats[4] [Saa03]. CRS and CCS provide compact storage and efficient multiplication operations at the expense of a higher initialization cost. Other libraries use more basic representation formats, such as LIL (List of Lists) or DOK (Dictionary of Keys). These allow for quick initialization, but provide limited scalability for multiplication operations on large matrices.

B.2 Graph Metrics as Graph Queries

As discussed in Sec. 2.7.2, many graph metrics can be expressed as graph queries. Here, we give such an example by formulating the *local clustering coefficient* metric (Def. 38) in Cypher:

[2] http://eigen.tuxfamily.org

[3] http://faculty.cse.tamu.edu/davis/suitesparse.html

[4] CRS is also known as *Compressed Sparse Row* (CSR) or the *Yale* format, while CCS is known as *Compressed Sparse Column* (CSC).

```
1  MATCH (v)
2  OPTIONAL MATCH (u)-[e1]-(v)-[e2]-(w) // wedges
3  OPTIONAL MATCH (u)-[e3]-(w)          // triangles
4  WITH
5    v,
6    count(e1) AS wedges,
7    count(e3) AS triangles
8  RETURN
9    v,
10   CASE triangles
11     WHEN 0 THEN 0
12     ELSE triangles/toFloat(wedges)
13   END AS lcc
```

B.3 Specialized Languages for Graph Analytics

Several systems [SW13; Car+15; Sun+15b; Hon+15; Sak+16; Wan+17], define frameworks for specifying graph algorithms. These put a number of constraints on the programming model (e.g. they require the algorithm to follow the Pregel model [Mal+10]), but at the same time, they allow developers to define the analytical computations in a general-purpose programming language such as C++, Java, or Scala. This approach comes at a cost: developers typically need to modify their algorithms to fit the programming model, and resulting algorithms are often difficult to optimize. To allow more productive development and make more efficient evaluation possible, increasingly more systems provide domain-specific languages (DSLs) that define high-level primitives for implementing graph algorithms. In the following, we briefly present three *graph analytical languages* in order of their appearance.

Green-Marl *Green-Marl* [Hon+12] is an imperative DSL designed to for writing custom graph algorithms. Users formulate graph queries as high-level *iterations* and *traversals*, which are then compiled into optimized parallel C++ code. Green-Marl is supported by the Oracle PGX.D graph analytical engine (Sec. 3.9.1).

GraphScript *GraphScript* [Par+17] is an imperative DSL for graph analytics, supported by the SAP HANA database. It adopts many graph-specific language constructs from Green-Marl for graph iterations and traversals. It also builds on the type system of HANA, facilitating both simple (e.g. *integer*) and complex data types (e.g. *spatial geometries*), complementing them with *graph-specific types*. These include basic graph elements (*nodes* and *edges*) from Green-Marl, and compound extensions, such as *paths* and *subgraphs*, which are not supported by the type system of Green-Marl.

GraphIt *GraphIt* [Zha+18b] is a DSL for graph analytics. Its distinguishing feature is that it allows developers to optimize the execution of the analytical process. To this end, it provides two languages: the *algorithm language* to specify the computation and the *scheduling language* to specify performance optimizations.

Appendix C

Train Benchmark Queries

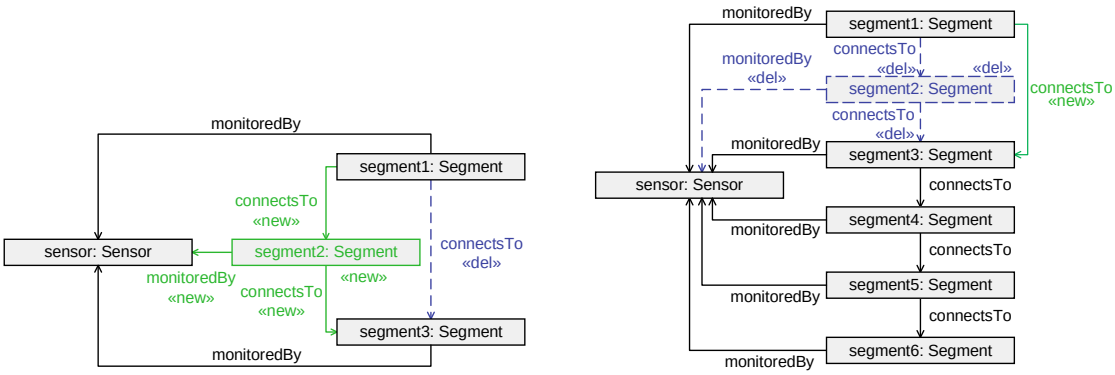
In this chapter, we present the textual and formal specifications of the graph queries of the Train Benchmark, along with the textual and graphical representations of the transformations. For defining the queries, we use the relations given in Sec. 2.4.1.

C.1 ConnectedSegments

Description The ConnectedSegments constraint requires that each sensor must have at most five segments attached to it. Therefore, the query (Fig. 5.2c) checks for sensors (sensor) that have at least six segments (segment1, …, segment6) attached to them.

Relational calculus formula

$$
\begin{aligned}
&\{\mathsf{sensor}, \mathsf{segment1}, \mathsf{segment2}, \mathsf{segment3}, \mathsf{segment4}, \mathsf{segment5}, \mathsf{segment6} \mid \\
&\mathsf{Sensor}(\mathsf{sensor}) \wedge \\
&\mathsf{Segment}(\mathsf{segment1}) \wedge \mathsf{Segment}(\mathsf{segment2}) \wedge \mathsf{Segment}(\mathsf{segment3}) \wedge \\
&\mathsf{Segment}(\mathsf{segment4}) \wedge \mathsf{Segment}(\mathsf{segment5}) \wedge \mathsf{Segment}(\mathsf{segment6}) \wedge \\
&\mathsf{connectsTo}(\mathsf{segment1}, \mathsf{segment2}) \wedge \mathsf{connectsTo}(\mathsf{segment2}, \mathsf{segment3}) \wedge \\
&\mathsf{connectsTo}(\mathsf{segment3}, \mathsf{segment4}) \wedge \mathsf{connectsTo}(\mathsf{segment4}, \mathsf{segment5}) \wedge \\
&\mathsf{connectsTo}(\mathsf{segment5}, \mathsf{segment6}) \wedge \\
&\mathsf{monitoredBy}(\mathsf{segment1}, \mathsf{sensor}) \wedge \mathsf{monitoredBy}(\mathsf{segment2}, \mathsf{sensor}) \wedge \\
&\mathsf{monitoredBy}(\mathsf{segment3}, \mathsf{sensor}) \wedge \mathsf{monitoredBy}(\mathsf{segment4}, \mathsf{sensor}) \wedge \\
&\mathsf{monitoredBy}(\mathsf{segment5}, \mathsf{sensor}) \wedge \mathsf{monitoredBy}(\mathsf{segment6}, \mathsf{sensor})\}
\end{aligned}
$$



(a) Inject transformation. (b) Repair transformation.

Figure C.1: The transformations for query ConnectedSegments.

Inject transformation A random segment segment1 is selected. The connectsTo edge running from segment1 to segment3 is deleted. A new segment segment2 is created and connected from segment1 to segment3. Node segment2 is also connected to node sensor, connected to segment1 via a sensor edge.

Repair transformation The segment2 node and its edges are deleted from the matches. The segment1 and segment3 nodes are connected with a connectsTo edge.

C.2 PosLength

Description The PosLength constraint requires that a segment must have a positive length. Therefore, the query (Fig. 5.2a) checks for segments (segment) with a length less than or equal to zero.

Relational calculus formula

$$\{\mathsf{segment}, \mathsf{length} \mid \mathsf{Segment}(\mathsf{segment}, \mathsf{length}) \wedge \mathsf{length} \leq 0\}$$

(a) Inject transformation. (b) Repair transformation.

Figure C.2: The transformations for query PosLength.

Inject transformation The length property of randomly selected segments is updated to 0.

Repair transformation The length property of the segment in the match is updated to $-\mathsf{length}+1$.

C.3 RouteSensor

The Inject and Repair transformations for the RouteSensor query are discussed in Sec. 5.3.6.

Relational calculus formula

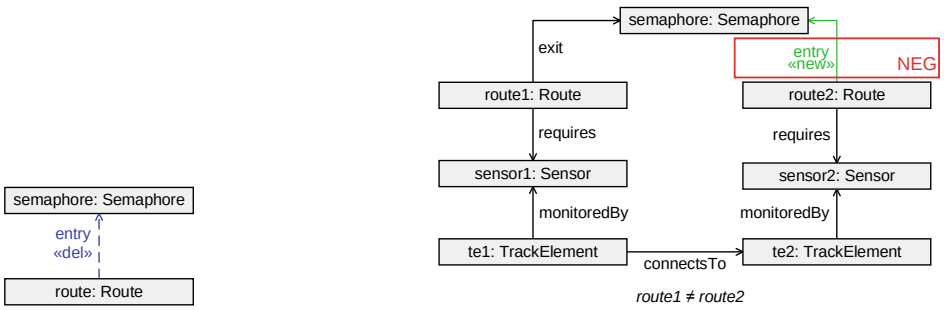
$$\{\mathsf{route}, \mathsf{sensor}, \mathsf{swP}, \mathsf{sw} \mid$$
$$\mathsf{Route}(\mathsf{route}) \wedge \mathsf{Sensor}(\mathsf{sensor}) \wedge$$
$$(\exists \mathsf{currentPosition} : \mathsf{SwitchPosition}(\mathsf{swP}, \mathsf{currentPosition}) \wedge$$
$$(\exists \mathsf{position} : \mathsf{Switch}(\mathsf{sw}, \mathsf{position}) \wedge \mathsf{follows}(\mathsf{route}, \mathsf{swP}) \wedge \mathsf{target}(\mathsf{swP}, \mathsf{sw}) \wedge$$
$$\mathsf{monitoredBy}(\mathsf{sw}, \mathsf{sensor}) \wedge \neg \mathsf{requires}(\mathsf{route}, \mathsf{sensor})))\}$$

C.4 SemaphoreNeighbor

Description The SemaphoreNeighbor constraint requires that the routes that are connected through sensors and track elements have to belong to the same semaphore. Therefore, the query (Fig. 5.2f) checks for routes (route1) which have an exit semaphore (semaphore) and a sensor (sensor1) connected to a track element (te1). This track element is connected to another track element (te2) which is connected to another sensor (sensor2) which (partially) defines another, different route (route2), while the semaphore is not on the entry of this route (route2).

