UNIVERSITY OF CALIFORNIA SAN DIEGO

Semantic Optimizations in Modern Hybrid Stores

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Rana Bijad M Alotaibi

Committee in charge:

Professor Alin Deutsch, Chair
Professor Michael J. Carey
Professor Ranjit Jhala
Professor Arun Kumar
Professor Ramesh Rao

2022

The dissertation of Rana Bijad M Alotaibi is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

This dissertation is gratefully dedicated to

My beloved parents, Bijad Alotaibi and Hussah Alzyadi

My siblings, Norah, Mohammed, Fares, and Rose.

EPIGRAPH

It is good to have an end to journey towards; but it is the journey that matters, in the end.

—Ursula K.Leguin, The Left Hand of Darkness, 1969.

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures ix

List of Tables xii

Acknowledgements xiii

Vita xvii

Abstract of the Dissertation xviii

Chapter 1 Introduction 1
1.1 Summary of Research Contributions 3
1.1.1 ESTOCADA: Bringing Semantic Optimization to Hybrid Stores 3
1.1.2 HADAD: Extending Benefits of Semantic Optimization to Hybrid Relational and Linear Algebra Computation 4
1.2 Dissertation Organization 5
1.3 Acknowledgement 6

Chapter 2 Heterogeneous Data Management 7
2.1 Mediator: Data Integration and Federated Systems 7
2.2 Hybrid Warehouses 9
2.3 Modern Hybrid Stores 10
2.4 Relationship to This Work 11

Chapter 3 Background 13
3.1 Conjunctive Query 13
3.1.1 Semantics 14
3.1.2 Query Containment and Equivalence 15
3.2 Integrity Constraints (Dependencies) 16
3.2.1 Expressing Constraints in First-Order Logic 16
3.2.2 Reasoning With Integrity Constraints: The Chase 17
3.3 Query Reformulation Under Integrity Constraints 20
3.3.1 The Chase&Backchase (C&B) Algorithm 20
3.3.2 The Provenance-Directed C&B: PACB 23

Chapter 4 ESTOCADA: Bringing Semantic Optimization to Hybrid Stores 26
4.1 Introduction 26
4.1.1 Motivating example 26
4.2 Problem statement 30
4.3 ESTOCADA Overview 32
4.4 The QBT^{XM} Language 35
4.4.1 Query Block Trees (QBT) 35
4.4.2 Integration Language: QBT^{XM} 37
4.5 Reduction from Cross-Model to Single-Model Setting 38
4.5.1 JSON Model 39
4.5.2 Key-Value Model 45
4.5.3 Graph Model 46
4.5.4 XML Model 49
4.5.5 Relational Model 51
4.5.6 Binding Patterns 51
4.5.7 Equality and Multiple Treatment of Nulls 54
4.5.8 Comparison and Arithmetic Operators 56
4.5.9 Denial Constraints 57
4.6 Encoding QBT^{XM} Queries into the Pivot Language 58
4.7 Encoding QBT^{XM} Views as Constraints 59
4.8 From Rewritings to Integration Plans 60
4.9 PACB Optimization and Extension 63
4.9.1 PACB^{OPT}: Optimized PACB 64
4.10 Extending PACB^{OPT} to Bag Semantics 66
4.11 PACB$^{OPT}_{QBT}$: Extending PACB^{OPT} to QBTs 67
4.12 Guarantees on the Reduction 67
4.13 Experimental Evaluation 69
4.13.1 Experiment Setup 70
4.13.2 Cross-Store Rewritings Evaluation 71
4.13.3 PACB vs. PACB^{OPT} 78
4.13.4 Summary of Experimental Findings 80
4.14 Conclusion 80
4.15 Acknowledgement 80

Chapter 5 HADAD: Extending Benefits of Semantic Optimization to Hybrid Relational and Linear Algebra Computation 82
5.1 Introduction 82
5.2 HADAD Optimizations 83
5.2.1 LA Pipeline Optimization 83
5.2.2 Hybrid RA-LA Optimization 84
5.3 Problem Statement 86
5.4 HADAD Overview 87
5.5 LA Reduction to the Relational Model 90

5.5.1 Supported Matrix Algebra . . . 90
5.5.2 VREM Schema and Relational Encoding . . . 91
5.5.3 LA Relational Rewriting Using Constraints . . . 98
5.6 Choice of an Efficient Rewriting . . . 99
5.7 Cost Model . . . 99
5.7.1 LA-based Sparsity Estimators . . . 100
5.7.2 Pruning Rewritings: $PACB^{++}$ . . . 101
5.8 Guarantees on the Reduction . . . 105
5.9 Experimental Evaluation . . . 107
5.9.1 Experiment Setup . . . 107
5.9.2 LA-based Experiments . . . 108
5.9.3 Hybrid (RA and LA) Experiments . . . 118
5.9.4 Experiments Takeaway . . . 127
5.10 Limitations . . . 128
5.11 Conclusion . . . 129
5.12 Acknowledgement . . . 129

Chapter 6 Related Wrok . . . 131
6.1 Related Work for ESTOCADA . . . 131
6.2 Related Work for HADAD . . . 133
6.3 Acknowledgement . . . 134

Chapter 7 Conclusion and Future Directions . . . 136

Appendix A Appendix: ESTOCADA . . . 138
A.1 Query Templates Example . . . 138
A.2 Snippet of QBT^{XM} Query Language Grammar . . . 140
A.3 Encoding of QBT^{XM} Views: V_1 and V_2 . . . 142
A.4 Encoding of QBT^{XM} Query Q_1 and Decoding RWQ_1 . . . 143

Appendix B Appendix: HADAD . . . 144
B.1 LA Operators Encoding Relations . . . 144
B.2 Key Constraints of LA Encoding Relations . . . 146
B.3 Properties of LA Operations Encoded as Constraints . . . 148
B.4 SystemML Rewrite Rules Encoded as Constraints . . . 153
B.5 Additional Results: $\mathcal{P}^{\neg Opt}$ and $\mathcal{P}^{Views}$ Pipelines Rewrites . . . 157
B.6 Additional Results: $\mathcal{P}^{\neg Opt}$ Pipelines: Naïve Cost Model . . . 159
B.7 Additional Results: $\mathcal{P}^{\neg Opt}$ Pipelines: MNC Cost Model . . . 184
B.8 Additional Results: $\mathcal{P}^{Views}$ Pipelines . . . 199

Bibliography . . . 205

LIST OF FIGURES

Figure 4.1: ESTOCADA reduction outline. 31
Figure 4.2: QBT query example. 36
Figure 4.3: View definitions expressed in QBT^{XM}. 38
Figure 4.4: JSON view V and query Q. 42
Figure 4.5: Graph view V and query Q. 47
Figure 4.6: Snippet of supported denail constraints. 58
Figure 4.7: Motivating scenario QBT^{XM} query Q_1. 59
Figure 4.8: Relational encoding of QBT^{XM} view V_2 defined in Figure 4.3. 59
Figure 4.9: Rewriting RWQ_1 of QBT^{XM} query Q_1. 60
Figure 4.10: Integration plan of the motivating scenario using Tatooine hybrid engine. . . 61
Figure 4.11: Rewriting time (28 relevant views). 72
Figure 4.12: Rewriting time (128 relevant views). 73
Figure 4.13: ESTOCADA (Relational and JSON views in PostgreSQL, text view in Solr) vs. single-store evaluation. 74
Figure 4.14: ESTOCADA (Relational and JSON views in PostgreSQL) vs. single-store evaluation. 75
Figure 4.16: PACB vs PACB^{OPT} rewriting performance. 78
Figure 4.15: Queries and rewritings evaluation in polystore engines. 79

Figure 5.1: HADAD reduction outline. 88
Figure 5.2: Snippet of $\mathcal{MMC}_{LA_{prop}}$ constraints 94
Figure 5.3: Relational encoding of view V 94
Figure 5.4: Equivalent rewritings of Q_p. 99
Figure 5.5: Relational equivalent rewritings of Q_p. 99
Figure 5.6: Naïve sparsity estimation scheme. 101
Figure 5.7: P1.1, P1.4, P1.15, and P2.12 evaluation before and after rewriting. 112
Figure 5.8: P1.13, and P1.25 evaluation before and after rewriting. 113
Figure 5.9: R speedup on $\mathcal{P}^{\neg Opt}$. 115
Figure 5.10: P2.14, P2.21, P2.25 and P2.27 evaluation before and after rewriting using V_{exp}. 116
Figure 5.11: Speedups of MorpheusR (with HADAD rewrites) over MorpheusR (without HADAD Rewrites) for pipelines P1.12, P2.10, P2.11 and P2.15 on synthetic data for a PK-FK join. 118
Figure 5.12: HADAD RW_{find} overhead as a percentage (%) of the total time ($Q_{exec} + RW_{find}$) for pipelines P1.10, P1.16, and P1.18 running on MorpheusR. . . . 119
Figure 5.13: Results of micro-hybrid benchmark using Twitter dataset. 121
Figure 5.14: Results of micro-hybrid benchmark using MIMIC dataset. 126

Figure A.1: Query templates QT_0, QT_1 and QT_2. 138
Figure A.2: Query Q produced from combining templates QT_0, QT_1 and QT_2. 139
Figure A.3: Query Q (in AsterixDB SQL++ syntax) produced from combining QT_0, QT_1 and QT_2. 139

Figure A.4: View V_1 forward and backward constraints. 142
Figure A.5: View V_2 backward constraint. 142
Figure A.6: Relational encoding of QBT^{XM} query Q_1. 143
Figure A.7: Decoding of rewriting RWQ_1. 143

Figure B.1: P1.2 evaluation before and after rewriting. 159
Figure B.2: P1.2 evaluation before and after rewriting. 160
Figure B.3: P1.6 evaluation before and after rewriting. 160
Figure B.4: P1.8 evaluation before and after rewriting. 161
Figure B.5: P1.8 evaluation before and after rewriting. 162
Figure B.6: P1.9 evaluation before and after rewriting. 162
Figure B.7: P1.10 evaluation before and after rewriting. 163
Figure B.8: P1.10 evaluation before and after rewriting. 164
Figure B.9: P1.11 evaluation before and after rewriting. 165
Figure B.10: P1.11 evaluation before and after rewriting. 166
Figure B.11: P1.12 evaluation before and after rewriting. 167
Figure B.12: P1.14 evaluation before and after rewriting. 167
Figure B.13: P1.15 evaluation before and after rewriting. 168
Figure B.14: P1.16 evaluation before and after rewriting. 168
Figure B.15: P1.16 evaluation before and after rewriting. 169
Figure B.16: P1.17 evaluation before and after rewriting. 169
Figure B.17: P1.18 evaluation before and after rewriting. 170
Figure B.18: P1.18 evaluation before and after rewriting. 171
Figure B.19: P1.25 evaluation before and after rewriting. 171
Figure B.20: P2.1 evaluation before and after rewriting. 172
Figure B.21: P2.2 evaluation before and after rewriting. 172
Figure B.22: P2.3 evaluation before and after rewriting. 172
Figure B.23: P2.4 evaluation before and after rewriting. 173
Figure B.24: P2.4 evaluation before and after rewriting. 174
Figure B.25: P2.5, P2.6 and P2.8 evaluation before and after rewriting. 175
Figure B.26: P2.9 evaluation before and after rewriting. 175
Figure B.27: P2.10 evaluation before and after rewriting. 176
Figure B.28: P2.11 evaluation before and after rewriting. 177
Figure B.29: P2.11 evaluation before and after rewriting. 178
Figure B.30: P2.13 evaluation before and after rewriting. 179
Figure B.31: P2.14 evaluation before and after rewriting. 179
Figure B.32: P2.15 evaluation before and after rewriting. 180
Figure B.33: P2.15 evaluation before and after rewriting. 181
Figure B.34: P2.16 evaluation before and after rewriting. 181
Figure B.35: P2.18 evaluation before and after rewriting. 182
Figure B.36: P2.18 evaluation before and after rewriting. 183
Figure B.37: P1.2 evaluation before and after rewriting. 184
Figure B.38: P1.6 evaluation before and after rewriting. 184

Figure B.39: P1.8 evaluation before and after rewriting. 185
Figure B.40: P1.9 evaluation before and after rewriting. 185
Figure B.41: P1.10 evaluation before and after rewriting. 186
Figure B.42: P1.11 evaluation before and after rewriting. 187
Figure B.43: P1.12 evaluation before and after rewriting. 188
Figure B.44: P1.14 evaluation before and after rewriting. 188
Figure B.45: P1.15 evaluation before and after rewriting. 189
Figure B.46: P1.16 evaluation before and after rewriting. 189
Figure B.47: P1.17 evaluation before and after rewriting. 190
Figure B.48: P1.18 evaluation before and after rewriting. 190
Figure B.49: P1.25 evaluation before and after rewriting. 191
Figure B.50: P2.1 evaluation before and after rewriting. 191
Figure B.51: P2.4 evaluation before and after rewriting. 192
Figure B.52: P2.5, P2.6, P2.8 and P2.9 evaluation before and after rewriting. 193
Figure B.53: P2.10 evaluation before and after rewriting. 194
Figure B.54: P2.11 evaluation before and after rewriting. 195
Figure B.55: P2.13 evaluation before and after rewriting. 196
Figure B.56: P2.14 evaluation before and after rewriting. 196
Figure B.57: P2.15 evaluation before and after rewriting. 197
Figure B.58: P2.16 evaluation before and after rewriting. 197
Figure B.59: P2.18 evaluation before and after rewriting. 198
Figure B.60: P1.2 and P1.3 evaluation before and after rewriting. 199
Figure B.61: P1.4 and P1.11 evaluation before and after rewriting. 200
Figure B.62: P1.17, P1.19, P1.20 and P1.21 evaluation before and after rewriting. 201
Figure B.63: P1.22, P1.23, P1.24 and P1.29 evaluation before and after rewriting. 202
Figure B.64: P2.2, P2.4 and P2.5 evaluation before and after rewriting. 203
Figure B.65: P2.9, P2.11 and P2.16 evaluation before and after rewriting. 204

LIST OF TABLES

Table 4.1: Models and languages supported by ESTOCADA. 29
Table 4.2: Snippet of $\mathcal{VRELJ}$ schema. 39
Table 4.3: Snippet of $\mathcal{VRELG}$ schema. 46
Table 4.4: Snippet of $\mathcal{VRELX}$ schema. 49

Table 5.1: Snippet of the $\mathcal{VRELM}$ schema. 90
Table 5.2: LA benchmark pipelines (part 1). 108
Table 5.3: LA benchmark pipelines (part 2). 109
Table 5.4: Overview of used real datasets. 109
Table 5.5: Syntactically generated dense datasets. 109
Table 5.6: Matrices used for each matrix name in a pipeline. 110
Table 5.7: LA pipelines used in micro-hybrid benchmark. 122

Table B.1: The $\mathcal{VRELM}$ schema (part 1). 144
Table B.2: The $\mathcal{VRELM}$ schema (part 2). 145
Table B.3: Key constraints of LA operators relations (part 1). 146
Table B.4: Key constraints of LA encoding relations (part 2). 147
Table B.5: Properties of addition of matrices encoded as integrity constraints. 148
Table B.6: Properties of addition and transposition of matrices encoded as integrity constraints. 149
Table B.7: Properties of inverse, determinant and trace of matrices encoded as integrity constraints. 150
Table B.8: Properties of direct sum and exponential of matrices encoded as integrity constraints. 151
Table B.9: Matrix decompositions properties captured as integrity constraints (part1). . . 151
Table B.10: Matrix decompositions properties captured as integrity constraints (part2). . 152
Table B.11: UnnecessaryAggregates rewrite rules encoded as integrity constraints. . . . 153
Table B.12: PushdownUnaryAggTransposeOp rewrite rules encoded as integrity constraints. 154
Table B.13: SimplifyTraceMatrixMult rewrite rules encoded as integrity constraint. . . . 154
Table B.14: SimplifySumMatrixMult rewrite rules encoded as integrity constraints. . . . 155
Table B.15: SimplifyColWiseAgg rewrite rules encoded as integrity constraints. 155
Table B.16: SimplifyRowWiseAgg rewrite rules encoded as integrity constraints. 156
Table B.17: PushdownSumOnAdd rewrite rules encoded as integrity constraints. 156
Table B.18: ColSums/rowSumsMVMult rewrite rules encoded as integrity constraints. . 156
Table B.19: $\mathcal{P}^{\neg Opt}$ pipelines rewrites (part 1). 157
Table B.20: $\mathcal{P}^{\neg Opt}$ pipelines rewrites (part 2). 157
Table B.21: The set of views V_{exp}. 157
Table B.22: $\mathcal{P}^{Views}$ pipelines rewrites. 158

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of many people.

First and foremost, I am deeply thankful for the mentorship and support that my advisor Professor Alin Deutsch provided over the past several years. Through his mentorship, he challenged me to think about the practical implications and impact of my research. He has been instrumental in my development as a researcher. Despite his busy schedule, his door has always been open to me every time I need his advice. Alin leads by example and makes sure his students thrive not just as researchers but as individuals. He is an exceptional advisor and a wonderful human being. I could not have asked for a better Ph.D. advisor and role model during this period of my professional development.

I would like to profoundly thank Professor Ioana Manolescu for her involvement in my research and her commitment to this work. She is a dedicated and caring mentor who always sets the bars high. She is supportive, always there for me, and willing to help. It has been an immense pleasure working with her, and I look forward to having an opportunity to collaborate with her in the future.

I was also fortunate to have productive collaborations with Bogdan Cautis, Damian Bursztyn, and Stamatis Zampetakis. Their collaborations helped immensely in making progress towards this thesis. Working with them has been indeed a pleasure and an amazing experience. None of this would have been possible without them, and I have learned so much from them. Thank you, Damian and Stamatis for the great time I spent with you and your families in Paris.

My sincere thanks also go to my committee members: Professor Arun Kumar, Professor Michael Carey, Professor Ramesh Rao, and Professor Ranjit Jhala, for agreeing to be on my thesis committee. I thank them for their constructive feedback and comments during my thesis proposal. I also thank Professor Yannis Papakonstantinou and Professor Victor Vianu for their valuable feedback about my work and presentations during UCSD's database lab seminars.

I have had the privilege of working with some world-class researchers in their respective

fields: Abdul Quamar, Alekh Jindal, Brandon Haynes, Carlo Curino, Chuan Lei, Fatma Ozcan, Jesús Camacho Rodríguez, and Jyoti Leek, whose mentorship and feedbacks were immensely invaluable during my internships at IBM Research and Microsoft Gray Systems Lab. These internships enriched my experiences in applied industrial research. They taught me to see research from different angles, emphasizing product impact.

This journey would not be nearly as enjoyable without all the fantastic people I met and shared experiences with at UC San Diego. I am lucky to call them my friends: Ainur Smagulova, Abdulrahman Alkhelaifi, Angel Zhang, Chunbin Lin, Coko Kuratomi, Ganz Chockalingam, Ian Chen, Jianguo Wang, Konstantinos Zarifis, Naif Alhajri, Nikolaos Koulouris, Supun Nakandala, Tara Mirmira, Vicky Papavasileiou, Vraj Shah, Wael Farhan, Yifei Yang, Yuhao Zhang, Zach Meyer, and many others. Thank you for being on this journey with me.

I extend my gratitude to King Abdulaziz City for Science and Technology (KACST), Riyadh, Saudi Arabia, for providing me with generous fully-funded Master and Ph.D. fellowships.

Outside of academia, I owe my deepest gratitude to my friend back home: Muneera Alhoshan, whose calls and support kept me motivated through the tough times even when I was not sure I would make it.

Last but not least, I owe this dissertation to my parents: Bijad and Hussah, and my siblings: Norah, Mohammed, Fares, and Rose. I cannot thank them enough for their endless love, encouragement, and support over the past years. It may not have been easy for them to have me away from home. I thank them for being my biggest believer and encouraging me to pursue my dreams. No words can express my deepest gratitude towards them, and I could not have gotten this far without them.

Chapter 1 of this dissertation contains material from "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue" by Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis, which appeared in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019). The dissertation author was the

primary investigator of this paper.

Chapter 1 of this dissertation contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu, which appeared in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2021). The dissertation author was the primary investigator of this paper.

Chapter 4 of this dissertation contains material from "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue" by Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis, which appeared in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019). The dissertation author was the primary investigator of this paper.

Chapter 4 of this dissertation contains material from "ESTOCADA: Towards Scalable Polystore Systems" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, Moustafa Latrache, Ioana Manolescu, and Yifei Yang, which appeared in Proceedings of the VLDB Endowment (PVLDB 2020). The dissertation author was the primary investigator of this paper.

Chapter 5 of this dissertation contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu, which appeared in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2021). The dissertation author was the primary investigator of this paper.

Chapter 5 of this dissertation contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries (Extended Version)" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. This paper is on ArXiv. The dissertation author was the primary investigator of this paper.

Chapter 6 of this dissertation contains material from "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue" by Rana Alotaibi, Damian Bursztyn, Alin Deutsch,

Ioana Manolescu, and Stamatis Zampetakis, which appeared in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019). The dissertation author was the primary investigator of this paper.

Chapter 6 of this dissertation contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu, which appeared in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2021). The dissertation author was the primary investigator of this paper.

Chapter 6 of this dissertation contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries (Extended Version)" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. This paper is on ArXiv. The dissertation author was the primary investigator of this paper.

VITA

2005-2010	B. S. in Information Systems, Al-Imam University, Riyadh, Saudi Arabia
2014-2016	M. S. in Computer Science, University of California San Diego
2017	Research Intern, IBM Research, Almaden, San Jose, CA
2018	Research Intern, IBM Research, Almaden, San Jose, CA
2021	Research Intern, Microsoft Gray Systems Lab (GSL), Redmond, WA
2016-2022	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu, "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics", *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 23-35, 2021.

Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu, "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics (Extended Version)", *arXiv preprint arXiv:2103.12317*, 2021.

Rana Alotaibi, Chuan Lei, Abdual Quamar, Vasilis Efthymiou, and Fatma Özcan, "Property Graph Schema Optimization for Domain-Specific Knowledge Graphs", *Proceedings of International Conference on Data Engineering (ICDE))*, pages 924-935, 2021.

Chuan Lei, **Rana Alotaibi**, Abdual Quamar, Vasilis Efthymiou, and Fatma Özcan, "Property Graph Schema Optimization for Domain-Specific Knowledge Graphs", *arXiv preprint arXiv:2003.11580*, 2020.

Rana Alotaibi, Bogdan Cautis, Alin Deutsch, Moustafa Latrache, Ioana Manolescu, and Yifei Yang, "Estocada: Towards Scalable Polystore Systems", *Proceedings of Very Large Data Bases Conference (VLDB)*, pages 2949-2952, 2020.

Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis, "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue.", *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 1660-1677, 2019.

Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis, "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue.", *BDA Conference on Data Management*, 2019.

ABSTRACT OF THE DISSERTATION

Semantic Optimizations in Modern Hybrid Stores

by

Rana Bijad M Alotaibi

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Alin Deutsch, Chair

In recent years, big data applications often involve dealing with diverse datasets in terms of structure: relations flat or nested, complex-structure graphs, documents (JSON or XML), poorly structured logs, or even text data. To handle the heterogeneity of the data, application designers usually rely on several data stores used side-by-side, each supporting a different data model, associated query language (or data access API), and very efficient for some, but not all, kinds of processing on the data. Systems capable of querying disparate data in this fashion are advocated by the database community under terms such as hybrid- or poly-stores.

These systems provide no support for semantic query optimizations, which include (*i*) exploiting possible data redundancy when the same data may be accessible (with different

performance) from distinct data stores; (*ii*) taking advantage of partial query results (in the style of materialized views), which may be available in the stores; and (*iii*) reasoning semantically about various data models and query operations' properties, which can enhance the hybrid workload performance. Motivated by these optimization opportunities, this dissertation makes the following two main contributions:

We design and demonstrate ESTOCADA, an extensible lightweight framework for providing semantic query optimization on top of hybrid stores without modifying their internals. ESTOCADA transparently enables each query to benefit from the best combination of stored data and available processing capabilities. It leverages recent advances in the area of view-based query rewriting under constraints, which we use to describe various data models and stored data. We demonstrate the effectiveness of our approach with an experimental evaluation using the MIMIC real-world dataset and show significant performance gains achieved by ESTOCADA.

Going beyond query workloads covering a variety of data models (relational, JSON, Graph, XML) in hybrid stores, modern applications increasingly need to blend querying and learning on the data, which is primarily expressed using a mix of relational algebra (RA)- and linear algebra (LA)-based languages. Existing specialized solutions for evaluating such hybrid analytical tasks either optimize RA and LA tasks separately, exploiting only RA properties while leaving LA-specific optimizations unexploited, or focus heavily on physical optimizations, leaving semantic query optimization opportunities unexplored. In our second contribution, we take a major step towards filling this gap by proposing HADAD. The novelty of HADAD is to extend the benefits of semantic query rewriting and view-based optimization introduced in ESTOCADA to LA computations, crucial for ML hybrid analytical tasks. Our solution can be naturally and portably applied on top of pure LA and hybrid RA-LA platforms. An extensive empirical evaluation shows that HADAD yields significant performance gains on diverse workloads, ranging from LA-centered to hybrid RA-LA workloads.

Chapter 1

Introduction

Nowadays, big data applications often involve dealing with diverse datasets in terms of structure: relations flat or nested, complex-structure graphs, documents (JSON or XML), poorly structured logs, or even text data. Such datasets are routinely hosted in heterogeneous stores. One reason is that the fast pace of application development prevents consolidating all the sources into a single data format and loading them into a single store. In addition, a system efficient on some tasks may perform poorly or not support other tasks, making it impossible to use a single data management system for a given application. Instead, the data model often dictates the choice of the store, e.g., relational data gets loaded into a relational or "Big Table"-style system, JSON documents in a JSON store [4, 11], graphs in a Graph store [14]. Heterogeneous stores also "accumulate" in an application along the time, e.g., at some point, one decision is made to host dataset D_1 in PostgreSQL and D_2 in MongoDB. Later on, another application needs to use D_1 and D_2 together. The database community advocates systems capable of exploiting diverse big data systems in this fashion under the term *hybrid stores*[1] *(a.k.a. polystores)* [68, 136, 27, 95, 35, 55].

Current hybrid stores support multiple query languages to seamlessly query datasets that reside in multiple backends. This feature distinguishes them from previous federations [49, 134],

[1] We use the terms hybrid stores, hybrid systems and polystores interchangeably in this monograph.

which were based mainly on SQL. Users are often wedded to SQL for querying structured data, semi-structured language for querying JSON data, path-oriented language for querying Graph data, etc. Such mixed query notations workload is often referred to as a *hybrid* worklaod [68].

Query evaluation in hybrid systems recalls, to some extent, mediator systems [49, 134, 41, 103, 64, 110, 139]; in both cases, sub-queries, each expressed in its own query language, are delegated to their corresponding underlying stores, while the remaining operations are applied in the upper integration layer. Hybrid systems process a query assuming that each of its input datasets is available in exactly one store (often chosen for its support of a given data model).

Further, going beyond query workloads covering a variety of data models (relational, JSON, Graph, XML) in hybrid stores, modern applications increasingly need to blend querying and learning on the data, which is primarily expressed using a mix of relational algebra (RA)- and linear algebra (LA)-based languages. To run such hybrid RA-LA analytical tasks, various works propose *specialized* hybrid RA-LA integration solutions, where both algebraic styles can be used together [105, 52, 18, 94, 97, 137, 44]. Some solutions [105, 94, 18] suggest extending RDBMS to treat LA objects as first-class citizens by using built-in functions to express LA operations. Others offer to call LA packages through user-defined functions (UDFs), where libraries like NumPy are embedded in the host language. Moreover, some systems [137, 44] only optimize LA expressions by converting them into RA expressions, optimizing the latter, and then converting the result back to an (optimized) LA expression. They only focus on optimizing LA pipelines containing operations that can be expressed in RA. More advanced hybrid RA-LA soultion [52] speeds up LA pipelines over large joins by pushing computation into each joined table, thereby avoiding expensive materialization. Recently, several soultions [97] focus on *low-level* physical optimization by exploiting data layouts (e.g., column-wise) to choose LA operators' physical implementations.

This dissertation identifies optimization opportunities in the scope of hybrid solutions/systems, of which special hybrid RA-LA integration systems are a subset. These solutions provide

no support for *cross-models semantic query optimization*. Semantic optimization uses semantic knowledge about the data and query operations to *rewrite* a query into another equivalent form that can be executed more efficiently than the original query [26]. In a hybrid setting, this includes (*i*) exploiting possible data redundancy: the same data could be stored in several stores, some of which may support a query operation much more efficiently than others; (*ii*) taking advantage of the presence of partially computed query results (in the style of materialized views), which may be available in one or several stores, when the data model of the queried dataset differs from the data model of the store hosting the view; and (*iii*) reasoning semantically about various data models and query operations supported by them, which can enhance hybrid queries performance.

Enabling semantic query optimizations in a hybrid setting is challenging since user queries, views, and query rewritings are expressed across different data models and query languages. Existing semantic query optimization solutions were designed to work within a *single* data model and query language (i.e., mostly relational or XML) [103, 34, 64, 26].

1.1 Summary of Research Contributions

We critically need a new and innovative solution to enable the aforementioned optimization opportunities and tackle challenges in a hybrid setting. With this in mind, this dissertation presents new innovations in enabling *cross-models semantic query optimization* on top of hybrid systems, with no need to modify their internals. In the following subsections, we describe the contributions of this dissertation in further detail.

1.1.1 ESTOCADA: Bringing Semantic Optimization to Hybrid Stores

We propose ESTOCADA (Chapter 4), a novel approach for allowing an application to transparently exploit data stored in a set of heterogeneous stores (in a hybrid setting) as a set of potentially overlapping data fragments (views). Further, if fragments store results of partial

computations applied on the data, we show how to speed up queries using them as materialized views. This reduces query processing time and seeks to take the maximum advantage of each store's efficient query processing features. Importantly, our approach does not require any change to the application code. ESTOCADA enables such semantic optimization based on a common abstraction that facilitates unified reasoning: a relational model endowed with integrity constraints, which we use to capture the key aspects of the data models in circulation today, properties of query operation, and stored views. The system fully appeared in SIGMOD'19 [31] and later was demonstrated in PVLDB'20 [32]. The technical contributions we make in ESTOCADA can be summarized as follows:

- We propose an extensible lightweight approach for bringing semantic query optimization to hybrid systems.
- We formalize the cross-models rewriting problem in the context of hybrid systems.
- We design a reduction from cross-models view-based rewriting to relational rewriting under integrity constraints.
- We prototype our cross-models views-based rewriting approach on top of some existing hybrid systems.
- We optimize the state-of-the-art rewriting algorithm to scale to a hybrid setting.
- We show that our approach improves the performance in natural scenarios for both cross-models and single model user queries.

1.1.2 HADAD: Extending Benefits of Semantic Optimization to Hybrid Relational and Linear Algebra Computation

As described previously, existing specialized hybrid RA-LA solutions do not support semantic query optimization, which includes views-based, integrity constraint-based, and LA

property-based rewriting and can bring significant performance saving in hybrid RA-LA and even plain LA settings.

To enable such fruitful optimizations, we introduce HADAD (Chapter 5), which extends the benefits of semantic query rewriting and view-based optimization introduced in ESTOCADA to LA computations, which are crucial for ML-hybrid workloads. Our solution can be naturally and portably applied on top of hybrid RA-LA and pure LA platforms. The system was partially demonstrated at PVLDB'20 [32] and later fully appeared in SIGMOD'21 [33]. The technical contributions we make in this work can be summarized as follows:

- We propose an extension to ESTOCADA [31] (Chapter 4). In particular, we extend ESTOCADA's intermediate relational abstraction to support reasoning about LA operations. The RA part is already captured in the target formalism.
- To the best of our knowledge, our approach is the first work that brings views-based rewriting under integrity constraints in the context of LA-based pipelines and hybrid analytical tasks.
- We extend the query rewriting engine, integrating two different cost models, to help prune out inefficient rewritings at the early stages of the rewriting candidates' search phase.
- We prototype our approach on top of popular LA-oriented and hybrid RA-LA platforms.
- We conduct an extensive set of empirical experiments on typical LA- and hybrid-based expressions, which show the benefits of HADAD.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 briefly gives an overview of heterogeneous data management systems and their relationship to this work. In

Chapter 3, we describe the necessary background to understand the details in this monograph. Chapter 4 introduces ESTOCADA, a novel lightweight framework that enables semantic query optimization on top of hybrid systems. Chapter 4 describes HADAD, which capitalizes on ESTOCADA and extends its benefits of semantic query rewriting and view-based optimization to LA computations. We discuss ESTOCADA and HADAD related work in Chapter 6. Finally, Chapter 7 concludes this dissertation and discusses potential future research directions.

1.3 Acknowledgement

This chapter contains material from "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue" by Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis, which appears in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019). The dissertation author was the primary investigator of this paper.

This chapter contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu, which appears in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2021). The dissertation author was the primary investigator of this paper.

Chapter 2

Heterogeneous Data Management

Since the 1970's, the data management field has faced several challenges raised by dealing with heterogeneous data sources, yielding a proliferation of approaches and systems which range from mediator systems [49, 134, 41, 103, 64, 110, 139, 47], of which federated database systems (FDSs) are a subset, and some of which use XML or Ontology as the integration model [64, 110, 116, 139] to hybrid warehouses [132, 133, 66, 99], which are special FDSs providing an integration between Hadoop-like big data platforms and parallel data warehouses and finally modern hybrid stores (a.k.a. polystores) [68, 136, 27, 95, 35, 55] (also called hybrid systems).

This chapter briefly gives an overview of the evolution of heterogeneous data management systems over the years. This overview is not intended to be comprehensive, and we refer a reader to a more robust treatment of these topics [126, 96, 119, 140, 46] for details beyond the scope of this thesis.

2.1 Mediator: Data Integration and Federated Systems

Traditional heterogeneous data management systems aim at providing transparent access to a collection of autonomous (i.e., managed independently) heterogeneous data sources. Two

main centralized system architectures were devised for that purpose: data warehouse [86] and mediator [67]. They both rely on the design of a global integration schema (*global schema* in short), which provides an *homogeneous* view of the source data, and w.r.t. which queries are formulated in a single language (e.g., SQL, XQuery). This way, the existence of multiple heterogeneous data sources is hidden from applications and users. However, the two architectures significantly differ in how they answer queries based on the source data. In a warehouse, data is first extracted from the sources and then transformed so as to be loaded as a global schema instance within a database. Thus, the warehouse approach eliminates heterogeneity by bringing all the data into a single store. In contrast, in a mediator, data remains in the heterogeneous sources, and every incoming query is decomposed into subqueries to be evaluated by the sources. Possible post-processing steps are usually applied in the mediator to integrate their results and provide the final query answer to the user.

Mediator systems follow *mediator-wrapper* architecture. In this architecture, each data source has a *wrapper* that passes on information about the source query processing capabilities and the local schema to the mediator (middleware). Wrappers translate queries received from the mediator, expressed in a common query language, to the query language of the source. Moreover, they reformat answers of each subquery from the source to conform to the internal data model of the middleware.

In mediator systems, the data is stored in the data sources and organized under local schemas. Hence, in order to for the mediator system to answer queries, there must be some descriptions (or explicit mapping rules) of how the relations in the sources are related to the ones in the global schema. These mapping rules virtually specify the global schema instance that applications and users can access through the mediator. There are two main types of mapping [101]:

Global-As-View (GAV). The global schema is defined as a set of views over the data source schemas. These view definitions indicate how the tuples (i.e., elements) of the global

schema relations can be derived from the tuples of the data source schemas. That is, for each relation *R* in the global schema, we write a query over the source relations specifying how to obtain R's tuples from the data sources schema. In a GAV mediator [101, 49, 134, 41, 79, 47], a query reformulation is simply unfolding, i.e., replacing each global schema relation in the query with its view definition. The reason is that relations in the global schema are defined in terms of the source relations; we only need to unfold the definitions of the global schema relations.

Local-As-View (LAV). The LAV approach is the opposite of the GAV approach. In a LAV mediator [103, 64, 110, 102], the global schema definition exists, and each data source schema is defined as a view over it. In other words, the contents of a data source are described as a query over the global schema. The query reformulation in LAV is not relatively straightforward as in GAV because it is not possible to simply unfold the definitions of the relations in the global schema. It amounts to finding combinations of views, hence of local schema tuples, that satisfy the query.

In [75], the authors describe the GLAV language that combines the expressive power of LAV and GLAV. They show that the query reformulation complexity is the same as for LAV.

Data Integration vs. Federated Systems. Data integration systems offer a level of data integration beyond what FDSs provide in that their global schema relation's tuples can be combined from multiple data sources' relations. In contrast, FDSs do not offer the ability to create such a composite "virtual" relation. Instead, the global schema in FDSs is just a collection of data sources' relations conforming to the schema model (primarily relational).

2.2 Hybrid Warehouses

With the emergence of Hadoop and its advantages in high-volume data processing, there has been a substantial move towards "Big Data" systems for massive data storage (i.e., HDFS) and analysis. Many applications (e.g., healthcare and enterprise businesses) require co-analyzing

data stored in HDFS and parallel relational database management systems (RDBMS), which have long been a superior storage and query processing engine for data warehousing applications. This has created a need for a special integration between Hadoop-like big data processing platforms (e.g., SQL-on- Hadoop) and parallel RDBMS.

Several solutions have emerged and most of these solutions statically *pre-load* HDFS data into the relational databases [22, 7, 23] through connectors [8, 22], bulk loading [7, 23] or external tables [1, 7], whereas some other solutions move the relational database data to the HDFS side [23, 22]. As stated in [133, 132], it is not always feasible to move the HDFS data into the relational database side since the HDFS data is typically very large, and bulk reading of the HDFS data into the database might incur an unnecessary load for the RDBMS resources [66]. Moreover, the parallel RDBMS data gets updated frequently. Thus, it is not beneficial to move the data to the HDFS side since it does not support updates properly.

Hybrid warehouses [132, 133, 66, 99, 38] propose to leave data in the storage engine where the data was originally located and allow splitting the query processing between the parallel RDBMS and HDFS side (i.e., Hadoop-like big data processing platform). They share some similarities with federated database systems. Both provide a global schema that conforms to a single data model (relational) and describes the data stored in RDBMS and HDFS. Further, they feature a single standard query language to the application layer. However, the key feature that distinguishes hybrid warehouses from other federation systems is that they follow the *lightweight-middleware* principle [46]. They mainly utilize the parallel RDBMS query engine as a *middleware* to orchestrate the query processing between two massively parallel systems.