Relational calculus formula

$$\{\mathsf{semaphore}, \mathsf{route1}, \mathsf{route2}, \mathsf{sensor1}, \mathsf{sensor2}, \mathsf{te1}, \mathsf{te2} \mid$$
$$\mathsf{Semaphore}(\mathsf{semaphore}) \wedge \mathsf{Route}(\mathsf{route1}) \wedge \mathsf{Route}(\mathsf{route2}) \wedge \mathsf{Sensor}(\mathsf{sensor1}) \wedge \mathsf{Sensor}(\mathsf{sensor2}) \wedge$$
$$\mathsf{TrackElement}(\mathsf{te1}) \wedge \mathsf{TrackElement}(\mathsf{te2}) \wedge \mathsf{exit}(\mathsf{route1}, \mathsf{semaphore}) \wedge \mathsf{requires}(\mathsf{route1}, \mathsf{sensor1}) \wedge$$
$$\mathsf{monitoredBy}(\mathsf{te1}, \mathsf{sensor1}) \wedge \mathsf{connectsTo}(\mathsf{te1}, \mathsf{te2}) \wedge$$
$$\mathsf{monitoredBy}(\mathsf{te2}, \mathsf{sensor2}) \wedge \mathsf{requires}(\mathsf{route2}, \mathsf{sensor2}) \wedge \neg \mathsf{entry}(\mathsf{route2}, \mathsf{semaphore})\}$$



(a) Inject transformation. (b) Repair transformation.

Figure C.3: The transformations for query SemaphoreNeighbor.

Inject transformation Errors are introduced by disconnecting the entry edge of the selected routes. (According to the metamodel, a route may only have 0 or 1 entry edges.)

Repair transformation The route2 node is connected to the semaphore node with an entry edge.

C.5 SwitchMonitored

Description The SwitchMonitored constraint requires that every switch must have at least one sensor connected to it. Therefore, the query (Fig. 5.2b) checks for switches (switch) that have no sensors (sensor) associated with them.

Relational calculus formula

$$\{\mathsf{sw} \mid \mathsf{Switch}(\mathsf{sw}) \wedge (\neg\exists \mathsf{sensor} : \mathsf{Sensor}(\mathsf{sensor}) \wedge \mathsf{monitoredBy}(\mathsf{switch}, \mathsf{sensor}))\}$$

(a) Inject transformation.

(b) Repair transformation.

Figure C.4: The transformations for query SwitchMonitored.

Inject transformation Errors are injected by randomly selecting switches (switch) and deleting all their edges to sensors. If the selected switch was invalid, it did not have such an edge, so no edges are deleted and the switch stays invalid. If the chosen switch was valid, it will become invalid.

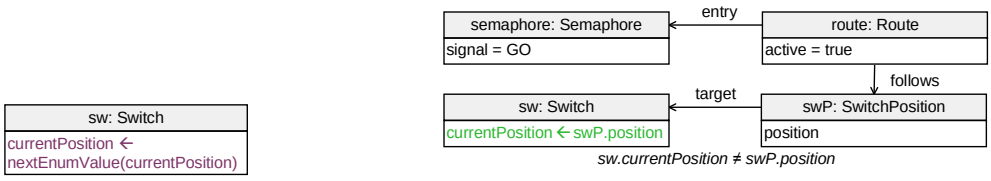
Repair transformation A sensor is created and connected to the switch.

C.6 SwitchSet

Description The SwitchSet constraint requires that an entry semaphore of a route may only show GO if all switches along the route are in the position prescribed by the route. Therefore, the query (Fig. 5.2e) checks for routes (route) which have an entry semaphore (semaphore) that shows the GO signal. Additionally, the route follows a switch position (swP) that is connected to a switch (sw), but the switch position (swP.position) defines a different position from the current position of the switch (sw.currentPosition).

Relational calculus formula

$$\{\mathsf{semaphore}, \mathsf{route}, \mathsf{swP}, \mathsf{sw}, \mathsf{currentPosition}, \mathsf{position} \mid$$
$$\mathsf{Route}(\mathsf{route}) \wedge \mathsf{SwitchPosition}(\mathsf{swP}, \mathsf{position}) \wedge \mathsf{Switch}(\mathsf{sw}, \mathsf{currentPosition}) \wedge$$
$$\mathsf{currentPosition} \neq \mathsf{position} \wedge \mathsf{Semaphore}(\mathsf{semaphore}, \text{“GO”}) \wedge \mathsf{entry}(\mathsf{route}, \mathsf{semaphore}) \wedge$$
$$\mathsf{follows}(\mathsf{route}, \mathsf{swP}) \wedge \mathsf{target}(\mathsf{swP}, \mathsf{sw})\}$$



(a) Inject transformation.

(b) Repair transformation.

Figure C.5: The transformations for query SwitchSet.

Inject transformation Errors are injected by randomly selecting switches (switch) and setting their currentPosition property to the next enum value, e.g. from LEFT to RIGHT (see Fig. 2.4).

Repair transformation The currentPosition property of switch is set to the position of swP.

Appendix D

Choke Points and Queries of the LDBC SNB's Business Intelligence Workload

This chapter contains the specification of the choke points (Sec. D.1) and queries (Sec. D.2) defined in the LDBC Social Network Benchmark's Business Intelligence workload. The specifications presented here were edited for conciseness; the complete specification of the benchmark is available in [r22].

D.1 Choke Points

In this section, we describe the *choke points* of the LDBC SNB benchmarks. The first set of these choke were first identified in [BNE13] with extensions added in [Erl+15] and [l18]. The connection between choke points and queries in the LDBC SNB is displayed in Tab. D.1.

D.1.1 Aggregation Performance

CP-1.1 Interesting orders This choke point tests the ability of the query optimizer to exploit the interesting orders induced by some operators. For example, a neighbourhood expansion operator often preserves the implicit sortedness of the adjacency, which can be subsequently used to perform cheaper duplicate elimination on the set of discovered nodes.

CP-1.2 High Cardinality group-by performance This choke point tests the ability of the execution engine to parallelize group-by operations with a large number of groups. Real property graphs are usually rich in terms of different property values (e.g. person names, topics, cities, etc.) and thus is very common that aggregate queries over these attributes result in a large number of groups (especially when grouping over multiple properties). In such a case, intra query parallelization can be exploited by making each thread to partially aggregate a subset of the results. In order to avoid the merging the partial aggregations and to avoid costly critical sections, the results to group can be partitioned by hashing the grouping key and be sent to the appropriate thread/partition.

CP-1.3 Top-k push down This choke point tests the ability of the engine to optimize queries requesting top-k results. The search space of Graph BI queries can easily explode given the complexity of the patterns in search. Many times, the search space can be pruned by imposing extra restrictions once k results have been obtained and the query advances. Applying this kind of optimizations can significantly reduce the search space.

	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	4.1	4.2	4.3	5.1	5.2	5.3	6.1	7.1	7.2	7.3	7.4	8.1	8.2	8.3	8.4	8.5	8.6
BI 1		⊗								⊗		⊗															⊗	
BI 2	⊗	⊗	⊗		⊗		⊗		⊗	⊗														⊗			⊗	
BI 3								⊗	⊗	⊗		⊗		⊗			⊗	⊗						⊗			⊗	
BI 4	⊗	⊗	⊗		⊗	⊗		⊗			⊗													⊗				
BI 5		⊗	⊗		⊗	⊗	⊗	⊗			⊗						⊗	⊗						⊗		⊗		
BI 6		⊗					⊗																	⊗				
BI 7		⊗					⊗			⊗	⊗							⊗						⊗				
BI 8				⊗							⊗					⊗							⊗					
BI 9		⊗	⊗		⊗		⊗	⊗																⊗				
BI 10		⊗			⊗		⊗			⊗														⊗		⊗	⊗	
BI 11	⊗				⊗	⊗	⊗		⊗	⊗								⊗					⊗		⊗			
BI 12		⊗				⊗			⊗									⊗									⊗	
BI 13		⊗				⊗	⊗			⊗								⊗							⊗		⊗	
BI 14		⊗				⊗	⊗			⊗										⊗	⊗	⊗	⊗				⊗	
BI 15		⊗					⊗			⊗	⊗						⊗	⊗						⊗		⊗		
BI 16		⊗	⊗				⊗	⊗			⊗								⊗	⊗	⊗		⊗					⊗
BI 17	⊗						⊗																					
BI 18	⊗	⊗		⊗						⊗			⊗	⊗									⊗	⊗	⊗	⊗	⊗	
BI 19	⊗		⊗		⊗		⊗	⊗			⊗				⊗						⊗	⊗	⊗				⊗	
BI 20				⊗	⊗													⊗					⊗					
BI 21		⊗			⊗		⊗	⊗		⊗	⊗				⊗		⊗							⊗		⊗	⊗	
BI 22			⊗	⊗	⊗				⊗		⊗				⊗	⊗	⊗								⊗	⊗		
BI 23				⊗			⊗	⊗			⊗			⊗													⊗	
BI 24				⊗	⊗		⊗	⊗		⊗				⊗													⊗	
BI 25		⊗			⊗	⊗		⊗			⊗				⊗		⊗			⊗	⊗		⊗	⊗	⊗	⊗	⊗	⊗
IC 1					⊗												⊗							⊗				
IC 2	⊗					⊗	⊗			⊗																	⊗	
IC 3					⊗				⊗						⊗									⊗			⊗	
IC 4							⊗																	⊗			⊗	
IC 5							⊗				⊗													⊗			⊗	
IC 6															⊗									⊗				
IC 7						⊗	⊗				⊗				⊗								⊗		⊗			
IC 8								⊗		⊗	⊗						⊗											
IC 9	⊗	⊗				⊗	⊗			⊗	⊗																⊗	
IC 10							⊗				⊗	⊗	⊗		⊗	⊗		⊗	⊗									⊗
IC 11			⊗				⊗	⊗			⊗																	
IC 12											⊗									⊗	⊗			⊗				
IC 13											⊗									⊗	⊗		⊗					⊗
IC 14											⊗									⊗	⊗		⊗					⊗

Table D.1: Coverage of choke points by the LDBC SNB's Business Intelligence (BI) *"read"* queries [l18] (Sec. D.2) and the Interactive *"complex read"* (IC) queries [Erl+15]. The symbol ⊗ shows that the given choke point is covered by the given query.

CP-1.4 Low cardinality group-by performance This choke point tests the ability to efficiently group results when only a very limited set of groups is available. This can require special strategies for parallelization, e.g. pre-aggregation when possible. This case also allows using special strategies for grouping like using array lookup if the domain of keys is small. This is typically observed in graph BI queries, especially when grouping results by country or month of the year.

D.1.2 Join Performance

CP-2.1 Rich join order optimization This choke point tests the ability of the query optimizer to find optimal join orders. When looking for pattern occurrences, a graph can be traversed in very different ways, which is equivalent to performing different join orders in the relational model. The execution time of these orders may differ by orders of magnitude, thus finding an efficient traversal strategy is of high importance.

CP-2.2 Late projection This choke point tests the ability of the query optimizer to delay the projection of unneeded attributes until late in the execution. This is common in graph BI queries where we look for patterns with predicates on a reduced set of properties, but we are later interested in selecting other properties not used in such predicates. In such a situation, it might be better to omit such properties from initial scans and fetch them later by node/edge id, which can save temporal space, and therefore I/O. Late projection does have a trade-off involving locality, since late in the plan accessing the nodes by id lead to scattered I/O or memory access patterns. Thus, late projection specifically makes sense in queries where the late use of these columns happens at a moment where the amount of qualifying nodes have been considerably reduced, for example after an aggregation with only few unique group-by keys, or a top-k operator.

CP-2.3 Join type selection This choke point tests the ability of the query optimizer to select the proper join operator type (e.g. hash or index-based joins), which implies accurate estimates of cardinalities. Typically, graph databases will have neighbourhoods materialized/indexed, and thus accessing the neighbours of a reduced set of nodes is typically performed using such indexes (index-based join). However, there are situations where one is interested in obtaining the neighbourhood of a large frontier (a set of nodes). Depending on the cardinalities (size of the frontier and expected size of the neighbourhood), either a hash or an index-based join operator is more appropriate, thus a good estimation of cardinalities is of high importance. The same rationale applies when accessing node/edge properties.

CP-2.4 Sparse foreign key joins This choke point tests the performance of join operators when the join is sparse. Sometimes joins involve relations where only a small percentage of rows in one of the tables is required to satisfy a join. When tables are larger, typical join methods can be sub-optimal. Partitioning the sparse table, using Hash Clustered indexes or implementing Bloom filter [Blo70] tests inside the join are techniques to improve the performance in such situations.

D.1.3 Data Access Locality

CP-3.1 Detecting correlation This choke point tests the ability of the query optimizer to detect data correlations and exploiting them by means of clustered indexes, MinMax indexes, etc. Many real graphs contain correlations between property values (e.g. the country of residence is correlated with the language a person speaks) which can be used to improve data access locality.

CP-3.2 Dimensional clustering This choke point tests suitability of the identifiers assigned to entities by the storage system to better exploit data locality. Many graph database systems use internal identifiers for nodes and edges, thus they have some choice in assigning a value to this identifier. Many real graphs have a modular structure with correlations between neighbours (e.g. friends are likely to share friends) and property values and neighbours (e.g. Persons tend to connect to Persons with similar interests). These characteristics can be exploited in order to assign ids to nodes/edges smartly, which can be used to improve compression and data locality.

CP-3.3 Scattered index access patterns This choke point tests the performance of indexes when scattered accesses are performed. The efficiency of index lookup is very different depending on the locality of keys coming to the indexed access. The structure of real graphs might induce unpredictable index accesses (e.g. a graph neighbourhood traversal is an example of an operation with random access without predictable locality), thus locality sensitive optimizations might need to be disabled if these are costly when there is not locality.

D.1.4 Expression Calculation

CP-4.1 Common subexpression elimination This choke point tests the ability of the query optimizer to detect common sub-expressions and reuse their results. A basic technique helpful in multiple queries is common subexpression elimination (CSE). CSE should recognize also that *average* aggregates can be derived afterwards by dividing the *sum* by the *count* when those have been computed.

CP-4.2 Complex boolean expression joins and selections This choke point tests the ability of the query optimizer to reorder the execution of boolean expressions to improve the performance. Some boolean expressions are complex, with possibilities for alternative optimal evaluation orders. For instance, the optimizer may reorder conjunctions to test first those conditions with larger selectivity.

CP-4.3 Low overhead expressions interpretation This choke point tests the ability of efficiently evaluating simple expressions on a large number of values. A typical example could be simple arithmetic expressions, mathematical functions like floor and absolute or date functions like extracting a year.

CP-4.4 String matching performance This choke point tests the ability of efficiently evaluating complex string matching expressions (e.g. via regular expressions).

D.1.5 Correlated Sub-Queries

CP-5.1 Flattening sub-queries This choke point tests the ability of the query optimizer to flatten execution plans when there are correlated sub-queries. Many queries have correlated sub-queries and their query plans can be flattened, such that the correlated sub-query is handled using an equi-join, outer-join or anti-join. To execute queries well, systems need to flatten both sub-queries, the first into an equi-join plan, the second into an anti-join plan. Therefore, the execution layer of the database system will benefit from implementing these extended join variants. The ill effects of repetitive tuple-at-a-time sub-query execution can also be mitigated if execution systems by using vectorized, or block-wise query execution, allowing to run sub-queries with thousands of input parameters instead

of one. The ability to look up many keys in an index in one API call creates the opportunity to benefit from physical locality, if lookup keys exhibit some clustering.

CP-5.2 Overlap between outer and sub-query This choke point tests the ability of the execution engine to reuse results when there is an overlap between the outer query and the sub-query. In some queries, the correlated sub-query and the outer query have the same joins and selections. In this case, a non-tree, rather DAG-shaped [NM09] query plan would allow to execute the common parts just once, providing the intermediate result stream to both the outer query and correlated sub-query, which higher up in the query plan are joined together (using normal query decorrelation rewrites). As such, the benchmark rewards systems where the optimizer can detect this and the execution engine supports an operator that can buffer intermediate results and provide them to multiple parent operators.

CP-5.3 Intra-query result reuse This choke point tests the ability of the execution engine to reuse sub-query results when two sub-queries are mostly identical. Some queries have almost identical sub-queries, where some of their internal results can be reused in both sides of the execution plan, thus avoiding to repeat computations.

D.1.6 Parallelism and Concurrency

CP-6.1 Inter-query result reuse This choke point tests the ability of the query execution engine to reuse results from different queries. Sometimes with a high number of streams a significant amount of identical queries emerge in the resulting workload. The reason is that certain parameters, as generated by the workload generator, have only a limited amount of parameters bindings. This weakness opens up the possibility of using a query result cache, to eliminate the repetitive part of the workload. A further opportunity that detects even more overlap is the work on recycling, which does not only cache final query results, but also intermediate query results of a "high worth". Here, worth is a combination of partial-query result size, partial-query evaluation cost, and observed (or estimated) frequency of the partial-query in the workload.

D.1.7 Graph-Specific Choke Points

Detailed in Sec. 6.2.2.

D.1.8 Language Choke Points

Detailed in Sec. 6.2.3.

D.2 Query Descriptions

We present a short textual specification for each query in the LDBC SNB BI workload.

Q1. Posting summary Given a date, find all *Messages* created before that date. Group them by a 3-level grouping:

1 by year of creation
2 for each year, group into *Message* types: is *Comment* or not
3 for each year-type group, split into four groups based on length of their content

- 0: 0 <= length < 40 (short)
- 1: 40 <= length < 80 (one liner)
- 2: 80 <= length < 160 (tweet)
- 3: 160 <= length (long)

Q2. Top tags for country, age, gender, time Select all *Messages* created in the range of `[startDate, endDate]` by *Persons* located in `country1` or `country2`. Select the creator *Persons* and the *Tags* of these *Messages*. Split these *Persons*, *Tags* and *Messages* into a 5-level grouping:

1 name of country of *Person*,
2 month the *Message* was created,
3 gender of *Person*,
4 age group of *Person*, defined as years between person's birthday and end of simulation (2013-01-01), divided by 5, rounded down (partial years do not count),
5 name of tag attached to *Message*.

Consider only those groups where number of *Messages* is greater than 100.

Q3. Tag evolution Find the *Tags* that were used in *Messages* during the given `month` of the given `year` and the *Tags* that were used during the next month. For the *Tags* and for both months, compute the count of *Messages*.

Q4. Popular topics in a country Given a *TagClass* and a *Country*, find all the *Forums* created in the given *Country*, containing at least one *Post* with *Tags* belonging directly to the given *TagClass*. The location of a *Forum* is identified by the location of the *Forum*'s moderator.

Q5. Top posters in a country Find the most popular *Forums* for a given *Country*, where the popularity of a *Forum* is measured by the number of members that *Forum* has from the given *Country*. Calculate the top 100 most popular *Forums*. In case of a tie, the forum(s) with the smaller id value(s) should be selected. For each member *Person* of the 100 most popular *Forums*, count the number of *Posts* (`postCount`) they made in any of those (most popular) *Forums*. Also include those member *Persons* who have not posted any messages (have a `postCount` of 0).

Q6. Most active Posters of a given Topic Get each *Person* (`person`) who has created a *Message* (`message`) with a given *Tag* (direct relation, not transitive). Considering only these messages, for each *Person* node:

- Count its `messages` (`messageCount`).
- Count *likes* (`likeCount`) to its `messages`.
- Count *Comments* (`replyCount`) in reply to it `messages`.

The `score` is calculated according to the following formula: `1 * messageCount + 2 * replyCount + 10 * likeCount`.

Q7. Most authoritative users on a given topic Given a *Tag*, find all *Persons* (`person`) that ever created a *Message* (`message1`) with the given *Tag*. For each of these *Persons* (`person`) compute their "authority score" as follows:

- The "authority score" is the sum of "popularity scores" of the *Persons* (`person2`) that liked any of that *Person*'s *Messages* (`message2`) with the given *Tag*.

- A *Person*'s (`person2`) "popularity score" is defined as the total number of likes on all of their *Messages* (`message3`).

Q8. Related topics Find all *Messages* that have a given *Tag*. Find the related *Tags* attached to (direct) reply *Comments* of these *Messages*, but only of those reply *Comments* that do not have the given *Tag*. Group the *Tags* by name, and get the count of replies in each group.

Q9. Forum with related Tags Given two *TagClasses* (`tagClass1` and `tagClass2`), find *Forums* that contain

- at least one *Post* (`post1`) with a *Tag* with a (direct) type of `tagClass1` and
- at least one *Post* (`post2`) with a *Tag* with a (direct) type of `tagClass2`.

The `post1` and `post2` nodes may be the same *Post*. Consider the *Forums* with a number of members greater than a given `threshold`. For every such *Forum*, count the number of `post1` nodes (`count1`) and the number of `post2` nodes (`count2`).

Q10. Central Person for a Tag Given a *Tag*, find all *Persons* that are interested in the *Tag* and/or have written a *Message* (*Post* or *Comment*) with a `creationDate` after a given `date` and that has a given *Tag*. For each *Person*, compute the `score` as the sum of the following two aspects:

- 100, if the *Person* has this *Tag* as their interest, or 0 otherwise
- number of *Messages* by this *Person* with the given *Tag*

Also, for each *Person*, compute the sum of the score of the *Person*'s friends (`friendsScore`).

Q11. Unrelated replies Find those *Persons* of a given *Country* that replied to any *Message*, such that the reply does not have any *Tag* in common with the *Message* (only direct replies are considered, transitive ones are not). Consider only those replies that do no contain any word from a given `blacklist`. For each *Person* and valid reply, retrieve the *Tags* associated with the reply, and retrieve the number of *likes* on the reply. The detailed conditions for checking blacklisted words are currently as follows. Words do *not* have to stand separately, i.e. if the word "Green" is blacklisted, "South-Greenland" cannot be included in the results. Also, comparison should be done in a case-sensitive way. These conditions are preliminary and might be changed in later versions of the benchmark.