2.3 Modern Hybrid Stores

Recently, we have witnessed a proliferation of data sources of different data models and query languages, which has rendered a mediator approach [10, 130, 80, 46], where using a *single*

global schema to interact with all data sources is unfeasible. Developed specifically for this setting, hybrid systems (a.k.a. polystores) [68, 136, 27, 95, 35, 55] do not hide the heterogeneity of the data sources and are proposed to combine domain-specific databases that specialize in specific data models and execution. They provide integrated access to a number of heterogeneous data stores through one or more query languages [10, 46]. For example, BigDAWG [68], at its core, is the abstraction of an *island*, a collection of data stores of a common data model, which can be accessed with a single query language. For instance, SQL is used to pose queries in the scope of the relational island. CloudMDSQL [95] and SparkSQL [35] support SQL-like queries with the extended capabilities for subqueries expressed in terms of each data store's native query interface/language. Myria [136] provides an imperative-declarative hybrid language called MyriaL. The language resembles relational queries but with an imperative extension. RHEEM [27] supports a dataflow language for users to specify their tasks in an imperative form. The language can be seen as an extension of PigLatin for cross-platform settings, where users can specify which platform to execute their queries (or sub-queries).

Moreover, these systems utilize (or extend) various query processing techniques from classical distributed database systems (e.g., data/function shipping, query decomposition). Query optimization is also supported by either a cost- or heuristics-based approach [46].

2.4 Relationship to This Work

Our work mainly focuses on enabling semantic query optimization (including cross-models views- and integrity constraints-based rewriting) on top of hybrid systems. Mediator systems [103, 64, 110, 139] that follow the LAV approach address semantic optimization mostly in a single-model (typically relational or XML) setting, where cross-models query rewriting does not occur. Moreover, several information integration systems [102, 81] focus on finding not-necessarily equivalent (but maximally-contained) rewritings. In contrast, this thesis targets

equivalent query rewriting, which leads to a very different algorithm. Further, setting our work apart is the scale and usage of integrity constraints.

Chapter 3

Background

In this chapter, we provide a brief introduction to conjunctive queries [50, 26], query containment and equivalence [50, 26], integrity constraints [26] and query reformulation under constraints [84]. These concepts are at the core of the approaches proposed in this monograph.

3.1 Conjunctive Query

A conjunctive query (or simply CQ) is an expression of the form:

$$Q(\vec{x}) \text{:- } R_1(\vec{y}_1), \ldots, R_n(\vec{y}_n)$$

where each R_i for every $i = 1, ..., n$ is a predicate (relation) of some finite arity, and Q is a *fresh* relation name, and it is called intensional relation since its content is given by definition through the query. $\vec{x}, \vec{y}_1, \ldots, \vec{y}_n$ are free tuples of variables or constants. Each $R_i(\vec{y}_i)$ is called a relational atom or subgoal. It is also called extensional relation since it is provided as input to the query.

The expression $Q(\vec{x})$ is the *head* of the query, while the conjunction of relational *atoms* $R_1(\vec{y}_1), \ldots, R_n(\vec{y}_n)$ is its *body*. The body can contain a conjunction (possibly empty) of relational

equality *atoms* of the form $y = y'$. All variables in the head are called *distinguished* variables. Also, every variable in $\vec{x}$ ($\vec{x}$ can be empty) must appear at least once in $\vec{y}_1, \ldots, \vec{y}_n$ tuples. Below some examples of conjunctive queries.

Example 3.1.1 (Examples of Conjunctive Query)**.**

$$Q_1(x,z) \text{:- } R(x,y), S(y,z)$$

$$Q_2(x,z) \text{:- } R(x,y), S(y,z), R(x,e)$$

$$Q_3() \text{:- } R(x,y), S(y,z), T(y)$$

$$Q_4(x,y) \text{:- } R(x,y), y = c, \textit{where c is a constant}$$

Q_3 is a *boolean* conjunctive query since its head is empty. In other words, its head is in the form of $Q()$. Equivalently, a conjunctive query is a query expressible by a SQL expression of the form: SELECT-FROM-WHERE.

3.1.1 Semantics

Given a database instance D, a valuation v from Q to D is a total function (defined for all possible input values) that maps the variables of Q to constants in D and is the identity on the constants in Q. Moreover, the image of every relational atom R of Q under v is an R-tuple in D, and for every equality atom $u = v$ of Q, u, v have the same image under v.

Example 3.1.2 (Valuation From the Query to a Database Instance)**.** *For query Q_1 (shown in Example 3.1.1), the function v, where $v(x) = x'$, $v(y) = y'$, $v(z) = z'$, and $v(\langle x,y \rangle) = \langle v(x), v(y) \rangle = \langle x', y' \rangle$, is a valuation.*

The answer of a conjunctive query Q on an instance D, denoted $Q(D)$, is the set of all tuples t for which there is a valuation v from Q into D, such that the image of the head tuple under v is t.

3.1.2 Query Containment and Equivalence

We now formally define query containment and equivalence [50, 26] for conjunctive queries:

Definition 1 (Query Containment)**.** *We say that query Q_1 is contained in query Q_2, denoted $Q_1 \subseteq Q_2$, if for every database instance D, we have $Q_1(D) \subseteq Q_2(D)$ (the set $Q_1(D)$ is included in the set $Q_2(D)$).*

Definition 2 (Query Equivalence)**.** *We say that query Q_1 is equivalent to query Q_2, denoted $Q_1 \equiv Q_2$, if for every database instance D, we have $Q_1(D) \equiv Q_2(D)$*

Two queries Q_1 and Q_2 are equivalent if and only if they are contained in each other ($Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$).

To check whether $Q_1 \subseteq Q_2$, we need to introduce the following core concepts:

Definition 3 (Canonical Instance)**.** *Given a conjunctive query Q, the canonical database D^Q is the database instance where each atom in Q becomes a fact in the instance.*

Example 3.1.3 (Canonical Instance)**.** *The canonical database for the query Q_2 in Example 3.1.1 is the instance $D^{Q_2} = \{R(x,y), S(y,z), R(x,e)\}$.*

Definition 4 (Homomorphism)**.** *A homomorphism h (a.k.a. containment mapping) from Q_2 to Q_1 is a function h: **var**(Q_2) $\rightarrow$ **var**(Q_1) $\cup$ **const**(Q_1) such that:*
(i) for every atom $R(x_1, x_2, \dots)$ in Q_2, there is an atom $R(h(x_1), h(x_2), \dots)$ in Q_1
*(ii) h(**head**(Q_2)) = **head**(Q_1)*

*where **var** denotes the set of variables in a query Q and **head** denotes the denotes the head variables of Q.*

Example 3.1.4 (Homomorphism Example)**.** *Consider the following two queries Q_1 and Q_1':*

$$Q_1(x,y) \text{:- } R(x,y), S(y,y), R(x,e)$$

$$Q_1'(u,d) \text{:- } R(u,d), S(d,f), R(u,w)$$

There is a homomorphism $h_1 = \{u \to x, d \to y, f \to y, w \to e\}$ *from* Q_1' *to* Q_1.

We now state the core theorem for query containment.

Theorem 3.1.1 (Query Containment [50])**.** *Given two conjunctive queries* Q_1*,* Q_2*, the following statements are equivalent:*

(i) $Q_1 \subseteq Q_2$

(ii) There is a homomorphism h from Q_2 *to* Q_1.

(iii) $\boldsymbol{head}(Q_1) \in Q_2(D^{Q_1})$

3.2 Integrity Constraints (Dependencies)

Different forms of *integrity constraints* (a.k.a. database dependencies in the relational model) have been introduced and studied in the literature [53, 69, 72]. They are used to express properties that must be satisfied by all instances of a database schema. These properties arise in a specific application domain. We refer a reader to several survey papers [72, 90, 135] for more details on constraints and their history.

3.2.1 Expressing Constraints in First-Order Logic

Constraints can be expressed using a fragment of the language of first-order logic, known as the class of *embedded dependencies* [26].

$$\forall x_1, \ldots x_n \phi(x_1, \ldots x_n) \to \exists z_1, \ldots, z_k \Psi(y_1, \ldots, y_m)$$

where $\{z_1,\ldots,z_k\} = \{y_1,\ldots,y_m\}\backslash\{x_1,\ldots,x_n\}$. The constraint's *premise* ϕ is a conjunction (possibly empty) of relational atoms over variables $x_1,\ldots,x_n$ or constants. The constraint's *conclusion* ψ is a non-empty conjunction of relational atoms and/or equality atoms – of the form $w = w'$ – over variables $y_1,\ldots,y_m$ or constants. The case when all atoms in ψ are relational atoms is called Tuple Generating Dependencies (**TGD**s), while the case when all atoms in ψ are equalities defines Equality Generating Dependencies (**EGD**s).

Example 3.2.1 (Example of Dependencies From [58])**.** *Consider a relation* ***Review(paper, reviewer, track)*** *listing reviewers of papers submitted to a conference's tracks, and a relation* ***PC (member, affiliation)*** *listing the affiliation of every program committee member. The fact that a paper can only be submitted to a single track is captured by the following EGD:*

$$\forall p \forall r \forall t \forall r' \forall t' \, Review(p,r,t) \wedge Review(p,r',t') \rightarrow t = t'$$

We can also express that papers be reviewed only by PC members by the following TGD:

$$\forall p \forall r \forall t \; Review(p,r,t) \rightarrow \exists a \; PC(r,a)$$

3.2.2 Reasoning With Integrity Constraints: The Chase

The chase procedure is a fundamental algorithm for tackling the implication problem of data dependencies [108, 40, 26] and optimizing conjunctive queries under data dependencies [89]. The algorithm has played a central role in semantic query optimizations [122, 63, 124, 101, 71] for a variety of database applications such as query answering using views [122, 63], data integration and exchange [101, 71] and probabilistic databases [117].

The core concept of the chase is that it fixes constraint violations for a given *fixed* database instance D and a set of data dependencies Σ.

Chase Application. Given a conjunction of atoms A and an embedded dependency $\sigma \in \Sigma$ of the from: $\forall \vec{x}\vec{y} \; \phi(\vec{x},\vec{y}) \; \rightarrow \exists \vec{z} \; \psi(\vec{x},\vec{z})$, then σ is applicable to A with a premise homomorphism h

iff h is a homomorphism from ϕ to A (i.e., the premise holds in A), such that h cannot be extended to cover ϕ (i.e., the conclusion is not already satisfied). We then apply the dependency σ by adding its conclusion to A.

A chase step adds $\psi(h(\vec{x}), f(\vec{z}))$ to A, where h is the premise homomorphism, and f creates "fresh" variables, known as fresh null value or skolems for all of existential variables $\vec{z}$.

Example 3.2.2 (Example of the Chase Application From [111]). *Consider two atoms $S(X)$ and $E(X,Y)$, where E stores directed edges from node x to node y, and S contains nodes with some distinguished properties, which are enforced by constraints. Now, suppose we have only the following constraint Σ and database instance D:*

$$\Sigma : \forall x\, S(x) \rightarrow \exists y\, E(x,y)$$

$$D = \{S(x_1), S(x_2), E(x_1,x_2)\}$$

The constraint Σ states that each node in S has at least one outgoing edge. The example shows that the instance D does not satisfy the constraint Σ since it does not contain an outgoing edge from node x_2. The chase would fix the constraint violation in D by creating the tuple $t = E(x_2, y')$, where y' is a "fresh" variable. After applying the chase, the resulting database instance $D' = D \cup \{t\}$ satisfies the constraint Σ. At this point, the chase terminates and returns the database instance D'.

The standard chase is a series of chase steps. These steps may be finite or infinite depending on the constraints. One well-known termination condition is *weak acyclicity* [70].

Weak Acyclicity. The key idea of weak acyclicity [70] is to assert that no "fresh" values are created over and over again during the execution of the chase.

Definition 5 (Weakly Acyclic Set of TGDs [70]). *Let Σ be a set of TGDs over a fixed schema. Construct a directed graph, called the dependency graph, as follows: (i) there is a node for every*

pair (R,a), where R is a relation symbol of the schema and a is an attribute of R; we call such pair (R,a) a position;(ii) for every TGD: $\forall\vec{x}\ \phi(\vec{x}) \rightarrow \exists\vec{y}\ \psi(\vec{x},\vec{y}) \in \Sigma$ and for every and for every x in $\vec{x}$ that occurs in ψ add the following edges:

- *For every occurrence of x in ϕ in position (R,a_i);*

    1. *for every occurrence of x in ψ in position (S,b_j), add an edge $(R,a_i) \rightarrow (S,b_j$ (if it does not already exist)*
    2. *for every existentially quantified variable $y \in \vec{y}$ and for every occurrence of y in ψ in position (E,c_k), add a special edge $(R,a_i) \xrightarrow{*} (E,c_k)$*

A set Σ of embedded dependencies is called weakly acyclic iff the dependency graph of Σ has no cycles going through a special edge.

Example 3.2.3 (Example of Weakly and Not Weakly Acyclic Constraints Adopted From [70])**.** *Consider a schema S that has a relation $DEPT(D\_ID, E\_ID, MGR\_N)$, listing departments with their manager IDs and names, and a separate relation for $EMP(E\_ID, D\_ID)$. The following set of constraints using $DEPT$ and EMP relations is not weakly acyclic since finite chase may not exist. The first constraint states that each manager of a department is an employee of some department (not necessarily the one they manage).*

$$\forall D \forall E \forall N\ DEPT(D,E,N) \rightarrow \exists d\ EMP(E,d)$$
$$\forall E \forall D\ EMP(E,D) \rightarrow \exists e \exists n\ DEPT(D,e,n)$$

However, if we assume that we know that each manager of a department is employed by the same department, then we have the following weakly acyclic constraints where it is guaranteed that every chase setp is finite.

$$\forall D \forall E \forall N\ DEPT(D,E,N) \rightarrow EMP(E,D)$$
$$\forall E \forall D\ EMP(E,D) \rightarrow \exists e \exists n\ DEPT(D,e,n)$$

We refer a reader to [111, 70] for more details on the chase termination under other permissive and sufficient conditions.

3.3 Query Reformulation Under Integrity Constraints

One of the main problems in query processing is reformulating a query Q expressed against a source schema S to an equivalent query R against a target schema T, by exploiting the relationship between S and T. Examples of query reformulation problems include semantic query optimization (e.g., eliminating redundant joins and reformulating queries under integrity constraints), rewriting queries using views and physical access path selection in query optimization.

In [62], the authors introduce a *uniform* solution for these problems in the form of the Chase & Backchase (C&B) algorithm for query reformulation under constraints. They show how the above query reformulation problems can be expressed as particular query reformulation instances where the relationship between the schemas S and T is expressed by constraints.

In the following sections, we first give an overview of the C&B algorithm (Section 3.3.1), and then we briefly introduce the *Provenance-Directed C&B* [84] (Section 3.3.2), which is an improved version of the C&B algorithm.

3.3.1 The Chase&Backchase (C&B) Algorithm

The Chase&Backchase (C&B) is an algorithm for rewriting queries under integrity constraints [62]. We illustrate the algorithm on a very restricted case of a query reformulation problem, which is *finding total rewriting (i.e., base tables are disallowed) of queries using materialized views*.

Reformulation Under Constraints. We write $D \models \Sigma$ if a database instance D satisfies all the constraints in a set Σ. Query Q_1 is contained in query Q_2 under the set Σ of constraints (denoted $Q_1 \sqsubseteq_\Sigma Q_2$) if and only if $Q_1(D) \subseteq Q_2(D)$ for every database $D \models \Sigma$, where $Q(D)$ denotes

the result of Q on D. Q_1 is equivalent to Q_2 under C (denoted $Q_1 \equiv_\Sigma Q_2$) if and only if $Q_1 \sqsubseteq_\Sigma Q_2$ and $Q_2 \sqsubseteq_\Sigma Q_1$.

Let S and T be relational schemas related by set Σ of constraints, and Q a query formulated against S. A reformulation of Q under Σ is a query R formulated against T, such that $Q \equiv_\Sigma R$

The C&B algorithm is based on expressing queries as conjunctive queries and the view definitions as a set $\mathcal{C}_\mathcal{V}$ of *embedded dependencies*. Then, chasing with $\mathcal{C}_\mathcal{V}$ as well as with other integrity constraints I. We denote the result of chasing a query Q with $\Sigma = \mathcal{C}_\mathcal{V} \cup I$ as Q^Σ.

Phases of The C&B algorithm: The algorithm proceeds in two phases:

Chase: The input query Q is chased with Σ, to obtain a chase result Q^Σ. Then, the subquery U of Q^Σ is obtained by restricting Q^Σ to the schema of the views (a.k.a. target schema). U is called the *universal plan*.

Backchase: In this phase, the subqueries of the universal plan U are checked for equivalence (under Σ) to Q. The output of this phase is all *minimal*-equivalent subqueries (i.e., contain no subqueries that are already equivalent to Q). This equivalence check is performed by chasing "back" each subquery sq in U, and checking whether Q has a containment mapping into sq^Σ.

We illustrate the algorithm on the following running example (the example adopted from [84, 59]). For simplicity, we discuss the example in the absence of integrity constraints I and only use $\mathcal{C}_\mathcal{V}$ constraints.

Example 3.3.1 (Example of The C&B Algorithm from [84, 59])**.** *Consider the query*

$$Q(x) : -R(x,w,y), S(y,z), T(z,e)$$

and assume that we have the following views defined:

$$V_R(x,y) : -R(x,w,y)$$

$$V_S(y,z) : -S(y,z)$$

$$V_{RS}(x,z) : -R(x,w,y), S(y,z)$$

$$V_T(z,e) : -T(z,e)$$

The reader realizes easily that

$$R_1(x) : -V_{RS}(x,z), V_T(z,e)$$

$$R_2(x) : -V_R(x,y), V_S(y,z), V_T(z,e)$$

are equivalent views-based rewritings of Q. In addition, R_1 and R_2 are minimal rewritings since non of them contain subqueries that are already equivalent to Q.

In order to find these two rewritings, the algorithm requires capturing the view definitions by a set $C_{\mathcal{V}}$ of embedded dependencies. $C_{\mathcal{V}}$ is obtained by stating the inclusion (in both directions) between the result of the query defining each view and the view's extent [34]. For example, $C_{\mathcal{V}}$ is the following set of embedded dependencies:

$$V_R^{IO} : \forall x \forall w \forall y \; R(x,w,y) \rightarrow V_R(x,y)$$

$$V_R^{OI} : \forall x \forall y \; V_R(x,y) \rightarrow \exists w \; R(x,w,y)$$

$$V_S^{IO} : \forall y \forall z \; S(y,z) \rightarrow V_S(y,z)$$

$$V_S^{OI} : \forall y \forall z \; V_S(y,z) \rightarrow S(y,z))$$

$$V_{RS}^{IO} : \forall x \forall w \forall y \forall z \; R(x,w,y), S(y,z) \rightarrow V_{RS}(x,z)$$

$$V_{RS}^{OI} : \forall x \forall z \; V_{RS}(x,z) \rightarrow \exists w \exists y \; R(x,w,y), S(y,z)$$

$$V_T^{IO} : \forall z \forall e \; T(z,e) \rightarrow V_T(z,e)$$

$$V_T^{OI} : \forall z \forall e \; V_T(z,e) \rightarrow T(z,e)$$

The Chase Phase*: When chasing Q with $C_{\mathcal{V}}$, the only applicable chase steps involve V_R^{IO}, V_S^{IO}, V_T^{IO} and V_{RS}^{IO}, yielding the following chase result:*

$$Q^{\mathcal{C}_\mathcal{V}}(x) : -R(x,w,y), S(y,z), T(z,e), V_R(x,y), V_S(y,z), V_T(z,e), V_{RS}(x,z)$$

Restricting $Q^{\mathcal{C}_\mathcal{V}}$ to the schema of the views results into the following universal plan U:

$$U(x) : -V_R(x,y), V_S(y,z), V_T(z,e), V_{RS}(x,z)$$

***The Backchase Phase**: Now, the input to the backchase is U, where the subqueries of U are inspected for equivalence with Q. The rewritings R_1 and R_2 above are among these subqueries. To determine that a subquery sq in U is equivalent to Q, we chase sq back with $\mathcal{C}_\mathcal{V}$, and we search for a containment mapping from Q into $sq^{\mathcal{C}_\mathcal{V}}$. In this example, we illustrate only for the subquery of U corresponding to R_1. The only applicable chase steps involve V_{RS}^{OI} and V_T^{OI}, which yields:*

$$R_1^{\mathcal{C}_\mathcal{V}}(x) : -V_{RS}(x,z), V_T(z,e), R(x,w,y), S(y,z), T(z,e)$$

Since there is a containment mapping $h(Q)$ from Q into $R_1^{\mathcal{C}_\mathcal{V}}$, R_1 is an equivalent and minimal rewriting of Q.

One major problem with the backchase is that it inspects exponentially many subqueries of U, even when there were only very few minimal rewritings. In the next section, we briefly introduce the PACB algorithm [84], which employs much more goal-directed search techniques and inspects up to exponentially fewer reformulation candidates than the C&B algorithm. PACB is provably sound and complete for conjunctive queries and large classes of constraints and has been shown to outperform by orders of magnitude the commercial optimizer of a major relational vendor even when limited to the key and foreign key constraints that commercial optimizers understand.

3.3.2 The Provenance-Directed C&B: PACB

The C&B algorithm enumerates the subqueries of the universal plan U in a bottom-up fashion and chases each of them in isolation to determine equivalence to the query Q. This

approach leads to redundant chasing of the atoms, and it also leads to useless chasing of subqueries that are not equivalent to Q.

The goal of the PACB algorithm [84] is to replace the many isolated subqueries chases with a single chase of the universal plan U. To do so, PACB keeps track of the provenance of each atom, where the provenance of an atom a gives the set of subqueries of U, whose chasing led to the creation of a. We illustrate the algorithm by revisiting Example 3.3.1 from Section 3.3.1.

Example 3.3.2 (The Example of PACB Approch From [84])**.** *Revisiting Example 3.3.1, applicable chase steps of U with $\mathcal{C}_\mathcal{V}$ involve V_R^{OI}, V_S^{OI}, V_T^{OI} and V_{RS}^{OI}, yielding*

$$U^{\mathcal{C}_\mathcal{V}}(x) :- V_R(x,y), V_S(y,z), V_T(z,u), V_{RS}(x,z),$$
$$R(x,w_1,y), S(y,z), T(z,e), R(x,w_2,y_1), S(y_1,z)$$

Notice that they query Q has two containment mappings into $U^{\mathcal{C}_\mathcal{V}}$:

$$h_1 = \{x \to x, y \to y_1, w \to w_2, z \to z, e \to e\}$$
$$h_2 = \{x \to x, y \to y, w \to w_1, z \to z, e \to e\}$$

Now, we show $U^{\mathcal{C}_\mathcal{V}}$, this time annotating its atoms with a unique ID called a provenance term. The provenance annotations appear as superscripts:

$$U^{\mathcal{C}_\mathcal{V}}(x) :- V_R(x,y)^{V_R}, V_S(y,z)^{V_S}, V_T(z,u)^{V_T}, V_{RS}(x,z)^{V_{RS}},$$
$$R(x,w_1,y)^{V_R}, S(y,z)^{V_S}, T(z,e)^{V_T}, R(x,w_2,y_1)^{V_{RS}}, S(y_1,z)^{V_{RS}}$$

The view atoms in $U^{\mathcal{C}_\mathcal{V}}$ are annotated by themselves since they are not introduced by chasing. The R, S and T atoms are introduced by chasing V_R^{OI}, V_S^{OI}, V_T^{OI} and V_{RS}^{OI}. For instance, the first R atom: $R(x,w_1,y)$ is introduced by chasing $V_R(x,y)$ with the dependency V_R^{OI}, while the second R atom: $R(x,w_2,y_1)$ is introduced by chasing $V_{RS}(x,z)$ with V_{RS}^{OI}. The provenance formula [1] $\pi(h_1)$ of the first image h_1 of Q is $V_{RS} \wedge V_T$, which corresponds to the rewriting R_1 in

[1] Provenance formulas are constructed from provenance terms using logical conjunction and disjunction with their expected properties such as commutativity, distributivity, idempotency and absorption.

Example 3.3.1, while the provenance formula $\pi(h_2)$ of the second image h_2 of Q is $V_R \wedge V_S \wedge V_T$, which corresponds to the rewriting R_2 in the same running example.

Notice that we immediately identify the two rewritings of Q by reading them off directly from the provenance formulas: $\pi(h_1)$ and $\pi(h_2)$ of Q images in $U^{\mathcal{C}_\mathcal{V}}$. With this approach, the PACB algorithm avoids fruitless individual chases of the subqueries in U.

Chapter 4

ESTOCADA: Bringing Semantic Optimization to Hybrid Stores

4.1 Introduction

This chapter presents ESTOCADA, an extensible lightweight framework for bringing semantic query optimization on top of hybrid stores. The novelty of ESTOCADA is to find a semantically equivalent views-based rewriting of a given hybrid query via a reduction from multi-models views-based query rewriting to relational rewriting under integrity constraints. We formalize our guarantees in terms of soundness and completeness and show that ESTOCADA improves the performance in natural scenarios for both cross-models and single model user queries.

4.1.1 Motivating example

Consider the Medical Information Mart for Intensive Care III (MIMIC-III) [88] dataset, comprising health data for more than 40.000 Intensive Care Unit (ICU) patients from 2001 to 2012. The total size of **46.6 GB**, and it consists of : (*i*) all charted data for all patients and

their hospital admission information, ICU stays, laboratory measurements, caregivers' notes, and prescriptions; (*ii*) the role of caregivers (e.g., MD stands for "medical doctor"), (*iii*) lab measurements (e.g., ABG stands for "arterial blood gas") and (*iv*) diagnosis-related groups (DRG) codes descriptions.

Our motivation query Q_1 is: "for the patients transferred into the ICU due to "coronary artery" issues, with abnormal blood test results, find the date/time of admission, their previous location (e.g., emergency room, clinic referral), and the drugs of type "additive" prescribed to them". Evaluating this query through the AsterixDB JSON store (v9.4) [4] took more than 25 minutes; this is because the system does not support full-text search by an index if the text occurs within a JSON array. In SparkSQL (v2.3.2), the query took more than an hour due to its lack of indexes for selective data access. In the MongoDB JSON store (v4.0.2) [11], it took more than 17 minutes due to its limited join capability. Finally, in PostgreSQL with JSON support (v9.6), denoted Postgres in the sequel, Q_1 took $\approx$ 12.6 minutes.

Now, consider we had at our disposal three *materialized views* which pre-compute partial results for Q_1. SOLR is a well-known highly efficient full-text server. It is also capable of handling JSON documents. Consider a SOLR server that stores a view V_1 storing the IDs of patients, their hospital admission IDs, and the caregivers' reports, including notes on the patients' stay (e.g., detailed diagnosis). Full-text search on V_1 for "coronary artery" allows retrieving the IDs of the respective patients efficiently. Further, consider that a Postgres server stores a view V_2 with the patient's meta-data information and their hospital admission information, such as admission time and patients' location prior to admission. Finally, assume available a view V_3, which stores all drugs that are prescribed for each patient that has "abnormal blood" test results as a JSON document stored in Postgres.

Now, we are able to evaluate Q_1 by a full-text search on V_1 followed by a BindJoin [123] with the result of filtering V_3, and projecting prescribed drugs as well as patients' admission time and prior location from V_2. Using Tatooine [45], a Java-based hybrid execution engine

(implementing select, project, join, etc.), to access the views and join them, this takes about ≈ 5.7 mins, or a **speedup of 5**$\times$ w.r.t. plain JSON query evaluation in SparkSQL and AsterixDB. This is also a **speedup of 2**$\times$ and **speedup of 3**$\times$ w.r.t. plain JSON query evaluation in Postgres and MongoDB, respectively.

Lessons learned. We can draw the following conclusions from the above example. **(1.)** Unsurprisingly, materialized views drastically improve query performance since they pre-compute partial query results. **(2.)** More interestingly: *materialized views can strongly improve performance even when stored across several data stores*, although such a hybrid scenario incurs a certain performance penalty due to the marshaling of data from one data model/store to another. In fact, *exploiting the different strengths of each system* (e.g., SOLR's text search, Postgres' efficient join algorithms, etc.) is the second reason (together with materialized view usage) for our performance gains. **(3.)** Different system combinations work best for different queries; thus *it must be easy to add/remove a view in one system*, without disrupting other queries that may be currently well-served. As known from classical data integration research [81], such flexibility is attained through the "local-as-view" (LAV) approach (see Chapter 2), where the content of each data source is described as a view. Thus, adding or removing a data source from the system is easily implemented by adding or removing the respective view definition. **(4.)** *Application data sets may come in a variety of formats*, e.g., Apache log data is often represented in CSV. However, while the storage model may change as data migrates, *applications should not be disrupted*. A simple way of achieving this is to guarantee them *access to the data in its native format*, regardless of where it is stored.

Observe that the combination of **2.**, **3.** and **4.** above goes well beyond the-state-of-the-art. Most LAV systems assume both the application data and the views are organized according to the same data model (mostly relational or XML). Thus, their view-based query rewriting algorithms are designed specifically within the bounds of that model, e.g., relational [102], or XML [51, 120, 118]. Different from these, some LAV systems [109, 64] allow different data

Table 4.1: Models and languages supported by ESTOCADA.

Data model	Query language/API	Systems
Relational	SQL	Major vendors
JSON	SparkSQL	Spark [35]
JSON	AQL/SQL++	AsterixDB [4]
JSON	SQLw/JSON	Postgres
Key-value	Redis API	Redis
Full-text and JSON	Solr API	Solr
XML	XQuery, XPath	Saxon [19]
Property Graph	Cypher	Neo4j [14]

models for the stored views but consider only the case when the application data model is XML. Consequently, the query answering approach adopted in these systems is tailored toward the XML data model and query language. In contrast, we aim at a unified approach, supporting *any data model both at the application and at the view level*. The core technical question to be answered in order to attain such performance benefits without disrupting applications is *view-based query rewriting across an arbitrary set of data models*. We introduce a novel approach for cross-models view-based rewriting. Our approach is currently capable of handling the systems listed in Table 4.1, together with their data models and query languages.

Chapter Outline. The chapter is organized as follows. Section 4.2 formalizes the rewriting problem we solve. Section 4.3 outlines our approach; at its core is a view-based query rewriting algorithm based on an internal relational model, invisible to users and applications, but which crucially supports rewriting under integrity constraints. Section 4.4 describes the language we use for (potentially cross-models) views and queries. Section 4.5 shows how we reduce the multi-data model rewriting problem to one within this internal pivot model, then transform rewritings obtained there into a hybrid integration query. Section 4.9 describes a set of extensions we contributed to the rewriting engine at the core of our approach to make it scale in a hybrid setting. Section 4.12 formalizes the guarantees of our approach. We present our experimental evaluation in Section 4.13, and conclude in Section 4.14.

4.2 Problem statement

We assume a set of data model-query language pairs $\mathcal{P} = \{(M_1, L_1), (M_2, L_2), \ldots, (M_n, L_n)\}$ such that for each $i \geq 1$, an L_i query evaluated against an M_i instance returns an answer which is also an M_i instance. The same model may be associated to several query languages; for instance, AsterixDB, MongoDB and Postgres have different query languages for JSON. As we shall see, we consider *expressive languages for realistic settings, supporting conjunctive querying, nesting, aggregation, and object creation*. Without loss of generality, we consider that a language is paired with one data model only.

We consider a hybrid setting comprising a set of stores $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ such that each store $S \in \mathcal{S}$ is characterized by a pair $(M_S, L_S) \in \mathcal{P}$, indicating that S can store instances of the model M_S and evaluate queries expressed in L_S.

We consider a set of datasets $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$, such that each data set $D \in \mathcal{D}$ is an instance of a data model M_D. A dataset D is stored as a set of (potentially overlapping) **materialized views** $\mathcal{V}$=$\{V_D^1, V_D^2 \ldots\}$, such that for every $j \geq 1$, V_D^j is stored within the storage system $S_D^j \in \mathcal{S}$. Thus, V_D^j is an instance of a data model supported by S_D^j [1].

We consider a *hybrid integration language* $\mathcal{IL}$, capable of expressing computations to be made within each store and across all of them, as follows:

- For every store S and query $q_S \in L_S$, $\mathcal{IL}$ allows that q_S should be evaluated over S directly;
- Further, $\mathcal{IL}$ allows expressing powerful processing over the results of one or several such source queries (e.g., join results).

Such integration language has been proposed in several hybrid systems [95, 68]; we use it to express queries and computations over the views. An $\mathcal{IL}$ expression bears obvious similarities with an execution plan in a wrapper-mediator system (see Chapter 2); its evaluation is split

[1] For uniformity, we describe any collection of stored data as a view, e.g., a table stored in an RDBMS is an (identity) view over itself.

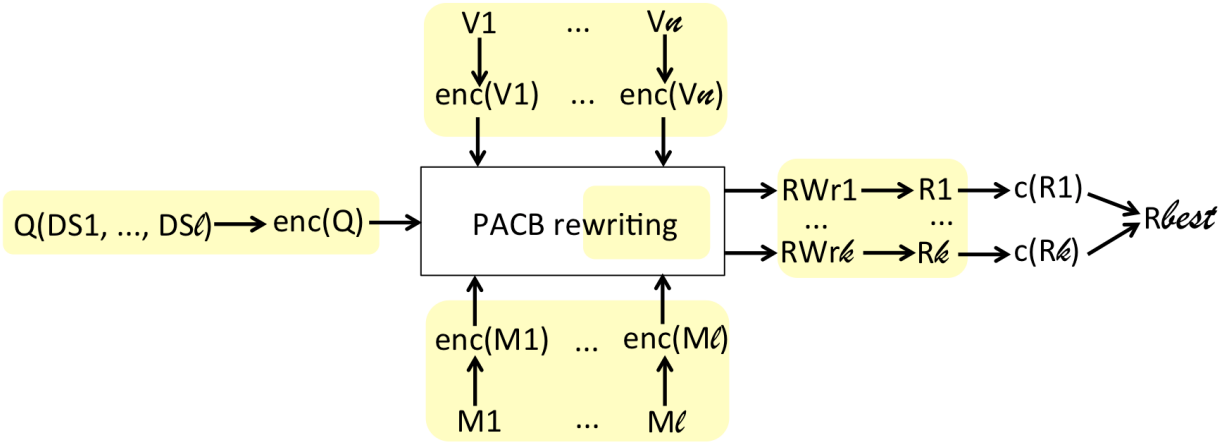


Figure 4.1: ESTOCADA reduction outline.

between computations pushed to the stores and subsequent operations applied by the mediator on top of their results. The main distinction is that $\mathcal{IL}$ remains declarative, abstracting the details of each in-store computation, which is simply specified by a query in the store's language.

We assume *available a cost model* which, given an $\mathcal{IL}$ expression e, returns an estimation of the cost of evaluating e (including the cost of its source sub-queries). The cost may reflect e.g., disk I/O, memory needs, CPU time, transfer time between distributed sites. *Devising a cost-based model for a hybrid setting is beyond the scope of this work.*

We term a *rewriting* of a query q as an integration query expressed in $\mathcal{IL}$, which is equivalent to q. We consider the following rewriting problem:

Definition 6 (Cross-Model Rewriting Problem)**.** *Given a query $q \in \mathcal{IL}$ over several datasets D_i, $1 \leq i \leq n$, and a set of views $\mathcal{V}$ materialized over these datasets, find the minimum-cost equivalent rewriting RW of q using the views.*

Most modern query languages include primitives such as arithmetic operations, aggregation, and calls to arbitrary UDFs; these suffice to preclude the decidability of checking whether two queries are equivalent. Therefore, we aim at finding rewritings *under black-box (uninterpreted) function semantics (UFS)*: we seek rewritings that make the same function calls, with the same arguments as the original query.

4.3 ESTOCADA Overview

We outline here our approach for solving the problem introduced in Section 4.2:

Our integration language: QBT^{XM}. We devised QBT^{XM}, a concrete integration language which supports queries over several data stores, each with its own data model and query language. QBT^{XM} follows a *block*-based design, with blocks organized into a tree in the spirit of the classical Query Block Trees (QBT) introduced in System R [125], slightly adapted to accommodate subsequent SQL developments, in particular, the ability to nest sub-queries within the select clause [85]. The main difference in our setting is that each block may be expressed in a different query language and carried over data of a different data model (e.g., SQL for relational data, key-based search API for key-value data, different JSON query languages). We call the resulting language QBT^{XM}, for *cross-models QBT* (detailed in Section 4.4.2). We note that exiting hybrid query languages [95, 68] resembles QBT^{XM}, and they can be translated directly to QBT^{XM}, as demonstrated in [32]. Excerpts of QBT^{XM} grammar are delegated to Appendix A.2.

QBT^{XM} Views. Each materialized view V is defined by an QBT^{XM} query (or other hybrid integration languages); it may draw data from one or several data sources of the same or different data models. Each view returns (holds) data following one data model, and is stored in a data store supporting that model.

Encoding into a Single Pivot Model. We reduce the cross-models rewriting problem to a single-model setting, namely *relational constraint-based query reformulation*, as follows (see also Figure 5.1; yellow background identifies the areas where we bring contributions in this work.). First, we *encode relationally*: the structure of original datasets, the view specifications, and the application query.

Note that the relations used in the encoding are *virtual*, i.e., no data is migrated into them; they are also *hidden*, i.e., invisible to both the application designers and to users. They only serve to support query rewriting via relational techniques.

The virtual relations are accompanied by *integrity constraints* that reflect the features of the underlying data models (for each model M, a set $enc(M)$ of constraints). For instance, we describe the organization of a JSON document data model using a small set of relations such as *Child(parentID, childID, key, type)* together with the constraint specifying that every child has just one parent. Such modeling had first been introduced in local-as-view XML integration works [109, 64]. The constraints are TGDs and EGDs (see Chapter 3), extended with such well-researched constraints as denial constraints [77] and disjunctive constraints [76]. We detail the pivot model in Section 4.5.

Reduction From Cross-Model to Single-Model Rewriting. Our reduction translates the declaration of each view V to additional constraints $enc(V)$ that reflect the correspondence between V's input data and its output. Constraints have been used to encode single-model views [65] and correspondences between source and target schemas in data exchange [71]. The novelty here is the rich collection of supported models, and the cross-models character of the views.

An incoming query Q over the original datasets $DS_1, \ldots, DS_l$, whose data models respectively are $M_1, \ldots, M_l$, is encoded as a relational query $enc(Q)$ over the dataset's relational encoding. $enc(Q)$ is centered around conjunctive queries, with extensions such as aggregates, UDFs, and nested sub-queries.

The reformulation problem is thus reduced to a purely relational setting: given a relational query $enc(Q)$, a set of relational integrity constraints encoding the views, $enc(V_1) \cup \ldots \cup enc(V_n)$, and the set of relational constraints obtained by encoding the data models $M_1, \ldots, M_l$, find the queries RW_r^i expressed over the relational views, for some integer k and $1 \leq i \leq k$, such that each RW_r^i is equivalent to $enc(Q)$ under these constraints.

The challenge in coming up with the reduction consists in designing a *faithful* encoding, i.e., one in which rewritings found by (*i*) encoding relationally, (*ii*) solving the resulting relational reformulation problem, and (*iii*) decoding each reformulation RW_r^i into a QBT^{XM} query $R =$

$dec(RW_r^i)$ over the views in the polystore, correspond to rewritings found by solving the original problem. That is, R is a rewriting of Q given the views $\{V_1,\ldots,V_n\}$ if $R = dec(RW_r^i)$, where RW_r^i is a relational reformulation of $enc(Q)$ under the constraints obtained from encoding $V_1,\ldots,V_n$, $M_1,\ldots,M_l$. The reduction is detailed in Sections 4.5, 4.6 and 4.7 .

Relational Rewriting Under Constraints. To solve the single-model rewriting problem, we need to reformulate relational queries under constraints. The algorithm of choice is known as Chase & Backchase (C&B, in short, see Chapter 3), introduced in [63] and improved in [84] to yield the *Provenance-Aware C&B algorithm* (PACB, in short, see Chapter 3). PACB was designed to work with relatively few views and constraints. In contrast, in the hybrid setting, each of the many data sources is described by a view, and each view is encoded by many constraints. For instance, the JSON views in our experiments (Section 4.13) are encoded via ≈ 45 TGDs involving 10-way joins in the premise. To cope with this complexity, we incorporated into PACB novel scale-up techniques (discussed in Section 4.9.1) to make it scalable in a hybrid setting. PACB was designed for conjunctive queries under set semantics. Towards supporting a richer class of queries, we extended it to bag semantics (Section 4.10) and QBTs (Section 4.11).

Decoding the Relational Rewritings. On any relational reformulation RW_r^i issued by our modified PACB rewriting, a *decoding step* $dec(RW_r^i)$ is performed to:

(*i*) Group the reformulation atoms by the view they pertain to by following the connections among atoms and knowledge of the encoded data model.

(*ii*) Reformulate each such atom group into a query that can be completely evaluated over a single view.

(*iii*) If several views reside in the same store, identify the largest subquery that can be delegated to that store, along the lines of query evaluation in wrapper-mediator systems [79].

Evaluation of Non-Delegated Operations. A decoded rewriting may be unable to push (delegate) some query operations to the store hosting a view if the store does not support them; for instance, most key-value stores do not support joins. In this case, operations have to be executed

outside of that store. For instance, to evaluate such "last-step" operations, our *lightweight execution (hybrid) engine* [45] supports many logical and physical operators, including bind joins [123] (with sideways information passing) to evaluate nested subqueries, and many other operators.

Choice of the Most Efficient Rewriting. Decoding may lead to several rewritings $R_1, \ldots, R_k$; for each R_i, several evaluation plans may lead to different performance. The problem of choosing the best rewriting and best evaluation plan in this setting recalls query optimization in distributed mediator systems [119]. For each rewriting R_i, we denote by $c(R_i)$ the lowest cost of an evaluation plan that we can find for R_i; we choose the rewriting R_{best} that minimizes this cost. While devising cost models for hybrid settings is beyond the scope of this paper, we architected ESTOCADA to take the cost model as a configuration parameter (using the hybrid engine's own cost estimator, typically accessible via an API).

4.4 The QBT^{XM} Language

We present here the language we use for views and queries. First, we recall the classical Query Block Trees (QBTs), then we extend them to cross-model views and queries.

4.4.1 Query Block Trees (QBT)

Since many of the languages that ESTOCADA supports allow for nested queries and functions (user-defined or built-in such as aggregates), our pivot language *Query Block Trees* (QBTs) also supports these features. These are essentially the classical System R QBTs [125], slightly adapted to accommodate subsequent SQL extensions, such as nesting in the select clause (introduced in SQL-1999 [85]). We first illustrate QBTs on a SQL example.

Example 4.4.1 (QBT Query)**.** *Consider the SQL query in Figure 4.2, which computes for each student who is not enrolled in any course for the Spring'16 quarter, the number of Spring'16*

```
SELECT s.ssn, COUNT (SELECT w.cno
                     FROM   Waitlist w
                     WHERE  w.ssn = s.ssn
                            AND w.qtr = 'Spring 2016'
                            ) AS cnt
FROM    Student s

WHERE   NOT EXISTS (SELECT  c.no
                    FROM    Course c, Enrollment e
                    WHERE   c.no = e.cno
                            AND s.ssn = e.ssn
                            AND e.qtr = 'Spring 2016')
```

Figure 4.2: QBT query example.

courses she is waitlisted for (a count of 0 is expected if the student is not waitlisted). While this query could be written using joins, outer joins, and group by operations, we show a natural alternative featuring nesting (which illustrates how we systematically normalize away such operations):

This query consists of three blocks. The outermost SELECT-FROM-WHERE expression corresponds to the root block; call it B_0. The two nested SELECT-FROM-WHERE expressions are modeled as children blocks of the root block, call them B_{00} and B_{01} for the block nested in the SELECT clause and the block nested in the WHERE clause, respectively

*The variables occurring in a block B can be either defined by B (in which case we call them **bound** in B), or by one of its ancestors (in which case they are called **free** in B). We assume W.l.o.g. that the bound variables of B have fresh names, i.e., they share no names with variables bound in B's ancestors. For example, variable s is bound in B_0, but it occurs free in both B_{00} and B_{01}.*

QBTs model SELECT-FROM-WHERE expressions as blocks, organized into a tree whose parent-child relationship reflects block nesting. Nested blocks always appear as arguments to functions, be they built-in (e.g., COUNT for B_{00} and NOT EXISTS for B_{01} in Example 4.4.1) or user-defined. While not shown in Example 4.4.1, the case of blocks nested within the FROM clause corresponds to the identity function.

4.4.2 Integration Language: QBT^{XM}

To specify cross-model queries and views, we adopted a *block-based design*, similar to QBTs, but in which each block is expressed in its own language, signaled by an annotation on the block. We call the resulting language QBT^{XM}, which stands for *cross-model QBT*. QBT^{XM} queries comprise FOR and RETURN clauses. The FOR clause introduces several variables and specifies their legal bindings. The RETURN clause specifies what data should be constructed for each binding of the variables. Variables can range over different data models, which is expressed by possibly several successive blocks, each pertaining to its own model. In SQL style, this defines a Cartesian product of the variable bindings computed by each block from the FOR clause; this Cartesian product can be filtered through WHERE clause. We impose the restriction that *variables shared by blocks spanning different data models must be of text or numeric type*, so as to avoid dealing with conversions of complex values across models. While there is no conceptual obstacle to handle such conversions automatically, the topic is beyond the scope of this work. We describe QBT^{XM} informally, by examples; the grammar of QBT^{XM} is delegated to Appendix A.2.

Example 4.4.2 (View Definition Expressed in QBT^{XM})**.** *We illustrate the QBT^{XM} definition of views V_1 and V_2 from Section 4.1 in Figure 4.3, using AsterixDB's SQL++ syntax (easier to read than the JSON syntax of PostgreSQL). FOR clauses bind variables, while RETURN clauses specify how to construct new data based on the variable bindings. Blocks are delimited by braces annotated by the language whose syntax they conform to. The annotations AJ, PR, and SJ stand for the SQL++ language of AsterixDB, the languages of Postgres and Solr, respectively. Also, below, PJ and RK stand respectively for PostgreSQL' JSON query language and for a simple declarative key-value query language that we designed for Redis.*

```
View V1:
FOR AJ:{SELECT  M.PATIENTID AS patientID,
                A.ADMISSIONID AS admissionID,
                NE.REPORT AS report
        FROM  MIMIC M, M.ADMISSIONS A, A.NOTEEVENTS NE}

RETURN SJ:{"PATIENTID":patientID, "ADMISSIONID":admissionID, "REPORT":report}

View V2:
FOR AJ:{SELECT  M.PATIENTID AS patientID,
                A.ADMISSIONID AS admissionID,
                A.ADMISSIONLOC AS admissionLoc
                A.ADMISSIONTIME AS admissionTime
         FROM  MIMIC M, M.ADMISSIONS A}

RETURN PR:{patientID, admissionID, admissionLoc, admissionTime}
```

Figure 4.3: View definitions expressed in QBT^{XM}.

4.5 Reduction from Cross-Model to Single-Model Setting

A key requirement in our setting is the ability to reason about queries and views involving multiple data models and query languages. This is challenging for several reasons. First, not every data model/query language setting supported in ESTOCADA comes with a known *view-based query rewriting (VBQR, in short) algorithm*: consider key-value pairs as the data model and declarative languages over them, or JSON data and its query languages in circulation, etc. Second, even if we had these algorithms for each model/language pair, neither would readily extend to a solution of the *cross-model* VBQR setting in which views may be defined over data from various models, materializing their result in yet again distinct models, and in which query rewritings access data from potentially different models than the original query. Third, for the feasibility of development and maintenance as the set of supported model/language pairs evolves, any cross-model VBQR solution needs to be modularly extensible to additional models/languages, in the sense that no modification to the existing code is required and it suffices to just add code dealing with the new model/language pair. Moreover, the developer of this additional code should not need to understand any of the other model/language pairs already supported in ESTOCADA.

To address these challenges, we ***reduce the cross-model VBQR problem to a single-***

Table 4.2: Snippet of $\mathcal{VREJ}$ schema.

$CollectionName_J(ID)$
$Child_J(parentID, childID, key, type)$
$Eq_J(x,y)$
$Value(x,y)$

model VBQR problem. That is, we propose a unique *pivot data model*, on which QBT (Section 4.4.1) serves as a *pivot query language*. Together, they allow us to capture data models, queries, and views so that cross-model rewritings can be found by searching for rewritings in the single-model pivot setting instead.

Our pivot model is relational, and it makes prominent use of expressive integrity constraints. This is sufficiently expressive to capture the key aspects of the data models in circulation today, including: freely nested collections and objects, object identity (e.g., for XML element nodes and JSON objects), limited binding patterns (as required by key-value stores), relationships (in the E/R and ODL sense), functional dependencies, and much more. The integrity constraints we use are TGDs or EGDs (see Chapter 3), extended to denial constraints [77] and disjunctive constraints [76].

4.5.1 JSON Model

We represent JSON documents as relational instances conforming to the schema $\mathcal{VREJ}$ (Virtual Relational Encoding of JSON) in Table 4.2. We emphasize that these relational instances are *virtual*, i.e., the JSON data is not stored as such. Regardless of the physical storage of the JSON data, we only use $\mathcal{VREJ}$ to encode JSON queries and views relationally, in order to reason about them.

$\mathcal{VREJ}$ constraints express the fact that JSON databases are organized into named collections of JSON values, which in turn can be objects (consisting of a set of key-value pairs), arrays of values, and scalar values (numeric or text).

We model objects, arrays, and values in an object-oriented fashion, i.e. they have identities. This is because some JSON stores, e.g., MongoDB and AsterixDB, support query languages that refer to identity and/or distinguish between equality by value versus equality by identity. Our modeling supports both the identity-aware and the identity-agnostic view of JSON via appropriate translation of queries and views into pivot language expressions over the $\mathcal{VREJ}$ schema.

Relation $CollectionName_J$ (see Table 4.2) attaches to each persistent collection name an ID. $Value_J$ attaches a scalar value y to a given variable x. $Child_J$ states that the value identified by *childID* is immediately nested within the value identified by *parentID*. The *type* attribute records the type of the parent (array "a" or object "o") and determines the kind of nesting and the interpretation of the *key* attribute: if the parent is an array, *key* holds the position at which the child occurs; if the parent is an object, then *key* holds the name of the key whose associated value is the child.

Our modeling of the parent-child (immediate nesting) relationship between JSON values provides the type of value *only in conjunction with a navigation step where this value is a parent*, and the step leads to a child value of undetermined type. This modeling reflects the information one can glean statically (at query rewriting time, as opposed to query runtime) from JSON query languages in circulation today.

Recall that JSON data is semi-structured, i.e., it may lack a schema. Queries can, therefore, express navigation leading to values whose type cannot be determined statically if these values are not further navigated into. The type of a value can be inferred only from the type of navigation step into it.

Example 4.5.1 (JSON Navigation)**.** *If query navigation starts from a named JSON collection "coll" and proceeds to the fifth element e therein, we can infer that "coll" is of array type, but we do not know the type of e. Only if the query specifies an ensuing lookup of the value v associated to key "k" in e can we infer e's type (object). However, absent further navigation steps, we cannot tell the type of the child value v.*

Importantly, we assume that fields' values are homogenous in this work. The reason is that different systems have different semantics and behavior on handling queries that access fields with heterogeneous types (e.g., failing, warning, returning `null`, or `missing` (see Section 4.5.9)). We leave capturing such different semantics to future work.

Finally, relation Eq_J is intended to model value-based equality for JSON (id-based equality is captured directly as equality of the id attributes).

We capture the intended meaning of the $\mathcal{VREJ}$ relations via constraints that are inherent in the data model (i.e. they would hold if we actually stored JSON data as a $\mathcal{VREJ}$ instance). We illustrate a few of these below:

The fact that a persistent name uniquely refers to a value is expressed by the EGD (4.1) below. EGD (4.2) states that an object cannot have two distinct key-value pairs sharing the same key, or that an array cannot have two distinct elements at the same position. TGDs (4.3) and (4.4) state that value-based equality is symmetric, respectively transitive, and TGD (4.5) states that value-equal parents must have value-equal children for each parent-child navigation step. EGD (4.6) states that no two distinct scalar values may correspond to a given id.

$$\forall x \forall y\ CollectionName_J(x) \wedge CollectionName_J(y) \rightarrow x = y \tag{4.1}$$

$$\forall p \forall c_1 \forall c_2 \forall k \forall t\ Child_J(p, c_1, k, t) \wedge Child_J(p, c_2, k, t) \rightarrow c_1 = c_2 \tag{4.2}$$

$$\forall x \forall y\ Eq_J(x, y) \rightarrow Eq_J(y, x) \tag{4.3}$$

$$\forall x \forall y \forall z\ Eq_J(x, y) \wedge Eq_J(y, z) \rightarrow Eq_J(x, z) \tag{4.4}$$

$$\forall p \forall p' \forall c \forall k \forall t\ Eq_J(p, p') \wedge Child_J(p, c, k, t) \rightarrow \exists c'\ Eq_J(c, c') \wedge Child_J(p', c', k, t) \tag{4.5}$$

$$\forall x \forall v_1 \forall v_2\ Value(x, v_1) \wedge Value(x, v_2) \rightarrow v_1 = v_2 \tag{4.6}$$

Encoding JSON Views and Queries. We present in this subsection how we encode JSON queries and views using the $\mathcal{VREJ}$ schema. JSON queries (or views) can be nested:

```
V:
SELECT VALUE {"CAaddresses": (SELECT VALUE D
                              FROM P.Addresses AS D
                              WHERE D.state="CA")}
FROM persons AS P

Q:
SELECT D.zip, D.street
FROM persons AS P, P.Addresses AS D
WHERE D.state="CA"
```

Figure 4.4: JSON view V and query Q.

the SELECT clause (a.k.a. RETURN clause in some languages [4]) may (*i*) constructs fresh elements (i.e., JSON objects), (*ii*) refers to variables bound in the FROM clause, (*iii*) contains SELECT-FROM-WHERE[2] expressions (a.k.a. FWR expressions in some languages.)

We call a block b, a fragment of a nested JSON query delimited as follows: it has the "FROM" clause, "WHERE" clause (if present), and "SELECT" clause. If a JSON object is created in the SELECT clause, we refer to it hereafter as a return template for a block b, denoted as $rt(b)$. The local variables of a block b, denoted as $localDefVars(b)$, represent a collection of variables introduced in b. Moreover, free variables of b, denoted as $freeVars(b)$, represent variables mentioned in b but introduced in its ancestors. We call a collection of variables in a block b visible, denoted as $visibleVars(b)$ if other blocks use them.

In the example below, we show how we encode a JSON query relationally using the $\mathcal{VREJ}$ schema, and translate a JSON view into additional relational integrity constraints, capturing the correspondence between the view input and output, with the knowledge of each block information described above.

Example 4.5.2. *(Encoding JSON View and Query). Consider the JSON query Q and view V (using AsterixDB's SQL++ syntax) in Figure 4.4; the query Q asks to find the zip code and street name of "CA" addresses for each person. The nested view V constructs a JSON object that has a list of "CA" addresses for each person. The view V consists of two blocks. The outermost*

[2]The fragment of the supported JSON language in this work includes select-project-join queries with nested blocks, JSON object navigation, array unnest and object construction.

SELECT-FROM expression corresponds to the root block; call it V_0. The nested SELECT-FROM-WHERE block in the SELECT clause is modeled as a child block of V_0; call it V_{00}. Notice that finding the rewriting of Q using V requires reasoning about the nested blocks and the construction of a new JSON object in the SELECT clause of V. The view V cannot be directly matched against Q.

Encoding of View V as a Set of Constraints. *The view V will be partitioned into two blocks: V_0 and V_{00}. For each block b, we capture $rt(b)$, $localDefVars(b)$, $freeVars(b)$, and $visibleVars(b)$. For example, for the block V_0, we have:*

$$rt(V_0) = \{\{\text{“}CAaddresses\text{”} :\}\}$$
$$localDefVars(V_0) = \{P\}$$
$$freeVars(V_0) = \emptyset$$
$$visibleVars(V_0) = \{P\}$$

Notice that $freeVars(V_0)$ is empty since V_0 is the root block. However, $visibleVars(V_0) = \{P\}$ since P is visiable to the block V_{00}. In the same fashion, we capture the following information for the block V_{00}:

$$rt(V_{00}) = \emptyset$$
$$localDefVars(V_{00}) = \{D\}$$
$$freeVars(V_{00}) = \{P\}$$
$$visibleVars(V_{00}) = \emptyset$$

$rt(V_{00})$ is empty since there is no object creation in the SELECT clause in V_{00}.

Now, given the information above for each block b, b will be encoded as a set of constraints. First, we encode the root block V_0, the following constraints are introduced for V_0:

$$\forall P\ persons(P) \rightarrow V_0ExtrJ(P) \tag{4.7}$$

$$\forall P\ V_0ExtrJ(P) \rightarrow \exists\ d_0 \exists d_1\ V_0CreateJ(d_0,d_1) \wedge V_0SkJ0(d_0,P) \wedge V_0SkJ1(d_1,P) \tag{4.8}$$

$$\forall\ d_0 \forall\ d_1 V_0\ CreateJ(d_0,d_1) \rightarrow V(d_0) \wedge Child_J(d_0,d_1,\text{“}CAaddresses\text{”},\text{“}o\text{”}) \tag{4.9}$$

The TGD (4.7) extracts the visible variable P to be accessed later by the constraint of the child block V_{00}. To enable rewriting, we would like the view V head exposes the ID of the constructed JSON object in the SELECT clause ($rt(V_0)$). This can be achieved by introducing the TGDs (4.8) and (4.9). The TGD (4.8) creates two IDs: d_0 and d_1, the former represents the ID of the constructed JSON object, and the latter represents the ID of the array "CAaddresses", which is immediately nested within the parent ID d_0. These IDs are created using the Skolem functions, which are modeled relationally using the relations V_0SkJ0 and V_0SkJ1. The variable P in both relations corresponds to arguments of the Skolem call. Furthermore, we need to express the fact that these relations model function and injective functions. We do this by means of key constraints, basically stating that:

$$\forall d_0^1 \forall d_0^2 \forall P\ V_0SkJ0(d_0^1,P) \wedge V_0SkJ0(d_0^2,P) \rightarrow d_0^1 = d_0^2 \text{ (function)}$$
$$\forall d_0 \forall P_1 \forall P_2\ V_0SkJ0(d_0,P_1) \wedge V_0SkJ0(d_0,P_2) \rightarrow P_1 = P_2 \text{ (injective)}$$

Now, we encode the nested block V_{00} by introducing the following constraint:

$$V_0ExtrJ(P) \wedge V_0SkJ1(d_1,P) \wedge Child_J(P,A,\text{“}addresses\text{”},\text{“}o\text{”}) \wedge Child_J(A,D,\text{“}*\text{”},\text{“}a\text{”}) \wedge$$
$$Child_J(D,S,\text{“}state\text{”},\text{“}o\text{”}) \wedge Value(S,\text{“}CA\text{”}) \rightarrow$$
$$Child_J(d_1,D,\text{“}*\text{”},\text{“}a\text{”})$$

The nested block V_{00} is created within the return template $rt(V_0)$ of the root block V_0. The result of the nested block V_{00} is an array of "CA" addresses for each person P. The ID of the array is d_1, created by the Skolem function $V_0SkJ1(d_1,P)$. The constraint encodes the fact that each element in d_1 is represented by the variable D obtained from unnesting the "addresses" array and applying the condition (state="CA"), as shown in the premise of the constraint. To

encode the unnest operator, we use the $Child_J$ relation, where we specify the key *attribute as $*$. For example, the relation $Child_J(d_1, D, "*", "a")$ indicates that d_1 is an array and D binds to each element in d_1.*

***Encoding of Query** Q. The following is the relational encoding of the query Q using the $\mathcal{VREJ}$ schema:*

$$Q(Z,ST) :- persons(P), Child_J(P,A,"addresses","o"),$$
$$Child_J(A,D,"*","a"), Child_J(D,S,"state","o"),$$
$$Value(S,"CA"), Child_J(D,ST,"street","o"), Child_J(D,Z,"zip","o");$$

Taking as input the query Q, set of the view V constraints, and constraints of $\mathcal{VREJ}$ relations, the constraints-based rewriting algorithm (see Chapter 3) finds the following equivalent relational V-based rewriting of Q:

$$R(Z,ST) :- V(d_0), Child_J(d_0,d_1,"CAaddresses","o"),$$
$$Child_J(d_1,D,"*","a"), Child_J(D,ST,"street","o"), Child_J(D,Z,"zip","o");$$

For brevity, we omit introducing the encoding of the unnest operator hereafter.

4.5.2 Key-Value Model

Our interpretation of the key-value pairs model is compatible with many current systems, in particular *Redis*, supported by ESTOCADA. Calling a *map* a set of key-value pairs, the store accommodates a set of persistently named maps (called *outer* maps). Within each outer map, the value associated to a key may be either a map (called an *inner* map), or a scalar value. In the case of an *inner* map, the value associated to a key is a scalar value.

Given the analogy between key-value and JSON maps, we model the former similarly to the latter, as instances of the relational schema $\mathcal{VREK}$, consisting of the relations: $Map_{KV}(name, mapId)$, $Child_{KV}(parentID, childID, key, type)$ and $Eq_{KV}(x,y)$. Here, Map_{KV}

Table 4.3: Snippet of $\mathcal{VREG}$ schema.

$Connection_G(sID, dID, lID)$
$Label_G(lID, name)$
$Property_G(lID, key, vID)$
$Kind_G(lID, type)$
$Value(x, y)$

models the association between a persistent name and (the id of) an outer map. $Child_{KV}$ reflects the immediate nesting relationship between a map (the parent) and a value associated to one of its keys (the child). The *type* attribute tells us whether the parent map is an outer or an inner map (these are treated asymmetrically in some systems). The Eq_{KV} relation models value-based equality, analogously to Eq_J for JSON.

The intended semantics of these relations is enforced by similar constraints to the JSON model, e.g., in a map, there is only one value for a key

$$\forall p \forall c_1 \forall c_2 \forall k \forall t \; Child_{KV}(p, c_1, k, t), Child_{KV}(p, c_2, k, t) \rightarrow c_1 = c_2$$

persistent map names are unique

$$\forall n \forall x \forall y \; Map_{KV}(n, x), Map_{KV}(n, y) \rightarrow x = y$$

4.5.3 Graph Model

We discuss in this subsection our relational encoding of property graph queries and views using the schema $\mathcal{VREG}$ (Virtual Relational Encoding of Graph) in Table 4.3. The relation $Connection_G$ states that there is a relationship between a source node *sID* and a destination node *dID* with the label ID *lID*. Every label in a graph has a name, and we capture that using the $Label_G$ relation. In addition, a label in a graph can be an "edge" (relationship) or a "node", and the type of a label is modeled using the $Kind_G$ relation, where *lID* is the label ID, and *type* indicates whether *lID* is an edge or a node. Nodes and relationships can have properties (key-value pairs).

```
V:
MATCH (po:Post)<-[:REPLY-OF]-(m:Message)-[:HAS-CREATOR]->
      (p:Person {firstName: "Amelie"})

CREATE (po)<-[:REPLY-OF-V]-(:Message_V
                {creationDate:m.creationDate})
             -[:HAS-CREATOR-V]->(p)
Q:
MATCH (:Person {firstName: "Amelie"})<-[:HAS_CREATOR]-
      (m:Message)-[:REPLY_OF]->(p:Post)
MATCH (p)-[:HAS_CREATOR]->(c)
RETURN
        m.creationDate AS mCreationDate,
        p.postId AS pPostId,
        c.personId AS originalPostAuthorId,
        c.lastName AS originalPostAuthorLastName
```

Figure 4.5: Graph view V and query Q.

Property attached to a label *lID* is encoded using the $Property_G$ relation, where *vID* is the value associated with the key *key*.

We currently support a fragment of Cypher graph-based language. This fragment includes (*i*) pattern-matching (MATCH), (*ii*) WHERE clause, containing simple predicates, and (*ii*) construction of a new node and a relationship between nodes. We note that we currently do not support the encoding of "Variable-length pattern matching" in the form: (n_1)-[*]$\rightarrow(n_2)$. This kind of matching can not be expressed directly in a conjunctive query form. In addition, we do not support filtering on patterns in the WHERE clause. We leave the investigation of addressing these limitations to future work.

Example 4.5.3 (Encoding Graph View and Query)**.** *Consider the Gypher query Q and view V in Figure 4.5; the query Q asks to find the IDs of all posts that "Amelie" replies to, the information of the creators of these posts, and the creation dates of "Amelie"'s replies. The view V creates a new node "Message* $-$ *V" that stores the creation date of "Amelie"'s reply "m" to a post "po". In addition, the view creates the "REPLY-OF-V" relationship that connects the new node "Message* $-$ *V" with the original post "po" and the "HAS-CREATOR-V" relationship, which indicates that "Amelie" is the creator of the "Message* $-$ *V" message.* [3]

[3] We note that the query Q is a simplified version of the query "interactive-short-2.cypher" in idbc benchmark.

Encoding of View V as a Constraint. *The view V is translated into the following TGD using the set of relations in $\mathcal{VREG}$ schema:*

$$Label_G(po, \text{“}Post\text{”}), Label_G(m, \text{“}Message\text{”}), Label_G(id0, \text{“}REPLY - OF\text{”}),$$
$$Connection_G(m, po, id0), Label_G(p, \text{“}Person\text{”}), Property_G(p, \text{“}firstName\text{”}, vID0),$$
$$Label_G(id1, \text{“}HAS - CREATOR\text{”}), Connection_G(m, p, id1), Value(vId0, \text{“}Amelie\text{”}),$$
$$Property_G(m, \text{“}creationDate\text{”}, vID1) \rightarrow$$
$$Label_G(id2, \text{“}Message - V\text{”}), Property_G(id2, \text{“}creationDate\text{”}, vID1),$$
$$Label_G(id3, \text{“}REPLY - OF - V\text{”}), Connection_G(id2, po, id3),$$
$$Label_G(id4, \text{“}HAS - CREATOR - V\text{”}), Connection_G(id2, p, id4)$$

Encoding of Query Q. *The following conjunctive query is the relational encoding of the query Q:*

$$Q(vID1, vID2, vID2, vID4) : - Label_G(id0, \text{“}Person\text{”}), Property_G(id0, \text{“}firstName\text{”}, vId0),$$
$$Value(vId0, \text{“}Amelie\text{”}), Label_G(m, \text{“}Message\text{”}), Label_G(id1, \text{“}HAS - CREATOR\text{”}),$$
$$Connection_G(m, id0, id1), Label_G(p, \text{“}Post\text{”}), Label_G(id2, \text{“}REPLY - OF\text{”}),$$
$$Connection_G(m, p, id2),\ Label_G(id3, \text{“}HAS - CREATOR\text{”}), Connection_G(p, c, id3),$$
$$Property_G(m, \text{“}creationDate\text{”}, vID1), Property_G(p, \text{“}postId\text{”}, vID2),$$
$$Property_G(c, \text{“}personId\text{”}, vID3), Property_G(c, \text{“}lastName\text{”}, vID4);$$

The query Q is equivalent to the V-based relational rewriting R:

$$R < vID1, vID2, vID3, vID4 > : - Property_G(p, \text{“}postId\text{”}, vID2),$$
$$Property_G(c, \text{“}personId\text{”}, vID3), Property_G(c, \text{“}firstName\text{”}, vID0),$$
$$Property_G(c, \text{“}lastName\text{”}, vID4), Property_G(id4, \text{“}creationDate\text{”}, vID1),$$
$$Connection_G(p, c, id3), Connection_G(id4, p, id5),\ Connection_G(id4, id0, id6),$$
$$Label_G(id3, \text{“}HAS - CREATOR\text{”}), Label_G(id4, \text{“}Message - V\text{”}),$$
$$Label_G(id5, \text{“}REPLY - OF - V\text{”}), Label_G(id6, \text{“}HAS - CREATOR - V\text{”});$$

Table 4.4: Snippet of $\mathcal{VREX}$ schema.

$Root(x)$	$Elem(x)$
$Child_X(x,y)$	$DescOrSelf(x,y)$
$Attrib(x,n,v)$	$Tag(x,n,v)$

4.5.4 XML Model

Query reformulation for XML has already been reduced in prior work to a purely relational setting using constraints. We refer the reader to [65, 64] for more details. At high-level, an XML document was represented as a *virtual* instance of the relational schema $\mathcal{VREX}$ consisting of the relations: $\{Root, Elem, Child_X, DescOrSelf, Tag, Attrib, Id, Text\}$ (their schemas are shown in Table 4.4).

The intended meaning of the relations in $\mathcal{VREX}$ reflects the fact that XML data is a tagged tree. The unary predicate *Root* denotes the root element of the XML document, and the unary relation $Elem$ is the set of all elements. $Child_X$ and $DescOrSelf$ are subsets of $Elem \times Elem$ and they say that their second component is a child, respectively a descendant of the first component. $Tag \subseteq Elem \times string$ associates the tag in the second component to the element in the first. $Attrib \subseteq Elem \times string \times string$ gives the element, attribute name and attribute value in its first, second, respectively third component. $Id \subseteq string \times Elem$ associates the element in the second component to a string attribute in the first that uniquely identifies it (if DTD-specified ID-type attributes exist, their values can be used for this). $Text \subseteq Elem \times string$ associates to the element in its first component the string in its second component.

[65] shows that some of the intended meaning of schema $\mathcal{VREX}$ is captured by a set TIX (True In XML) of constraints expressible as TGDs and EGDs, in some cases extended with disjunction. We list below the most interesting ones:

$$\text{(base)} \quad \forall x \forall y \, Child_X(x,y) \rightarrow DescOrSelf(x,y)$$

$$\text{(trans)} \quad \forall x \forall y \forall z \; DescOrSelf(x,y) \wedge DescOrSelf(y,z) \rightarrow DescOrSelf(x,z)$$

$$\text{(refl)} \quad \forall x\, Elem(x) \rightarrow DescOrSelf(x,x)$$

$$\text{(someTag)} \quad \forall x\, Elem(x) \rightarrow \exists t\, Tag(x,t)$$

$$\text{(oneTag)} \quad \forall x \forall t_1 \forall t_2\, Tag(x,t_1) \wedge Tag(x,t_2) \rightarrow t_1 = t_2$$

$$\text{(keyID)} \quad \forall x \forall t_1 \forall t_2\, Id(s,e_1) \wedge Id(s,e_2) \rightarrow t_1 = t_2$$

$$\text{(oneAttrib)} \quad \forall x \forall n \forall v_1 \forall v_2\, Attrib(x,n,v_1) \wedge Attrib(x,n,v_2) \rightarrow v_1 = v_2$$

$$\text{(oneLoop)} \quad DescOrSelf(x,y) \wedge DescOrSelf(y,x) \rightarrow y = x$$

$$\text{(oneParent)} \quad \forall x \forall y \forall z\, Child_X(x,z) \wedge Child_X(y,z) \rightarrow x = y$$

$$\text{(oneRoot)} \quad \forall x \forall y\, Root(x) \wedge Root(y) \rightarrow x = y$$

$$\text{(topRoot)} \quad \forall x \forall y \quad DescOrSelf(x,y) \wedge Root(y) \rightarrow Root(x)$$

Intuitively, (oneRoot) states that the root of the XML tree is unique, and by (topRoot) has no ancestors beside itself. (someTag) and (oneTag) say that every element has precisely one tag.

The treeness of the data model is (partially) enforced by such constraints as (oneParent) (every element has at most one parent), (noLoop) (only trivial cycles are allowed). Observe that (base), (trans), (refl) above only guarantee that $DescOrSelf$ contains its intended interpretation, namely the reflexive, transitive closure of the $Child$ relation.

There are many models satisfying these constraints, in which $DescOrSelf$ is interpreted as a proper superset of its intended interpretation, and it is well-known that we have no way of ruling them out using first-order constraints because transitive closure is not first-order definable. Similarly, the "treeness" property of the $Child_X$ relation cannot be fully captured in first-order logic.

Example 4.5.4 (Relational Encoding of XML Tree Navigation)**.** *Consider an XPath expression Q defined as $//a$, which returns the set of nodes reachable by navigating to a descendant of the root and from there to a child tagged "a". Assume also that we materialize the view v defined as $//.//a$, i.e. which contains all "a"-children of descendants of descendants of the root. We can encode q, v as conjunctive queries $Q = enc(q)$, $V = enc(v)$ over schema $\mathcal{VREX}$.*

$$Q(y) : -Root(r), DescOrSelf(r,x), Child(x,y), Tag(y,\text{“}a\text{”})$$

$$V(y) : -Root(r), DescOrSelf(r,u), DescOrSelf(u,x), Child(x,y), Tag(y,\text{“}a\text{”})$$

It is easy to see that the decoding operation, which takes a relational query over $\mathcal{VREX}$ back to XQuery/XPath syntax, is straightforward.

Clearly, under arbitrary interpretations of the DescOrSelf relation, the two encodings are not equivalent, and Q cannot be reformulated to use V. But on intended interpretations, the DescOrSelf relation is transitive and reflexive, therefore v itself is equivalent to q and

R(y):- V(y)

is a relatioanl reformulation of Q using V.

4.5.5 Relational Model

It is well known that the relational data model endowed with key, uniqueness and foreign key constraints is captured by our pivot model (see Chapter 3): key/uniqueness constraints are expressible by EGDs, and foreign key constraints by TGDs.

4.5.6 Binding Patterns

We have seen above a natural way to model sets of key-value pairs using a relation. To simplify the discussion, we abstract from the *parentId* and *type* attributes of relation $Child_{KV}$ above, focusing on the binary relationship between keys and values:

$$KV(key, value).$$

Note that typical key-values store APIs require that values can be looked up only given their key, but not conversely. If we do not capture this limitation, the rewriting algorithm may

produce rewritings that correspond to no executable plan. For instance, consider a rewriting of the form:

$$R(v) : -KV(k, v).$$

R corresponds to no legal plan given the access limitation of KV, because R requires the direct extraction of all values stored in the store, without looking them up by key.

This setting involving relations with limited lookup access is well-known in the literature and is modeled via the concept of relations adorned with *binding patterns* [123]. Binding patterns are strings over the alphabet $\{i, o\}$. If position p^{th} in the pattern holds character i, this signals that the p^{th} attribute must be provided as input in a lookup. An o at position p signals that the p^{th} attribute can be extracted by the lookup.

In the key-value store modeling example, one would use KS^{io} to specify the lookup limitation, and the candidate rewriting would turn to a conjunctive query over binding-pattern-annotated atoms:

$$R'(v) : -KV^{io}(k, v).$$

The binding pattern now carries sufficient information to determine that R' requires k as input in order to look up v.

In a relational setting, query rewriting when access to the views is limited by binding patterns has been studied for conjunctive queries and views [74, 115], yet complete rewriting under both integrity constraints and binding patterns was not considered until [60]. [60] shows how to encode binding patterns using TGD constraints, which fit nicely into our pivot model. The idea is to introduce a unary relation, say $D(x)$ to state that x is accessible, and for each binding pattern a TGD stating that when all required input attributes of a relation are accessible, then the output attributes are accessible as well. This allows the chase with such TGDs to determine which attributes of a rewriting are accessible.

Example 4.5.5 (Binding Pattern Constraints)**.** *This example shows a query $Q(y, z) : -R(x, y, z)$ that can be rendered executable only after exploiting an integrity constraint and a binding pattern.*

Let us consider a store S that imposes a binding access constraint on a relation $R(x,y,z)$, which requires that y and z can be extracted (projected) only when x is given. Given this constraint, Q is not executable as such. Moreover, consider that we have a constraint that states an inclusion, which guarantees that all values of R's first column (x) are among those in the relation S's first column.

$$\forall x \forall y \forall z \; R(x,y,z) \rightarrow S(x)$$

In order to make the query executable, we first introduce a view for each access pattern on a given relation:

$$\forall x \forall y \forall z \; R(x,y,z) \rightarrow R^{ioo}(x,y,z)$$
$$\forall x \; S(x) \rightarrow S^{o}(x)$$

The above constraints state that x is required as input to project y and z from the relation R, and x is always accessible (can be extracted) from the relation S, respectively. Then, we need to introduce extractability constraints for access pattern modeling:

$$\forall x \; S^{o}(x) \rightarrow D(x) \tag{4.10}$$

$$\forall x \forall y \forall z \; R^{ioo}(x,y,z) \wedge D(x) \rightarrow D(y) \wedge D(z) \tag{4.11}$$

The extractability constraint 4.11 states that when x is accessible, then the attributes y and z are accessible as well.

The access pattern modeling above reduces the problem from rewriting under binding patterns to rewriting only under constraints. This means that if we have an algorithm for rewriting under constraints, we can "trick" it into rewriting under binding patterns even though it is unaware of them and does not treat them as first-class citizens. As a result, the found rewriting (the executable version of Q) is:

$$R'(y,z) \text{-} : R^{ioo}(x,y,z), S^{o}(x)$$

which extracts all x values from the relation S, and uses these values as a look-up key to extract y and z from the relation R.

4.5.7 Equality and Multiple Treatment of Nulls

An important issue raised by the polystore context is the fact that the notion of `null` values is treated differently across the spectrum of data models/stores, which in turn has implications on such fundamental primitives as equality and hence equi-joins, etc. Even in the single-model relational scenario, the classical theory of conjunctive queries, constraints, and view-based rewritings defines this real-life problem away by focusing on idealized relations without `null` values.

Direct application of classical theory would result in potentially unsound reformulation algorithms, because they would ignore the fact that for instance a MongoDB query joining [4] JSON data has a different semantics of equality from a SQL query joining two relations. For example, in MongoDB, join variables can match `null` values, but this is not the case in SQL. Such null-agnostic algorithms would generate a spurious reformulation of the query using the view.