Q12. Trending Posts Find all *Messages* created after a given `date` (exclusive), that received more than a given number of likes (`likeThreshold`).

Q13. Popular Tags per month in a country Find all *Messages* in a given *Country*, as well as their *Tags*. Group *Messages* by creation `year` and `month`. For each group, find the 5 most popular *Tags*, where popularity is the number of *Messages* (from within the same group) where the *Tag* appears. Note: even if there are no *Tags* for *Messages* in a given `year` and `month`, the result should include the `year` and `month` with an empty `popularTags` list.

Q14. Top thread initiators For each *Person*, count the number of *Posts* they created in the time interval `[startDate, endDate]` (equivalent to the number of threads they initiated) and the number of *Messages* in each of their (transitive) reply trees, including the root *Post* of each tree. When calculating *Message* counts only consider messages created within the given time interval. Return each *Person*, number of *Posts* they created, and the count of all *Messages* that appeared in the reply trees (including the *Post* at the root of tree) they created.

Q15. Social normals Given a *Country* `country`, determine the "social normal", i.e. the floor of average number of friends that *Persons* of `country` have in `country`. Then, find all *Persons* in `country`, whose number of friends in `country` equals the social normal value.

Q16. Experts in social circle Given a *Person*, find all other *Persons* that live in a given `country` and are connected to given *Person* by a transitive path with length in range `[minPathDistance, maxPathDistance]` through the *knows* relation. In the path, an edge can be only traversed once while nodes can be traversed multiple times. For each of these *Persons*, retrieve all of their *Messages* that contain at least one *Tag* belonging to a given *TagClass* (direct relation not transitive). For each *Message*, retrieve all of its *Tags*. Group the results by *Persons* and *Tags*, then count the *Messages* by a certain *Person* having a certain *Tag*.

Q17. Friend triangles For a given `country`, count all the distinct triples of *Persons* such that:

- `a` is friend of `b`,
- `b` is friend of `c`,
- `c` is friend of `a`.

Distinct means that given a triple t_1 in the result set R of all qualified triples, there is no triple t_2 in R such that t_1 and t_2 have the same set of elements.

Q18. How many persons have a given number of messages For each *Person*, count the number of *Messages* they made (`messageCount`). Only count *Messages* with the following attributes:

- Its `content` is not empty (and consequently, `imageFile` empty for *Posts*).
- Its `length` is below the `lengthThreshold` (exclusive).
- Its `creationDate` is after `date` (exclusive).
- It is written in any of the given `languages`. The language of a *Post* is defined by its `language` attribute. The language of a *Comment* is that of the *Post* that initiates the thread where the *Comment* replies to. The *Post* and *Comments* in the reply tree's path (from the *Message* to the *Post*) do not have to satisfy the constraints for `content`, `length` and `creationDate`.

For each `messageCount` value, count the number of *Persons* with exactly `messageCount` *Messages* (with the required attributes).

Q19. Stranger's interaction For all the *Persons* (`person`) born after a certain `date`, find all the `strangers` they interacted with, where `strangers` are *Persons* that do not *know* `person`. There is no restriction on the date that `strangers` were born. Consider only `strangers` that are

- members of *Forums* tagged with a *Tag* with a (direct) type of `tagClass1` and
- members of *Forums* tagged with a *Tag* with a (direct) type of `tagClass2`.

The *Tags* may be attached to the same *Forum* or they may be attached to different *Forums*. Interaction is defined as follows: the `person` has replied to a *Message* by the `stranger` *B* (the reply might be a transitive one). For each `person`, count the number of `strangers` they interacted with (`strangerCount`) and total number of times they interacted with them (`interactionCount`).

Q20. High-level topics For all given *TagClasses*, count number of *Messages* that have a *Tag* that belongs to that *TagClass* or any of its children (all descendants through a transitive relation).

Q21. Zombies in a country Find zombies within the given `country`, and return their zombie scores. A `zombie` is a *Person* created before the given `endDate`, which has created an average of `[0, 1)` *Messages* per month, during the time range between profile's `creationDate` and the given `endDate`. The number of months spans the time range from the `creationDate` of the profile to the `endDate` with partial months on both end counting as one month (e.g. a `creationDate` of Jan 31 and an `endDate` of Mar 1 result in 3 months). For each `zombie`, calculate the following:

- `zombieLikeCount`: the number of *likes* received from other zombies.
- `totalLikeCount`: the total number of *likes* received.
- `zombieScore`: `zombieLikeCount` / `totalLikeCount`. If the value of `totalLikeCount` is 0, the `zombieScore` of the `zombie` should be 0.

For both `zombieLikeCount` and `totalLikeCount`, only consider *likes* received from profiles created before the given `endDate`.

Q22. International dialog Consider all pairs of people `(person1, person2)` such that one is located in a *City* of *Country* `country1` and the other is located in a *City* of *Country* `country2`. For each *City* of *Country* `country1`, return the highest scoring pair. The score of a pair is defined as the sum of the subscores awarded for the following kinds of interaction. The initial value is `score = 0`.

1. `person1` has created a reply *Comment* to at least one *Message* by `person2`: `score += 4`
2. `person1` has created at least one *Message* that `person2` has created a reply *Comment* to: `score += 1`
3. `person1` and `person2` *know* each other: `score += 15`
4. `person1` liked at least one *Message* by `person2`: `score += 10`
5. `person1` has created at least one *Message* that was liked by `person2`: `score += 1`

To break ties, order by (1) `person1.id` ascending and (2) `person2.id` ascending.

Q23. Holiday destinations Count the *Messages* of all residents of a given *Country*, where the message was written abroad. Group the messages by `month` and `destination`. A *Message* was written abroad if it *is located in* a *Country* different than `home`.

Q24. Messages by Topic and Continent Find all *Messages* tagged with a *Tag* that has the (direct) type of the given `tagClass`. Count all *Messages* and their likes grouped by *Continent*, `year`, and `month`.

Q25. Weighted interaction paths Given two *Persons*, find all (unweighted) shortest paths between these two *Persons*, in the subgraph induced by the *knows* relationship. Then, for each path calculate a weight. The nodes in the path are *Persons*, and the weight of a path is the sum of weights between every pair of consecutive *Person* nodes in the path. The weight for a pair of *Persons* is calculated based on their interactions:

- Every direct reply (by one of the *Persons*) to a *Post* (by the other *Person*) contributes 1.0.
- Every direct reply (by one of the *Persons*) to a *Comment* (by the other *Person*) contributes 0.5.

Only consider *Messages* that were created in a *Forum* that was created within the timeframe `[startDate, endDate]`. Note that for *Comments*, the containing *Forum* is that of the *Post* that the comment (transitively) replies to. Return all paths with the *Person* ids ordered by their weights descending.

Appendix E

Foundations of Incremental Query Evaluation

In this chapter, we present the key maintenance rules for Rete networks. In this section, we discuss Rete nodes in detail. For unary and binary nodes, we formulate the *maintenance operations*, which are performed upon receiving an update message. For these operations, we denote the output relation by t, the updated output relation by t', and the propagated update message on the output by Δt and ∇t for positive and negative change sets, respectively. Using these, the updated output relation can be determined as $t' = t \cup \Delta t - \nabla t$. To provide a running example, we revisit the railway network of Fig. 2.2b, repeated in Fig. E.1. This chapter is partially based on [j3].

E.1 Nullary Nodes

Input nodes provide the relations for each label of the graph. For example, the input node for the requires edge label of the example graph in Fig. 2.2c (see Sec. 2.1) returns tuples that are currently in the *requires* relation: $\{\langle 1, d, 5\rangle, \langle 1, e, 6\rangle, \langle 2, f, 5\rangle, \langle 2, g, 7\rangle\}$. This input node is also responsible for propagating changes to worker nodes in the network:

- If a requires edge 'z' is inserted from node 2 to 6, the input node sends a positive update message to its subscriber nodes with the change set $\{\langle 2, z, 6\rangle\}$.
- If the edge 'd' between nodes 1 and 5 is deleted, the input node sends a negative update to its subscriber nodes with the change set $\{\langle 1, d, 5\rangle\}$.

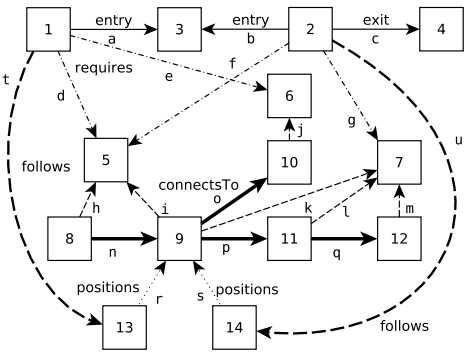


Figure E.1: Railway instance as an edge-typed graph (repeated from Fig. 2.2b).

The relations contained by input nodes can be defined with nullary operators: input nodes indexing vertices implement the *get-vertices* operator, while input nodes indexing edges implement the *get-edges* operator.

E.2 Unary Nodes

Unary nodes have one input slot. They filter or transform the tuples of the parent node according to certain criteria. In the following, the relation representing the input tuples is denoted with r, the relation representing the output tuples is denoted with t, and the operator processing the input is denoted with α:

$$t = \alpha(r).$$

Maintenance In the following, we assume that the α operator is *distributive* w.r.t. the union ($\cup$) and set minus ($-$) operators. If a unary node receives a *positive update* Δr, it performs the operation and computes the change set. The result (t') and the change set (Δt) are:

$$\begin{aligned} t' &\equiv \alpha(r \cup \Delta r) \\ &= \alpha(r) \cup \alpha(\Delta r) \\ &= t \cup \underbrace{\alpha(\Delta r)}_{\Delta t} \end{aligned}$$

Similarly, for a *negative update* ∇r:

$$\begin{aligned} t' &\equiv \alpha(r - \nabla r) \\ &= \alpha(r) - \alpha(\nabla r) \\ &= t - \underbrace{\alpha(\nabla r)}_{\nabla t} \end{aligned}$$

Unary nodes are often implemented as *stateless* nodes, i.e. they do not store the results of the previous executions. Instead, these results are cached in their subscribers, e.g. indexers of *binary nodes* (Sec. E.3).

As their name suggests, unary nodes implement unary relational algebraic operators (Sec. 2.4.3):

- The *projection node* performs a projection operation on the input relation.
- The *selection node* performs a selection operation on the input relation.

As both the projection and the selection operators are distributive w.r.t. the union and set minus operators, their results can be maintained by performing the operation for change sets Δr and ∇r.