To mitigate this issue, our current idea is to model each equality flavor by its own relational predicate, capturing *as much as possible of their intended semantics using constraints*. We use constraints to capture relationships between various equality flavors. For instance, the fact that whenever two values are equal according to flavor Eq_{S1} (e.g., Store S_1) they are also equal according to flavor Eq_{S2} (e.g., Store S_2), but only if they are both non-`null` could be captured as follows:

$$\forall x \forall y \; Eq_{S1}(x,y) \wedge \texttt{notnull}(x) \wedge \texttt{notnull}(y) \rightarrow Eq_{S2}(x,y) \tag{4.12}$$

$$\forall x \forall y \; Eq_{S2}(x,y) \rightarrow Eq_{S1}(x,y) \wedge \texttt{notnull}(x) \wedge \texttt{notnull}(y) \tag{4.13}$$

[4] Whenever we mention join, we refer to equi-join.

We introduce an `isnull` predicate, as well as a `notnull` predicate, relating them to equality. To correctly model the fact that `null` and `notnull` are mutually exclusive, we add denial constraints (see Section 4.5.9) such as :

$$\forall x\ \texttt{isnull}(x) \wedge \texttt{notnull}(x) \rightarrow \texttt{false}$$

Example 4.5.6 (Null Treatment Accross Different Equality Flavors)**.** *Consider the query*

$$Q(y,z) : -R(x_1,y), S(x_2,z), Eq_{S2}(x_1,x_2)$$

which joins R and S relations (in the store S_2) on x_1 and x_2 using the equality flavor Eq_{S2}, where join variables cannot match `null` values (the standard SQL semantics).

Now, suppose we have the following two views, which are materialized in the store S_1, where join variables can match `null` values (i.e., `null` = `null` is evaluated to `true`):

$$V_1^{S_1}(x_1,y) : -R(x_1,y)$$
$$V_2^{S_1}(x_2,z) : -S(x_2,z)$$

Taking as inputs Q, $V_1^{S_1}$, $V_2^{S_1}$, and the set of constraints 4.12 and 4.13, the relational rewriting algorithm under integrity constraints will find the following rewriting:

$$R(y,z) : -V_1^{S_1}(x_1,y), V_2^{S_1}(x_2,z), Eq_{S1}(x_1,x_2), \texttt{notnull}(x_1), \texttt{notnull}(x_2)$$

which joins $V_1^{S_1}$ and $V_1^{S_1}$ (in the store S_1) using the equality flavor Eq_{S1} and filtering out the `null` values of x_1 and x_2.

The rewriting R is equivalent to Q under `null`-aware semantics. Now, suppose that we do not appropriately capture different `null`-semantics across stores, in other words, we treat equality similarly across stores. For example, the constraint $Eq_{S2}(x,y) \rightarrow Eq_{S1}(x,y)$ indicates that the semantic of equality in S_1 is the same as the one in S_2. This will result in the following rewriting:

$$R'(y,z) : -V_1^{S_1}(x_1,y), V_2^{S_1}(x_2,z), Eq_{S1}(x_1,x_2)$$

which is not equivalent to Q under `null`*-aware semantic. The reason is that join variables can match* `null` *values when using the equality flavor* Eq_{S1}*, which is different from the equality flavor used in the original query.*

We use `isnull` and `notnull` predicates not only to model equality constraints but also to translate "is null" and "is not null" conditions in a query. For example, consider the SQL WHERE clause: "r.b is null AND s.b is null", it will be translated to :

$$R(b_1, y), S(b_2, x), \texttt{isnull}(b_1), \texttt{isnull}(b_2)$$

Given a rewriting with `isnull` and `notnull` predicates, they will be decoded to "is null" and "is not null" conditions, respectively.

In addition to supporting `null` value, JSON stores have a notion of `missing` data value [4, 11], which is produced by attempts to access non-existent fields or out-of-bound array elements. We leave the investigation of how far this constraint-based approach can be used to capture `missing` semantics across these stores to future work.

4.5.8 Comparison and Arithmetic Operators

To maintain extensibility, we adopt a uniform way to incorporate any additional language primitives by treating them as *user-defined functions with worst-case black-box semantics*, attempting to "make the box more transparent" by adding constraints that capture some of their intended semantics.

Comparison Operators. We use the relations $Lt(x,y)$, $Gt(x,y)$, $Lte(x,y)$, and $Gte(x,y)$ to encode "x is less than y", "x is greator than y", "x is less than or equal y", "x is greator than or equal y" comparison operators, respectively. We capture capture some of their properties via

constraints as follows:

$$\forall x \forall y \forall z \; Lt(x,y) \wedge Lt(y,z) \rightarrow Lt(x,z) \tag{4.14}$$

$$\forall x \forall y \forall z \; Gt(x,y) \wedge Gt(y,z) \rightarrow Gt(x,z) \tag{4.15}$$

$$\forall x \forall y \forall z \; Lte(x,y) \wedge Lte(y,z) \rightarrow Lte(x,z) \tag{4.16}$$

$$\forall x \forall y \forall z \; Gte(x,y) \wedge Gte(y,z) \rightarrow Gte(x,z) \tag{4.17}$$

$$\forall x \forall y \; Lt(x,y) \rightarrow Gt(y,x) \tag{4.18}$$

$$\forall x \forall y \; Gt(x,y) \rightarrow Lt(y,x) \tag{4.19}$$

The constraints from (4.14) to (4.17) capture the transitive property, and the constraints from (4.18) to 4.18) capture the reversal property.

Arithmetic Operators. The arithmetic operators such as addition operator would be modeled as a relation $add(arg_1, arg_2, res)$ with intended meaning $res = arg_1 + arg_2$, which one could partially capture with constraints stating that add is a functional relation:

$$\forall x \forall y \forall res_1 \forall res_2 \; add(x,y,res_1) \wedge add(x,y,res_2) \rightarrow res_1 = res_2$$

that is commutative, associative, and has zero:

$$\text{(comm)} \quad \forall x \forall y \forall res_1 \; add(x,y,res_1) \rightarrow add(y,x,res_1)$$

$$\text{(asso)} \quad \forall x \forall y \forall res_1 \forall z \forall res_3 \; add(x,y,res_1) \wedge add(res_1,z,res_3) \rightarrow$$

$$\exists res_4 \; add(y,z,res_4) \wedge add(x,res_4,res_3)$$

$$\text{(identity)} \quad \forall x \forall res \; add(x,0,res) \rightarrow res = x$$

4.5.9 Denial Constraints

A natural way of handling mutual exclusion of domains in the various models considered is by extending our pivot model to support *denial constraints*. They are constraints whose premise

$$\forall x \forall y\ Eq(x,y) \wedge Lt(x,y) \rightarrow \texttt{false}$$
$$\forall x \forall y\ Eq(x,y) \wedge Gt(x,y) \rightarrow \texttt{false}$$
$$\forall x \forall y\ Lt(x,y) \wedge Gt(x,y) \rightarrow \texttt{false}$$
$$\forall x \forall y\ Lte(x,y) \wedge Gt(x,y) \rightarrow \texttt{false}$$
$$\forall x \forall y\ Gte(x,y) \wedge Lt(x,y) \rightarrow \texttt{false}$$
$$\forall x \forall y\ Lt(x,y) \wedge Lt(y,x) \rightarrow \texttt{false}$$
$$\forall x \forall y\ Gt(x,y) \wedge Gt(y,x) \rightarrow \texttt{false}$$
$$\forall x\ notnull(x) \wedge null(x) \rightarrow \texttt{false}$$

Figure 4.6: Snippet of supported denail constraints.

has the same form as that of TGDs and EGDs, but whose conclusion contains simply the boolean value `false`.

Example 4.5.7 (Denail Constraints)**.** *Let us consider the relations $Gt(x,y)$ and $Lt(x,y)$, which we use to encode that x is greator than y, and x is less than y, respectively. Then to state that x cannot be both greator than y and less than y, we would use the denial constraint $\forall x,y\ Gt(x,y) \wedge Lt(x,y) \rightarrow$ `false`. Chasing with this denial constraint, a query Q that attempts to specify a condition, where x is greator than y and x is less than y, would add the atom `false` to its body, thus signaling Q's unsatisfiability (the query returns an empty answer).*

Figure 4.6 illustrates a snippet of denial constraints that ESTOCADA currently supports.

4.6 Encoding QBT^{XM} Queries into the Pivot Language

The purpose of the pivot language is to reduce the VBQR problem to a single-model setting. The pivot model enables us to represent relationally queries expressed in the various languages supported in our system (recall Table 4.1), and which can be combined into a single QBT^{XM} query. As shown in Figure 4.1, an incoming query Q is *encoded* as a relational conjunctive $enc(Q)$ over the relational encoding of the datasets it refers to (with extensions such as aggregates,

```
 FOR AJ:{SELECT M.PATIENTID AS patientID,
                A.ADMISSIONLOC AS admissionLoc,
                A.ADMISSIONTIME AS admissionTime,
                P.DRUG AS drug
          FROM  LABITEM L, MIMIC M, M.ADMISSIONS A,
                A.LABEVENTS LE, A.NOTEEVENTS NE, A.PRESCRIPTIONS P
          WHERE L.ITEMID=LE.LABITEMID AND
                L.CATEGORY='blood'    AND
                L.FLAG ='abnormal'    AND
                P.DRUGTYPE='additive' AND
                contains(NE.REPORT,'coronary artery')}
RETURN patientID, admissionLoc, admissionTime, drug
```

Figure 4.7: Motivating scenario QBT^{XM} query Q_1.

```
    MIMIC(M),
    Childƒ(M,PID,"PATIENTID","o"),
    Childƒ(M,A,"ADMISSIONS","o"),
    Childƒ(A,AID,"ADMISSIONID","o"),
    Childƒ(A,ALOC,"ADMISSIONLOC","o"),
    Childƒ(A,ATIME,"ADMISSIONTIME","o")->
    V2(PID,AID,ALOC,ATIME);
```

Figure 4.8: Relational encoding of QBT^{XM} view V_2 defined in Figure 4.3.

user-defined functions, and nested queries). Figure 4.7 shows the QBT^{XM} query Q_1 of the motivating scenario, and its relational encoding $enc(Q_1)$ appears in Appendix A.4.

4.7 Encoding QBT^{XM} Views as Constraints

We translate each view definition V expressed in QBT^{XM} into additional relational integrity constraints $enc(V)$ showing how the view inputs are related to its output. Figure 4.8 illustrates the relational encoding of QBT^{XM} view V_2 from Section 4.4, and constraints resulting from V_1 appears in Appendix A.3. We note that QBT^{XM} view definitions do not have aggregation. We leave the investigation of extending the rewriting algorithm to reason about views involving aggregations to future work.

4.8 From Rewritings to Integration Plans

We translate each query rewriting into a *logical plan* specifying (*i*) the (native) query to be evaluated within each store, and (*ii*) the remaining logical operations to be executed within the integration engine. The translation first decodes RW_r into QBT^{XM} syntax. Based on the heuristic that the store where a view resides is more efficient than the middleware, and thus it should be preferred for all the operations it can apply, decoding tries to delegated to the stores as much as possible.

To that end, the decoding phase first partitions each rewriting RW_r into view-level subqueries, which are sets of atoms referring to the virtual relations of a *same view*. If a group of view-level subqueries pertains to the same store (we record the store reference where each view is stored in a catalog) and if the store supports joins, we translate the group to a single query to be pushed to the store.

```
RWQ1<PID,ALOC,ATIME,DRUG>:-
V1(d1),
Child_J(d1,PID,"PATIENTID","o"),
Child_J(d1,AID,"ADMISSIONID","o"),
Child_J(d1,REPORT,"REPORT","o"),
   Value(report,"contains-coronary artery"),

V2(PID,AID,ALOC,ATIME),

V3(d2),
Child_J(d2,PID,"PATIENTID","o"),
Child_J(d2,AID,"ADMISSIONID","o"),
Child_J(A,P,"PRESCRIPTIONS","o"),
Child_J(P,DRUG,"DRUG","o"),
Child_J(P,DRUGTYPE,"DRUGTYPE","o"),
     Value(DRUGTYPE,"additive");
```

Figure 4.9: Rewriting RWQ_1 of QBT^{XM} query Q_1.

Example 4.8.1 (Delegation-Aware Decoding)**.** *Consider the rewriting RWQ_1 of QBT^{XM} query Q_1 shown in Figure 4.9. First, we delimit the view-wide subqueries (delimited by empty lines in the figure), calling them RWQ_1^1, RWQ_1^2 and RWQ_1^3 in order from the top. The subquery heads contain variables from the head of RWQ_1 and join variables shared with other subqueries. For*

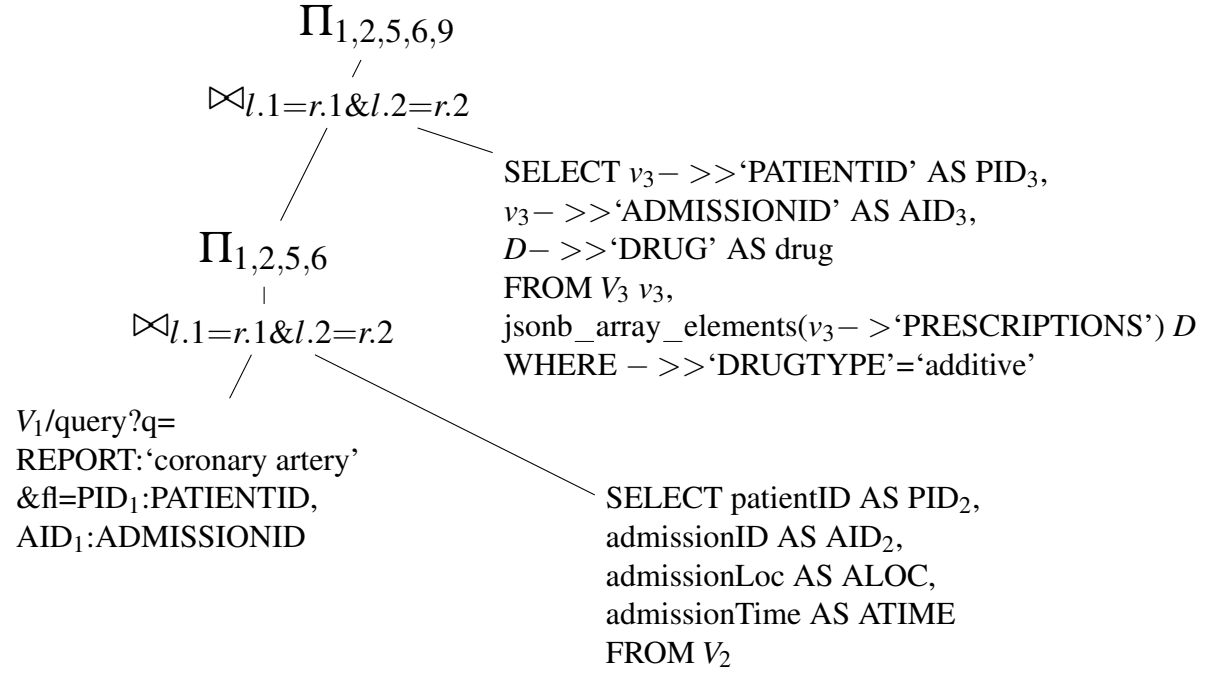


Figure 4.10: Integration plan of the motivating scenario using Tatooine hybrid engine.

example, the head of RWQ_1^2 contains the variables ALOC and ATIME (in the head of RWQ_1), and also PID, AID, needed to join with RWQ_1^1, and RWQ_1^3. These relational subqueries are then decoded to the native syntax of their respective stores, each constituting a block of the resulting QBT^{XM} rewriting $dec(RWQ_1)$ (shown in Appendix A.4).

ESTOCADA's plan generator next translates the decoded QBT^{XM} rewriting to a logical plan that pushes leaf blocks to their native store, applying last-step operators on the results. The integration plan for RWQ_1 is shown in Figure 4.10. We note that when using ESTOCADA's polystore engine Tatooine [45], the results of sources' subqueries are translated into Tatooine's internal nested-valued tuple-based model (nested relation [57]).

Algorithm 1: Delegation-aware decoding QBT^{XM}.

Input : CQ $rw(\bar{x}) : -a_1 \wedge \cdots \wedge a_n$, Catalog c
Output : Delegation-Aware Decoded Rewriting into QBT^{XM}

$cqs \leftarrow$ split rw atoms by store reference
foreach $cq_1 \in cqs, cq_2 \in cqs$ **do**
    if *$cq_1 \neq cq_2$ and cq_1, cq_2 share variables and cq_1, cq_2 are in the same store s and s supports join* **then**
        remove cq_1 from cqs;
        remove cq_2 from cqs;
        add $cq_1 \cup cq_2$ to cqs;
    end
end
$returnClause \leftarrow \emptyset$;
foreach $x \in \bar{x}$ **do**
    add x to $returnClause$;
end
$whereClause \leftarrow \emptyset$;
foreach $(cq_1, s_1) \in cqs$, $(cq_2, s_2) \in cqs$, **do**
    if *head(cq_1) share same variable z in head(cq_2)* **then**
        $(cq_2', s_2) \leftarrow$ rename z in head(cq_2) and body(cq_2) to z_i;
        remove (cq_2, s_2) from cqs;
        add (cq_2', s_2) to cqs;
        add $z = z_i$ to $whereClause$;
    end
end
$forBlocks \leftarrow \emptyset$;
foreach $(cq, s) \in cqs$ **do**
    add decode(cq, s) to $blocks$;
end
$decodedRW \leftarrow concat(forBlocks, whereClause, returnClause)$
return $plan$;

4.9 PACB Optimization and Extension

We detail below just enough of the PACB algorithm's inner working to explain our optimization. Then, Section 4.9.1 introduces our optimization in the original PACB setting (conjunctive relational queries and set semantics). Section 4.10 extends this to bag semantics, Section 4.11 extends it to QBT^{XM}.

A key ingredient of the PACB algorithm is to *capture views as constraints*, in particular TGDs, thus reducing the view-based rewriting problem to constraints-only rewriting. For a given view V, the constraint V_{IO} states that for every match of the view body against the input data there is a corresponding tuple in the view output; the constraint V_{OI} states the converse inclusion, i.e., each view tuple is due to a view body match. Then, given a set $\mathcal{V}$ of view definitions, PACB defines a set of view constraints $C_{\mathcal{V}} = \{V_{IO}, V_{OI} \mid V \in \mathcal{V}\}$.

The constraints-only rewriting problem thus becomes: given the source schema σ with a set of integrity constraints I, a set $\mathcal{V}$ of view definitions over σ and the target schema τ which includes $\mathcal{V}$, given a conjunctive query Q expressed over σ, find reformulations ρ expressed over τ that are equivalent to Q under the constraints $I \cup C_{\mathcal{V}}$.

For instance, if $\sigma = \{R,S\}$, $I = \emptyset$, $\tau = \{V\}$ and view V materializes the join of tables R and S, $V(x,y) :- R(x,z), S(z,y)$, the constraints capturing V are:

$$V_{IO}: \quad R(x,z) \wedge S(z,y) \rightarrow V(x,y)$$
$$V_{OI}: \quad V(x,y) \rightarrow \exists z\, R(x,z) \wedge S(z,y).$$

For input σ-query $Q(x,y) :- R(x,z), S(z,y)$, PACB finds the τ-reformulation $\rho(x,y) :- V(x,y)$. Algorithmically, this is achieved by:

(*i*) chasing Q with the set of constraints $I \cup C_{\mathcal{V}}^{IO}$ where $C_{\mathcal{V}}^{IO} = \{V_{IO} \mid V \in \mathcal{V}\}$;

(*ii*) restricting the chase result to only the τ-atoms (the result is the *universal plan*) U;

(*iii*) chasing U with the constraints in $I \cup C_{\mathcal{V}}^{OI}$, where $C_{\mathcal{V}}^{OI} = \{V_{OI} \mid V \in \mathcal{V}\}$; the result is

denoted B and called the *backchase*; finally:

(*iv*) matching Q against B and outputting as rewritings those subsets of U that are responsible for the introduction (during the backchase) of the atoms in the image of Q. [5]

In our example, I is empty, $C_{\mathcal{V}}^{IO} = \{V_{IO}\}$, and the result of the chase in phase (*i*) is $Q_1(x,y) :- R(x,z), S(z,y), V(x,y)$. The universal plan obtained in phase (*ii*) by restricting Q_1 to τ is $U(x,y) :- V(x,y)$. The result of backchasing U with $C_{\mathcal{V}}^{OI}$ in phase (*iii*) is $B(x,y) :- V(x,y), R(x,z), S(z,y)$ and in phase (*iv*) we find a match from Q's body into the R and S atoms of B, introduced during the backchase due to U's atom $V(x,y)$. This allows us to conclude that $\rho(x,y) :- V(x,y)$ is an equivalent rewriting of Q.

4.9.1 PACB^{OPT}: Optimized PACB

The idea for our optimization was sparked by the following observation. The backchase phase (step (*iii*)) involves the V_{OI} constraints for all $V \in \mathcal{V}$. The backchase attempts to match the left-hand side of each V_{OI} for each V repeatedly, leading to wasted computation for those views that have no match. In a polystore setting, the large number of data sources and stores lead to a high number of views, most of which are often irrelevant for a given query Q.

This observation leads to the following modified algorithm. Define the set of views mentioned in the universal plan U as *relevant to Q under I*, denoted $relev_I(Q)$. Define the optimized PACB algorithm PACB^{OPT} identical with PACB except phase (*iii*) where PACB^{OPT} replaces $C_{\mathcal{V}}^{OI}$ with $C_{relev_I(Q)}^{OI}$. That is, PACB^{OPT} performs the backchase only with the *OI*-constraints of the views determined in phase (*i*) to be relevant to Q, ignoring all others. This modification is likely to save significant computation when the polystore includes many views. In our example assume that, besides V, $\mathcal{V}$ contained 1000 other views $\{V^i\}_{1 \le i \le 1000}$, each irrelevant to Q. Then the universal plan obtained by PACB would be the same as U above, and yet

[5] Recall that a chase step s with constraint c matches c's premise against existing atoms e and adds new atoms n corresponding to c's conclusion. To support fast detection of responsible atoms in Phase (*iv*), s records that the e atoms are responsible for the introduction of the n atoms [84]. Our optimization does not affect Phase (*iv*).

$\{V_{OI}^i\}_{1\leq i\leq 1000}$ would still be considered by the backchase phase, which would attempt at every step to apply each of these 1000 constraints. In contrast, PACB$^{\mathcal{OPT}}$ would save this work in particular, in our setting, where the premises of V_{OI} views can have up to 10-way joins.

In general, ignoring even a single constraint c during backchase may lead to missed rewritings [121]. This may happen *even when c mentions elements of schema* σ *that are disjoint from those mentioned in U,* if the backchase with I exposes semantic connections between these elements. In our setting, we prove that this is not the case:

Theorem 4.9.1. *Algorithm PACB$^{\mathcal{OPT}}$ finds the exact same rewritings as the original PACB.*

This is because we only ignore some *view-capturing constraints*, which can be shown (by analyzing the backchase evolution) *never* to apply, regardless of the constraints in I.

Proof. Assume for the sake of contradiction that the set $\mathcal{RW}^{\mathcal{OPT}}$ of rewritings found by PACB$^{\mathcal{OPT}}$ is not the same as the set $\mathcal{RW}$ of rewritings found by the original PACB ($\mathcal{RW}^{\mathcal{OPT}} \neq \mathcal{RW}$). Let $\mathcal{V}$ be the set of views and $V_{irr} \in \mathcal{V}$ be the set of irrelevant views to the query Q.

By applying the PACB$^{\mathcal{OPT}}$ algorithm, the chase introduces the relevant views $V_{rel} \in \mathcal{V}$ (i.e., $V_{rel} = \mathcal{V} - V_{irr}$) into the universal plan U. Now, in the backchase phase, only the premise of each backward constraint $v \in V_{rel}$ can be fully matched against U. This makes the chase step applicable, where the conclusion of each v will be added to the evolving backchase instance B. When no further chase steps are applied, the query Q will be matched against B, and the output will be the rewritings $\mathcal{RW}^{\mathcal{OPT}}$, which are those subsets of U that are responsible for the introduction (during the backchase) of the atoms in the image of Q in B. When applying the original PACB, similarly, the chase phase introduces only V_{rel} into U. During the backchase phase, V_{irr} will never be applied since their premises cannot be fully matched into U, resulting into *the same instance B* as in the PACB$^{\mathcal{OPT}}$ algorithm. Thus, when Q is matched against B, this outputs the rewritings $\mathcal{RW}$, which are exactly the same as $\mathcal{RW}^{\mathcal{OPT}}$. However, this contradicts that $\mathcal{RW}^{\mathcal{OPT}} \neq \mathcal{RW}$. Hence, our original statement is true that the algorithm PACB$^{\mathcal{OPT}}$ finds

the exact same rewritings as the original PACB. □

Combined with the main result from [84], Theorem 4.9.1 yields the completeness of PACB^{OPT} in the following sense:

Corollary 4.9.1. *Whenever the chase of input conjunctive query Q with the constraints in I terminates*[6]*, we have:*

(a) algorithm PACB^{OPT} without a cost model enumerates all join-minimal[7] *$\mathcal{V}$-based rewritings of Q under I, and*

(b) algorithm PACB^{OPT} equipped with a cost model c finds the c-optimal join-minimal rewritings of Q.

In Section 4.13.3, we evaluate the performance gains of PACB^{OPT} over the original PACB, measuring up to 40x speedup when operating in the polystore regime.

4.10 Extending PACB^{OPT} to Bag Semantics

We extended PACB^{OPT} to find equivalent rewritings of an input conjunctive query under bag semantics (the original PACB only addresses set semantics). The extension involves a modification to phase (*iv*). In its original form, the matches from Q into the backchase result B are not necessarily injective, being based on *homomorphisms* [26]. These allow multiple atoms from Q to map into the same atom of B. To find bag-equivalent rewritings, we disallow such matches, requiring match homomorphisms to be injective.

[6] Termination of the chase with TGDs is undecidable [61], but many sufficient conditions for termination are known, starting with weak acyclicity [70]. Our constraints are chosen so as to ensure termination.

[7] A rewriting of Q is join-minimal if none of its joins can be removed while preserving equivalence to Q.

4.11 PACB$^{\mathcal{OPT}}_{\mathcal{QBT}}$: Extending PACB$^{\mathcal{OPT}}$ to QBTs

The original PACB algorithm (and our extensions thereof) have so far been defined for conjunctive queries only. However, recall that QBTs (Section 4.4.1) are nested.

We extend the PACB$^{\mathcal{OPT}}$ algorithm to nested QBTs as follows. Each block B is rewritten *in the context of its ancestor blocks* $A_1, \ldots, A_n$, to take into account the fact that the free variables of B are instantiated with the results of the query corresponding to the conjunction of its ancestor blocks. We therefore replace B with the rewriting of the query $A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge B$. We call the resulting algorithm PACB$^{\mathcal{OPT}}_{\mathcal{QBT}}$, using the notation PACB$^{\mathcal{OPT}}_{\mathcal{QBT}}\langle C, c\rangle$ to emphasise the fact that it is parameterized by the constraints C and the cost model c.

Recalling the uninterpreted (black-box) function semantics from Section 4.2, Corollary 4.9.1 implies:

Corollary 4.11.1. *Under uninterpreted-function semantics, Corollary 4.9.1 still holds when we replace conjunctive queries with QBT queries and PACB$^{\mathcal{OPT}}$ with PACB$^{\mathcal{OPT}}_{\mathcal{QBT}}$.*

4.12 Guarantees on the Reduction

We provide the following formal guarantees for our solution to the cross-model rewriting problem based on the reduction to single-model rewriting. Recall from Section 5.4 that $enc(M)$ are the relational constraints used to encode $M \in M^i$ in virtual relations, $enc(Q)$ encodes a QBT^{XM} query Q as a QBT query over the virtual relations, and $dec(R)$ decodes a QBT query over the virtual relations into a QBT^{XM} query. Also recall from Section 4.9 that given a set $\mathcal{V}$ of QBT^{XM} views, $C_{\mathcal{V}}$ are the relational constraints used to capture $\mathcal{V}$. We have:

Theorem 4.12.1 (Soundness of the reduction)**.** *Let Q be a QBT^{XM} query over a polystore over the set of data models M^i. Let I be a set of integrity constraints satisfied by the data in the polystore*

and $enc(I)$ be their relational encoding. For every rewriting R obtained via our reduction, i.e.

$$R = dec(\,PACB^{\mathcal{OPT}}_{\mathcal{QBT}}\langle enc(I) \cup \bigcup_{M \in M^i} enc(M) \cup C_{\mathcal{V}}, c\rangle(enc(Q))\,),$$

R is a c-optimal $\mathcal{V}$-based rewriting equivalent to Q under I, assuming black-box function semantics (Section 4.2).

In the above, $enc(I) \cup \bigcup_{M \in M^i} enc(M) \cup C_{\mathcal{V}}$ is the set of constraints used by the $\text{PACB}^{\mathcal{OPT}}_{\mathcal{QBT}}$ algorithm; Theorem 4.12.1 states the correction of our approach, outlined in Figure 5.1.

Proof sketch. Let F_r be a family of virtual relations detailed in Section 4.5. The core strategy of our approach is to compile the operations in a query Q and a set of views V to virtual relations in F_r. The relations in F_r may not fully specify the exact properties of the operations they capture. They are interpreted as a proper superset of their intended interpretations. However, they can satisfy some constraints I. In other words, this large class of relations can contain the intended relations interpretations, which we capture via constraints $enc(I)$. While the encoded constraints $enc(I)$ do not contradict the intended operations interpretations and properties (this holds for our encoded constraints $enc(I)$), the theoretical result in [65] combined with the soundness of PACB [84] and Thereom 4.9.1 guarantee that soundness is preserved for any query Q that is compilable relationally. If a relational reformulation is found, then it is the encoding of QBT^{XM} rewriting, which can be retrieved by the decoding phase that follows the connections among atoms and knowledge of the encoded data models. □

Completeness Discussion. The natural question to be asked is whether we capture all constraints that follow from the original data model. If we miss some constraints, then our rewriting approach, which includes encoding, rewriting with constraints, and finally decoding the rewriting, remains sound, returning only equivalent rewritings. It may not be complete, meaning that it may miss existing rewritings.

The question has to be answered in a nuanced way because of the following reason: classical results from logic [129] tell us that for any tree-like models (including XML and JSON) the the answer is no, and it is impossible to capture all constraints that follow from these models even when using full-fledged first-order logic to express constraints (TGDs and EGDs are merely particular cases of first-order constraints).

But it is possible to capture all constraints that matter to a certain language L, in the sense that L does not have sufficient expressivity to distinguish among models that satisfy all constraints and models that satisfy only the ones we capture. For practicality, we support the entire language via black-box encodings of the primitives with undecidable/ computationally expensive reasoning, giving up on the completeness guarantee while adopting a solution that allows progressive enhancements of a best-effort approach to completeness. In other words, the more relevant constraints we add, the fewer rewritings we miss, allowing progressive refinement of the rewriting algorithm without changing code, simply by adding constraints.

Not guaranteeing completeness has no discernible practical impact, since already for SQL absolute completeness of the rewriting procedure is precluded by the undecidability of reasoning about primitives that render SQL (and related languages) Turing-complete (e.g., user-defined functions and arithmetic). In practice, completeness is not attained by commercial SQL optimizers, even when the user queries fall in a restricted class with theoretically feasible rewriting completeness, due to the combinatorial explosion of searching for all possible plans and the imperfection of the cost model.

4.13 Experimental Evaluation

In this section, we describe our experimental evaluation to show the effectiveness of our cross-models rewriting technique. Section 4.13.1 describes our experiment setup. In Sections 4.13.2 and 4.13.3, we discuss the results of our noval cross-models rewriting approach. We

summarize our experimental finding and takeaways in Section 4.13.4.

4.13.1 Experiment Setup

We use a single machine with an Intel(R) Xeon(R) CPU E5-2640 v4@2.40GHz, 20 physical cores (40 logical cores), and 123GB RAM of main memory. The machine is equipped with a 1TB SSD storage device, where the read speed is 616 MB/s, and the write speed is 455 MB/s (interesting since some systems write intermediate results to disk).

Polystore Configuration. For our main polystore setting (called "ESTOCADA polystore engine" hereafter), we use Tatooine [45], a lightweight execution engine and a set of data stores, selected for their capabilities and popularity: an RDBMS (PotsgreSQL v9.6), JSON document stores/engines (PostgreSQL v9.6, MongoDB v4.0.2 , AsterixDB v0.9.4, and SparkSQL v2.3.2) and a text search engine (Solr v6.1). We set a memory budget of 60GB for all systems. We configure the number of utilizable cores to 40 (for systems with such a configurable parameter). We set the compiler frame size of AsterixDB (for its batch-at-a-time query processor) to 6MB.

Dataset. We use the real-life **46.6 GB** MIMIC-III dataset [88] described in Section 4.1.1.

Generating Query and View Families. We create a micro-benchmark based on MIMIC dataset. We define a set QT of 25 query templates, each checking meaningful conditions against patients' data. These are parameterized by selection constants, and they involve navigation into the JSON document structure. Each query/view is obtained by joining a subset of QT and picking values for the selection constants, leading to an exponential space of meaningful queries and views. Among them, we identify those with non-empty results: first, non-empty instantiations of QT queries then join of two such queries, then three etc., in an adaptation of the Apriori algorithm [29].

Example 4.13.1 (Generating Queries/Views based on Query Templates)**.** *Consider the query templates QT_0, QT_1 and QT_2 in Appendix A, shown directly in relationally encoded form. QT_0 asks for patients' information including patient's PATIENTID, DOB, and GENDER. QT_1 asks*

for all "abnormal" lab measurement results for patients. QT_2 asks for bloodwork-related lab measurements. The natural join of QT_0, QT_1 and QT_2 yields a query Q which seeks information on patients with abnormal blood test results. The query Q and its translation in AsterixDB SQL++ syntax appear in Appendix A.

The Queries. We choose 50 queries Q_{EXP} among those described above. 58% of the queries $(Q_{01}, \ldots, Q_{29})$ contain full-text operations; all involve joins, selections, and at least two-level-deep navigation into the JSON document structure.

The Views. We materialized a set of views V_{EXP} as follows: We store in PostgreSQL six relational views $V_{PostgreSQL} \subset V_{EXP}$ of the MIMIC-III dataset, comprising the uniformly structured part of the dataset (including all patient metadata and admission under specific services such as Cardiac Surgery, Neurologic Medical, etc).

We also stored in PostgreSQL a set $V_{PostgresJSON} \subset V_{EXP}$ of 21 views which are most naturally represented as nested JSON documents. These views store for instance: (*i*) lab tests conducted for each patient with "abnormal" test results (e.g., blood gas, Cerebrospinal Fluid (CSF)); (*ii*) data about drugs prescribed to patients sensitive to certain types of antibiotics (e.g., CEFEPIME); (*iii*) data about drugs and lab tests prescribed to each patient who underwent specific types of procedures (e.g., carotid endarterectomy); (*iv*) microbiology tests were conducted for each patient; (*v*) lab tests for each patient who was prescribed certain drug types (e.g. additive, base); etc. We placed in Solr a view $V_{Solr} \in V_{EXP}$, storing for each admitted patient the caregivers' reports (unstructured text). The usage of AsterixDB, SparkSQL, and MongoDB is detailed below.

4.13.2 Cross-Store Rewritings Evaluation

In this section, we study the effectiveness of ESTOCADA cross-store query rewriting: (*i*) compared to single-store query evaluation and (*ii*) improving performance in the pre-existing polystore engines: BigDAWG [68] and ESTOCADA polystore engine (Tatooine) [45].

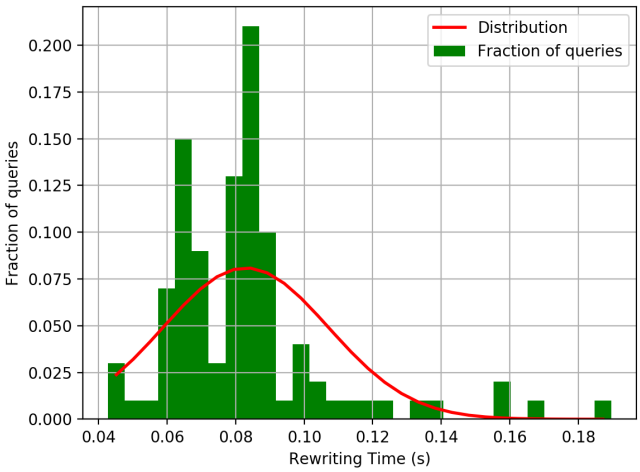


Figure 4.11: Rewriting time (28 relevant views).

Single-Store Evaluation Comparison. Figure 4.11 shows the rewriting time for the query set using the views described above. Notice that most queries are rewritten within 100ms, and the few outliers require at most 190ms, which is negligible compared to the query execution time (in the order of minutes, as seen in Figures 4.13 and 4.14). Figure 4.12 shows the distribution of rewriting time over the same query set when we scale up the number of relevant views (we did not materialize them) to 128 views (this is for stress-test purposes, as 128 views relevant to the data touched by a single query is implausible).

Queries with Text Search Predicates. Figure 4.13 reports the total execution time plus rewriting time of ESTOCADA using the views V_{EXP} for Q_{01} to Q_{29}, all of which feature a text search predicate against the text notes in the caregiver's report. For each query, cross-model rewriting and evaluation significantly improve the performance of a direct query evaluation in a single store.

For SparkSQL and AsterixDB, queries (Q_{01},..., Q_{29}) took over 25 minutes (the timeout value we used). The bottleneck is the text search operation, and a JSON array unnest operator: full-text indexing is lacking in SparkSQL, and limited to exclude text occurring within JSON arrays in AsterixDB [8]. This confirms our thesis that cross-models redundant storage of data into

[8] AsterixDB v9.8 supports array indexing.

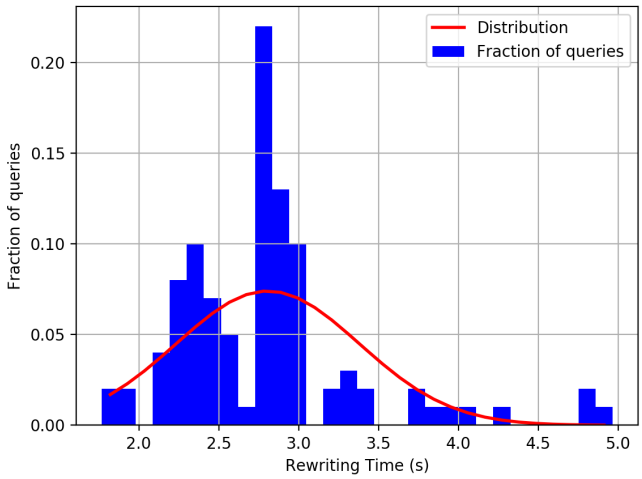


Figure 4.12: Rewriting time (128 relevant views).

the stores most suited for an expected operation is worthwhile.

For MongoDB and PostgreSQL, the execution time is correlated with the number of JSON array unnest operators. For instance, query Q_{25} has five unnest operators, whereas query Q_{17} has 2. PostgreSQL outperforms MongoDB because the latter lacks join-reordering optimization, and it does not natively support inner joins. These must be simulated by left outer joins – using the $lookup operator – followed by a scan and selection for non-null values –using the $match operator.

Queries without Text Search Predicates. Figure 4.14 repeats this experiment for queries $Q_{30},\ldots,Q_{50}$, which do not perform a text search. These queries each feature join, selection and navigation into the JSON document structure (at least two levels deep). The relevant views for these queries are $V_{PostgreSQL} \cup V_{PostgresJSON}$; again, exploiting them improves single-store evaluation. SparkSQL has the highest query execution time; this is because it supports navigation into JSON arrays through the EXPLODE function, which is highly inefficient. We observe that in a single-store evaluation, PostgreSQL is more efficient; this is why we choose it to store $V_{PostgresJSON}$.

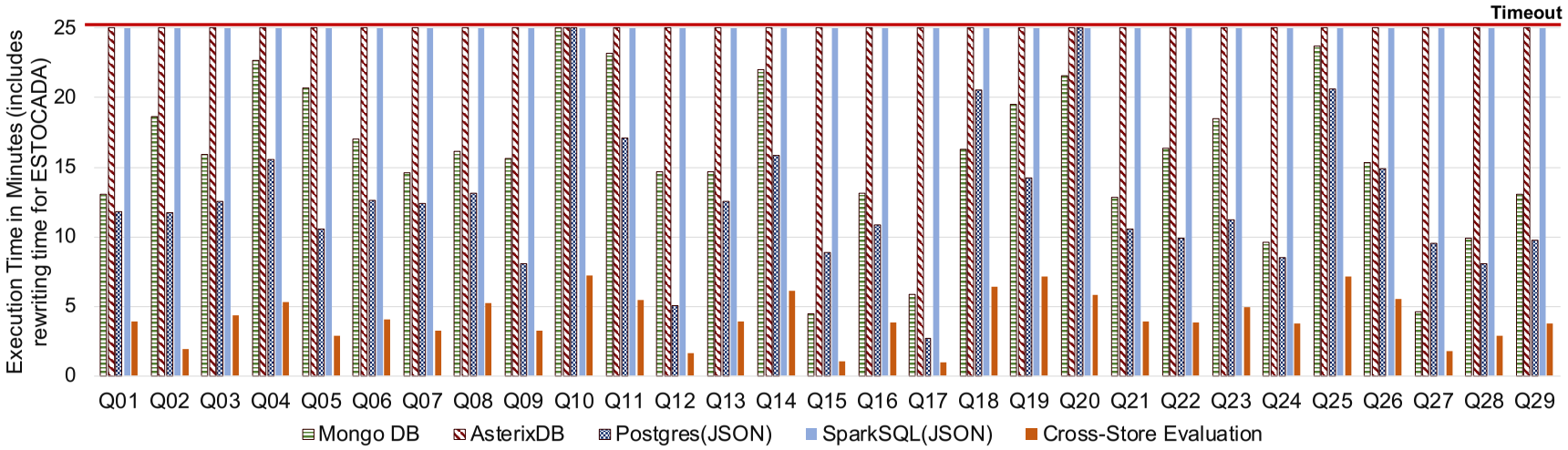


Figure 4.13: ESTOCADA (Relational and JSON views in PostgreSQL, text view in Solr) vs. single-store evaluation.

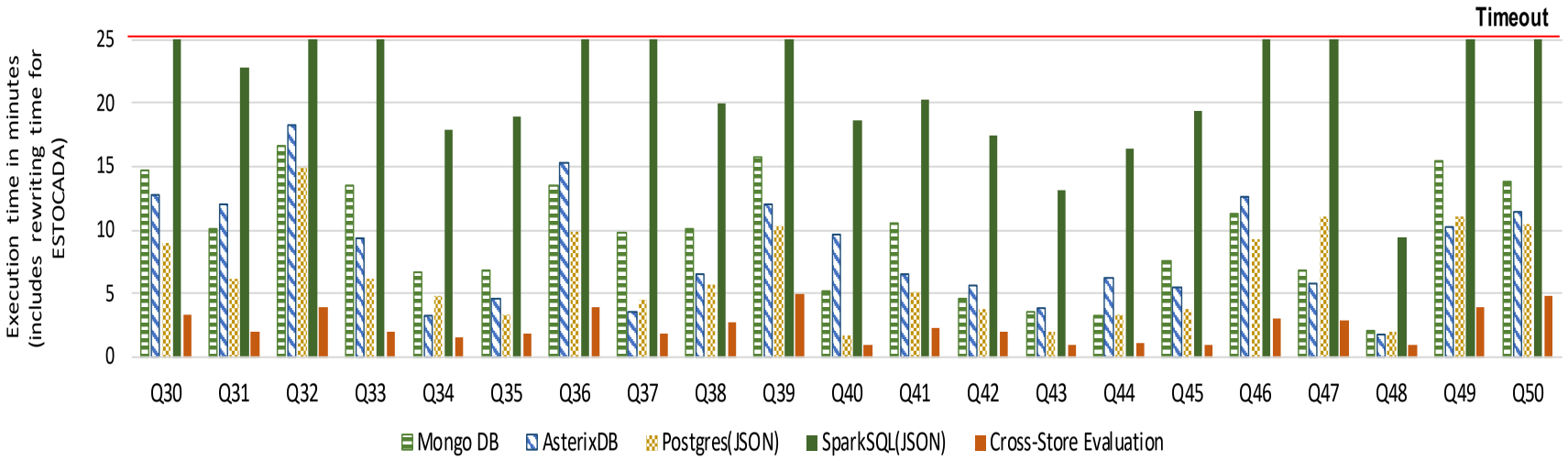


Figure 4.14: ESTOCADA (Relational and JSON views in PostgreSQL) vs. single-store evaluation.

Polystore Evaluation Comparison. We study the benefits that ESTOCADA can bring to an existing polystore system by transparently rewriting queries using materialized views stored across different stores. We used two polystore engines: (*i*) ESTOCADA polystore engine instantiated with two PostgreSQL servers (instances), one stores relational data while the other stores JSON, and Solr v6.1 for storing text; (*ii*) the latest release of the BigDAWG polystore. A key BigDAWG concept is an *island*, or collection of data stores accessed with a single query language. BigDAWG supports islands for relational, array, and text data, based on the PostgreSQL, SciDB [20] and Apache Accumulo [3] stores, respectively. BigDAWG queries use explicit CAST operators to *migrate* an intermediary result from one island to another.

To work with BigDAWG, we extended it with two new CAST operators: (*i*) to migrate Solr query results to PostgreSQL; (*ii*) to migrate PostgreSQL JSON query results to a PostgreSQL relational instance and vice-versa. The main difference between our ESTOCADA polystore engine and BigDAWG is that we join subquery results *in the mediator* using a BindJoin[123], whereas BigDAWG *migrates* such results to a store in an island capable of performing the join.

Data Storage. We store the MIMIC-III dataset (in both systems) as follows: patient metadata in PostgreSQL relational instance; caregivers' reports in Solr; patients' lab events, prescriptions, microbiology events, and procedures information in the PostgreSQL JSON instance.

Polystore Queries. We rewrite $Q_{01}, \ldots, Q_{29}$ in BigDAWG syntax, referring to each part of the data from its respective island (as BigDAWG requires); we refer to the resulting query set as $Q_{BigDAWG}$. We have the same set of queries in QBT^{XM} syntax; we call these queries $Q_{\text{ESTOCADA } Polystore}$.

The Views. To the view set V_{EXP} introduced above, we have added a new set of views V_{NEW} which can benefit $Q_{BigDAWG}$ and $Q_{\text{ESTOCADA } Polystore}$ queries as we detail below.

The queries vary in terms of full-text search predicates selectivity. $\approx 60\%$ of $Q_{BigDAWG}$ and Q_{ESTOCADA} queries consist of full-text search predicates, which are *not highly selective* (e.g., "low blood pressure"). We refer to these queries as $Q^{60\%}_{BigDAWG}$ and $Q^{60\%}_{\text{ESTOCADA } Polystore}$. We

observed that the cost of such queries in BigDAWG is dominated by moving and ingesting the results of Solr queries in PostgreSQL (JSON instance) to join them with the results of other sub-queries. To alleviate this overhead, we materialized *in the PostgreSQL instance storing JSON* a set of 6 views V_{NEW}, *which join the data from Solr (using those full-text predicates in the views definitions) with JSON data from the PostgreSQL JSON instance*.

Given the $Q^{60\%}_{BigDAWG}$ queries, $V_{PostgresJSON}$, $V_{PostgreSQL}$ and V_{NEW}, our cross-model views-based rewriting approach finds rewritings using views from PostgreSQL (relational instance) and PostgreSQL (JSON instance). The performance saving is due to the fact that we no longer have to move data from Solr to a PostgreSQL instance (see Figure 4.15 for queries labeled *). Although ESTOCADA polystore engine does not require any data movement, it still benefits from utilizing $V_{PostgresJSON}$, $V_{PostgreSQL}$ and V_{NEW} to answer $Q^{60\%}_{\text{ESTOCADA } Polystore}$ queries as shown in Figure 4.15 (queries labeled with *). In contrast, the remaining 40% of $Q_{BigDAWG}$ and $Q_{\text{ESTOCADA } Polystore}$ queries have *highly selective full-text search predicates* (we refer to these as queries $Q^{40\%}_{BigDAWG}$ and $Q^{40\%}_{\text{ESTOCADA } Polystore}$).

The high selectivity of these queries reduces the overhead of moving the data from Solr to PostgreSQL in BigDAWG, and in general the data movement is not a bottleneck for these queries. However, both systems can benefit from the materialized views V_{EXP} to evaluate these queries, as shown in Figure 4.15 (queries labeled with +).

As mentioned earlier, ESTOCADA polystore engine and BigDAWG differ in terms of multi-store join evaluation strategies, leading to different performance variations when the queries and/or views change. On the queries $Q^{60\%}_{BigDAWG}$ and $Q^{60\%}_{\text{ESTOCADA } Polystore}$, where the full-text search predicates are not very selective, BigDAWG execution time is dominated by moving partial results to the join server. In contrast, ESTOCADA polystore engine performs better since it computes the join in memory in the mediator; thus, it does not need to pay the intermediary result storage cost.

For the other 40% of queries (with very selective full-text search predicates), BigDAWG data movement cost is negligible; thus its evaluation time is dominated by evaluating the join

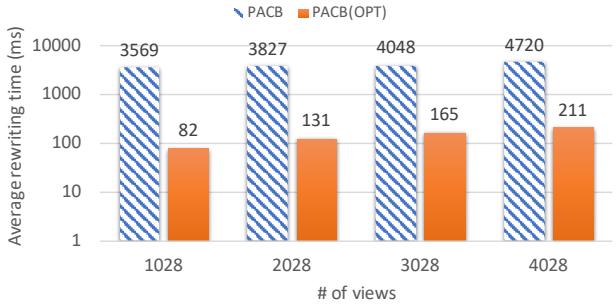


Figure 4.16: PACB vs PACB^{OPT} rewriting performance.

between sub-queries results in the PostgreSQL island, where join algorithms and orders may be better chosen than in the ESTOCADA polystore engine (the two platform support different sets of join algorithms). However, the differences are negligible. The main conclusion of this experiment, however, is that our cross models views-based rewriting approach *improves performance in both polystore engines*.

4.13.3 PACB vs. PACB^{OPT}

This experiment demonstrates the performance gains of PACB^{OPT} over PACB in a polystore setting. We consider the queries Q_{EXP} and the 28 views V_{EXP} that can be utilized to answer Q_{EXP}, introduced above. We add to V_{EXP} some *irrelevant* views $V_{irrel} \subseteq (\mathcal{V} - V_{EXP})$. The backward constraints of V_{irrel} can partially match against the universal plan of chasing each query in the benchmark. However, they cannot be used to answer a query; therefore, we call them "irrelevant". The premises of the backward constraints of V_{irrel} can have up to 10-way joins. PACB^{OPT} would save attempts performed by PACB at every step to apply each of these constraints. We scale the size of V_{irrel} from 1000 to 4000 (from 2000 to 4000, the premises (backward) of the newly added constraints have up to 3-ways joins). Figure 4.16 presents the *average rewriting time* of Q_{EXP} in the presence of $V_{EXP} \cup V_{irrel}$; the y axis is in log scale.

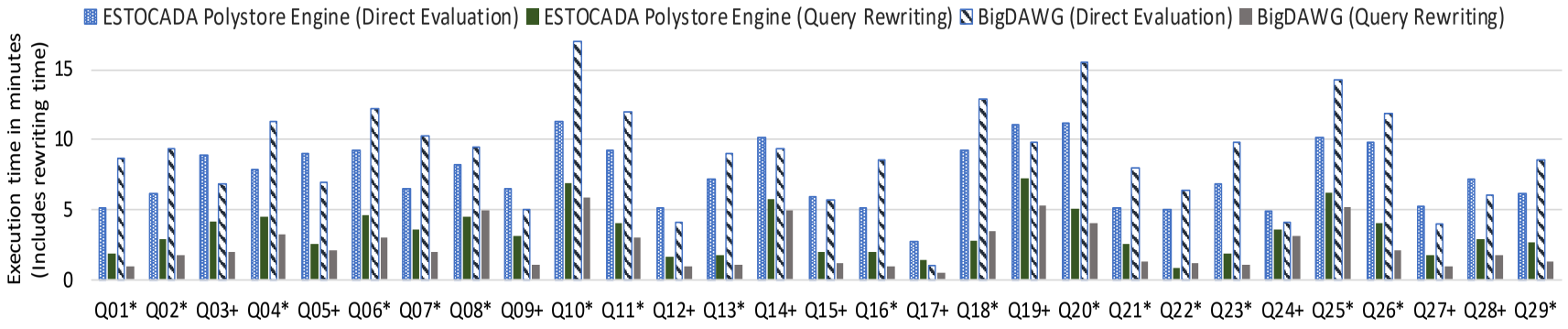


Figure 4.15: Queries and rewritings evaluation in polystore engines.

4.13.4 Summary of Experimental Findings

We have shown that our cross-models views-based rewriting approach is *portable* across polystore engines. Moreover, it is *worthwhile*, as it improves their performance in natural scenarios for both cross-models and even single-model user queries; the latter are improved by rewritings using a cross-store (and cross-models) set of materialized views (even when accounting for the time it takes to find the rewriting). We have also shown that the time spent searching for rewritings is a small fraction of the query execution time and hence a worthwhile investment. As we confirm experimentally, the performance of the rewriting search is due to our optimized PACB$^{\mathcal{OPT}}$ algorithm, shown to outperform standard PACB by up to 40×.

4.14 Conclusion

We have shown that multi-store architectures have the potential to significantly speed up query evaluation by materializing views in the systems most suited to expected workload operations, even when these views are distributed across stores and data models. ESTOCADA supports this functionality by a local-as-view approach whose immediate benefit is flexibility since it requires no work when the underlying data storage changes. In our experiments, we achieve performance gains by simply placing the materialized views according to a few common-sense guidelines (e.g., place large unstructured text collections in a store with good full-text indexing support, and place inherently nested data in JSON document stores).

4.15 Acknowledgement

This chapter contains material from "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue" by Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis, which appeared in Proceedings of the 2019 International Conference

on Management of Data (SIGMOD 2019). The dissertation author was the primary investigator of this paper.

This chapter contains material from “ESTOCADA: Towards Scalable Polystore Systems” by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, Moustafa Latrache, Ioana Manolescu, and Yifei Yang, which appeared in Proceedings of the VLDB Endowment (PVLDB 2020). The dissertation author was the primary investigator of this paper.

Chapter 5

HADAD: Extending Benefits of Semantic Optimization to Hybrid Relational and Linear Algebra Computation

5.1 Introduction

This chapter presents HADAD, which capitalizes on ESTOCADA framework previously introduced in [31] (Chapter 4) for rewriting queries across many data models, using materialized views in a hybrid setting that does not include the LA model. The novelty of HADAD is to extend the benefits of semantic rewriting and views optimizations under integrity constraints to pure LA and hybrid RA-LA computations, which are crucial for ML-hybrid workloads.

This approach makes it very easy to extend HADAD's semantic knowledge of LA operations, RA-LA, or LA rewrite rules by simply declaring appropriate constraints, with no need to change HADAD code. Moreover, as we will show, constraints are sufficiently expressive to declare (and thus allow HADAD to exploit) more properties of LA operations than previous work could consider.

Chapter Outline. The rest of this chapter is organized as follows: Section 5.2 highlights HADAD's optimizations that go beyond the state-of-the-art based on *real-world* scenarios. In Section 5.3, we formalize the query optimization problem in the context of pure LA and hybrid RA-LA settings. Section 5.4 provides an end-to-end overview of our approach. We present our novel reduction of the rewriting problem, focusing on LA computation, into one that can be solved by existing techniques from the relational setting (Chapter 4) in Section 5.5. Section 5.6 describes our extension to the query rewriting engine, integrating two different cost models to help prune out inefficient rewritings as soon as they are enumerated. We formalize our guarantees in Section 5.8 and present the experiments in Section 5.9. We conclude in Section 5.11.

5.2 HADAD Optimizations

We highlight below examples of performance-enhancing opportunities that are exploited by HADAD and not being addressed by LA-oriented and cross RA-LA existing solutions.

5.2.1 LA Pipeline Optimization

Example 5.2.1. *Consider the Ordinary Least Squares Regression (OLS) pipeline:* $(X^TX)^{-1}(X^Ty)$*, where* X *is a square matrix of size 10K*$\times$*10K and* y *is a vector of size 10K*$\times$*1. Suppose available a materialized view* $V = X^{-1}$*.*

HADAD *rewrites the pipeline to* $(V(V^T(X^Ty))$*, by exploiting the LA properties* $(CD)^{-1} = D^{-1}C^{-1}$*,* $(CD)E = C(DE)$ *and* $(D^T)^{-1} = (D^{-1})^T$ *as well as the view* V*. The rewriting is more efficient than the original pipeline since it avoids computing the expensive inverse operation. Moreover, it optimizes the matrix chain multiplication order to minimize the intermediate result size. This leads to a 70*$\times$ *speedup on R. Current popular LA-oriented systems [17, 16, 42, 25, 112] are not capable of exploiting such rewrites due to the lack of systematic exploration of standard LA properties and views.*

5.2.2 Hybrid RA-LA Optimization

Cross RA-LA platforms such as MorpheusR [52], SparkSQL [35] and others [97, 91] can greatly benefit from HADAD's cross-model optimizations, which can find rewrites that they miss.

Example 5.2.2 (Factorization of LA Operations over Joins)**.** *Morpheus R [52] implements a powerful optimization that factorizes an LA operation on a matrix* $\boldsymbol{M}$ *obtained by joining tables* $\boldsymbol{R}$ *and* $\boldsymbol{S}$ *and casting the join result as a matrix. Factorization pushes the LA operation on* $\boldsymbol{M}$ *to operate on* $\boldsymbol{R}$ *and* $\boldsymbol{S}$*, cast as matrices.*

Consider a specific instantiation of factorization: $\texttt{colSums}(\boldsymbol{M}N)$*, where matrix* $\boldsymbol{M}$ *has size 20M*$\times$ *120 and N has size 120*$\times$*100; both matrices are dense. The* `colSums` *operation sums up the elements in each column, returning the vector of these sums (the operation is common in ML algorithms such as K-means clustering [107]). On this pipeline, MorpheusR applies a left matrix multiplication factorization rule to push the multiplication by N down to* $\boldsymbol{R}$ *and* $\boldsymbol{S}$*. The size of* $\boldsymbol{M}N$ *intermediate result is 20M*$\times$*100. Finally,* `colSums` *is applied to the intermediate result, reducing to a 1*$\times$*100 vector.*

HADAD *can help MorpheusR do better by pushing the* `colSums` *operator to* $\boldsymbol{R}$ *and* $\boldsymbol{S}$ *(instead of the multiplication with N), then concatenating the resulting vectors. This leads to much smaller intermediate results, since the combined size of vectors* $\texttt{colSums}(\boldsymbol{R})$ *and* $\texttt{colSums}(\boldsymbol{S})$ *is only 1*$\times$*120.*

To this end, HADAD *rewrites the pipeline to* $\texttt{colSums}(\boldsymbol{M})N$ *by exploiting the property* $\texttt{colSums}(AB) = \texttt{colSums}(A)B$ *and applying its cost estimator, which favors rewritings with a small intermediate result size. Evaluating this* HADAD*-produced rewriting, MorpheusR's multiplication pushdown rule no longer applies, while the* `colSums` *pushdown rule is now enabled, leading to 125*$\times$ *speedup.*

Example 5.2.3 (Cross-Model Optimizations)**.** *Consider another hybrid example on a Twitter dataset [24]. The JSON dataset contains tweet ids, extended tweets, entities including hashtags,*

filter-level, media, URL, tweet text, etc. We implemented it on SparkSQL (with SystemML [42]). In the preprocessing stage, our SparkSQL query constructs a tweet-hashtag filter-level matrix $\boldsymbol{N}$ *of size* $2M \times 1000$*, for all tweets posted from "USA" mentioning "covid", where rows are tweets, columns are hashtags, and values are filter-levels*[1].

$\boldsymbol{N}$ *is then loaded into SystemML, where rows with filter-level less than four are selected. The result undergoes an Alternating Least Square (ALS) [114] computation. A core building block of the ALS computation is the LA pipeline* $(uv^T - \boldsymbol{N})v$*. In our example,* u *is a tweet feature vector (of size* $2M \times 1$*) and* v *is a hashtag feature vector (of size* 1000×1*).*

We have two materialized views available: V_1 *stores the tweet id and text as a* text data source in Solr*, and* V_2 *stores tweet id, hashtag id, and filter-level for all tweets posted from "USA", and is materialized on disk as CSV file. The rewriting* modifies the preprocessing *of* $\boldsymbol{N}$ *by introducing* V_1 *and* V_2*; it also* pushes the filter-level selection *from the LA pipeline into the preprocessing stage. To this end, it rewrites* $(uv^T - \boldsymbol{N})v$ *to* $uv^T v - \boldsymbol{N}v$*, which is more efficient for two reasons. First,* $\boldsymbol{N}$ *is ultra sparse (0.00018% non-zero), which renders the computation of* $\boldsymbol{N}v$ *extremely efficient. Second, SystemML evaluates the chain* $uv^T v$ *efficiently, computing* $v^T v$ *first, which results in a scalar, instead of computing* uv^T*, which results in a dense matrix of size* $2M \times 1000$ *(*HADAD*'s cost model realizes this). Without the rewriting help from* HADAD*, SystemML is unable to exploit its own efficient operations for lack of awareness of the distributivity property of vector multiplication over matrix addition,* $Av + Bv = (A + B)v$*. The rewriting achieves* $14\times$ *speedup.*

HADAD detects and applies all the above-mentioned optimizations combined. It captures RA-, LA-, and cross-model optimizations precisely because it reduces all rewrites to a single setting in which they can interact and synergize: *relational rewrites under integrity constraints*.

[1] Matrix N is represented in MatrixMarket Format (MTX) since it is sparse.

5.3 Problem Statement

We consider a set of *value domains* $\mathcal{D}_i$, e.g., $\mathcal{D}_1$ denotes integers, $\mathcal{D}_2$ denotes real numbers, and two basic data types: *relations (sets of tuples)* and *matrices* (bi-dimensional arrays). Any attribute in a tuple or cell in a matrix is a value from some $\mathcal{D}_i$. We assume each tuple would become a matrix line, in some order that is unknown, unless the relation was explicitly sorted before the conversion. In other words, resulting relations are cast as matrices.

We consider a hybrid RA-LA language $\mathcal{L}$, comprising a set R_{ops} of (unary or binary) *RA operators*; concretely, R_{ops} comprises the standard relational *selection, projection, and join*. We also consider a set L_{ops} of *LA operators*, comprising: unary (e.g., inversion and transposition) and binary (e.g., matrix product) operators. The full set L_{ops} of LA operations we support is detailed in Section 5.5.1. A *hybrid expression* in $\mathcal{L}$ is defined as follows:

- any value from a domain $\mathcal{D}_i$, any matrix, and any relation, is an expression;
- (RA operators): given some expressions E, E', $ro_1(E)$ is also an expression, where $ro_1 \in R_{ops}$ is a unary relational operator, and E's type matches ro_1's expected input type. The same holds for $ro_2(E, E')$, where $ro_2 \in R_{ops}$ is a binary relational operator (i.e., the join);
- (LA operators): given some expressions E, E' that are either numeric matrices or numbers (which can be seen as 1×1 matrices), and some real number r, the following are also expressions: $lo_1(E)$ where $lo_1 \in L_{ops}$ is a unary operator, and $lo_2(E, E')$ where $lo_2 \in L_{ops}$ is a binary operator (again, provided that E, E' match the expected input types of the operators).

Clearly, an important set of *equivalence rules* that we focus on in this work are LA equivalence rules (i.e., properties of LA operations). These equivalences lead to *alternative evaluation strategies* for each expression. Further, we assume given a (possibly empty) set of *materialized views* $\mathcal{V} \in \mathcal{L}$, which have been previously computed over some inputs (matrices

and/or relations), and whose results are directly available (e.g., as a file on disk). Detecting when a materialized view can be used instead of evaluating (part of) an expression is another important source of alternative evaluation strategies.

Given an expression E and a *cost model* that assigns a cost (a real number) to an expression, we consider the problem of **identifying the most efficient rewrite** derived from E by: (*i*) exploiting integrity constraints, (*ii*) exploiting equivalence rules, and/or (*iii*) replacing part of an expression with a scan of a materialized view equivalent to that expression.

Below, we detail our approach, the equivalence rules we capture, and two alternative cost models we devised for this setting. Importantly, our solution (based on a *relational encoding with integrity constraints*) capitalizes on ESTOCADA framework previously introduced in (Chapter 4) [31], where it was used to *rewrite queries using materialized views in a polystore setting*, where the data, views and queries cover a variety of data models (relational, JSON, XML, etc.). Those queries can be expressed in a combination of query languages, including SQL, JSON query languages, etc. **The ability to rewrite such queries using heterogeneous views directly and fully transfers to HADAD**: thus, instead of a relation, we could have the (tuple-structured) results of an XML or JSON query; views materialized by joining an XML document with a JSON one and a relational database could also be used. The novelty of HADAD is to **extend the benefits of rewriting and view-based optimization to LA computations, crucial for ML workloads**. In Section 5.5, we focus on capturing matrix data and LA computations in the relational framework, along with relational data naturally; this enables our novel, holistic optimization.

5.4 HADAD Overview

We outline here our approach as an extension to ESTOCADA [31] (see Chapter4) for solving the rewriting problem introduced in Section 5.3.

RA-LA Hybrid Expressions and Views. A hybrid RA-LA expression (whether asked

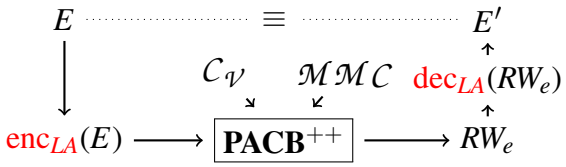


Figure 5.1: HADAD reduction outline.

as a query or describing a materialized view) can be purely relational (RA), in which case we assume it is specified as a conjunctive query [50]. Other expressions are purely LA ones; we assume that they are defined in a dedicated LA language such as R [17], DML [42], etc. , using LA operators from our set L_{ops} (see Section 5.5.1), commonly used in real-world ML workloads. Finally, a hybrid expression can combine RA and LA, e.g., an RA expression (resulting in a relation) is treated as a matrix input by an LA operator.

Our approach is based on a reduction to a relational model. Below, we focus on showing how to bring LA components under a relational form (the RA part of each expression is already in the target formalism, see Chapter4).

Encoding of LA Operations into the Relational Model. Let E be an LA expression (query), and $\mathcal{V}$ be a set of materialized views. We reduce the LA views-based rewriting problem to the relational rewriting problem under integrity constraints, as follows (see Figure 5.1). First, we *encode relationally* E, $\mathcal{V}$, and the set L_{ops} of LA operators. Note that the relations used in the encoding are *virtual* and *hidden*. They only serve to support query rewriting via relational techniques, as we show in Chpater 4. These virtual relations are accompanied by a set of relational *integrity constraints* $enc_{LA}(LA_{prop})$ that reflect a set LA_{prop} of LA properties of the supported L_{ops} operations. For instance, we model the *matrix addition* operation using a relation $\texttt{add}_M(M,N,R)$, denoting that R is the result of $M+N$, together with a set of constraints (see Section 5.5.2) stating that $\texttt{add}_M$ is a *functional* relation that is *commutative*, *associative*, etc. We detail our relational encoding in Section 5.5.

Reduction From LA-based to Relational Rewriting. Our reduction translates the dec-

laration of each view $V \in \mathcal{V}$ to constraint $enc_{LA}(V)$ that reflects the correspondence between V's input data and its output. Separately, E is also encoded as a relational query $enc_{LA}(E)$ over the relational encodings of L_{ops} and its matrices.

The reformulation problem is reduced to a purely relational setting (as shown before in Chapter 4) as follows. We are given a (purely LA or RA-LA hybrid) query expression $E \in \mathcal{L}$ and a set $\mathcal{V} \subseteq \mathcal{L}$ views. We encode E as the relational query $enc_{LA}(E)$, the views as the relational integrity constraints $\mathcal{C}_{\mathcal{V}} = enc_{LA}(V_1) \cup \ldots \cup enc_{LA}(V_n)$. We add as further input the set of relational constraints $enc_{LA}(LA_{prop})$ mentioned above, which specify properties of the LA operators. We call them Matrix-Model encoding Constraints, or $\mathcal{MMC}$ in short. We must find a rewriting RW_e expressed over the relational views $\mathcal{C}_{\mathcal{V}}$, such that RW_e is equivalent to $enc_{LA}(E)$ under the constraints $(\mathcal{C}_{\mathcal{V}} \bigcup \mathcal{MMC})$ and is optimal according to our cost model. Note that RW_e is a *relationally encoded rewriting* of E; a final *decoding* step is needed to obtain $E' \in \mathcal{L}$, the (LA or hybrid) rewriting of E.

The challenge in coming up with the reduction consists in designing an encoding, i.e., one in which rewritings found by (*i*) encoding relationally, (*ii*) solving the resulting relational rewriting problem, and (*iii*) decoding a resulting rewriting over the views is guaranteed to produce an equivalent expression E' (see Section 5.5).

Relational Rewriting Using Constraints. To solve the relational rewriting problem under constraints, the engine of choice is PACB [84] (previously used in ESTOCADA, see Chapter 4). We extend PACB engine ($PACB^{++}$ hereafter) to utilize the $Pruned_{prov}$ algorithm discussed in [83], which prunes inefficient rewritings during the equivalent rewritings search phase, based on simple cost models (see Section 5.6).

Decoding of the Relational Rewriting. In a similar fashion as what we have seen in Chapter 4, the selected relational reformulation RW_e found by $PACB^{++}$, a *decoding step* $dec(RW_e)$ is performed to translate RW_e into the native syntax of its respective underlying system language (e.g., R, DML, etc.).

Table 5.1: Snippet of the $\mathcal{VREM}$ schema.

Operation	Encoding	Operation	Encoding
Matrix scan	$name(M,n)$	Inversion	$\texttt{inv}_M(M,R)$
Multiplication	$\texttt{multi}_M(M,N,R)$	Scalar Multiplication	$\texttt{multi}_{MS}(s,M,R)$
Addition	$\texttt{add}_M(M,N,R)$	Cells Sum	$\texttt{sum}(M,s)$
Division	$\texttt{div}_M(M,N,R)$	Trace	$\texttt{trace}(M,s)$
Hadamard product	$\texttt{multi}_E(M,N,R)$	Row sum	$\texttt{rowSums}(M,R)$
Transposition	$\texttt{tr}(M,R)$	Colsums	$\texttt{colSums}(M,R)$

5.5 LA Reduction to the Relational Model

Our internal model is relational, and it makes prominent use of expressive integrity constraints. This framework suffices to describe the features and properties of most data models used today, notably including relational, XML, JSON, graph, etc [31, 32] (see Chapter 4).

Going beyond, in this section, we present a novel way to *reason relationally about LA operations* by treating them as un-interpreted functions with black-box semantics and adding *constraints that capture their important properties*. First, we give an overview of a wide range of LA operations that we consider (Section 5.5.1). Then, in Section 5.5.2, we show how matrices and their operations can be *encoded* using a set of *virtual* relations, part of a schema we call $\mathcal{VREM}$ (for *Virtual Relational Encoding of Matrices*), together with the integrity constraints $\mathcal{MMC}$ that capture the properties of these operations. Section 5.5.3 exemplifies relational rewritings obtained via our reduction.

5.5.1 Supported Matrix Algebra

We consider a wide range of matrix operations [98, 37], which are common in real-world ML algorithms [9]: element-wise multiplication ($\texttt{multi}_E$), matrix-scalar multiplication ($\texttt{multi}_{MS}$), matrix multiplication ($\texttt{multi}_M$), addition ($\texttt{add}_M$), division ($\texttt{div}_M$), transposition (`tr`), inversion ($\texttt{inv}_M$), determinant (`det`), trace (`trace`), diagonal (`diag`), exponential (`exp`),

adjoints (adj), direct sum ($\texttt{sum}_D$), direct product ($\texttt{product}_D$), summation (sum), rows/columns summation (rowSums, colSums, respectively), QR (QR), Cholesky (CHO), LU (LU), and pivoted LU (LUP) decompositions. The full list of supported operations appear in Appendix B.1.

5.5.2 VREM Schema and Relational Encoding

To model LA operations on the $\mathcal{VREM}$ relational schema (part of which appears in Table 5.1), we also rely on a set of integrity constraints $\mathcal{MMC}$, which are encoded using relations in $\mathcal{VREM}$. We detail the encoding below.

Base Matrices and Dimensionality Modeling. We denote by $\mathrm{M}_{k\times z}(\mathcal{D})$ a matrix of k rows and z columns, whose values come from a domain $\mathcal{D}$, e.g., the domain of real numbers $\mathbb{R}$. For brevity we just use $\mathrm{M}_{k\times z}$. We define a virtual relation $name(M,n) \in \mathcal{VREM}$ attaching a unique ID M to any matrix identified by a name denoted n (which may be e.g., of the form "/M.csv"). This relation (shown at the top left in Table 5.1) is accompanied by an EGD key constraint $I_{name} \in \mathcal{MMC}_m$, where $\mathcal{MMC}_m \subset \mathcal{MMC}$, stating that *two matrices with the same name n have the same ID*:

$$I_{name}\colon \forall M \forall N\ name(M,n) \wedge name(N,n) \rightarrow M = N$$

Note that the matrix ID in *name* relation (and all the other virtual relations used in our encoding) are not IDs of individual matrix objects: rather, each identifies an *equivalence class* (induced by value equality) of expressions. That is, two expressions are assigned the same ID iff they yield value-based-equal matrices. In Table 5.1, we use M and N to denote input matrices' IDs, R for the resulting matrix ID, and s for scalar input and output.

Accessing a particular cell in matirx M can be modeled using a $cell(M,i,j,v)$ relation, where i is the row, j is the column and v is the cell value. To state that there is no cell that has two distinct values, we need to introduce the following EGD:

$$I_{cell}\colon \forall M \forall i \forall j \forall v_1 \forall v_2\ cell(M,i,j,v_1) \wedge cell(M,i,j,v_2) \rightarrow v_1 = v_2$$

Currently, we do not support encoding *matrix slicing operator*, which involves selecting certain rows and columns of a matrix, and forming a new matrix, possibly in a different dimension from the original. We leave this to future work.

The dimensions of a matrix are captured by a $size(M,k,z)$ relation, where k and z are the number of rows, resp. columns and M is an ID. An EGD constraint $I_{size} \in \mathcal{MMC}_m$ holds on the *size* relation, stating that the ID determines the dimensions:

$$I_{size}\colon \forall M \forall k_1 \forall z_1 \forall k_2 \forall z_2\ size(M,k_1,z_1) \wedge size(M,k_2,z_2) \rightarrow k_1 = k_2 \wedge z_1 = z_2$$

The identity and zero matrices are captured by $Zero(O)$ and $Identity(I)$ relations, where O and I denote their IDs, respectively. They are accompanied by EGD constraints I_{iden}, $I_{zero} \in \mathcal{MMC}_m$, stating that zero matrices with the same sizes have the same IDs, and this also applies for identity matrices with the same size:

$$I_{zero}\colon \forall O_1 \forall O_2 \forall k \forall z\ Zero(O_1) \wedge size(O_1,k,z) \wedge Zero(O_2) \wedge size(O_2,k,z) \rightarrow O_1 = O_2$$

$$I_{iden}\colon \forall I_1 \forall I_2\ Identity(I_1) \wedge size(I_1,k,k) \wedge Identity(I_2) \wedge size(I_2,k,k) \rightarrow I_1 = I_2$$

Encoding Matrix Algebra Expressions. LA operations are encoded into dedicated relations, as shown in Table 5.1. We now illustrate the encoding of an LA expression on the $\mathcal{VREM}$ schema. The full list of encoding relations appears in Appendix B.1.