E.3 Binary Nodes

Binary nodes have two input slots: the *primary* (p) and the *secondary* (s). Their positive and negative change sets are denoted with $\Delta p/\nabla p$ and $\Delta s/\nabla s$. Binary node implementations typically cache both their input relations in *indexers* and updates rely on the efficient lookups provided by these indexers. Most of the maintenance rules defined in this section are illustrated with the example base tables and changes in Fig. E.2. The complete examples were presented in Fig. 8.4.

p

A	B
x	1
y	2
z	3

s

B	C
1	0.4
1	0.5
2	0.6

Δp

A	B
m	2
n	4

∇p

A	B
y	2
z	3

Δs

B	C
2	0.7
3	0.8

∇s

B	C
1	0.5
2	0.6

$p \cup \Delta p$

A	B
x	1
y	2
z	3
m	2
n	4

$p - \nabla p$

A	B
x	1
~~y~~	~~2~~
~~z~~	~~3~~

$s \cup \Delta s$

B	C
1	0.4
1	0.5
2	0.6
2	0.7
3	0.8

$s - \nabla s$

B	C
1	0.4
~~1~~	~~0.5~~
~~2~~	~~0.6~~

Figure E.2: Example base relations for primary and secondary slots (p and s) with positive and negative change sets ($\Delta p, \nabla p, \Delta s, \nabla s$) for demonstrating IVM operations.

E.3.1 Join Node

In the following, we define the maintenance operations for join nodes.

Positive update on the primary slot (Δp)

If a join node receives a *positive update* Δp on its *primary* input slot, the result (t') and the change set (Δt) are determined as follows:

$$\begin{aligned} t' &\equiv (p \cup \Delta p) \bowtie s \\ &= (p \bowtie s) \cup (\Delta p \bowtie s) \\ &= t \cup \underbrace{(\Delta p \bowtie s)}_{\Delta t} \end{aligned}$$

> **Example 33** $\Delta t = \Delta p \bowtie s = \{\langle \mathrm{m}, 2\rangle, \langle \mathrm{n}, 4\rangle\} \bowtie \{\langle 1, 0.4\rangle, \langle 1, 0.5\rangle, \langle 2, 0.6\rangle\} = \{\langle \mathrm{m}, 2, 0.6\rangle\}$

Negative update on the primary slot (∇p)

For a *negative update* ∇p, the derivation is the same, but the deltas are propagated as a *negative update*:

$$t' = t - (\nabla p \bowtie s)$$

Example 34 $\nabla t = \nabla p \bowtie s = \{\langle \mathrm{y}, 2, 0.6 \rangle\}$;

Positive update on the secondary slot (Δs)

If the node receives a *positive update* Δs on its *secondary* input slot, the result (t') and the change set (Δt) are the following:

$$\begin{aligned} t' &\equiv p \bowtie (s \cup \Delta s) \\ &= (p \bowtie s) \cup (p \bowtie \Delta s) \\ &= t \cup \underbrace{(p \bowtie \Delta s)}_{\Delta t} \end{aligned}$$

Example 35

$$\Delta t = p \bowtie \Delta s = \{\langle \mathrm{x}, 1 \rangle, \langle \mathrm{y}, 2 \rangle, \langle \mathrm{z}, 3 \rangle\} \bowtie \{\langle 2, 0.7 \rangle, \langle 3, 0.8 \rangle\} = \{\langle \mathrm{y}, 2, 0.7 \rangle, \langle \mathrm{z}, 3, 0.8 \rangle\}$$

Negative update on the secondary slot (∇s)

For a *negative update* ∇p, the derivation is the same, but it is propagated as a *negative update*:

$$t' = t - (p \bowtie \nabla s)$$

Example 36

$$\nabla t = p \bowtie \nabla s = \{\langle \mathrm{x}, 1 \rangle, \langle \mathrm{y}, 2 \rangle, \langle \mathrm{z}, 3 \rangle\} \bowtie \{\langle 1, 0.5 \rangle, \langle 2, 0.6 \rangle\} = \{\langle \mathrm{x}, 1, 0.5 \rangle, \langle \mathrm{y}, 2, 0.6 \rangle\}$$

E.3.2 Semijoin Node

We omit the expressions of the semijoin node but note that they are similar to the expressions for antijoin in terms of complexity.

E.3.3 Antijoin Node

We recall the definition of the *antijoin* operator (Def. 17) for relations p and s:

$$t \equiv p \overline{\ltimes} s = p - (p \ltimes s)\,,$$

As the antijoin operator is not commutative, handling update messages requires us to distinguish between the cases of Δp, ∇p, Δs, and ∇s.

Equalities for simplifying expressions To allow us simplifying the delta expressions, we present the following equalities:

E1. For sets $A, B \subseteq C$: $(C - A) - (C - B) = B - A$.
E2. For sets X, Y: $X \cup Y - X = Y - X$.

E3. For relations q of schema $\mathrm{sch}(q) = Q$ and r_1, r_2 of schema $\mathrm{sch}(r_1) = \mathrm{sch}(r_2) = R$:

$$(q \ltimes r_1) - (q \ltimes r_2) = q \ltimes (r_1 \underset{Q \cap R}{\overline{\ltimes}} r_2)$$

It is important to note that the antijoin operator in $r_1 \underset{Q \cap R}{\overline{\ltimes}} r_2$ is a *theta-antijoin* (Def. 20) as it checks the equivalence of attributes in $Q \cap R$. Applying the definition reveals that the theta-antijoin operator in $r_1 \underset{Q \cap R}{\overline{\ltimes}} r_2 = r_1 - (r_1 \underset{Q \cap R}{\ltimes} r_2)$ checks tuples in r_1 against tuples in r_2 based on the equality of attributes $Q \cap R$.[1] For a concrete case, see the example for $p \overline{\ltimes} (s \cup \Delta s)$.

Positive update on the primary slot (Δp)

There is a *positive update* in the result for each incoming tuple which have no matching tuples on the secondary slot.

$$\begin{aligned} t' &\equiv (p \cup \Delta p) \overline{\ltimes} s \\ &= (p \overline{\ltimes} s) \cup (\Delta p \overline{\ltimes} s) \\ &= t \cup \underbrace{(\Delta p \overline{\ltimes} s)}_{\Delta t} \end{aligned}$$

> **Example 37** $\Delta t = \Delta p \overline{\ltimes} s = \{\langle \mathrm{m}, 2\rangle, \langle \mathrm{n}, 4\rangle\} \overline{\ltimes} \{\langle 1, 0.4\rangle, \langle 1, 0.5\rangle, \langle 2, 0.6\rangle\} = \{\langle \mathrm{n}, 4\rangle\}$

Negative update on the primary slot (∇p)

There is a *negative update* in the results for each incoming tuple which have no matching tuple on the secondary slot. The expression can be derived similarly to Δp:

$$\nabla t = \nabla p \overline{\ltimes} s$$

> **Example 38** $\nabla t = \nabla p \overline{\ltimes} s = \{\langle \mathrm{y}, 2\rangle, \langle \mathrm{z}, 3\rangle\} \overline{\ltimes} \{\langle 1, 0.4\rangle, \langle 1, 0.5\rangle, \langle 2, 0.6\rangle\} = \{\langle \mathrm{z}, 3\rangle\}$

Positive update on the secondary slot (Δs)

For positive updates on the secondary slot, the result set can be expressed as:

$$\begin{aligned} t' &\equiv p \overline{\ltimes} (s \cup \Delta s) \\ &= p - (p \ltimes (s \cup \Delta s)) \end{aligned}$$

[1] This expression is based on equality R1 presented in the SIGMOD Record 1998 Griffin–Kumar paper [GK98]. However, it omits the details of the exact join predicates. As the ICDE 2007 Larson–Zhou paper [LZ07a] puts it: *"Griffin's and Kumar's algorithm [2] produces maintenance expressions of the correct form but they are incomplete because the predicates of the semi and anti-semi joins used are not specified. Getting the predicates right is not trivial."* In this chapter, we provide the predicates for cases when not the default (natural join) semantics are used.

Note that positive updates on the secondary slot result in *negative updates* in the result set, so that $t' = t - \nabla t$, which in turn results in:

$$
\begin{aligned}
\nabla t = t - t' &= \overbrace{\big[\underbrace{p}_{C} - \underbrace{(p \ltimes s)}_{A}\big]}^{t} - \overbrace{\big[\underbrace{p}_{C} - \underbrace{(p \ltimes (s \cup \Delta s))}_{B}\big]}^{t'} && \text{(by E1)}\\
&= (p \ltimes (s \cup \Delta s)) - (p \ltimes s)\\
&= \underbrace{(p \ltimes s)}_{X} \cup \underbrace{(p \ltimes \Delta s)}_{Y} - \underbrace{(p \ltimes s)}_{X} && \text{(by E2)}\\
&= (\underbrace{p}_{q} \ltimes \underbrace{\Delta s}_{r_1}) - (\underbrace{p}_{q} \ltimes \underbrace{s}_{r_2}) && \text{(by E3)}\\
&= p \ltimes (\Delta s \underset{P \cap S}{\overline{\ltimes}} s)
\end{aligned}
$$

Note that the condition of the antijoin operator in $\Delta s \overline{\ltimes} s$ is based on the common attributes of $\mathrm{sch}(p) = P$ and $\mathrm{sch}(s) = S$ as discussed when defining rule E3 at the beginning of this section.

> **Example 39** $\nabla t = p \ltimes (\Delta s \overline{\ltimes} s) = p \ltimes \big(\{\langle 2, 0.7\rangle, \langle 3, 0.8\rangle\} \overline{\ltimes} \{\langle 1, 0.4\rangle, \langle 1, 0.5\rangle, \langle 2, 0.6\rangle\}\big) = \{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 2\rangle, \langle \mathrm{z}, 3\rangle\} \ltimes \{\langle 3, 0.8\rangle\} = \{\langle \mathrm{z}, 3\rangle\}$.

Negative update on the secondary slot (∇s)

For negative updates on the secondary slot, the result set can be expressed as:

$$
\begin{aligned}
t' &\equiv p \overline{\ltimes} (s - \nabla s)\\
&= p - (p \ltimes (s - \nabla s))
\end{aligned}
$$

Negative updates may result in *positive updates* in the result set. Since $t' = t \cup \Delta t$, we define Δt as:

$$
\begin{aligned}
\Delta t = t' - t &= \overbrace{\big[\underbrace{p}_{C} - \underbrace{(p \ltimes (s - \nabla s))}_{A}\big]}^{t'} - \overbrace{\big[\underbrace{p}_{C} - \underbrace{(p \ltimes s)}_{B}\big]}^{t} && \text{(by E1)}\\
&= \underbrace{(p \ltimes s)}_{B} - \underbrace{(p \ltimes (s - \nabla s))}_{A} && \text{(break up } p \ltimes s \text{ by } s = (s - \nabla s) \cup \nabla s)\\
&= \underbrace{(p \ltimes (s - \nabla s))}_{X} \cup \underbrace{(p \ltimes \nabla s)}_{Y} - \underbrace{(p \ltimes (s - \nabla s))}_{X} && \text{(by E2)}\\
&= \underbrace{(p \ltimes \nabla s)}_{Y} - \underbrace{(p \ltimes (s - \nabla s))}_{X} && \text{(the tuples in } X \text{ to be removed from } Y \text{ are in } p \ltimes \nabla s)\\
&= (p \ltimes \nabla s) - ((p \ltimes \nabla s) \ltimes (s - \nabla s)) && \text{(following the definition of antijoin)}\\
&= (p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s)
\end{aligned}
$$

Although this change set may seem difficult to calculate, we point out that a few peculiarities can be exploited for its implementation: (1) in general, ∇s is small, which renders $p \ltimes \nabla s$ also small, and (2) $s - \nabla s$ is the maintained version of s. Based on these observations, the expression can be evaluated efficiently without additional data structures.