Example 5.5.1 (Encoding LA Expression Using $\mathcal{VREM}$ Schema)**.** *Consider the LA expression* E*:* $((MN)^T)$*, where the two matrices* $M_{100\times1}$ *and* $N_{1\times10}$ *are stored as "M.csv" and "N.csv", respectively. The encoding function* $enc_{LA}(E)$ *takes as argument the LA expression* E *and returns a conjunctive query whose:* (i) *body is the relational encoding of* E *using* $\mathcal{VREM}$ *(see below), and* (ii) *head has one distinguished variable, denoting the equivalence class of the result. For instance:*

$enc(((MN)^T)$ =

Let $enc(MN)$ =

Let $enc(M) = Q_0(M)$:- $name(M,$"$M.csv$"$)$;

$enc(N) = Q_1(N)$:- $name(N,$"$N.csv$"$)$;

R_1 = *freshId()*

in

$Q_2(R_1)$:- $\texttt{multi}_M(M,N,R_1), Q_0(M), Q_1(N)$;

R_2 = *freshId()*

in

$Q(R_2)$:- $\texttt{tr}(R_1,R_2), Q_2(R_1)$;

In the above, nesting is dictated by the syntax of E. From the inner (most indented) to the outer, we first encode M and N as small queries using the name relation, then their product (to whom we assign the newly created identifier R_1 *), using the* $multi_M$ *relation and encoding the relationship between this product and its inputs in the definition of* $Q_2(R_1)$*. Next, we create a fresh ID* R_2 *used to encode the full E (the transposed of* Q_2*) via relation* $\texttt{tr}$*, in* $Q(R_2)$*. For brevity, we omit the matrices' size relations in this example and hereafter. Unfolding* $Q_2(R_1)$ *in the body of Q yields:*

$$Q(R_2) \text{ :- } \texttt{tr}(R_1,R_2), \texttt{multi}_M(M,N,R_1), Q(R_2)\ Q_0(M), Q_1(N);$$

Now, by unfolding Q_0 *and* Q_1 *in Q, we obtain the final encoding of* $((MN)^T)$ *as a conjunctive query Q:*

$$Q(R_2) :- \texttt{tr}(R_1,R_2), \texttt{multi}_M(M,N,R_1), Q(R_2), name(M,\text{"}M.csv\text{"}), name(N,\text{"}N.csv\text{"});$$

Encoding LA Properties as Integrity Constraints. Figure 5.2 shows some of the constraints $\mathcal{MMC}_{LA_{prop}} \subset \mathcal{MMC}$, which capture textbook LA properties [98, 37] of our LA operations (Section 5.5.1). The TGDs (5.1), (5.2) and (5.3) state that matrix addition is commutative, matrix transposition is distributive with respect to addition, and the transposition of the

$$\forall M \forall N \forall R \; \texttt{add}_M(M,N,R) \rightarrow \texttt{add}_M(N,M,R) \tag{5.1}$$

$$\forall M \forall N \forall R_1 \forall R_2 \; \texttt{add}_M(M,N,R_1) \wedge \texttt{tr}(R_1,R_2) \rightarrow$$
$$\exists R_3 \exists R_4 \; \texttt{tr}(M,R_3) \wedge \texttt{tr}(N,R_4) \wedge \texttt{add}_M(R_3,R_4,R_2) \tag{5.2}$$

$$\forall M \forall R_1 \forall R_2 \; \texttt{inv}_M(M,R_1) \wedge \texttt{tr}(R_1,R_2) \rightarrow$$
$$\exists R_3 \; \texttt{tr}(M,R_3) \wedge \texttt{inv}_M(R_3,R_2) \tag{5.3}$$

Figure 5.2: Snippet of $\mathcal{MMC}_{LA_{prop}}$ constraints

$$\forall M \forall N \forall R_1 \forall R_2 \forall R_3 \forall R_4$$
$$name(M, \text{“}M.csv\text{”}) \wedge \; name(N, \text{“}N.csv\text{”}) \wedge$$
$$\texttt{tr}(N,R_1) \wedge \; \texttt{tr}(M,R_2) \wedge$$
$$\texttt{inv}_M(R_2,R_3) \wedge \; \texttt{add}_M(R_1,R_3,R_4) \rightarrow name(R_4, \text{“}V.csv\text{”})$$

Figure 5.3: Relational encoding of view V

inverse of matrix M is equivalent to the inverse of the transposition of M, respectively. We also express that the *virtual relations are functional* by using EGD key constraints. For example, the following $I_{multi_M} \in \mathcal{MMC}_{LA_{prop}}$ constraint states that $multi_M$ is functional, that is the products of pairwise equal matrices are equal.

$$I_{multi_M} : \forall M \forall N \forall R_1 \forall R_2$$
$$\texttt{multi}_M(M,N,R_1) \wedge \texttt{multi}_M(M,N,R_2) \rightarrow R_1 = R_2$$

Other properties [37, 98] of the LA operations we consider are similarly encoded (See Appendix B.3).

Encoding LA Views as Constraints. We translate each view definition V (defined in LA language) into relational constraints $enc_{LA}(V) \in \mathcal{C}_{\mathcal{V}}$, where $\mathcal{C}_{\mathcal{V}}$ is the set of relational constraints used to capture the views $\mathcal{V}$. These constraints show how the view's inputs are related to its output over the $\mathcal{VREM}$ schema. Figure 5.3 illustrates the encoding as a TGD constraint of the view $V : (N)^T + (M^T)^{-1}$ stored in a file "V.csv" and computed using matrices N and M (e.g., stored as "N.csv" and "M.csv", respectively).

Encoding Matrix Decompositions. Matrix decompositions play a crucial role in many LA computations. For instance, for every symmetric positive definite matrix M there exists a unique Cholesky Decomposition (CD) of the form $M = LL^T$, where L is a lower triangular matrix. We model CD, as well as other well-known decompositions (LU, QR, and Pivoted LU or PLU) as a set of virtual relations $\mathcal{VREM}_{dec}$, which we add to $\mathcal{VREM}$. For instance, for CD, we associate a relation $\texttt{CHO}(M,L)$, which denotes that L is the output of the CD decomposition for a given matrix M whose ID is M. CHO is a functional relation, meaning every symmetric positive definite matrix has a unique CD decomposition. This functional aspect is captured by an EGD, conceptually similar to the constraint I_{multi_M} (Section 5.5.2). The property $M = LL^T$ is captured as a TGD constraint $I_{cho} \in \mathcal{MMC}_{LA_{prop}}$:

$$I_{cho} : \forall M\ type(M,\text{“}S\text{”}) \rightarrow \exists\, L_1 \exists L_2\ cho(M,L_1) \wedge$$
$$type(L_1,\text{“}L\text{”}) \wedge tr(L_1,L_2) \wedge multi_M(L_1,L_2,M) \tag{5.4}$$

The atom $type(M,\text{“S”})$ indicates the type of matrix M, where the constant “S” denotes a matrix that is *symmetric positive definite*; similarly, $type(L_1,\text{“L”})$ denotes that the matrix L_1 is a lower triangular matrix. For each base matrix, its type (if available) (e.g., symmetric, upper triangular, etc.) is specified as TGD constraint. For example, we state that a certain matrix M (and any other matrix value-equal to M) is symmetric positive definite as follows:

$$\forall M\ name(M,\text{“}M.csv\text{”}) \rightarrow type(M,\text{“}S\text{”}) \tag{5.5}$$

Example 5.5.2 (Cholesky Decomposition Rewriting)**.** *Consider a view V=$N + LL^T$, where $L = cho(M)$ and M is a symmetric positive definite matrix encoded as in (5.5). Let E be the LA expression $M + N$. The reader realizes easily that V can be used to answer E directly, thanks to the specific property of the CD decomposition (5.4), and since $M + N = N + M$, which is encoded*

in (5.1). However, at the syntactic level, V and E are very dissimilar. Knowledge of (5.1) and (5.4) and the ability to reason about them is crucial in order to efficiently answer E based on V.

The output matrix of CD decomposition is a lower triangular matrix L, which is not necessarily a symmetric positive definite matrix, meaning that CD decomposition can not be applied again on L. For other decompositions, such as $QR(M) = [Q,R]$ decomposition, where M is a real square matrix, Q is an orthogonal matrix [98] and R is an upper triangular matrix, there exists a QR decomposition for the orthogonal matrix Q such that $QR(Q) = [Q,I]$, where I is an identity matrix and $QR(R) = [I,R]$. We say the *fixed point* of the QR decomposition is $QR(I) = [I,I]$. These properties of the Q decompositions are captured with the following constraints, which are part of $\mathcal{MMC}_{LAprop}$:

$$\forall M \forall n \forall k \; name(M,n) \wedge size(M,k,k) \rightarrow \; \exists Q \exists R$$

$$QR(M,Q,R) \wedge type(Q,\text{“}O\text{”}) \wedge type(R,\text{“}U\text{”}) \wedge multi_M(Q,R,M) \tag{5.6}$$

$$\forall Q \; type(Q,\text{“}O\text{”}) \rightarrow \exists I \; QR(Q,Q,I) \wedge identity(I) \wedge multi_M(Q,I,Q) \tag{5.7}$$

$$\forall R \; type(R,\text{“}U\text{”}) \rightarrow \exists I \; QR(R,I,R) \wedge identity(I) \wedge multi_M(I,R,R) \tag{5.8}$$

$$\forall I \; identity(I) \rightarrow QR(I,I,I) \tag{5.9}$$

Known LA properties of the other matrix decompositions (LU and PLU) are similarly encoded in Appendix B.3.

Encoding LA-Oriented System Specific Rewrite Rules. Most LA-oriented systems [17, 16] execute an incoming LA expression *as-is*, that is: run operations in a sequence, whose order is dictated by the expression syntax. Such systems do not exploit basic LA properties, e.g., reordering a chain of multiplied matrices in order to reduce the intermediate size. SystemML [42] is the only system that models *some* LA properties as static rewrite rules. It also comprises a set of *rewrite rules* which modify the given expressions to avoid large intermediates for aggregation and statistical operations such as `rowSums`(M), `sum`(M), etc. For example, SystemML uses the

following rule to rewrite $\texttt{sum}(MN)$ (summing all cells in the matrix product), where $\odot$ is a matrix element-wise multiplication, to avoid actually computing MN and materializing it.

$$\texttt{sum}(MN) = \texttt{sum}(\texttt{colSums}(M)^T \odot \texttt{rowSums}(N)) \quad (i)$$

However, the performance benefits of rewriting depend on the rewriting power (or, in other words, on *how much the system understands the semantics of the incoming expression*), as the following example shows.

Example 5.5.3. *Consider an LA expression* E=$((M^T)^k(M+N)^T)$*, where* M *and* N *are square matrixes, and expression* E'=$\texttt{sum}(E)$*, which computes the sum of all cells in* E*.* E' *can be rewritten to*

$$RW_1 : \texttt{sum}(\texttt{colSums}(M+N)^T \odot \texttt{rowSums}(M^k))$$

Failure to exploit the LA properties $M^T N^T = (NM)^T$, $(M^n)^T = (M^T)^n$ *and* $\texttt{sum}((MN)^T) = \texttt{sum}(MN)$ *together prevents from finding the rewriting* RW_1.
E' *admits the alternative rewriting:*

$$RW_2\text{: } \texttt{sum}((\texttt{colSums}((M^T)^k))^T \odot (\texttt{colSums}(M+N)^T))$$

which can be obtained by directly applying the rewrite rule (i) *given previously and the LA property* $\texttt{rowSums}(M^T)$=$\texttt{colSums}(M)^T$*. However,* RW_2 *creates more intermediate results than* RW_1.

To fully exploit the potential of rewrite rules (for statistical or aggregation operations), they should be accompanied by sufficient knowledge of, and reasoning on, known properties of LA operations.

To bring such fruitful optimization to other LA-oriented systems lacking support of such rewrite rules, we have incorporated SystemML's rewrite rules into our framework, encoding

them as a set of integrity constraints over the virtual relations in the schema $\mathcal{VREM}$, denoted $\mathcal{MMC}_{StatAgg} \subset \mathcal{MMC}$. Thus, these rewrite rules can be exploited together with other LA properties. For instance, the rewrite rule (i) is modeled by the following constraint:

$$I_{sum} : \forall M \forall N \forall R \; \texttt{multi}_M(M,N,R) \wedge \texttt{sum}(R,s) \rightarrow$$
$$\exists R_1 \exists R_2 \exists R_3 \exists R_4 \texttt{colSums}(M,R_1) \wedge \texttt{tr}(R_1,R_2)$$
$$\wedge \texttt{rowSums}(N,R_3) \wedge \texttt{multi}_E(R_2,R_3,R_4) \wedge \texttt{sum}(R_4,s)$$

We refer the reader to Appendix B.4 for a list of supported SystemML's rewrite rules and their encoding into integrity constraints.

5.5.3 LA Relational Rewriting Using Constraints

With the set of views constraints $\mathcal{C}_\mathcal{V}$ and $\mathcal{MMC} = \mathcal{MMC}_m \cup \mathcal{MMC}_{LA_{prop}} \cup \mathcal{MMC}_{StatAgg}$, we rely on *PACB*$^{++}$ to rewrite a given expression under integrity constraints. We exemplify this below, and detail *PACB*$^{++}$'s inner workings in Section 5.6.

Example 5.5.4 (LA View-based Rewriting)**.** *The view V shown in Figure 5.3 can be used to* fully *rewrite (return the answer for) the pipeline* $Q_p = (M^{-1}+N)^T$ *by exploiting the TGDs (5.1), (5.2) and (5.3) listed in Figure 5.2, which describe the following three LA properties, denoted* LA_{prop_1}*:* $M+N = M+N$*;* $((M+N))^T = (M)^T + (N)^T$ *and* $((M)^{-1})^T = ((M)^T)^{-1}$*. The relational rewriting* RW_0 *of* Q_p *using the view V is* $RW_0(R_4)$*:- name*$(R_4,$"*V.csv*"*). In this example,* RW_0 *is the only views-based rewriting of* Q_P*. However,* five *other rewritings exist (shown in Figure 5.4 and their relational form in Figure 5.5), which reorder its operations just by exploiting the set* LA_{prop_1} *of LA properties.*

Rewritings have different evaluation costs. We discuss next how we estimate which among these alternatives (including evaluating Q_p directly) is likely the most efficient.

$$RW_1 : (M^{-1})^T + N^T \qquad RW_2 : (M^T)^{-1} + N^T$$
$$RW_3 : N^T + (M^{-1})^T \qquad RW_4 : N^T + (M^T)^{-1}$$
$$RW_5 : (N + M^{-1})^T$$

Figure 5.4: Equivalent rewritings of Q_p.

$$RW_1(R_4) : -name(M, \text{“}M.csv\text{”}), name(N, \text{“}N.csv\text{”}),$$
$$\texttt{tr}(N, R_1), \texttt{inv}_M(M, R_2), tr(R_2, R_3), \texttt{add}_M(R_3, R_1, R_4)$$
$$RW_2(R_4) : -name(M, \text{“}M.csv\text{”}), name(N, \text{“}N.csv\text{”}),$$
$$\texttt{tr}(N, R_1), \texttt{tr}(M, R_2), \texttt{inv}_M(R_2, R_3), \texttt{add}_M(R_3, R_1, R_4)$$
$$RW_3(R_4) : -name(M, \text{“}M.csv\text{”}), name(N, \text{“}N.csv\text{”}),$$
$$\texttt{tr}(N, R_1), \texttt{inv}_M(M, R_2), tr(R_2, R_3), \texttt{add}_M(R_1, R_3, R_4)$$
$$RW_4(R_4) : -name(M, \text{“}M.csv\text{”}), name(N, \text{“}N.csv\text{”}),$$
$$\texttt{tr}(N, R_1), \texttt{tr}(M, R_2), \texttt{inv}_M(R_2, R_3), \texttt{add}_M(R_1, R_3, R_4)$$
$$RW_5(R_4) : -name(M, \text{“}M.csv\text{”}), name(N, \text{“}N.csv\text{”}),$$
$$\texttt{inv}_M(M, R_1), \texttt{add}_M(N, R_1, R_2), \texttt{tr}(R_2, R_4)$$

Figure 5.5: Relational equivalent rewritings of Q_p.

5.6 Choice of an Efficient Rewriting

We introduce our cost model in Section 5.7, which can take two different sparsity estimators (Section 5.7.1). Then, we detail our extension to the PACB rewriting engine based on the $Prune_{prov}$ algorithm (Section 5.7.2) to prune out inefficient rewritings.

5.7 Cost Model

We estimate the cost of an expression E, denoted $\gamma(E)$, as the sum of the intermediate result sizes if one evaluates E “as stated”, in the syntactic order dictated by the expression. Real-world matrices may be *dense* (most or all elements are non-zero) or *sparse* (a majority of zero elements). The latter admits more economical representations that do not store zero elements,

which our intermediate result size measure excludes. To estimate the number of non-zeros (*nnz*, in short), we incorporated two different sparsity estimators from the literature (discussed in Section 5.7.1) into our framework.

Example 5.7.1. *Consider* $E_1 = (MN)M$ *and* $E_2 = M(NM)$*, where we assume the matrices* $M_{50K\times100}$ *and* $N_{100\times50K}$ *are dense. The total cost of* E_1 *is* $\gamma(E_1) = 50K \times 50K$ *and* $\gamma(E_2) = 100 \times 100$.

5.7.1 LA-based Sparsity Estimators

We outline below two existing *sparsity estimators* [137, 42] that we have incorporated into our framework to estimate *nnz*[2].

Naïve Metadata Estimator. The naïve metadata estimator [43, 137] derives the sparsity of the output of LA expression solely from the base matrices' sparsity. This incurs no runtime overhead since metadata about the base matrices, including the *nnz*, columns and rows are available before runtime in a specific metadata file. The most common estimator is the worst-case estimator [43, 137, 44], which aims to provide an upper bound for worst-case memory estimates. Figure 5.6 illustrates the naïve worst-case sparsity estimation scheme that we use[3]. S_E denotes the estimated sparsity of an input LA expression. We define *sparsity* = *nnz/size*. We note that we assign "zero" as the sparsity of the scalar output (e.g., S_E[trace(D)] = 0, S_E[sum(M)] = 0, S_E[det(D)] = 0, etc.). The operator $\odot$ denotes a matrix-element wise multiplication. For complex operations such as inverse, we assume the sparsity of the output matrix is the same as the sparsity of the input matrix[4].

Matrix Non-zero Count (MNC) Estimator. The MNC estimator [128] exploits matrix structural properties such as single non-zero per row or columns with varying sparsity for efficient,

[2]Solving the problem of sparsity estimation is beyond the scope of this work.

[3]The figure illustrates the sparsity estimation for LA operators used in our experiments (Section 5.9)

[4]Most of the existing sparsity estimators focus on basic matrix operations such as matrix products, transportation, element-wise multiply, etc. To the best of our knowledge, we are not aware of existing work on estimating the sparsity of matrices resulting from complex operators (e.g., inverse)

$S_E[\boldsymbol{cM}] = S_E[M]$
where $\boldsymbol{c}$ is a non -zero scalar
$S_E[M^T] = S_E[M]$
$S_E[M \odot N] = min(S_E[M], S_E[N])$
$S_E[M + N] = min(\mathbf{1}, (S_E[M] + S_E[N]))$
$S_E[M - N] = min(\mathbf{1}, (S_E[M] + S_E[N]))$
$S_E[M/N] = min(\mathbf{1}, (S_E[M] + (1 - S_E[N])))$
$S_E[MN] = min(\mathbf{1}, (S_E[M] * \mathbf{n}) * min(\mathbf{1}, (S_E[N] * \mathbf{n})$
where $\boldsymbol{n}$ is the common dimension of M and N
$S_E[rowSums(M)] = min\ (\mathbf{1}, \mathbf{m} * S_E[M])$
where $\boldsymbol{m}$ is the number of rows of M
$S_E[colSums(M)] = min\ (\mathbf{1}, \mathbf{n} * S_E[M])$
where $\boldsymbol{n}$ is the number of columns of M

Figure 5.6: Naïve sparsity estimation scheme.

accurate, and general sparsity estimation. It relies on count-based histograms that exploit these properties. We have also adopted this framework into our approach, and compute histograms about the *base* matrices offline. However, the MNC framework still needs to derive and construct histograms for *intermediate results* online (during rewriting cost estimation). We study this overhead in our experiments (Section 5.9). MNC also lacks the support of estimating the sparsity of complex matrix operations. For this reason, we made the same assumptions discussed in the naïve metadata estimator.

5.7.2 Pruning Rewritings: *PACB*$^{++}$

We extended the PACB rewriting engine with the $Prune_{prov}$ algorithm discussed in [83, 84], to eliminate inefficient rewritings during the rewriting search phase. The naïve PACB algorithm generates all minimal (by join count) rewritings before choosing a *minimum-cost* one. While this suffices on the scenarios considered in [84, 31], the settings we obtain from our LA encoding stress-test the naïve algorithm, as commutativity, associativity, etc. blow up the space

of alternate rewritings exponentially. Scalability considerations forced us to further optimize naïve PACB to find only *minimum-cost rewritings*, aggressively pruning the others during the generation phase. In this Section, we recall below just enough of the *PACB*'s inner working to explain $Prune_{prov}$ (see Section 3.3.2 for details).

Provenance-Directed Rewritings Search. Given a source schema σ with a set of integrity constraints I, a set $\mathcal{V}$ of views defined over σ, and a conjunctive query Q over σ, the **rewriting problem** thus becomes: find every reformulation query ρ over the schema of view names $\mathcal{V}$ that is equivalent to Q under the constraints $I \cup C_{\mathcal{V}}$.

Example 5.7.2. *For instance, if* $\sigma = \{R,S\}$, $I = \emptyset$, $\tau = \{V\}$ *and we have a view V materializing the join of relations R and S,* $V(x,y)$*:-* $R(x,z) \wedge S(z,y)$*, the pair of constraints capturing V is the following:*

$$V_{IO} : \forall x \forall z \forall y \; R(x,z) \wedge S(z,y) \rightarrow V(x,y)$$
$$V_{OI} : \forall x \forall y V(x,y) \rightarrow \exists z \; R(x,z) \wedge S(z,y)$$

Given the query $Q(x,y)$*:-* $R(x,z) \wedge S(z,y)$*, PACB finds the rewriting* $RW(x,y)$*:-* $V(x,y)$*. Algorithmically, this is achieved by:*

(i) chasing Q with the constraints $I \cup C_{\mathcal{V}}^{IO}$*, where* $C_{\mathcal{V}}^{IO} = \{V_{IO} \mid V \in \mathcal{V}\}$*; intuitively, this enriches (extends) Q with all the consequences that follow from its atoms and the constraints* $I \cup C_{\mathcal{V}}^{IO}$*.*

(ii) restricting the chase result to only $\mathcal{V}$*-atoms; the result is called a* universal plan U*.*

(iii) annotating each atom of U with a unique ID called a provenance term*.*

(iv) chasing U with the constraints in $I \cup C_{\mathcal{V}}^{OI}$*, where* $C_{\mathcal{V}}^{OI} = \{V_{OI} \mid V \in \mathcal{V}\}$*, and annotating each relational atom* $\boldsymbol{a}$ *introduced by these chase steps with a* provenance formula $\pi(\boldsymbol{a})$*, which gives the set of U-subqueries whose chasing led to the creation of* $\boldsymbol{a}$*; the result of this phase, called the* backchase*, is denoted B.*

(v) matching Q against B and outputting as rewritings the subsets of U that are responsible

for the introduction (during the backchase) of the atoms in the image $h(Q)$ of Q; these rewritings are read off directly from the provenance formula $\pi(h(Q))$.

In our example, I is empty, $C_V^{IO} = \{V_{IO}\}$, and the result of the chase in phase (i) is $Q_1(x,y)$:- $R(x,z) \wedge S(z,y) \wedge V(x,y)$. The universal plan obtained in (ii) by restricting Q_1 to the schema of view names is $U(x,y)$:- $V(x,y)^{p_0}$, where p_0 denotes the provenance term of atom $V(x,y)$. The result of backchasing U with C_V^{OI} in phase (iv) is $B(x,y)$:- $V(x,y)^{p_0} \wedge R(x,z)^{p_0} \wedge S(z,y)^{p_0}$. Note that the $\pi(R)$ and $\pi(S)$ of the R and S atoms (a simple term p_0, in this example) are introduced by chasing the view V. Finally, in phase (v) we find one match image given by h from Q's body into the R and S atoms from B's body. The provenance $\pi(h(Q))$ of the image h of Q is p_0, which corresponds to an equivalent rewriting: $RW(x,y)$:- $V(x,y)$.

Prune$_{prov}$ **Minimum-Cost Rewriting.** The minimal rewritings of a query Q are obtained by first finding the set $\mathcal{H}$ of all matches (i.e., containment mappings) from Q to the result B of backchasing the universal plan U. Denoting with $\pi(A)$ the provenance formula of a set of atoms A, PACB computes the DNF form D of $\bigvee_{h \in \mathcal{H}} \pi(h(Q))$. Each conjunct c of D determines a subquery $sq(c)$ of U which is guaranteed to be a rewriting of Q.

The idea behind cost-based pruning is that, whenever the naïve PACB backchase would add a provenance conjunct c to an existing atom a's provenance formula $\pi(a)$, *Prune$_{prov}$* does so more conservatively: if the cost $\gamma(sq(c))$ is larger than the minimum cost threshold T found so far, then c will never participate in a minimum-cost rewriting and need not be added to $\pi(a)$. Moreover, atom a itself need not be chased into B in the first place if all its provenance conjuncts have above-threshold cost.

Example 5.7.3 (Applying *Prune$_{prov}$*)**.** *Let $E = M(NM)$, where we assume for simplicity that $M_{50K \times 100}$ and $N_{100 \times 50K}$ are dense. Exploiting the associativity of matrix-multiplication $(MN)M = M(NM)$ during the chase leads to the following universal plan U annotated with provenance terms:*

$$U(R_2) :- name(M, \text{“}M.csv\text{”})^{p_0}, size(M, \text{“}50000\text{”}, \text{“}100\text{”})^{p_1}, name(N, \text{“}N.csv\text{”})^{p_2},$$
$$size(N, \text{“}100\text{”}, \text{“}50000\text{”})^{p_3}, \texttt{multi}_M(M,N,R_1)^{p_4}, \texttt{multi}_M(R_1,M,R_2)^{p_5},$$
$$\texttt{multi}_M(N,M,R_3)^{p_6}, \texttt{multi}_M(M,R_3,R_2)^{p_7}$$

Now, consider in the backchase the associativity constraint C:

$$\forall M \forall N \forall R_1 \forall R_2$$
$$\texttt{multi}_M(M,N,R_1) \wedge \texttt{multi}_M(R_1,M,R_2) \rightarrow$$
$$\exists R_4 \texttt{multi}_M(N,M,R_4) \wedge \texttt{multi}_M(M,R_4,R_2)$$

There exists a match h embedding the two atoms in the premise P of C into the U atoms whose provenance annotations are p_4 and p_5. The provenance conjunct collected from P's image is $\pi(h(P))$=$p_4 \wedge p_5$.

Without pruning*, the backchase would chase U with the constraint C, yielding U' which has additional $\pi(h(P))$-annotated atoms:* $\texttt{multi}_M(N,M,R_4)^{p_4 \wedge p_5} \wedge \texttt{multi}_M(M,R_4,R_2)^{p_4 \wedge p_5}$. *E has precisely two matches h_1, h_2 into U'. $h_1(E)$ involves the newly added atoms as well as those annotated with p_0, p_1, p_2, p_3. Collecting all their provenance annotations yields the conjunct $c_1 = p_0 \wedge p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$. c_1 determines the U-subquery $sq(c_1)$ corresponding to the rewriting $(MN)M$, of cost $(50K)^2$.*

$h_2(E)$'s image yields the provenance conjunct $c_2 = p_0 \wedge p_1 \wedge p_2 \wedge p_3 \wedge p_6 \wedge p_7$, which determines the rewriting $M(NM)$ that happens to be the original expression E of cost 100^2.

The naïve PACB would find both rewritings, cost them, and drop the former ,which introduces a large intermediate result ($\gamma(sq(\pi(h(P))) = (50K)^2$), above the threshold, in favor of the latter.

With pruning*, the threshold T is the cost of the original expression 100^2. The chase step with C is never applied, as it would introduce the provenance conjunct $\pi(h(P))$ which determines U-subquery $sq(\pi(h(P)) =$* $\texttt{multi}_M(M,N,R_1)^{p_4} \wedge \texttt{multi}_M(R_1,M,R_2)^{p_5}$ *of cost $(50K)^2$ exceeding T. The atoms needed as an image of E under h_1 are thus never produced while*

backchasing U, so the expensive rewriting is never discovered. This leaves only the match image $h_2(E)$, which corresponds to the efficient rewriting: $M(NM)$.

Our improvements on $Prune_{prov}$. Whenever the pruned chase step is applicable and applied for each constraint, the original algorithm searches for all *minimal-rewritings* $\mathcal{RW}$ that can be found "so far", then it costs each $rw \in \mathcal{RW}$ to find the "so far" *minimum-cost one* rw_e and adjusts the threshold T to the cost of rw_e. However, this strategy can cause *redundant* costing of $rw \in \mathcal{RW}$ whenever the pruned chase step is applied again for another constraint.

To address this issue, in our modified version of $Prune_{prov}$, we keep track of the rewritings that their costs are already estimated to prevent such redundant work. Additionally, the search for *minimal-rewritings* "so far" (matches of the query Q into the evolving universal plan instance U'), whenever the pruned chase step is applied, is modeled as a query evaluation of Q against U' (viewed as a symbolic/canonical database [26]). This involves repeatedly evaluating the same query plan. However, the query is evaluated over evolutions of the same instance. Each pruned chase step *adds a few new tuples* to the evolving instance, corresponding to atoms introduced by that step, while most of the instance is unchanged. Therefore, instead of evaluating the query plan from scratch, we employ incremental evaluation as in [84]. The plan is kept in memory along with the populated hash tables and, whenever new tuples are added to the evolving instance, we push them to the plan.

5.8 Guarantees on the Reduction

We detail the conditions under which we guarantee that our approach is *sound* (i.e., generates only equivalent, cost-optimal rewritings), and *complete* (i.e., finds all equivalent cost-optimal rewritings).

Let $\mathcal{V} \subseteq \mathcal{L}$ be a set of materialized view definitions, where $\mathcal{L}$ is the language of hybrid expressions described in Section 5.3.

Let LA_{prop} be a set of properties of the LA operations in L_{ops} that admits relational encoding over $\mathcal{VREM}$. We say that LA_{prop} is *terminating* if it corresponds to a set of TGDs and EGDs with terminating chase (this holds for our choice of LA_{prop}).

Denote with γ a cost model for expressions from $\mathcal{L}$. We say that γ is *monotonic* if expressions are never assigned a lower cost than their subexpressions (this is true for both models we used).

We call $E \in \mathcal{L}$ $(\gamma, LA_{prop}, \mathcal{V})$-*optimal* if for every $E' \in \mathcal{L}$ that is $(LA_{prop}, \mathcal{V})$-equivalent to E we have $\gamma(E') \geq \gamma(E)$.

Let $\text{Eq}^{\gamma}\langle LA_{prop}, \mathcal{V}\rangle(E)$ denote the set of all $(\gamma, LA_{prop}, \mathcal{V})$-optimal expressions that are $(LA_{prop}, \mathcal{V})$-equivalent to E.

We denote with $HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle$ our parameterized solution based on relational encoding followed by PACB++ rewriting and next by decoding all the relational rewritings generated by the cost-based pruning PACB++ (recall Figure 5.1). Given $E \in \mathcal{L}$, $HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle(E)$ denotes all expressions returned by $HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle$ on input E.

Theorem 5.8.1 (Soundness)**.** *If the cost model γ is monotonic, then for every $E \in \mathcal{L}$ and every $rw \in HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle(E)$, we have $rw \in Eq^{\gamma}\langle LA_{prop}, \mathcal{V}\rangle(E)$.*

The proof sketch parallels that of Theorem 4.12.1 (shown in Section 4.12).The only difference is that we extended HADAD to use $Prune_{prov}$, where the soundness still holds.

Completeness Discussion. We state that our approach is complete within theoretically imposed limitations, meaning if γ is monotonic and LA_{prop} is terminating, then for every $E \in \mathcal{L}$ and every $rw \in \text{Eq}^{\gamma}\langle LA_{prop}, \mathcal{V}\rangle(E)$, we have $rw \in HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle(E)$. The completeness of PACB $Prune_{prov}$ [83] (which was proven under terminating chase and monotonic cost function) guarantees that it *returns* all and precisely the *minimum-cost reformulations* of $enc(E)$ under $\mathcal{C}_{\mathcal{V}} \cup enc_{LA}(LA_{prop}))$. However, LA operations can satisfy more properties that we cannot fully capture with our class of constraints. The rewriting algorithm will not exploit those properties.

Thus, with respect to those properties, our approach is incomplete necessarily so because these tasks are undecidable.

5.9 Experimental Evaluation

We evaluate HADAD to answer the research questions below about our approach:

- **Section 5.9.2** and **Section 5.9.3**: Can HADAD find rewrites with/without views that lead to a greater performance improvement than original pipelines without modifying the internals of the existing systems?. Are the identified rewrites found by state-of-the-art platforms?.
- **Section 5.9.2** and **Section 5.9.3**: Is HADAD's optimization overhead compensated by the performance gains in execution?.

We evaluate our approach, first on **LA-based pipelines** (Section 5.9.2), then on **hybrid expressions** (Section 5.9.3).

5.9.1 Experiment Setup

We use a single machine with an Intel(R) Xeon(R) CPU E5-2640 v4@2.40GHz, 20 physical cores (40 logical cores) and 123GB RAM, running CentOS Linux release 7.9.2009. The machine is equipped with a 1TB SSD storage device, where the disk read and write speeds are 616 MB/s and 455 MB/s, respectively. We run our PACB++ engine on OpenJDK Java 8 VM. As for **LA systems/libraries**, we use **R 3.6.0**, **Numpy 1.16.6 (python 2.7)**, **TensorFlow r1.4.2**, **Spark 2.4.5 (MLlib)**, **SystemML 1.2.0**. For hybrid experiments, we use **MorpheusR** [12] and **SparkSQL** [35]. We use Scala version 2.11.12 for SparkMLlib and SystemML.

We use a JVM-based linear algebra library for SystemML as recommended in [131], at the optimization level 4. Additionally, we enable multi-threaded matrix operations in a single node. We run Spark in a single node setting and configure Spark to import OpenBLAS (we

Table 5.2: LA benchmark pipelines (part 1).

No.	Expression	No.	Expression	No.	Expression
P1.1	$(MN)^T$	P1.2	$A^T + B^T$	P1.3	$C^{-1}D^{-1}$
P1.4	$(A+B)v_1$	P1.5	$((D)^{-1})^{-1}$	P1.6	$\texttt{trace}(s_1 D)$
P1.7	$((A)^T)^T$	P1.8	$s_1 A + s_2 A$	P1.9	$\texttt{det}(D^T)$
P1.10	$\texttt{rowSums}(A^T)$	P1.11	$\texttt{rowSums}(A^T + B^T)$	P1.12	$\texttt{colSums}((MN))$
P1.13	$\texttt{sum}(MN)$	P1.14	$\texttt{sum}(\texttt{colSums}((N^T M^T))$	P1.15	$(MN)M$
P1.16	$\texttt{sum}(A^T)$	P1.17	$\texttt{det}(CDC)$	P1.18	$\texttt{sum}(\texttt{colSums}((A))$
P1.19	$(C^T)^{-1}$	P1.20	$\texttt{trace}(C^{-1})$	P1.21	$(C + D^{-1})^T$
P1.22	$\texttt{trace}((C+D)^{-1})$	P1.23	$\texttt{det}((CD)^{-1}) + D)$	P1.24	$\texttt{trace}((CD)^{-1}))$ $+\texttt{trace}(D)$
P1.25	$M \odot (N^T /$ $(MNN^T))$	P1.26	$N \odot (M^T / (M^T MN))$	P1.27	$\texttt{trace}(D(CD)^T)$
P1.28	$A \odot (A \odot B + A)$	P1.29	$DCCC$	P1.30	$NM \odot NMR^T$

compiled from the source as detailed in [13]) as a linear algebra backend library to take advantage of its accelerations [131]. SparkMLlib's datatypes do not support many basic LA operations, such as scalar-matrix multiplication, Hadamard-product. To support them, we use the `Breeze` Scala library [5], convert MLlib's datatypes to Breeze types and express the basic LA operations. The driver memory allocated for Spark and SystemML is 115GB.

To maximize TensorFlow performance, we compile it from the source to enable architecture-specific optimizations (e.g., SMID instructions). For all systems/libraries, we set the number of **cores** to **24** (i.e., we use the command **`taskset -c 0-23`** when running R, NumPy and TensorFlow scripts, as used in [131]). All system use double precision numbers (**`double`**) by default, while TensorFlow uses single precision floating point numbers (**`float`**). To enable fair companion with other systems, we use a double precision (**`tf.float64`**) for Tensorflow.

5.9.2 LA-based Experiments

In this experiment, we study the performance benefits of our approach on LA-based pipelines as well as our optimization overhead.

Table 5.3: LA benchmark pipelines (part 2).

No.	Expression	No.	Expression	No.	Expression
P2.1	$\texttt{trace}(C+D)$	P2.2	$\texttt{det}(D^{-1})$	P2.3	$\texttt{trace}(D^T)$
P2.4	s_1A+s_1B	P2.5	$\texttt{det}((C+D)^{-1})$	P2.6	$C^T(D^T)^{-1}$
P2.7	$DD^{-1}C$	P2.8	$\texttt{det}(C^TD)$	P2.9	$\texttt{trace}(C^TD^T+D)$
P2.10	$\texttt{rowSums}(MN)$	P2.11	$\texttt{sum}(A+B)$	P2.12	$\texttt{sum}(\texttt{rowSums}(N^TM^T))$
P2.13	$((MN)M)^T$	P2.14	$((MN)M)N$	P2.15	$\texttt{sum}(\texttt{rowSums}(A))$
P2.16	$\texttt{trace}(C^{-1}D^{-1})$ $+\texttt{trace}(D)$	P2.17	$((((C+D)^{-1})^T)$ $((D^{-1})^{-1})C^{-1}C$	P2.18	$\texttt{colSums}((A^T+B^T)$
P2.19	$(C^TD)^{-1}$	P2.20	$(M(NM))^T$	P2.21	$(D^TD)^{-1}(D^Tv_1)$
P2.22	$\texttt{exp}((C+D)^T)$	P2.23	$\texttt{det}(C)*\texttt{det}(D)$ $*\texttt{det}(C)$	P2.24	$(D^{-1}C)^T$
P2.25	$(u_1v_2^T-X)v_2$	P2.26	$\texttt{exp}((C+D)^{-1})$	P2.27	$((((C+D)^T)^{-1})D)C$

Table 5.4: Overview of used real datasets.

Name	Rows n	Cols$_m$	Nnz $\|\|X\|\|_0$	$\mathbf{S}_X$
Amazon/(AS)	50K	100	378	0.0075%
Amazon/(AM)	100K	100	673	0.0067%
Amazon/(AL_1)	1M	100	6539	0.0065%
Amazon/(AL_2)	10M	100	11897	0.0011%
Amazon/(AL_3)	100K	50K	103557	0.0020%
Netflix/(NS)	50K	100	69559	1.3911%
Netflix/(NM)	100K	100	139344	1.3934%
Netflix/(NL_1)	1M	100	665445	0.6654%
Netflix/(NL_2)	10M	100	665445	0.0665%
Netflix/(NL_3)	100K	50K	15357418	0.307%

Table 5.5: Syntactically generated dense datasets.

Name	Rows n	Cols$_m$	Name	Rows n	Cols$_m$
Syn_1	50K	100	Syn_6	20K	20K
Syn_2	100	50K	Syn_7	100	1
Syn_3	1M	100	Syn_8	50K	1
Syn_4	5M	100	Syn_9	100K	1
Syn_5	10K	10K	Syn_{10}	100	100

Datasets. We used several **real-world, sparse matrices**, for which Table 5.4 lists the dimensions and the sparsity ($\mathbf{S}_X$): (i) we used several subsets of an Amazon books review dataset [2] (in JSON), and similarly (ii) subsets of a Netflix movie rating dataset [15]. They

Table 5.6: Matrices used for each matrix name in a pipeline.

Matrix Name	Used Data
A and B	AM, AL_1, AL_2, NM, NL_1, NL_2, Syn_3 or Syn_4
C and D	Syn_5 or Syn_6
M	AS, NS, or Syn_1
N	Syn_2
R	Syn_{10}
X	AL_3 or NL_3
v_1,v_2 and u_1	Syn_7, Syn_8 and Syn_9, respectively.

were easily converted into matrices where columns are items and rows are customers [137]; we extracted smaller subsets of all real datasets to ensure the various computations applied on them fit in memory (e.g., Amazon/(AS) denotes the small version of the Amazon dataset). We also used a set of **synthetic, dense matrices**, described in Table 5.5.