> **Example 40** $\Delta t = (p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s) = \{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 2\rangle\} \overline{\ltimes} \{\langle 1, 0.4\rangle\} = \{\langle \mathrm{y}, 2\rangle\}$

E.3.4 Left Outer Join Node

We recall the definition of the *left outer join* operator (Def. 18) for relations p and s:

$$p \mathbin{⟕} s = (p \bowtie s) \cup ((p \overline{\ltimes} s) \times \langle \mathsf{NULL}\rangle_{|\mathrm{sch}(s)-\mathrm{sch}(p)|}),$$

where $\langle \mathsf{NULL}\rangle_{|\mathrm{sch}(s)-\mathrm{sch}(p)|}$ denotes a tuple $\langle \mathsf{NULL}, \mathsf{NULL}, \ldots, \mathsf{NULL}\rangle$ of width $|\mathrm{sch}(s) - \mathrm{sch}(p)|$. For the sake of conciseness, we will simply write $\langle \mathsf{NULL}\rangle$ instead of $\langle \mathsf{NULL}\rangle_{|\mathrm{sch}(s)-\mathrm{sch}(p)|}$.

Positive update on the primary slot (Δp)

$$\begin{aligned}
(p \cup \Delta p) \mathbin{⟕} s &= ((p \cup \Delta p) \bowtie s) \cup [((p \cup \Delta p) \overline{\ltimes} s) \times \langle \mathsf{NULL}\rangle] \\
&= ((p \bowtie s) \cup (\Delta p \bowtie s)) \cup [((p \overline{\ltimes} s) \cup (\Delta p \overline{\ltimes} s)) \times \langle \mathsf{NULL}\rangle] \\
&= (p \bowtie s) \cup (\Delta p \bowtie s) \cup [(p \overline{\ltimes} s) \times \langle \mathsf{NULL}\rangle] \cup [(\Delta p \overline{\ltimes} s)) \times \langle \mathsf{NULL}\rangle] \\
&= \underbrace{(p \bowtie s) \cup [(p \overline{\ltimes} s) \times \langle \mathsf{NULL}\rangle]}_{p \mathbin{⟕} s} \cup \underbrace{(\Delta p \bowtie s) \cup [(\Delta p \overline{\ltimes} s)) \times \langle \mathsf{NULL}\rangle]}_{\Delta p \mathbin{⟕} s} \\
&= \underbrace{(p \mathbin{⟕} s)}_{t} \cup \underbrace{(\Delta p \mathbin{⟕} s)}_{\Delta t}
\end{aligned}$$

> **Example 41**
>
> $\Delta t = \Delta p \mathbin{⟕} s = \{\langle \mathrm{m}, 2\rangle, \langle \mathrm{n}, 4\rangle\} \mathbin{⟕} \{\langle 1, 0.4\rangle, \langle 1, 0.5\rangle, \langle 2, 0.6\rangle\} = \{\langle \mathrm{m}, 2, 0.6\rangle, \langle \mathrm{n}, 4, \mathsf{NULL}\rangle\}$

Negative update on the primary slot (∇p)

Similarly to Δp:

$$(p - \nabla p) \mathbin{⟕} s = \underbrace{(p \mathbin{⟕} s)}_{t} - \underbrace{(\nabla p \mathbin{⟕} s)}_{\nabla t}$$

> **Example 42**
>
> $\nabla t = \nabla p \mathbin{⟕} s = \{\langle \mathrm{y}, 2\rangle, \langle \mathrm{z}, 3\rangle\} \mathbin{⟕} \{\langle 1, 0.4\rangle, \langle 1, 0.5\rangle, \langle 2, 0.6\rangle\} = \{\langle \mathrm{y}, 2, 0.6\rangle, \langle \mathrm{z}, 3, \mathsf{NULL}\rangle\}$

Positive update on the secondary slot (Δs)

$$
\begin{aligned}
p ⟕ (s \cup \Delta s) &= (p \bowtie (s \cup \Delta s)) \cup [(p \overline{\ltimes} (s \cup \Delta s)) \times \langle \mathsf{NULL} \rangle] \\
&= (p \bowtie s) \cup (p \bowtie \Delta s) \cup [(p \overline{\ltimes} (s \cup \Delta s)) \times \langle \mathsf{NULL} \rangle] \qquad (\text{use } \nabla(p \overline{\ltimes} (s \cup \Delta s))) \\
&= (p \bowtie s) \cup (p \bowtie \Delta s) \cup [(p \overline{\ltimes} s) - (p \ltimes (\Delta s \overline{\ltimes} s)) \times \langle \mathsf{NULL} \rangle] \\
&= (p \bowtie s) \cup (p \bowtie \Delta s) \cup [(p \overline{\ltimes} s) \times \langle \mathsf{NULL} \rangle] - [(p \ltimes (\Delta s \overline{\ltimes} s)) \times \langle \mathsf{NULL} \rangle] \\
&= \underbrace{(p \bowtie s) \cup [(p \overline{\ltimes} s) \times \langle \mathsf{NULL} \rangle]}_{p ⟕ s} \cup (p \bowtie \Delta s) - [(p \ltimes (\Delta s \overline{\ltimes} s)) \times \langle \mathsf{NULL} \rangle] \\
&= \underbrace{(p ⟕ s)}_{t} \cup \underbrace{(p \bowtie \Delta s)}_{\Delta t} - \underbrace{\left[(p \ltimes (\Delta s \underset{P \cap S}{\overline{\ltimes}} s)) \times \langle \mathsf{NULL} \rangle\right]}_{\nabla t}
\end{aligned}
$$

> **Example 43** There are both positive and negative change sets:
>
> $$\Delta t = p \bowtie \Delta s = \{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 2\rangle, \langle \mathrm{z}, 3\rangle\} \bowtie \{\langle 2, 0.7\rangle, \langle 3, 0.8\rangle\} = \{\langle \mathrm{y}, 2, 0.7\rangle, \langle \mathrm{z}, 3, 0.8\rangle\}$$
>
> $$
> \begin{aligned}
> \nabla t &= (p \ltimes (\Delta s \overline{\ltimes} s)) \times \langle \mathsf{NULL} \rangle \\
> &= (p \ltimes (\{\langle 2, 0.7\rangle, \langle 3, 0.8\rangle\} \overline{\ltimes} \{\langle 1, 0.4\rangle, \langle 1, 0.5\rangle, \langle 2, 0.6\rangle\})) \times \langle \mathsf{NULL} \rangle \\
> &= (\{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 2\rangle, \langle \mathrm{z}, 3\rangle\} \ltimes \{\langle 3, 0.8\rangle\}) \times \langle \mathsf{NULL} \rangle \\
> &= \{\langle \mathrm{z}, 3, \mathsf{NULL}\rangle\}
> \end{aligned}
> $$

Negative update on the secondary slot (∇s)

$$
\begin{aligned}
p ⟕ (s - \nabla s) &= (p \bowtie (s - \nabla s)) \cup [(p \overline{\ltimes} (s - \nabla s)) \times \langle \mathsf{NULL} \rangle] \\
&= (p \bowtie s) - (p \bowtie \nabla s) \cup [(p \overline{\ltimes} (s - \nabla s)) \times \langle \mathsf{NULL} \rangle] \qquad (\text{use } \Delta(p \overline{\ltimes} (s - \nabla s))) \\
&= (p \bowtie s) - (p \bowtie \nabla s) \cup [((p \overline{\ltimes} s) - ((p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s))) \times \langle \mathsf{NULL} \rangle] \\
&= (p \bowtie s) - (p \bowtie \nabla s) \cup [(p \overline{\ltimes} s) \times \langle \mathsf{NULL} \rangle] \cup [((p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s)) \times \langle \mathsf{NULL} \rangle] \\
&= \underbrace{(p \bowtie s) \cup [(p \overline{\ltimes} s) \times \langle \mathsf{NULL} \rangle]}_{p ⟕ s} - (p \bowtie \nabla s) \cup [((p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s)) \times \langle \mathsf{NULL} \rangle] \\
&= \underbrace{p ⟕ s}_{t} - \underbrace{(p \bowtie \nabla s)}_{\nabla t} \cup \underbrace{[((p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s)) \times \langle \mathsf{NULL} \rangle]}_{\Delta t}
\end{aligned}
$$

> **Example 44** There are both positive and negative change sets:

$$
\begin{aligned}
\Delta t &= \big((p \ltimes \nabla s) \overline{\ltimes} (s - \nabla s)\big) \times \langle \mathsf{NULL} \rangle \\
&= \big(\{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 2\rangle, \langle \mathrm{z}, 3\rangle\} \ltimes \{\langle 1, 0.5\rangle, \langle 2, 0.6\rangle\}\big) \\
&\quad \overline{\ltimes} \big(\{\langle 1, 0.4\rangle, \langle 1, 0.5\rangle, \langle 2, 0.6\rangle\} - \{\langle 1, 0.5\rangle, \langle 2, 0.6\rangle\}\big) \times \langle \mathsf{NULL} \rangle \\
&= (\{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 2\rangle\} \overline{\ltimes} \{\langle 1, 0.4\rangle\}) \times \langle \mathsf{NULL} \rangle \\
&= \{\langle \mathrm{y}, 2\rangle\} \times \langle \mathsf{NULL} \rangle = \{\langle \mathrm{y}, 2, \mathsf{NULL}\rangle\}
\end{aligned}
$$

$$
\nabla t = p \bowtie \nabla s = \{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 2\rangle, \langle \mathrm{z}, 3\rangle\} \bowtie \{\langle 1, 0.5\rangle, \langle 2, 0.6\rangle\} = \{\langle \mathrm{x}, 1, 0.5\rangle, \langle \mathrm{y}, 2, 0.6\rangle\}
$$

E.4 Incremental Performance for Query SemaphoreNeighbor

E.4.1 Query Plans

Fig. E.3–E.8 show possible Rete layouts for query SemaphoreNeighbor. *Input nodes* are marked with dashed lines, while *worker nodes* are marked with solid lines. Due to the sake of conciseness, *production nodes* were omitted in the figures. All Rete networks have a single production node as a parent of their depicted root node.

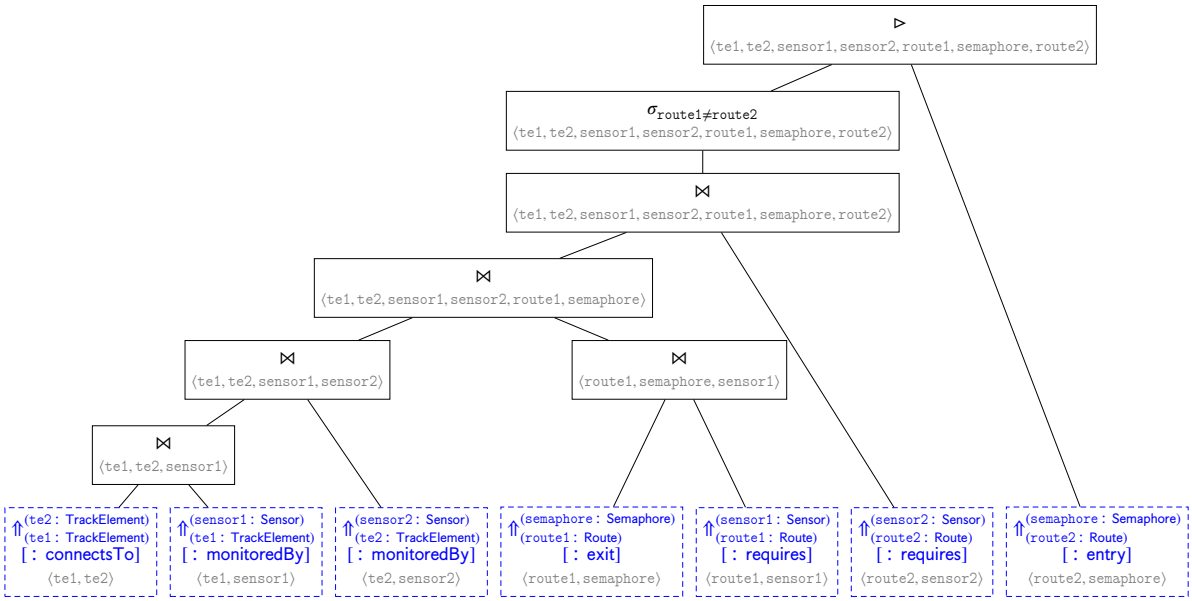


Figure E.3: Rete layout A for query SemaphoreNeighbor.

E.4.2 Execution Time and Memory Consumption

Fig. E.9 shows the execution times for different plans of query SemaphoreNeighbor. The x-axis shows the graph model size, while the y-axis shows the time required for each phase. Both axes use logarithmic scale. Plans A-D show similar performance, but plans E and F perform significantly worse as both compute a Cartesian product as the first step of their evaluation.

Fig. E.10 shows the memory consumption for different plans of query SemaphoreNeighbor. Similarly to the execution times, plans A-D require almost the same amount of memory. However, plan E, which requires the evaluation and maintenance of a Cartesian product, runs out of memory for small

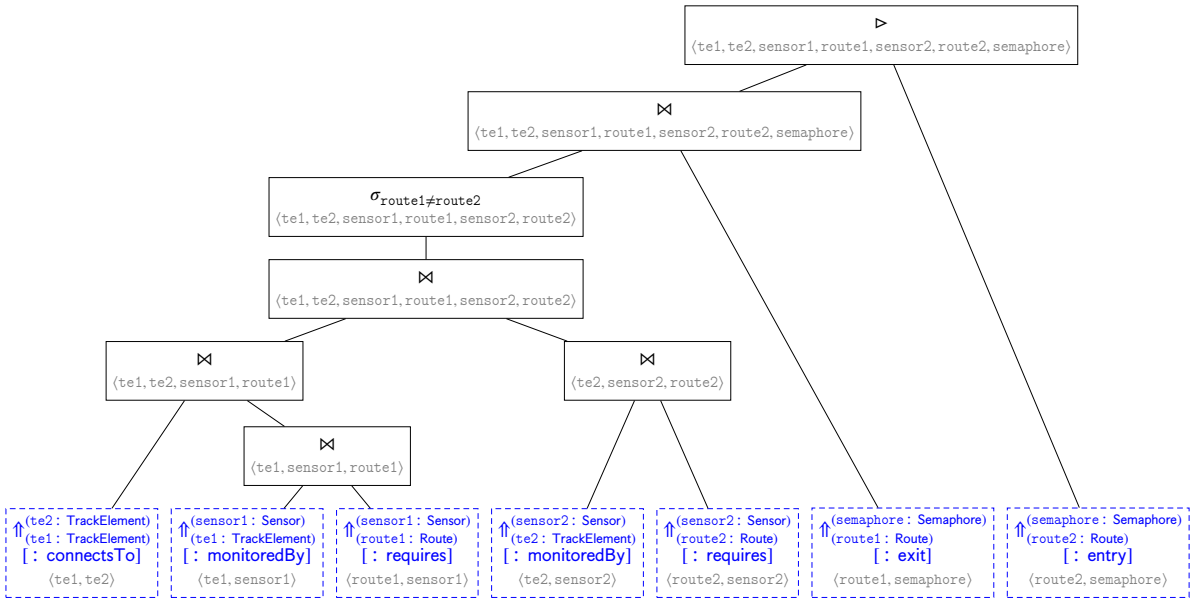


Figure E.4: Rete layout B for query SemaphoreNeighbor.

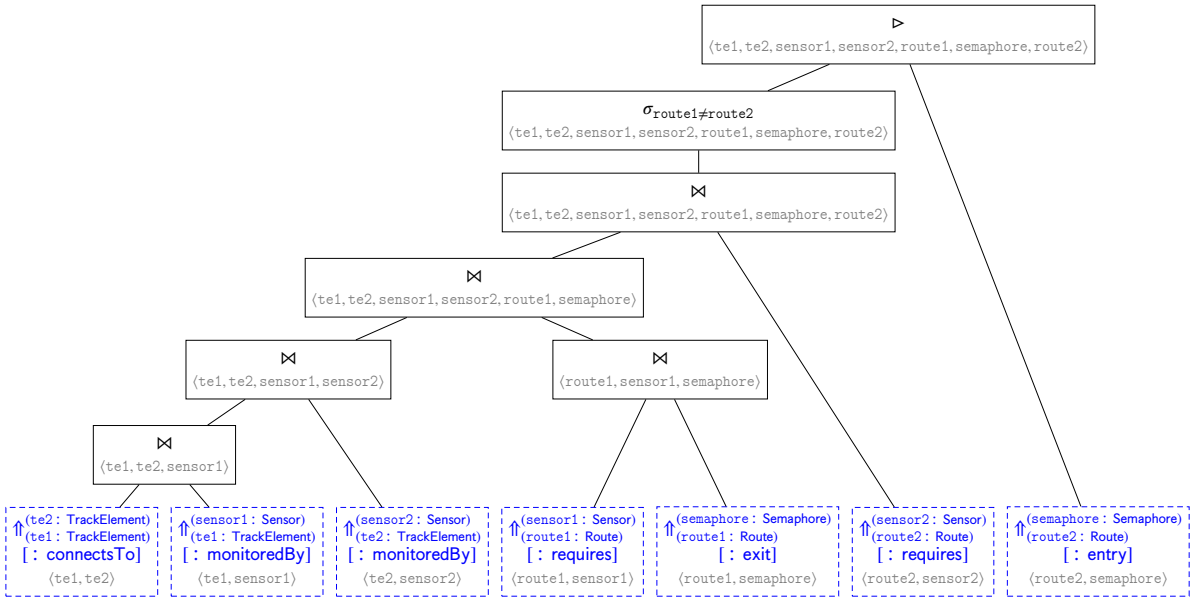


Figure E.5: Rete layout C for query SemaphoreNeighbor.

models (even for graph models with 136k elements). It is interesting to observe that while plan F – which also requires a Cartesian product – consumes significantly less memory than E, is unable to scale for model graphs with 67k elements due to timeout.

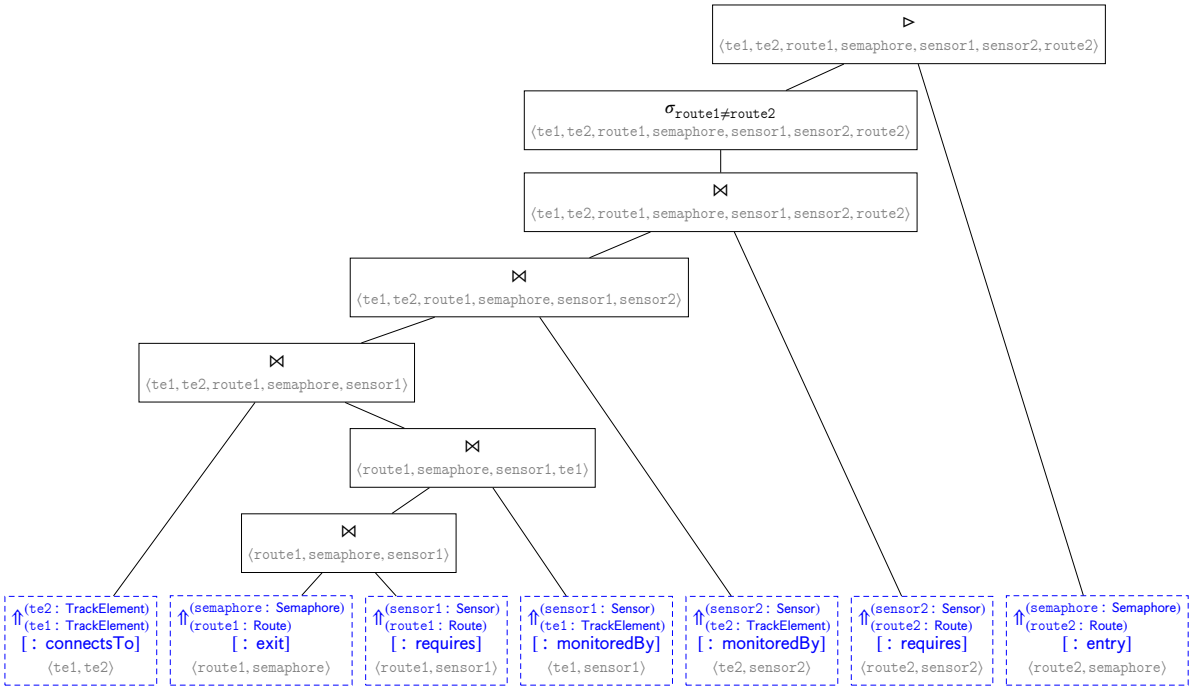


Figure E.6: Rete layout D for query SemaphoreNeighbor.