LA benchmark. We use a set $\mathcal{P}$ of **57 LA pipelines** used in prior studies and/or frequently occurring in real-world LA computations, as follows:

- **Real-world Pipelines (10):** include: a chain of matrix self products used for reachability queries and other graph analytics [128] (**P1.29** in Table 5.2); expressions used in Alternating Least Square Factorization (ALS) [137] (**P2.25** in Table 5.3); Poisson Nonnegative Matrix Factorization (PNMF)(**P1.13** in Table 5.2) [137]; Nonnegative Matrix Factorization (NMF)(**P1.25** and **P1.26** in Table 5.2) [131]; recommendation computation [128] (**P1.30** in Table 5.2); finally, Ordinary Least Squares Regression (OLS) [131] (**P2.21** in Table 5.3).

- **Synthetic Pipelines (47):** were also generated, based on a set of basic matrix operations (inverse, multiplication, addition, etc.), and a set of combination templates, written as a Rule-Iterated Context-Free Grammar (RI-CFG) [113]. Examples of synthetic pipelines include **P2.16**, **P2.16**, **P2.23**, **P2.24** in Table 5.2. Expressions thus generated include **P2.16**, **P2.16**, **P2.23**, **P2.24** in Table 5.3.

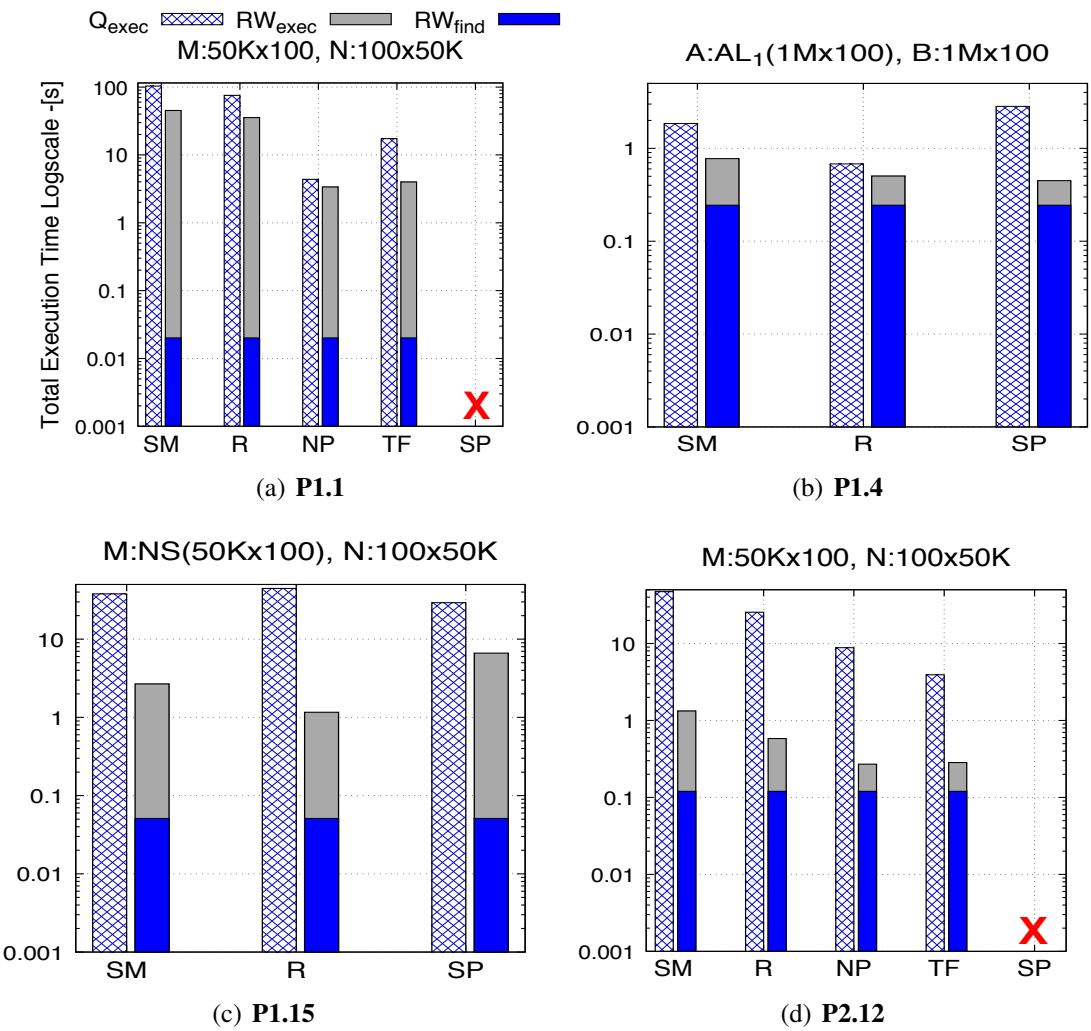
Methodology. We first show the performance benefits of our approach to LA-oriented

systems/tools mentioned above using a set $\mathcal{P}^{\neg Opt} \subset \mathcal{P}$ of **38** pipelines in Table 5.2 and Table 5.3, and the matrices in Table 5.6. The performance of these pipelines can be improved *just by exploiting LA properties* (in the absence of views). For TensorFlow and NumPy, we present the results only for dense matrices due to limited support for sparse matrices. Then, we show how our approach improves the performance of **30** pipelines from $\mathcal{P}$, denoted $\mathcal{P}^{Views}$, using pre-materialized views. Finally, we study our rewriting performance and *optimization overhead* for the set $\mathcal{P}^{Opt} = \mathcal{P} \setminus \mathcal{P}^{\neg Opt}$ of **19** pipelines that are already optimized. For the purpose of the discussion, we discuss a selection of these experiments below. The detailed set of results appears in Appendix B.5, Appendix B.6, Appendix B.7, and Appendix B.8.

Effectiveness of LA Rewriting (No Views). For each system, we run the original pipeline and our rewriting five times; we report the average of the last four running times. We exclude the data loading time [131]. For fairness, we ensure SparkMLib and SystemML compute the entire pipeline (despite their lazy evaluation mode). So, we print a random matrix cell value to force the systems to compute the entire pipeline. Additionally, for SystemML, we insert a "break block" (e.g., `WHILE(FALSE)`) after each pipeline and before the cell print statement to prevent computing only the value of a single cell and force it to compute all the outputs.

Discussion. Figure 5.7 illustrates the original pipeline execution time Q_{exec} and the selected rewriting execution time RW_{exec} for **P1.1**, **P1.3**, **P1.4**, and **P1.15**, including the rewriting time RW_{find}, using the MNC cost model. For each pipeline, the used datasets are on top of the figure. For brevity in the figures, we use **SM** for SystemML, **NP** for NumPy, **TF** for Tensorflow, and **SP** for SparkMLlib.

For **P1.1** (see Figure 5.7(a)), both matrices are dense. The speedup ($1.3\times$ to $4\times$) comes from rewriting $(MN)^T$ (intermediate result size to $(50K)^2$) into $N^T M^T$, much cheaper since both N^T and M^T are of size $50K \times 100$. We exclude MLlib from this experiment since it failed to allocate memory for the intermediate matrix (Spark/MLLib limits the maximum size of a dense



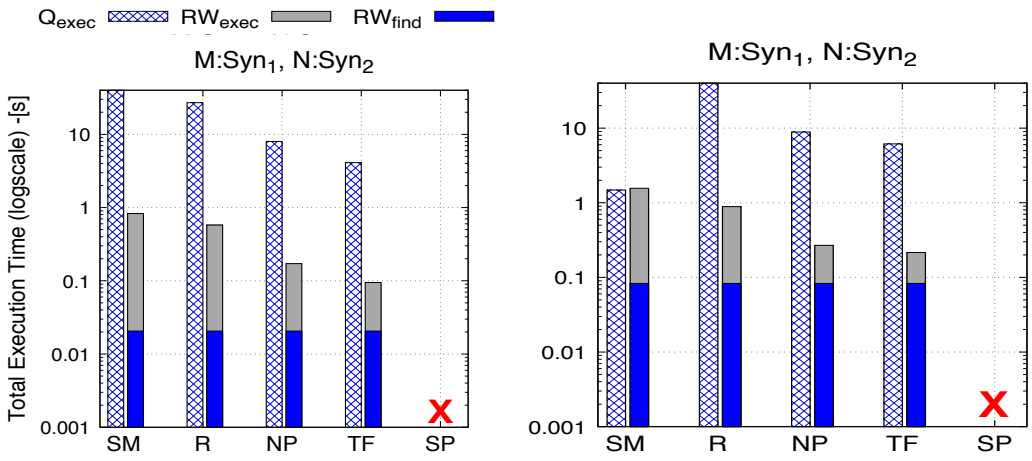
(a) **P1.1** (b) **P1.4**

(c) **P1.15** (d) **P2.12**

Figure 5.7: P1.1, P1.4, P1.15, and P2.12 evaluation before and after rewriting.

matrix). As a variation (not plotted in the Figure), we run the same pipeline with the ultra-sparse *AS* matrix (0.0075% non-zeros) used as M. The Q_{exec} and RW_{exec} time are very comparable using SystemML because we avoid large dense intermediates. In R, this scenario leads to a runtime exception, and to avoid it, we cast M during load time to a dense matrix type. Thus, the speedup achieved is the same as if M and N were both dense. If, instead, *NS* (1.3911% non-zeros) plays the role of M, our rewrite achieves $\approx 1.8\times$ speedup for SystemML.

For **P1.4** (Figure 5.7(b)), we rewrite $(A+B)v_1$ to $Av_1 + Bv_1$. Adding Amazon sparse matrix AL_1 (0.0065% nnz) used as A to a dense matrix B results into a dense intermediate of size $1M \times 100$. Instead, $Av_1 + Bv_1$ has intermediate results of sizes $1M \times 1$ and $1M \times 1$, respectively.



(a) **P1.13** (b) **P1.25**

Figure 5.8: P1.13, and P1.25 evaluation before and after rewriting.

In addition, Av_1 can be computed efficiently thanks to the sparsity in A. The MNC sparsity estimator has a noticeable overhead. We run the same pipeline, where the dense $5M \times 100$ matrix plays both A and B (not shown in the Figure). This leads to speedup of up to $2.7\times$ for SystemML, $1.4\times$ for R, and $9\times$ for MLlib, which does not natively support matrix addition; thus, we convert its matrices to Breeze types in order to perform it [131].

P1.15 (Figure 5.7(c)) is a matrix chain multiplication. The naïve left-to-right evaluation plan $(MN)M$ computes an intermediate matrix of size $O(n^2)$, where n is $50K$. Instead, the rewriting $M(NM)$ only needs an $O(m^2)$ intermediate matrix, where m is 100, and is much faster. To avoid MLLib memory failure on **P1.15**, we use the distributed matrix of type BlockMatrix, which we can run locally for both matrices. While M thus converted has the same sparsity, Spark views it as being of a dense type (multiplication on BlockMatrix is considered to always produce dense matrices) [21]. SystemML does optimize the multiplication order if the user does not enforce it (based on our experiments). Further (not shown in the Figure), we run **P.15** with AS in the role of M. This is $4\times$ faster in SystemML since with an ultra sparse M, multiplication is more efficient. This is not the case for MLlib which views it as dense. For R, we again had to densify M during loading to prevent crashes (discussed earlier).

Figure 5.7(d) shows speedups of up to $42\times$ when rewriting **P2.12** into $\texttt{sum}((\texttt{colSums}(M))^T$

$\odot$rowSums(N)). This exploits several properties:

(i) $(MN)^T = N^T M^T$,

(ii) sum(M^T) =sum(M),

(iii) sum(rowSums(M)) = sum(M), and

(iv) sum(MN)=sum((colSums(M))$^T \odot$ rowSums(N)).

SystemML captures (ii), (iii), and (iv) as rewrite rules, however, it is unable to find this rewrite since it is unaware of (i). Other systems do not exploit these properties. *In this experiment and subsequently, whenever MLlib is absent, this is due to its lack of support for LA operations* (here, sum of all cells in a matrix) on BlockMatrix.

Figures 5.8(a) and 5.8(b) study **P1.13** and **P1.25**, two real-world pipelines involved in ML algorithms, using the MNC cost model; note the log-scale y axis. Rewriting **P1.13** to sum(t(colSums((M)) $*$ rowSums(N)) yields a speedup of 50×; while SystemML *has* this rewrite as a dynamic rewrite rule, it did not apply it during optimization. In addition, our rewrite allows other systems to benefit from it. Not shown in the Figure, we re-run this pipeline with M ultra sparse (using *AS*) and SystemML: the rewrite did not bring benefits, since MN is already efficient. For **P1.25**, the important optimization is selecting the multiplication order in MNN^T (Figure 5.8(b)). SystemML is efficient here, due to its dedicated operator tsmm for transpose-self matrix multiplication and mmchain for matrix multiply chains. MLlib is excluded as it does not support divisions on BlockMatrixes.

Figure 5.9 shows the distribution of the significant rewriting speedup on $\mathcal{P}^{\neg Opt}$ running on R, and using the MNC-based cost model. For clarity, we split the distribution into two figures: on the left, 25 $\mathcal{P}^{\neg Opt}$ pipelines with speedup lower than 10×; on the right, the remaining 13 with greater speedup. Among the former, 87% achieved at least 1.5× speedup. The latter are sped up by 12× to 58×. **P1.5** is an extreme case here (not plotted): it is sped up by about 1000×, simply by rewriting $((D)^{-1})^{-1}$ into D.

Effectiveness of Views-based LA Rewriting. We define a set V_{exp} of 12 views that

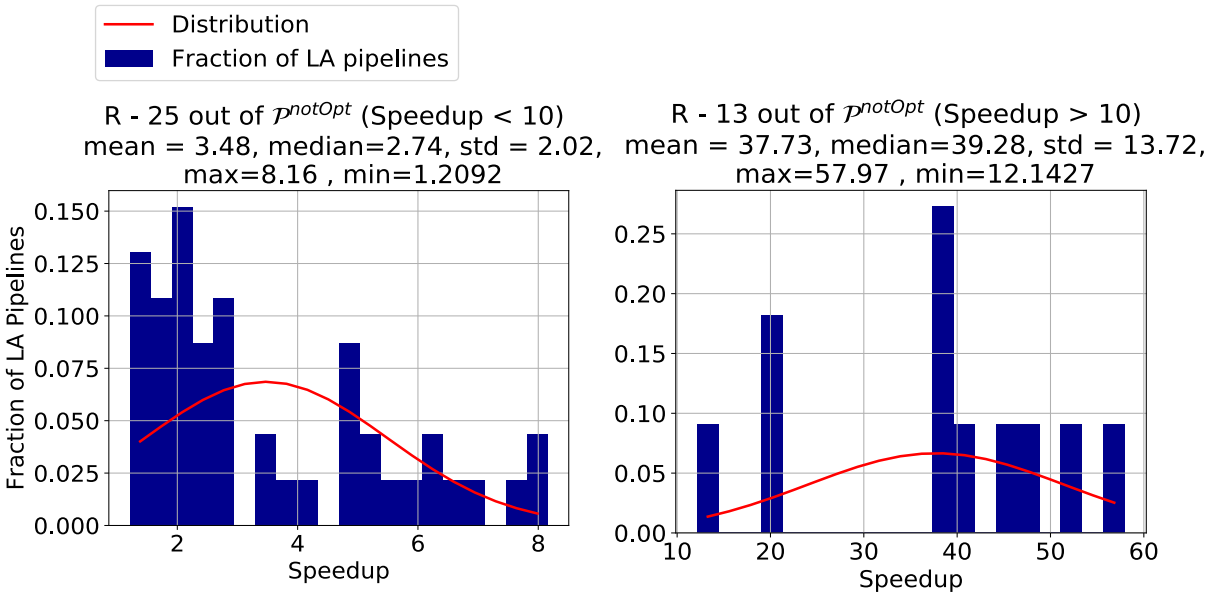


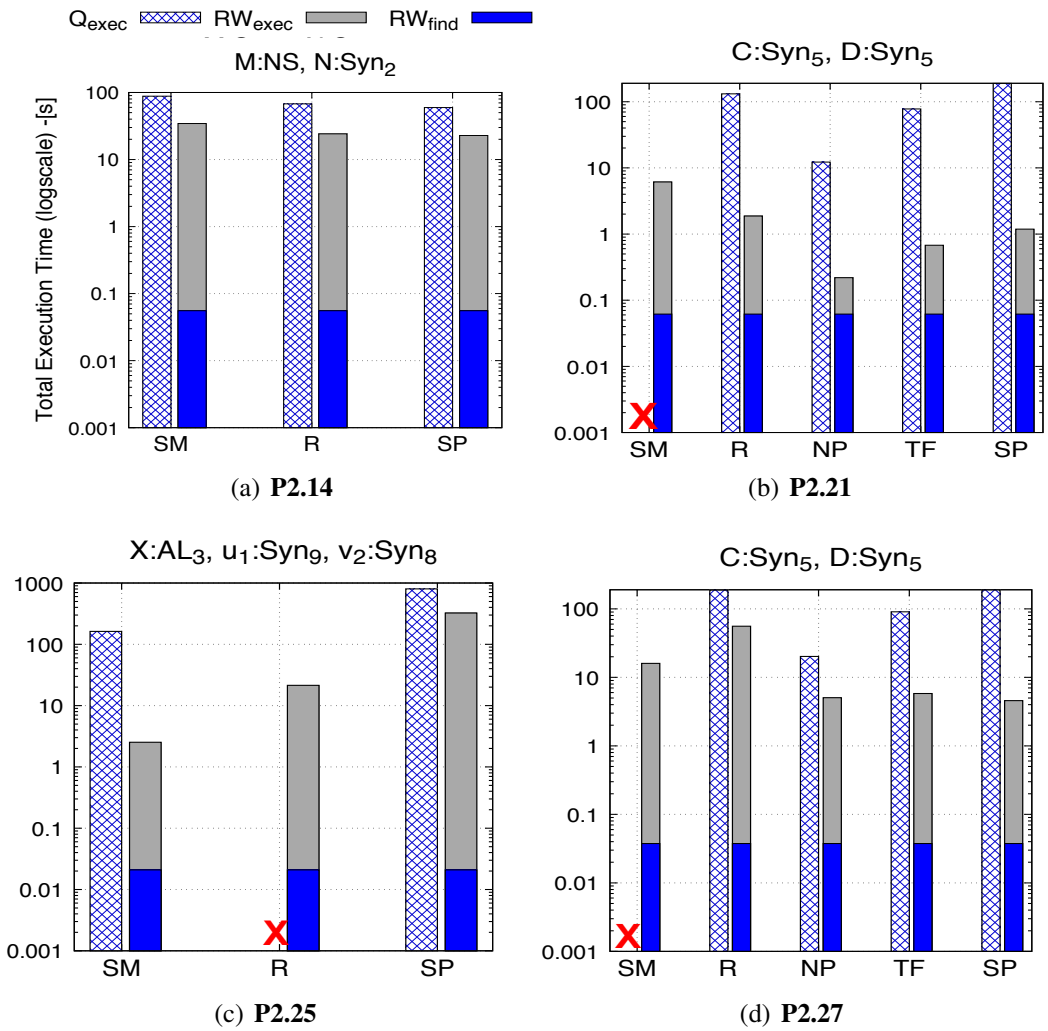
Figure 5.9: R speedup on $\mathcal{P}^{\neg Opt}$.

pre-compute the result of some expensive operations (multiplication, inverse, determinant, etc.), which can be used to answer our $\mathcal{P}^{Views}$ pipelines and materialize them on disk as CSV files. The experiments outlined below used the naïve cost model; all graphs have a log-scale y axis. The list of all view definitions appears in Appendix B.5.

Discussion. For **P2.14** (Figure 5.10(a)), using the view $V_3 = NM$ by and the multiplication associativity leads to up to 2.8× speedup.

Figure 5.10(b) shows the performance gain due to using the view $V_1 = D^{-1}$, for the ordinary-least regression (OLS) pipeline **P2.21**. It has 8 rewritings, 4 of which use V_1; they are found thanks to the properties $(CD)^{-1} = D^{-1}C^{-1}$, $(CD)E = C(DE)$ and $(D^T)^{-1} = (D^{-1})^T$ among others. The cheapest rewriting is $V(V^T(D^T v_1))$, since it introduces small intermediate results due to the optimal matrix chain multiplication order. This rewrite leads to 70×, 55×, 115×, and 150× speedups on R, NumPy, TensorFlow, and MLlib, respectively; On SystemML, the original pipeline timed out (> 1000 seconds).

Pipeline **P2.25** (Figure 5.10(c)) benefits from the view $V_4 = u_1 v_2^T$, which pre-computes a dense intermediate vector multiplication result. The rewrite of $(u_1 v_2^T - X)v_2$ to $V_4 v_2 - X v_2$ is



(a) **P2.14** (b) **P2.21**

(c) **P2.25** (d) **P2.27**

Figure 5.10: P2.14, P2.21, P2.25 and P2.27 evaluation before and after rewriting using V_{exp}.

based on exploiting the property $(A+B)v = Av+Bv$, which leads to 65× speedup in SystemML. For MLlib, as discussed before, to avoid memory failure, we used BlockMatrix types. for all matrices and vectors, thus they were treated as dense. In R, the original pipeline triggers a memory allocation failure for the intermediate result, which the rewriting avoids.

Figure 5.10(d) shows that for **P2.27** exploiting the views $V_9 = (D+C)^{-1}$ and $V_5 = DC$ leads to speedups of 4× to 41× on different systems. Properties enabling rewriting here are $C+D = D+C$, $(D^T)^{-1} = (D^{-1})^T$ and $(CD)E = C(DE)$. The matrix multiplication order does not have an effect for this pipeline since all matrices are squares.

Rewriting Performance and Overhead. We now study the running time RW_{find} of our

rewriting algorithm, and the *rewrite overhead* defined as:

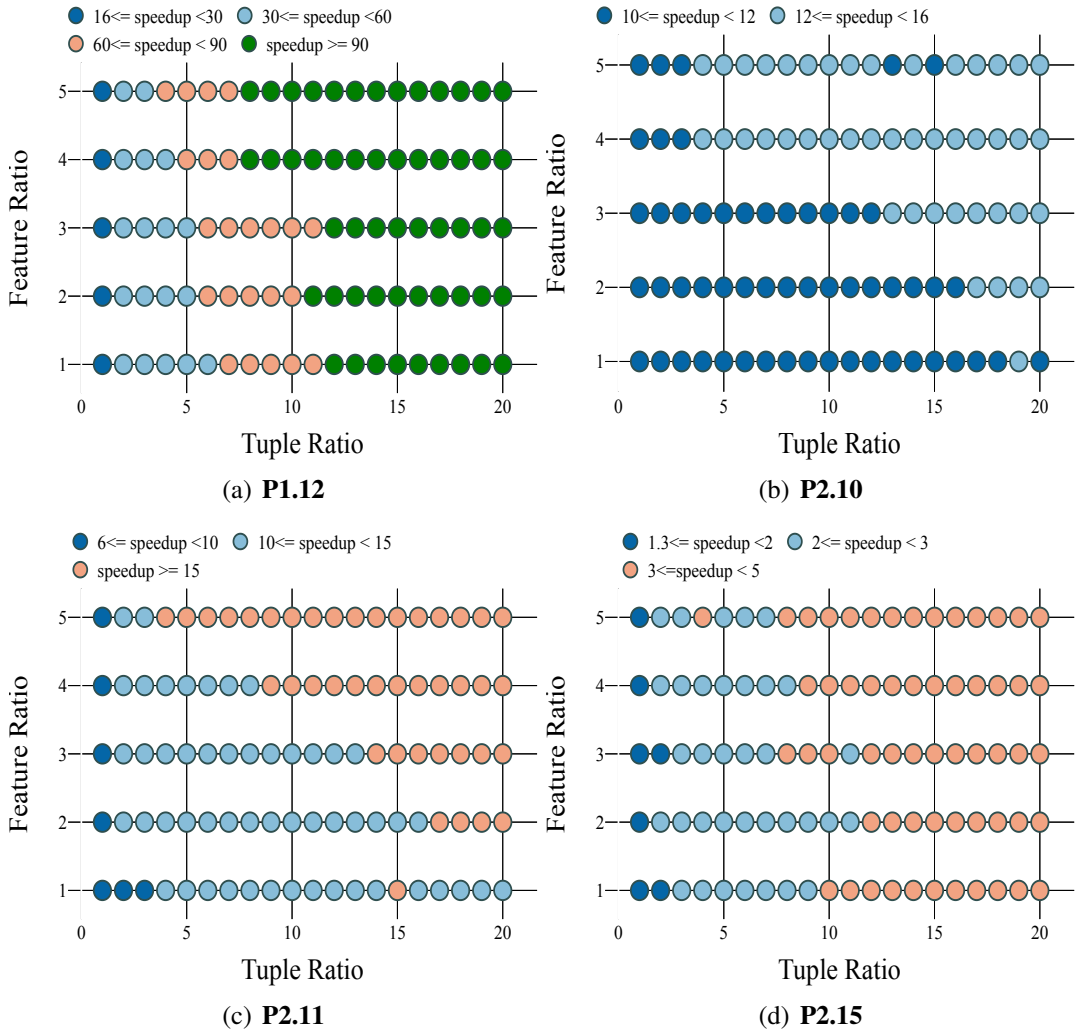
$$RW_{find}/(Q_{exec}+ RW_{find})$$

where Q_{exec} is the time to run the pipeline "as stated".

We run each experiment 100 times and report the average of the last 99 times. The global trends are as follows. (*i*) For a fixed pipeline and set of data matrices, *the overhead is slightly higher using the MNC cost model*, since histograms are built during optimization. (*ii*) For a fixed pipeline and cost model, *sparse matrices lead to a higher overhead* simply because Q_{exec} tends to be smaller. (*iii*) *Some* (system, pipeline) pairs lead to a low Q_{exec} when *the system applies internally the same optimization* that HADAD finds "outside" of the system.

Concretely, for the $\mathcal{P}^{\neg Opt}$ pipelines, on the dense and sparse matrices listed in Table 5.6, using the naïve cost model, 64% of the RW_{find} times are under 25ms (50% are under 20ms), and the longest is about 200m. Using the MNC estimator, 55% took less than 20ms, and the longest (outlier) took about 300ms. Among the 38 $\mathcal{P}^{\neg Opt}$ pipelines, SystemML finds efficient rewritings for a set of 9, denoted $\mathcal{P}_{SM}^{\neg Opt}$, while TensorFlow optimizes a different set of 11, denoted $\mathcal{P}_{TF}^{\neg Opt}$. On these subsets, where HADAD's optimization is redundant, using *dense* matrices, the rewriting overhead is very low: with the *MNC* model, 0.48% to 1.12% on $\mathcal{P}_{SM}^{\neg Opt}$ (0.64% on average), and 0.0051% to 3.51% on $\mathcal{P}_{TF}^{\neg Opt}$ (1.38% on average). Using the *naïve* estimator slightly reduces this overhead, but across $\mathcal{P}^{\neg Opt}$, this model misses 4 efficient rewritings. On *sparse* matrices, the overhead is at most 4.86% with the *naïve* estimator and up to 5.11% with the *MNC* one.

Among the already-optimal pipelines $\mathcal{P}^{Opt}$, 70% involve expensive operations such as inverse, determinant, and matrix exponential, leading to rather long Q_{exec} times. Thus, the rewriting overhead is less than 1% of the total time on all systems, using sparse or dense matrices, and the naïve or the MNC-based cost models. For the other $\mathcal{P}^{Opt}$ pipelines with short Q_{exec}, mostly matrix multiplications chains already in the optimal order, on *dense* matrices, the overhead reaches 0.143% (SparkMlLib) to 9.8% (TensorFlow) using the naïve cost model, while the MNC cost model leads to an overhead of 0.45% (SparkMlib) up to 10.26% (TensorFlow). On *sparse*



(a) **P1.12** (b) **P2.10**

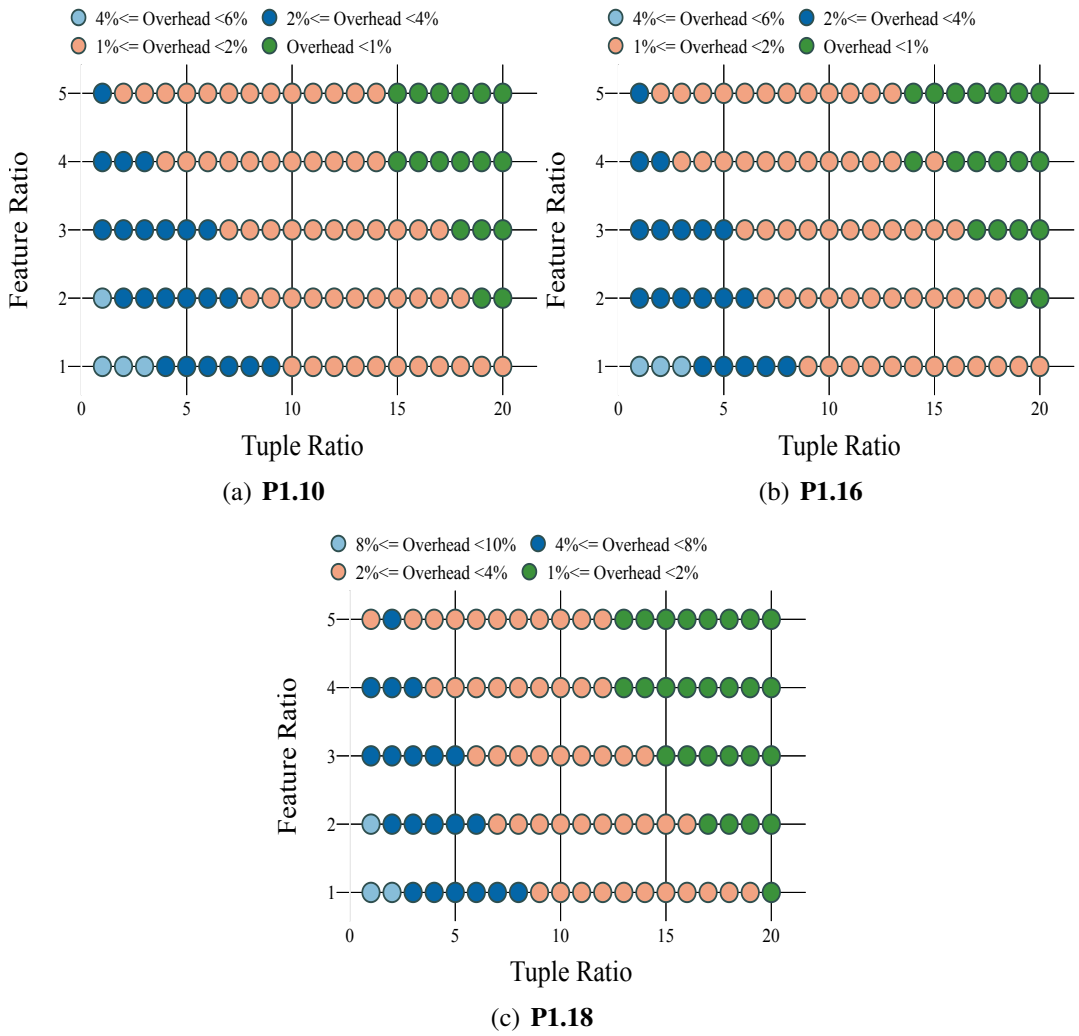
(c) **P2.11** (d) **P2.15**

Figure 5.11: Speedups of MorpheusR (with HADAD rewrites) over MorpheusR (without HADAD Rewrites) for pipelines P1.12, P2.10, P2.11 and P2.15 on synthetic data for a PK-FK join.

matrices, using the naïve and MNC cost models, the overhead reaches up to 0.18% (SparkMLlib) to 1.94% (SystemML), and 0.5% (SparkMLlib) to 2.61% (SystemML), respectively.

5.9.3 Hybrid (RA and LA) Experiments

We now study the benefits of rewriting on hybrid scenarios combining RA and LA operations. We first show the performance benefits of HADAD to a cross RA-LA platform, MorpheusR [12]. We then evaluate our hybrid micro-benchmark on SparkSQL+SystemML.



(a) **P1.10** (b) **P1.16**

(c) **P1.18**

Figure 5.12: HADAD RW_{find} overhead as a percentage (%) of the total time ($Q_{exec} + RW_{find}$) for pipelines P1.10, P1.16, and P1.18 running on MorpheusR.

MorpheusR Experiments. We use the same experimental setup introduced in [52] for generating synthetic datasets for the PK-FK join of tables **R** and **S**. The quantities varied are the *tuple ratio* (n_S/n_R) and feature ratio (d_R/d_S), where n_S and n_R are the number of rows and d_R and d_S are the number of columns (features) in **R** and **S**, respectively. We fix $n_R = 1M$ and $d_S = 20$. The join of **R** and **S** outputs $n_s \times (d_R + d_S)$ matrix **M**, which is always dense. We evaluate on MorpheusR a set of 8 pipelines and their rewritings found by HADAD using the naïve cost model.

Discussion. **P1.12**: `colSums`(**M***N*) is the example from Section 5.2, with **M** the output

(viewed as dense matrix) of joining tables **R** and **S** generated as described above. N is a $ncol(\mathbf{M}) \times 100$ dense matrix. HADAD's rewriting yields up to 125× speedup (see Figure 5.11(a)).

Figure 5.11(b) shows up to 15× speedup for **P2.10**: $\texttt{rowSums}(N\mathbf{M})$, where the size of N is $100 \times nrow(\mathbf{M})$. This is due to HADAD's rewriting: $N\texttt{rowSums}(\mathbf{M})$, which enables MorpheusR to push the rowSums operator to **R** and **S** instead of computing the matrix multiplication.

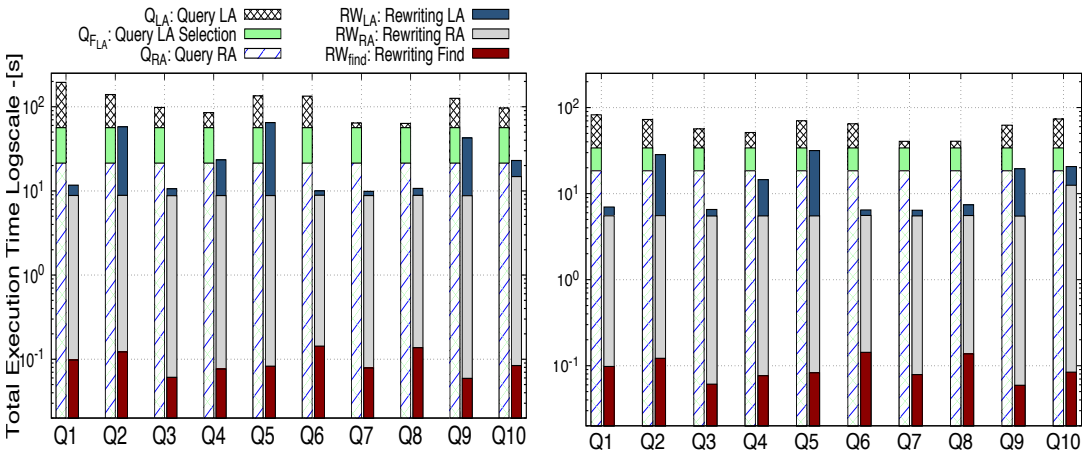
P2.11: $\texttt{sum}(N+\mathbf{M})$ is run *as-is* by MorpheusR since it does not factorize element-wise operations, e.g., addition. However, HADAD rewrites **P2.11** into $\text{sum}(N)+\texttt{sum}(\mathbf{M})$, which avoids the (large and dense) intermediate result of the element-wise matrix addition. The HADAD rewrite enables MorpheusR to execute $\texttt{sum}(\mathbf{M})$ by pushing sum to **R** and **S**, for up to 20× speedup (see Figure 5.11(c)).

MorpheusR evaluates **P2.15**: $\texttt{sum}(\texttt{rowSums}(\mathbf{M}))$ by pushing the rowSums operator to **R** and **S**. HADAD finds the rewriting $\texttt{sum}(\mathbf{M})$, which enables MorpheusR to push the sum operation instead, achieving up to 4.5× speedup (see Figure 5.11(d)).

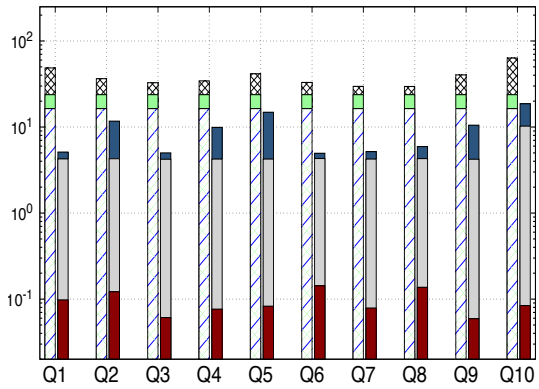
Since MorpheusR does not exploit the associative property of matrix multiplication, it cannot reorder multiplication chains to avoid large intermediate results, which leads to runtime exception in R (MorpheusR's backend). For example, for the chain $N^T N\mathbf{M}$ (variation of **P1.26**), when the size of **M** is 1M×40 and the size of N is 40×1M, the size of the $N^T N$ intermediate result is 1M×1M, which causes out of memory in R. HADAD exploits associativity and selects the rewriting $N^T(N\mathbf{M})$ of intermediate result size (40×40).

For pipelines **P1.14** and **P2.12**, that involve transpose operator, MorpheusR applies its special rewrite rules that replace an operation on $\mathbf{M}^T$ with an operation on **M** before pushing the operation to the base tables. HADAD rewrites both pipelines to $\texttt{sum}(N\mathbf{M})$, enabling MorpheusR to apply its factorized rewrite rule on $N\mathbf{M}$. This rewrite achieves speedup from 1.3× up to 1.5×.

Rewriting Overhead. For pipelines that are already optimized or MorpheusR finds the same rewriting found by HADAD, the rewriting overhead is very negligible compared to Q_{exec}. For example, for pipelines that contain matrix multiplication expressions, the rewriting time is



(a) **N matrix: 2M rows** (b) **N matrix: 1M rows**



(c) **N matrix : 0.5 rows**

Figure 5.13: Results of micro-hybrid benchmark using Twitter dataset.

generally less than 0.1% of the total time. However, for the other pipelines, such as **P1.10**, **P1.16**, and **P1.18**, which contain only aggregate operations, the rewriting time is up to 9% of the total time (when the data size is very small (0.32GB), and the computation is extremely efficient) and less than 1% (when the data size is large (19.2GB), and the computation is expensive) as shown in Figure 5.12.

Micro-hybrid Benchmark Experiments. In this experiment, we create a *micro-hybrid benchmark* on Twitter[24] and MIMIC [88] datasets to empirically study HADAD's rewriting benefits in a hybrid setting (SparkSQL+SystemML). The benchmark comprises ten different queries combining relational and linear algebra expressions.