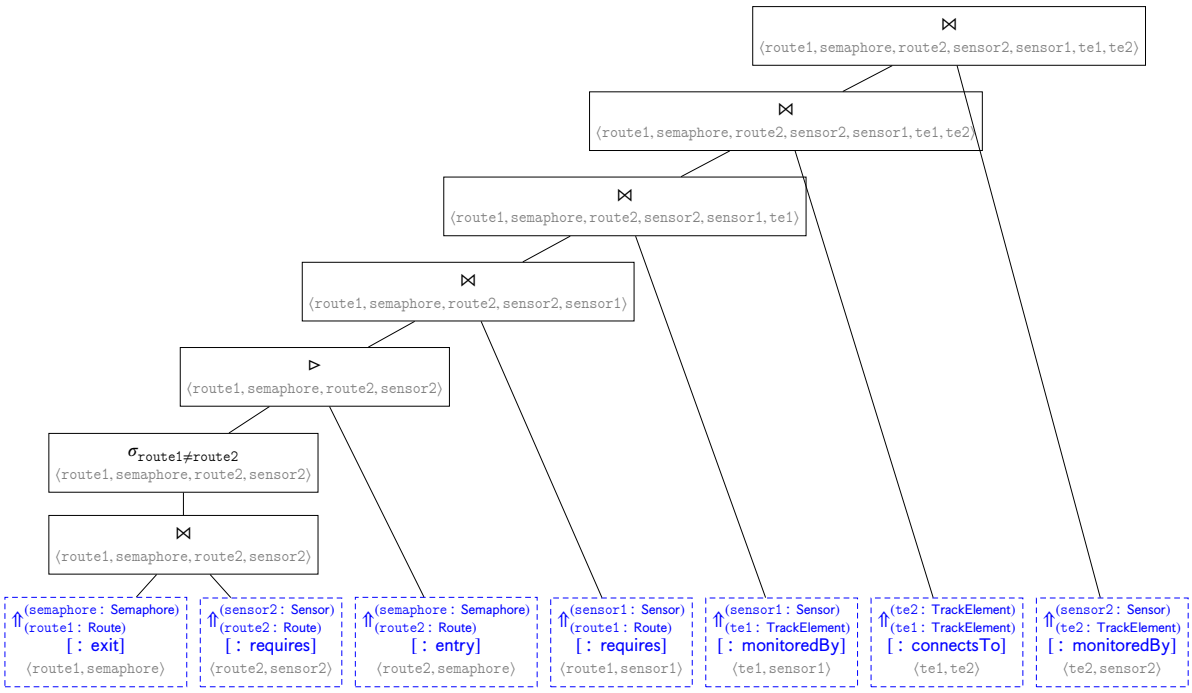


Figure E.7: Rete layout E for query SemaphoreNeighbor.

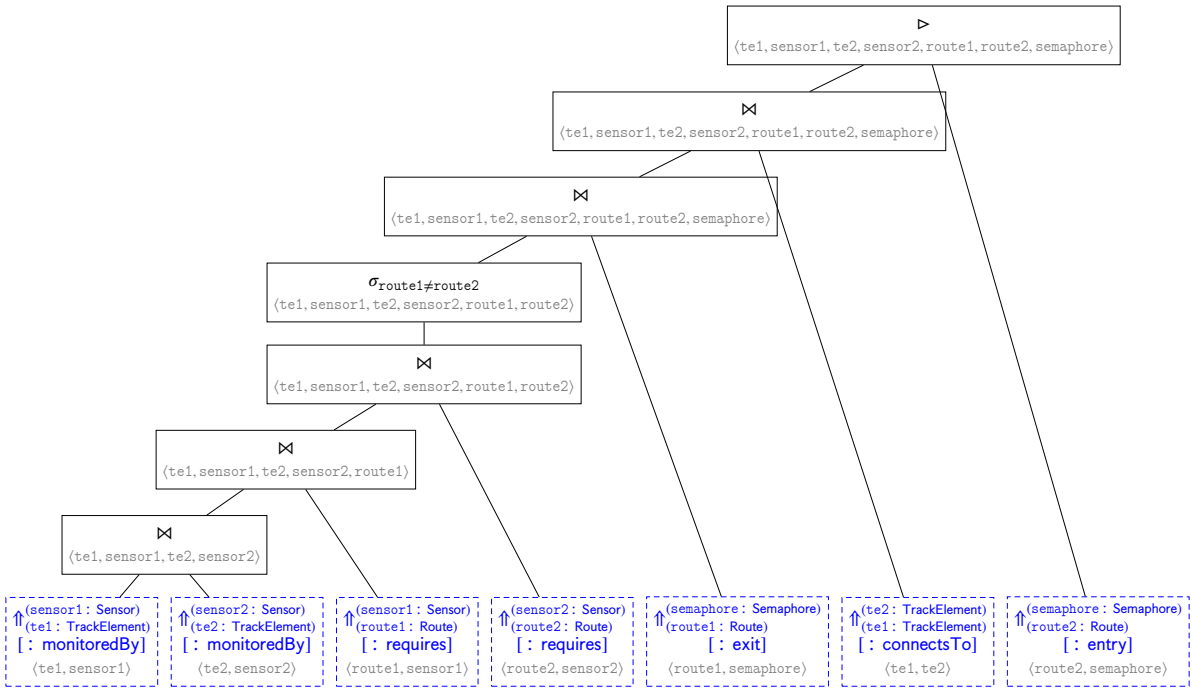


Figure E.8: Rete layout F for query SemaphoreNeighbor.

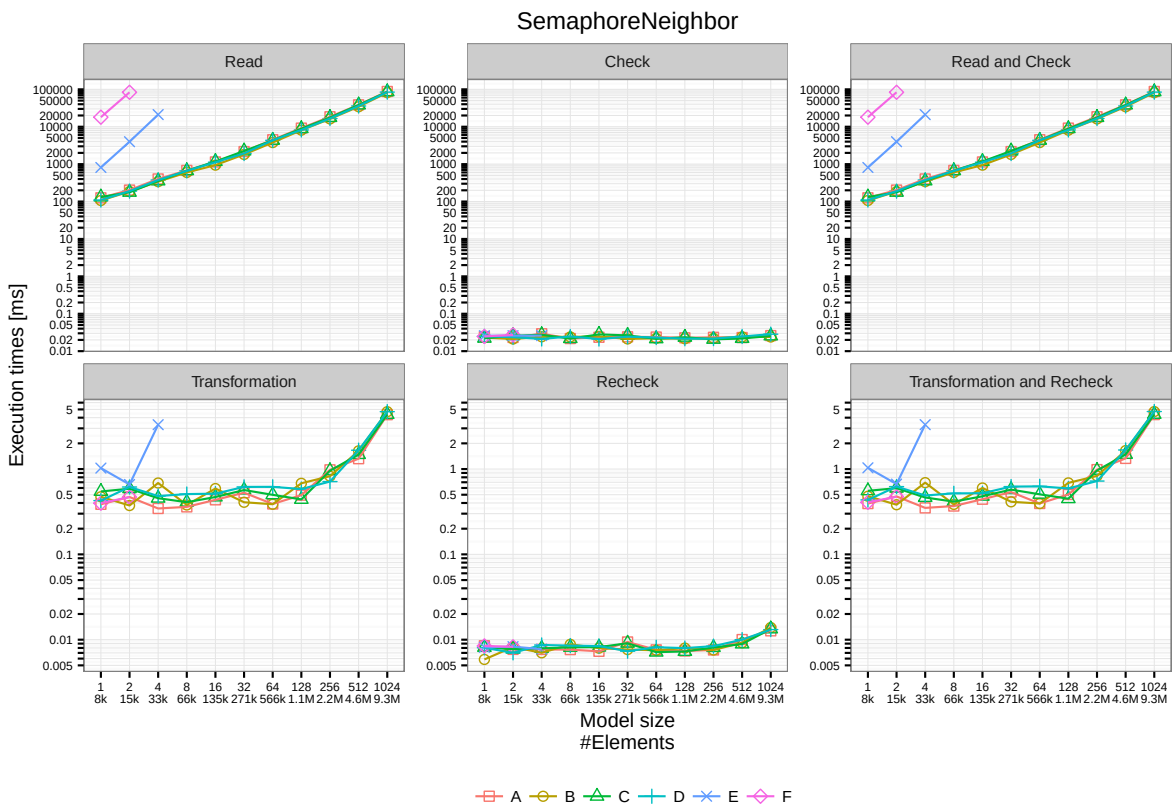


Figure E.9: Execution times for different plans of query SemaphoreNeighbor.

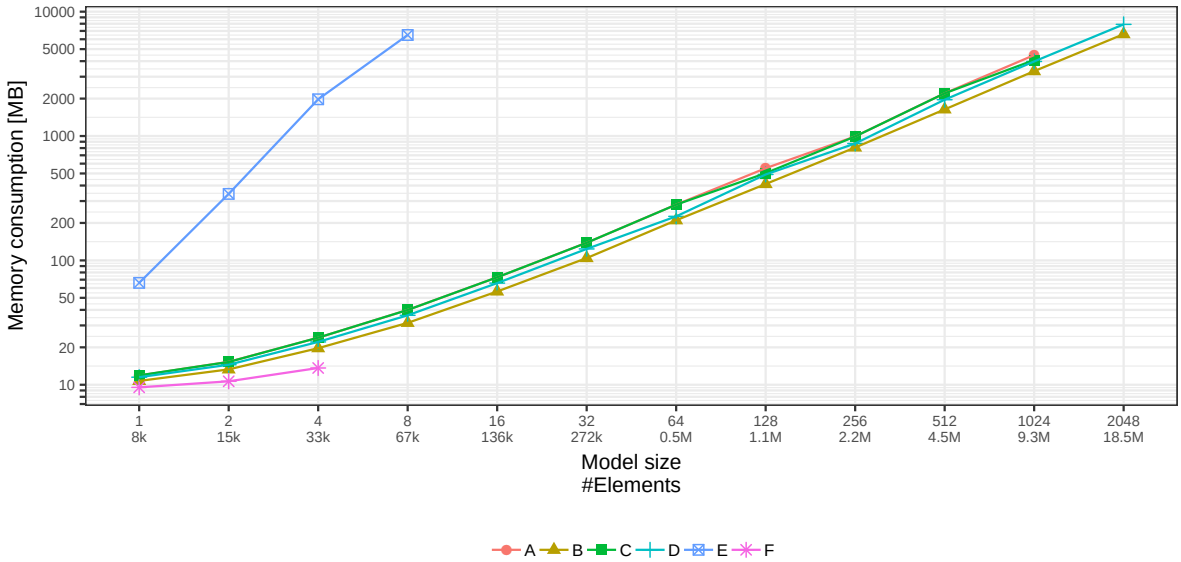


Figure E.10: Memory consumption for different plans of query SemaphoreNeighbor.