Twitter Dataset Preparation. We obtain from Twitter API [24] 16GB of tweets (in

Table 5.7: LA pipelines used in micro-hybrid benchmark.

No.	Expression
P3.1	$\texttt{rowSums}(X\mathbf{M}) + (uv^T + \mathbf{N}^T)v$
P3.2	$\mathbf{u}\texttt{colSums}((X\mathbf{M})^T)\mathbf{+N}$
P3.3	$((\mathbf{N}+X)v)\texttt{colSums}(\mathbf{M})$
P3.4	$\texttt{sum}(C+\mathbf{N}\texttt{rowSums}(X\mathbf{M})v)$
P3.5	$u\texttt{colSums}(\mathbf{M}X)\mathbf{+N}$
P3.6	$\texttt{rowSums}((\mathbf{M}X)^T) + (uv^T + \mathbf{N})v$
P3.7	$X\mathbf{N}u+\texttt{rowSums}((\mathbf{M})^T)$
P3.8	$\mathbf{N}\odot\texttt{trace}(C + v\texttt{colSums}(\mathbf{M}X)C)$
P3.9	$X\odot\texttt{sum}(\texttt{colSums}(\mathbf{C})^T\odot\texttt{rowSums}(\mathbf{M}))\mathbf{+N}$
P3.10	$\mathbf{N}\odot\ \texttt{sum}((X + C)\mathbf{M})$

JSON). We extract the structural parts of the dataset, which include user and tweet information, and store them in tables **User (U)** and **Tweet (T)**, linked via PK-FK relationships. The dataset is detailed in Section 5.2. The tables **User** and **Tweet** as well as **TweetJSON (TJ)** are stored in Parquet format.

Twitter Queries and Views. Queries consist of two parts: (*i*) RA preprocessing (Q_{RA}) and (*ii*) LA analysis (Q_{LA}). In the Q_{RA} part, queries construct two matrices: **M** and **N**. The matrix **M** (2M×12; *dense*) is the output of joining **T** and **U**. The construction of matrix **N** is described in Section 5.2. We fix the Q_{RA} part across all queries and vary the Q_{LA} part using a set of LA pipelines detailed below. In addition to the views defined in Section 5.2, we define three hybrid RA-LA materialized views: V_1^H, V_2^H and V_3^H, which store the result of applying `rowSums`, `colSums` and matrix multiplication operations over base tables **T** and **U** (viewed as matrices), respectively. Importantly, rewritings based on these views can only be found by *exploiting together LA properties* and *Morpheus's rewrites rules* (we incorporated them in our framework as a set of integrity constraints).

Discussion. After construction of **M** and **N** by the Q_{RA} part in SparkSQL, both matrices are loaded to SystemML to be used in the Q_{LA} part. Before evaluating an LA pipeline on **M** and **N**, all queries select **N**'s rows ($Q_{F_{LA}}$) with *filter-level* less than 4 (medium). For all of them,

HADAD rewrites the Q_{RA} part of $\mathbf{N}$ as described in Section 5.2.

Q1: For the Q_{LA} part, the query runs **P3.1** (see Table 5.7). HADAD applies several optimizations: (*i*) it rewrites $(uv^T + \mathbf{N}^T)v$ to $uv^Tv + \mathbf{N}^Tv$, where u and v are synthetic vectors of size 1000×1 and 2M×1. First, $\mathbf{N}$ is ultra sparse, which makes the computation of $\mathbf{N}^Tv$ extremely efficient. Second, SystemML evaluates uv^Tv efficiently in one go without intermediates, taking advantage of **`tsmm`** operator (discussed earlier) and **`mmchain`** for matrix multiply chains, where the best way to evaluate it computes v^Tv first, which results in a scalar, instead of computing uv^T, which results in a dense matrix of size 2M×1000. Alone, SystemML is unable to exploit its own efficient operations for lack of awareness of the LA property $Av + Bv = (A+B)v$; (*ii*) HADAD also rewrites $\texttt{rowSums}(X\mathbf{M})$ into XV_1^H, where $V_1^H = \texttt{rowSums}(\mathbf{T}) + K\,\texttt{rowSums}(\mathbf{U})$, by exploiting the property $\texttt{rowSums}(X\mathbf{M}) = X\,\texttt{rowSums}(\mathbf{M})$ together with Morpheus's rewrite rule: $\texttt{rowSums}(\mathbf{M}) \rightarrow \texttt{rowSums}(\mathbf{T}) + K\,\texttt{rowSums}(\mathbf{U})$[5] . The rewriting achieves up to 16.5× speedup.

Q2: The speedup of 2.5× comes from rewriting the pre-processing part and turning **P3.2**: $u\,\texttt{colSums}((X\mathbf{M})^T) + \mathbf{N}$ to $u(XV_1^H)^T + \mathbf{N}$, where u and X are synthetic matrices of size 2M×1 and 1000×2M, respectively. HADAD exploits $\texttt{colSums}((X\mathbf{M})^T) = (\texttt{rowSums}(X\mathbf{M}))^T$ and $\texttt{rowSums}(X\mathbf{M}) = X\,\texttt{rowSums}(\mathbf{M})$ together with Morpheus's rewrite rule: $\texttt{rowSums}(\mathbf{M}) \rightarrow \texttt{rowSums}(\mathbf{T}) + K\,\texttt{rowSums}(\mathbf{U})$. Both the rewriting and the original LA pipeline introduce an unavoidable large dense intermediate of size 2M×1000.

Q3: The query runs $((\mathbf{N}+X)v)\,\texttt{colSums}(\mathbf{M})$ in the Q_{LA} part, where dense matrices X and v are of size 2M×1000 and 1000×1, respectively. HADAD avoids the dense intermediate $(\mathbf{N}+X)$ by distributing the multiplication by v and realizing that the sparsity of $\mathbf{N}$ yields efficient multiplication. It also directly rewrites $\texttt{colSums}(\mathbf{M})$ to V_2^H, where $V_2^H = [\texttt{colSums}(\mathbf{T}), \texttt{colSums}(K)\mathbf{U}]$ by utilizing one of Morpheus's rewrite rules: $\texttt{colSums}(\mathbf{M}) \rightarrow [\texttt{colSums}(\mathbf{T}), \texttt{colSums}(\mathbf{K})\mathbf{U}]$. The rewriting $(\mathbf{N}v + Xv)\,V_2^H$ and including the rewriting of the Q_{RA} part of $\mathbf{N}$, achieves 9.2× speedup (Figure 5.13(a)-Q3).

[5] K is the unique sparse indicator matrix that captures the primary/foreign key dependencies between $\mathbf{T}$ and $\mathbf{U}$, introduced by Morpheus's rewrite rules [52].

Q4: In the Q_{LA} part, the query runs **P3.4**: $\texttt{sum}(C+\mathbf{N}\texttt{rowSums}(X\mathbf{M})v)$ (inspired by the COX proportional hazard regression model used in SystemML's test suite [6]) , where synthetic dense matrices C, X and v have size 2M×1000,1000×2M and 1×1000, respectively. HADAD (i) distributes the sum operation to avoid introducing the dense result of the addition (SystemML includes this rewrite rule but fails to apply it); (ii) rewrites $\texttt{rowSums}(X\mathbf{M})$ to XV_1^H, where $V_1^H = \texttt{rowSums}(\mathbf{T}) + K\texttt{rowSums}(\mathbf{U})$, by exploiting $\texttt{rowSums}(X\mathbf{M}) = X\texttt{rowSums}(\mathbf{M})$ together with Morpheus's rewrite rule $\texttt{rowSums}(\mathbf{M}) \rightarrow \texttt{rowSums}(\mathbf{T}) + K\texttt{rowSums}(\mathbf{U})$. The multiplication chain in the rewriting and the query is efficient since **N** is sparse. The rewriting of this query (including the rewriting of the Q_{RA} part of **N**) achieves 3.63× speedup (Figure 5.13(a)-Q4).

Q5: HADAD's rewriting speeds up this query by 2.3×. It rewrites $u\texttt{colSums}(\mathbf{M}X)$ in **P3.5** to uV_2^HX (see **Q3** for V_2^H's definition). The view is exploited by HADAD due to utilizing $\texttt{colSums}(\mathbf{M}X) = \texttt{colSums}(\mathbf{M})X$ together with the Morpheus's rewrite rule (shown in **Q3**) for pushing the colSums to the base tables (viewed as matrices) **U** and **T**. With the found rewriting, SystemML optimizes the matrix-chain multiplication by computing V_2^HX first, resulting in the intermediate 1×1000 matrix (X is 12×1000 dense matrix) instead of computing uV_2^H (u is a dense vector of size 2M×1), which results in the intermediate of size 2M×12. The rewriting and the original pipeline still introduce an unprevented dense intermediate of size 2M×1000.

Q6: The LA pipeline (**P3.6**) in this query is a variation of **P3.1**. In addition to distributing the multiplication of v, HADAD rewrites $\texttt{rowSums}((\mathbf{M}\,X)^T)$ to $(V_2^HX)^T$ (see **Q3** for V_2^H's definition) by exploiting $\texttt{rowSums}((\mathbf{M}X)^T) = \texttt{colSums}(\mathbf{M}X)^T$ and $\texttt{colSums}(\mathbf{M}X) = \texttt{colSums}(\mathbf{M})X$, all together with Morpheus's rewrite rule as illustrated in **Q3**. The obtained rewriting, including the rewriting of Q_{RA} part, achieves a speedup of 13.4×.

Q7: In the Q_{LA} part, the query runs the pipeline **P3.7**: $X\mathbf{N}u+\texttt{rowSums}((\mathbf{M})^T)$, where sizes of synthetic dense matrices X and u are 12×2M and 1000×1, respectively. HADAD rewrites the pipeline to $X\mathbf{N}u+(V_2^H)^T$. It discovers the view V_2^H by exploiting the property $\texttt{rowSums}((\mathbf{M})^T) = (\texttt{colSums}(\mathbf{M}))^T$ and Morpheus rewrite rule $\texttt{colSums}(\mathbf{M}) \rightarrow$[colSums(**T**),colSums(K)**U**]. Notice

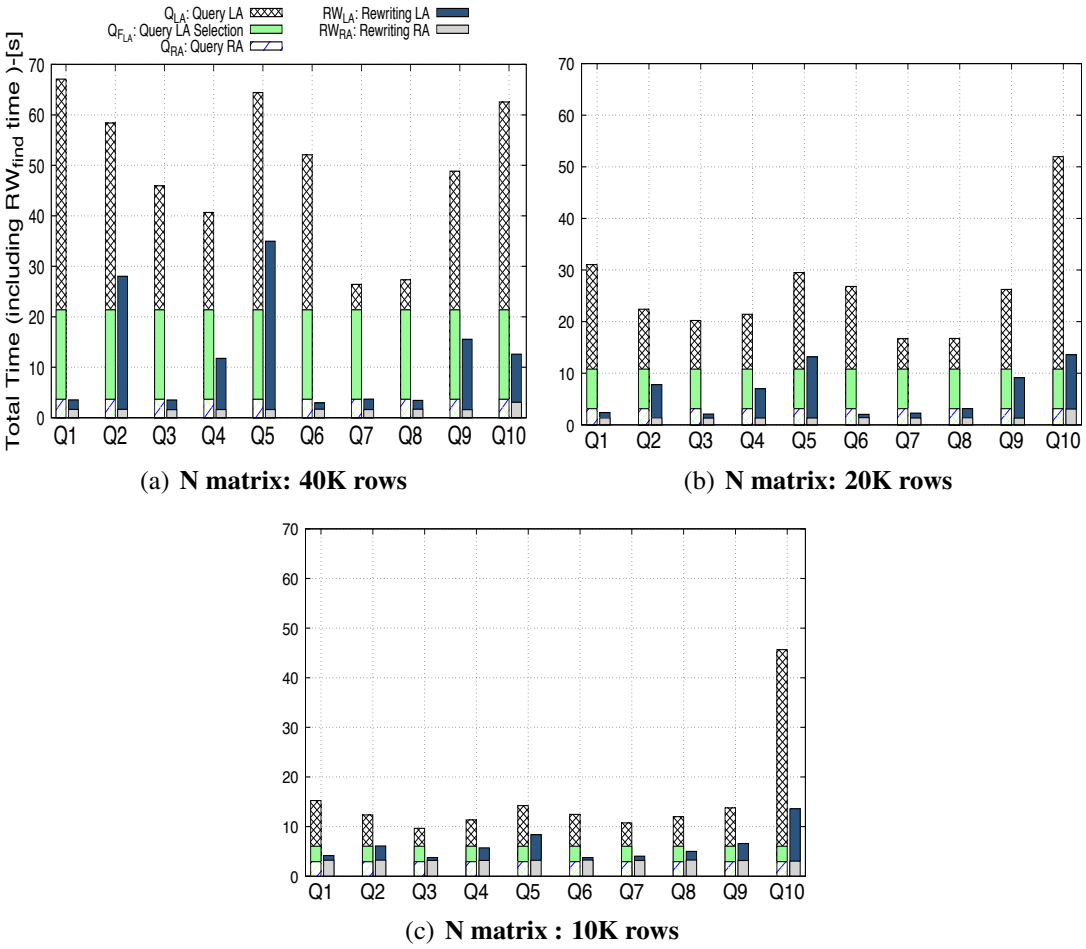
that SystemML computes $X\mathbf{N}u$ (in the rewriting and the original query) efficiently since $\mathbf{N}$ is ultra sparse.

Q8: The Q_{LA} part executes $\mathbf{N}\odot\texttt{trace}(C+v\texttt{colSums}(\mathbf{M}X)C)$, where the size of v, X and C are 20K×1, 12×20K and 20K×20K, respectively. First, HADAD distributes the `trace` operation (which SystemML does not apply) to avoid the dense intermediate addition. Second, HADAD enables the exploitation of view V_2^H (see **Q2**) by utilizing $\texttt{colSums}(\mathbf{M}X)$ = $\texttt{colSums}(\mathbf{M})X$. The resulting multiplication chain is optimized by the order $v((V_2^HX)C)$. The final element-wise multiplication with $\mathbf{N}$ is efficient since $\mathbf{N}$ is ultra-sparse. The combined Q_{RA} and Q_{LA} rewriting speeds up **Q8** by 5.94× (Figure 5.13(a)-Q8).

Q9: In addition to the rewriting of the Q_{RA} part, the speedup of 3× also attributes to turning $\texttt{sum}(\texttt{colSums}(C)^T\odot\texttt{rowSums}(\mathbf{M}))$ in **P3.9** to $\texttt{sum}(V_3^H)$, where $V_3^H=[C\mathbf{T},(CK)\mathbf{U}]$. The view V_3^H is utilized by exploiting the property $\texttt{sum}(C\mathbf{M})=\texttt{sum}(\texttt{colSums}(C)^T\odot\texttt{rowSums}(\mathbf{M}))$ together with Morpheus's rewrite rule: $C\mathbf{M}\rightarrow[C\mathbf{T},(CK)\mathbf{U}]$. The result of an element-wise multiplication with the dense matrix X in the rewriting and the original pipelines is a dene intermediate (see Figure 5.13(a)-Q9).

Q10: HADAD's rewrite the Q_{LA}: $\mathbf{N}\odot\texttt{sum}((X+C)\mathbf{M})$, where X and C are dense matrices of size 1000×1M, to $\mathbf{N}\odot(\texttt{sum}(X\mathbf{M})+\texttt{sum}(V_3^H))$. The $V_3^H=[C\mathbf{T},(CK)\mathbf{U}]$ is utilized by exploiting $(X+C)\mathbf{M}=X\mathbf{M}+C\mathbf{M}$ together with Morpheus's rewrite rule: $C\mathbf{M}\rightarrow[C\mathbf{T},(CK)\mathbf{U}]$. This optimization goes beyond SystemML's optimization since it does not consider distributing the multiplication of $\mathbf{M}$, which then enables exploiting the view and distributing the `sum` operation to avoid a dense intermediate. The obtained rewriting (with the rewriting of the Q_{RA} part of $\mathbf{N}$) achieves 3.91× (Figure 5.13(a)-Q8).

Twitter Varying Filter Selectivity. We repeat the benchmark for two different text-search selection conditions: "Trump" and "US election", obtaining 1M and 0.5M rows for $\mathbf{N}$, respectively (we adjust the size of the synthetic matrices for dimensional compatibility). As shown in Figures 5.13(b) and 5.13(c), the benefit of the combined Q_{RA} and Q_{LA} stage rewriting



(a) **N matrix: 40K rows** (b) **N matrix: 20K rows**

(c) **N matrix : 10K rows**

Figure 5.14: Results of micro-hybrid benchmark using MIMIC dataset.

increases with data size, remaining significant across the spectrum.

MIMIC Dataset Preparation. MIMIC dataset [88] comprises health data for patients. The total size of the dataset is 46.6 GB, and it consists of : (i) all charted data for all patients and their hospital admission information, ICU stays, laboratory measurements, caregivers' notes, and prescriptions; (ii) the role of caregivers (e.g., MD stands for "medical doctor"), (iii) lab measurements (e.g., ABG stands for "arterial blood gas") and (iv) diagnosis-related groups (DRG) codes descriptions. We use a subset of the dataset, which includes **Patients (P)**, **Admission (A)**, **Service (S)**, and **Callout (C)** tables. Using one-hot encoding, we convert tables' categorical features (columns) to numeric.

MIMIC Queries and Views. Similar to the Twitter's dataset benchmark, queries consist of two parts: (*i*) pre-processing (Q_{RA}) and (*ii*) analysis (Q_{LA}). In the Q_{RA} part, the queries construct two main matrices: **M** and **N**. The matrix **M** (40K×82; *dense*) is the output of joining tables **P** and **A**. The matrix **N** (40K×30K; *ultra-sparse*) is *patient-service* outcome (e.g., cancelled (1), serving (2), etc) matrix, constructed from joining **C** and **S** for all patients who are in "CCU" care unit. We fix the Q_{RA} part across all queries and vary the Q_{LA} part using a set of LA pipelines in Table 5.7. For views, we define three cross RA-LA materialized views: V_4^H, V_5^H and V_6^H, which store the result of applying `rowSums`, `colSums` and matrix multiplication operations over **P** and **A** base tables (matrices), respectively. These views can only be found by *exploiting together LA properties* and *Morpheus rewrites' rules* in the same fashion as we detailed in the Twitter's dataset experiment.

Discussion. The results exhibit similar trends to Twitter's benchmark. The first run of the benchmark is shown in Figure 5.14(a); both matrices **M** and **N** are loaded in SystemML to be used for the analysis part (varied using the set of pipelines in Table 5.7). Before evaluating an LA pipeline, all queries filter **N**'s rows, where the *outcome* is equal to 2. HADAD applies the same set of optimizations as described in Twitter's benchmark. For the second and third runs of the queries (see Figures 5.14(b) and 5.14(c)), we construct the **N** matrix for patients who are in "TSICU" and "MICU" care units, where **N**'s rows are 20K and 10K, respectively.

5.9.4 Experiments Takeaway

Our experiments with both real-life and synthetic datasets show performance gains across the board, for small rewriting overhead, in both pure LA and hybrid RA-LA settings. This is due to the fact that HADAD's rewriting power strictly subsumes that of optimizers of reference platforms like R, Numpy, TensorFlow, Spark (MLlib), SystemML and MorpheusR. Moreover, HADAD enables optimization where it was not previously feasible, such as across a cascade of unintegrated tools, e.g., SparkSQL for pre-processing followed by SystemML for analytics.

5.10 Limitations

In our LA relational-based reduction, we primarily focus on treating matrices and LA operations expressed over them as a block-box, where we capture important properties of LA operators via integrity constraints. However, it turns out that we can discover more interesting rewrites by opening this block-box, as we show in the example below:

Example 5.10.1 (LA Rewriting Follows From Standard Properties of Operations Over Numeric Values)**.** *Consider an LA expression $E = \texttt{sum}((X-UV)^2)$, which defines a typical loss function for approximating a matrix X with a low-rank matrix UV^T [137], and a view $B = XV$. After applying standard LA properties, which include expanding the square and distributing the sum, we obtain the following expression:*

$$\texttt{sum}(X^2)\text{-}2\texttt{sum}(X \odot UV^T) + \textit{sum}((UV^T)^2)$$

Interestingly, the expression $\texttt{sum}(X \odot UV^T)$ can be rewritten as $\texttt{sum}(U \odot XV)$, where we can match the view B. However, this rewrite does not seem to follow from just using standard LA properties, where we treat matrices as a black box. The equivalence of expressions $\texttt{sum}(X \odot UV^T)$ and $\texttt{sum}(U \odot XV)$ follows from standard properties of operations over numeric values.

$$\begin{aligned}
\texttt{sum}(X \odot UV^T) &= \sum_i \sum_k x_{ik} \circ (\sum_j u_{ij} \circ v_{kj}) \\
&= \sum_i \sum_k \sum_j x_{ik} \circ (u_{ij} \circ v_{kj}) \quad \textit{by distributivity of product over addition} \\
&= \sum_i \sum_j \sum_k u_{ij} \circ (v_{kj} \circ x_{ik}) \quad \textit{by associativity and commutativity of product} \\
&= \sum_i \sum_j u_{ij} \circ (\sum_k x_{ik} \circ v_{kj}) \quad \textit{by distributivity of product over addition} \\
&= \texttt{sum}(U \odot XV) \\
&= \texttt{sum}(U \odot B)
\end{aligned}$$

To obtain the rewriting in the example above, this requires extending HADAD rewriting capabilities to rewrite nested conjunctive queries with aggregation under integrity constraints where the body of views can have nested aggregation. Existing studies [54] consider rewriting conjunctive queries with aggregation in the absence of constraints.

Moreover, LA expressions can contain matrix slicing operations. Rewriting such operations requires complex index bounds analysis. Further, some complex analytics tasks can be naturally expressed by iterative LA programs, or LA expressions can be recursively rewritten given special constraints imposed over base matrices. We leave these extensions and investigations to future work. We plan to start from existing work on the safety and equivalence of index-based queries [106] and recent work on recursive query rewriting and optimization [138] and study their applicability in the presence of integrity constraints.

5.11 Conclusion

In this chapter, we presented HADAD, an extensible lightweight framework for optimizing hybrid analytics queries, based on the powerful intermediate abstraction of *a relational model with integrity constraints*. HADAD extends the capability of [31] (Chapter 4) with a reduction from LA (or LA view)-based rewriting to relational rewriting under constraints. It enables a full exploration of rewrites using a large set of LA operations, with no modification to the execution platform. Our experiments show significant performance gains on various LA and hybrid workloads across popular LA and cross RA-LA platforms.

5.12 Acknowledgement

This chapter contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana

Manolescu, which appeared in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2021). The dissertation author was the primary investigator of this paper.

This chapter contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries (Extended Version)" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. This paper is on ArXiv. The dissertation author was the primary investigator of this paper.

Chapter 6

Related Wrok

6.1 Related Work for ESTOCADA

Heterogeneous data integration is an old topic [102, 81, 79, 109] addressed mostly in a single-model (typically relational) setting (see Chapter 2), where cross-models query rewriting was therefore not an issue in these settings. The reference federated Garlic system [79] considered true multi-model settings, but the non-relational stores did not support their own declarative query language (they included, for instance, text documents, spreadsheets, etc.), being instead wrapped to provide an SQL API. Consequently, works on federated databases did not need cross-models rewriting.

The remark "one-size does not fit all" [130] has been recently revisited [104, 87] for heterogeneous stores. [99] uses a relational database as a "cache" for partial results of a MapReduce computation, while [100] considers view-based rewriting in a MapReduce setting. Unlike our work, these algorithms need the data, views, and queries to be in the same data model.

Polystores [68, 27] allow querying heterogeneous stores by grouping similar-model platform into "islands" and explicitly sending queries to one store or another; datasets can also be migrated by the users. Our Local-as-View (LAV) (see Chapter 3 for an overview of LAV systems)

approach is novel and, as we have shown, enables improving the performance of such stores also. The integration of "NoSQL" stores has been considered e.g., in [36] again in a Global-as-View (GAV) (see Chapter 3 for an overview of GAV systems) approach, without the benefits of LAV view-based rewriting.

Adaptive stores for a single data model have been studied e.g., in [82, 30, 56, 93, 92]; views have been also used in [127, 28] to improve the performance of a large-scale distributed relational store. The novelty of ESTOCADA here is to support multiple data models by relying on powerful query reformulation techniques under constraints.

Data exchange tools such as Clio [70, 78] allow migrating data between two different schemas of the same (typically relational and sometimes XML) model (and thus not focused on cross-models rewriting). View-based rewriting and view selection are grounded in the seminal works [102, 81]; the latter focuses on maximally contained rewritings, while we target exact query rewriting, which leads to very different algorithms. Further setting our work apart is the scale and usage of integrity constraints. Our pivot model recalls the ones described in [64, 109] but ESTOCADA generalizes these works by allowing multiple data models both at the application and storage level.

CloudMdsQL [95] is an integration language resembling QBT^{XM}, and our cross-models view-based rewriting approach could be easily adapted to use CloudMdsQL as its integration language, just like we adapted it to use BigDAWG's. The polystore engine supporting CloudMdsQL does not feature our cross-models view-based query rewriting functionality.

Works on publishing relational data as XML [73] followed the GAV paradigm thus did not raise the (cross-models) view-based rewriting problem we address here.

6.2 Related Work for HADAD

LA Systems/Libraries. SystemML [42] offers high-level R-like LA language, called Declarative Machine Learning (DML). The language offers physical data independence, where the users do not need to specify any data layout or formats. It applies some logical LA pattern-based rewrites and physical execution optimizations based on cost estimates for the latter. The system also supports sparse and dense matrices. SparkMLlib [112] provides LA operations and built-in function implementations of popular ML algorithms on Spark RDDs. The library supports sparse and dense matrices, but the user has to select this type explicitly.

R [17] and NumPy [16] are two of the most popular computing environments for statistical data analysis, widely used in academia and industry. They provide a high-level abstraction that can simplify the programming of numerical and statistical computations by treating matrices as first-class citizens and providing a rich set of built-in LA operations. However, LA properties in most of these systems remain unexploited, which makes them *miss opportunities to use their own highly efficient operators* (recall the scenario in Section 5.2). Our experiments (Section 5.9.2) show that evaluation of LA pipelines in these systems can be sped up by more than 10× using our rewriting that utilizes: (*i*) LA properties and (*ii*) materialized views.

Bridging the Gap: RA and LA. There has been a recent increase in research for unifying the execution of RA and LA expressions [105, 18, 94, 52]. A key limitation of these approaches is that *the semantics of LA operations remains hidden* behind built-in functions or UDFs, preventing performance-enhancing rewrites, as shown in Section 5.9.3.

SPORES [137], SPOOF [44], LARA [97] and RAVEN [91] are closer to our work. SPORES and SPOOF optimize LA expressions by converting them into RA, optimizing the latter, and then converting the result back to an (optimized) LA expression. They are restricted to a small set of selected LA operations (the ones that can only be expressed in RA), while we support significantly more (Section 5.5.1) and model properties, allowing to optimize with them.

LARA relies on a declarative domain-specific language for collections and matrices, which can enable optimization (i.e., *selection pushdown*) across the two algebraic abstractions. It heavily focuses on low-level optimization, such as exploiting the choice of data layouts for physical LA operators' optimization. RAVEN takes a step forward by providing intermediate representation to enable cross-optimization between ML and database operators and enhance in-database model inferencing performance. It transforms many classical ML models (e.g., decision tree) into equivalent neural networks (NN) to leverage highly optimized ML engines on CPU/GPU. [39, 48] study the expressive power of cross RA$-$LA [39] / LA [48] query languages, but do not address semantic optimizations / rewriting under integrity constraints and/or views.

In contrast to HADAD, as all aforementioned solutions do not reason with constraints, they provide no capabilities for *holistic* semantic query optimizations, including RA/LA views-based and LA pure rewritings; such optimizations can bring large performance savings, as shown in Section 5.9. We see HADAD as complementary to all of these platforms, on top of which it can be naturally and portably applied, providing an algorithm for RA/LA views- and constraint-based rewriting.

6.3 Acknowledgement

This chapter contains material from "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue" by Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis, which appears in Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019). The dissertation author was the primary investigator of this paper.

This chapter contains material from "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries" by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu, which appears in Proceedings of the 2019 International Conference on Management

of Data (SIGMOD 2021). The dissertation author was the primary investigator of this paper.

This chapter contains material from “HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries (Extended Version)” by Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. This paper is on ArXiv. The dissertation author was the primary investigator of this paper.

Chapter 7

Conclusion and Future Directions

In this dissertation, we first presented ESTOCADA (see Chapter 4), which is an extensible-lightweight framework designed to enable semantics query optimizations (including cross-models views-based and integrity constraint-based rewriting) in a polystores context. At the core of ESTOCADA lies a powerful intermediate abstraction of a *relational model with integrity constraints*, which we used to describe various data models' properties and cross model views. The novelty of ESTOCADA is to reduce the multi-data model rewriting problem to one within the internal relational pivot model, then transform rewritings obtained there into hybrid integration plans. For scalability, we presented a set of optimizations and extensions we contributed to the rewriting engine at the core of ESTOCADA, making it scalable in a polystore setting. Our experiments showed that ESTOCADA improves the performance in natural scenarios for both cross-models and single model user queries.

Then, in Chapter 5, we introduced HADAD, a framework that extends the benefits of semantic query optimizations introduced in ESTOCADA to LA computations, which are crucial for hybrid complex workloads. HADAD is the first framework that brings views-based rewriting under integrity constraints in the context of LA pipelines. It extends the intermediate abstraction used in ESTOCADA to enable reasoning about LA and cross RA-LA computations. This approach makes

it very easy to extend HADAD's semantic knowledge of LA operations or RA-LA rewrite rules by simply declaring appropriate constraints, with no need to change HADAD code, and ensures portability across pure LA and cross RA-LA platforms. Our extensive empirical evaluation showed significant performance gains on diverse LA and RA-LA workloads.

As a natural future work step in the space of hybrid systems, we are targeting the cost-based recommendation of optimal cross-models views placement, which suggests views to materialize within each underlying data store to obtain the best performance for a given *hybrid* workload. This task is related to the field of automatic view selection, with the significant difference that prior work was conducted in a single-model setting. View selection starts from an expected query workload and proposes a collection of views to materialize to best support the workload, subject to cost constraints (e.g., the available space). A majority of view selection results were developed for the relational model but mostly abstract away from integrity constraints and thus do not apply directly to our context, in which expressive constraints are prominent and views are expressed as a particular case of constraints. Extending these results to the presence of constraints is a fertile research proposition. Moreover, an interesting follow-up work would be designing efficient incremental maintenance of fragments as the native dataset is updated in a cross-models setting.

Appendix A

Appendix: ESTOCADA

A.1 Query Templates Example

```
QT0<PID,DOD,GENDER> :-
MIMIC(M),
ChildJ(M,PID,"PATIENTID","o"),
ChildJ(M,DOB,"DOB","o"),
ChildJ(M,DOD,"DOD","o"),
ChildJ(M,GENDER,"GENDER","o");

QT2<ITID>:-
LABITEMS(L),
ChildJ(L, ITID, "ITEMID","o"),
ChildJ(L,FLUID, "FLUID","o"),
  ValueJ(FLUID,"Blood");
```

```
QT1<PID,ITID,VAL,CHT> :-
MIMIC(M),
ChildJ(M,PID,"PATIENTID","o"),
ChildJ(M,A, "ADMISSIONS","o"),
ChildJ(A,LE,"LABEVENTS","o"),
ChildJ(LE,ITID,"ITEMID","o"),
ChildJ(LE,VAL, "VALUE","o"),
ChildJ(LE,CHT, "CHART","o"),
ChildJ(LE,FLAG, "FLAG","o"),
  ValueJ(FLAG,"abnormal");
```

Figure A.1: Query templates QT_0, QT_1 and QT_2.

```
Q<PID,DOB,GENDER,ITID,VAL,CHT> :-
MIMIC(M),
Child_J(M,PID,"PATIENTID","o"),
Child_J(M,DOB,"DOB","o"),
Child_J(M,DOD,"DOD","o"),
Child_J(M,GENDER,"GENDER","o"),
Child_J(M,A,"ADMISSIONS","o"),
Child_J(A,LE,"LABEVENTS","o"),
Child_J(LE,ITID, "ITEMID","o"),
Child_J(LE,VAL, "VALUE","o"),
Child_J(LE,CHT, "CHARTTIME","o"),
Child_J(LE,FLAG, "FLAG","o"),
  Value_J(FLAG,"abnormal");
LABITEMS(L),
Child_J(L, ITID, "ITEMID","o"),
Child_J(L,FLUID, "FLUID","o"),
    Value_J(FLUID,"Blood");
```

Figure A.2: Query Q produced from combining templates QT_0, QT_1 and QT_2.

```
SELECT  M.PATIENTID AS PID,
        M.DOB AS DOB,
        M.GENDER AS GENDER,
        L.ITEMID AS ITID,
        LE.VALUE AS VAL,
        LE.CHARTTIME AS CHT

FROM    MIMIC AS M, LABITEMS AS L,
        M.ADMISSIONS AS A,
        A.LABEVENTS AS LE
WHERE   LE.ITEMID=L.ITEMID AND
        LE.FLAG="abnormal" AND
        L.FLUID ="Blood"
```

Figure A.3: Query Q (in AsterixDB SQL++ syntax) produced from combining QT_0, QT_1 and QT_2.

A.2 Snippet of QBT^{XM} Query Language Grammar

⟨*Query*⟩ ::= ⟨*ForClause*⟩ ⟨*WhereClause*⟩? ⟨*ReturnClause*⟩

⟨*ForClause*⟩ ::= 'FOR' ⟨*Block*⟩ (',' ⟨*Block*⟩)*

⟨*WhereClause*⟩ ::= 'WHERE' ⟨*Condition*⟩

⟨*ReturnClause*⟩ ::= 'RETURN' ⟨*ReturnStatement*⟩

⟨*Block*⟩ ::= ⟨*Annotation*⟩ ':' '{'⟨*ModelBlock*⟩'}'

⟨*Annotation*⟩ ::= 'PR' | 'RK' | 'PJ' | 'AJ' ...

⟨*ModelBlock*⟩ ::= ⟨*PRModelBlock*⟩
| ⟨*RKModelBlock*⟩
| ⟨*PJModelBlock*⟩
| ⟨*AJModelBlock*⟩
| ...

⟨*Condition*⟩ ::= ⟨*Atom*⟩
| '('⟨*Condition*⟩')'
| ⟨*Condition*⟩ 'AND' ⟨*Condition*⟩

⟨*Atom*⟩ ::= ⟨*Term*⟩ '=' ⟨*Term*⟩

⟨*Term*⟩ ::= ⟨*Constant*⟩
| ⟨*Variable*⟩

⟨*Constant*⟩ ::= ⟨*StringConstant*⟩
| ⟨*NumericConstant*⟩

⟨*ReturnStatement*⟩ ::= ⟨*ModelConstructor*⟩
| ⟨*Variable*⟩ (',' ⟨*Variable*⟩)*

⟨*PRModelBlock*⟩ ::= ⟨*PR-Syntax-Specific*⟩

⟨*RKModelBlock*⟩ ::= ⟨*RK-Syntax-Specific*⟩

⟨*PJModelBlock*⟩ ::= ⟨*PJ-Syntax-Specific*⟩

⟨*AJModelBlock*⟩ ::= ⟨*AJ-Syntax-Specific*⟩

⟨*ModelConstructor*⟩ ::= ⟨*RKModelConstructor*⟩
| ⟨*PJModelConstructor*⟩
| ⟨*AJModelConstructor*⟩
| ...

⟨*RKModelConstructor*⟩ ::= ⟨*RK-Syntax-Specific-Constructor*⟩

⟨*PJModelConstructor*⟩ ::= ⟨*PJ-Syntax-Specific-Constructor*⟩

⟨*AJModelConstructor*⟩ ::= ⟨*AJ-Syntax-Specific-Constructor*⟩

Notes. Whenever mentioned ”syntax-specific,” the understanding is that the parser of the supported fragment of the language corresponding to the annotation will be called. Importantly, the comparison between variables of the same data model conforms to that data model. However, the comparisons between variables of distinct data models are only legal when both variables have string or numeric types.

A.3 Encoding of QBT^{XM} Views: V_1 and V_2

```
V1 Forward (IO) Constraints
MIMIC(M),
Child_J(M,PID,"PATIENTID","o"),
Child_J(M,A,"ADMISSIONS","o"),
Child_J(A,AID,"ADMISSIONID","o"),
Child_J(A,NE,"NOTEEVENTS","o"),
Child_J(A,REPORT,"REPORT","o")->
V1(d1),
Child_J(d1,PID,"PATIENTID","o"),
Child_J(d1,AID,"ADMISSIONID","o"),
Child_J(d1,REPORT,"REPORT","o");

V1 BACKWARD (OI) CONSTRAINTS:
V1(d1),
Child_J(d1,PID,"PATIENTID","o"),
Child_J(d1,AID,"ADMISSIONID","o"),
Child_J(d1,REPORT,"REPORT","o")->
MIMIC(M),
Child_J(M,PID,"PATIENTID","o"),
Child_J(M,A,"ADMISSIONS","o"),
Child_J(A,AID,"ADMISSIONID","o"),
Child_J(A,NE,"NOTEEVENTS","o"),
Child_J(NE,REPORT,"REPORT","o")
```

Figure A.4: View V_1 forward and backward constraints.

```
V2(PID,AID,ALOC,ATIME)->
MIMIC(M),
Child_J(M,PID,"PATIENTID","o"),
Child_J(M,A,"ADMISSIONS","o"),
Child_J(A,AID,"ADMISSIONID","o"),
Child_J(A,ALOC,"ADMISSIONLOC","o"),
Child_J(A,ATIME,"ADMISSIONTIME","o");
```

Figure A.5: View V_2 backward constraint.

A.4 Encoding of QBT^{XM} Query Q_1 and Decoding RWQ_1

```
Q1<PID,ALOC,ATIME,DRUG>:-
MIMIC(M),
Childƒ(M,PID,"PATIENTID","o"),
Childƒ(M,A,"ADMISSIONS","o"),
Childƒ(A,AID,"ADMISSIONID","o"),
Childƒ(A,NE,"NOTEEVENTS","o"),
Childƒ(NE,REPORT,"REPORT","o"),
  Value(REPORT,"contains-coronary artery"),
Childƒ(A,ALOC,"ADMISSIONLOC","o"),
Childƒ(A,ATIME,"ADMISSIONTIME","o"),
Childƒ(A,LE,"LABEVENTS","o"),
Childƒ(LE,FLAG,"FLAG","o"),
  Value(FLAG,"abnormal"),
Childƒ(LE,ITEMID,"LABITEMID","o"),
Childƒ(A,P,"PRESCRIPTIONS","o"),
Childƒ(P,DRUG,"DRUG","o"),
Childƒ(P,DRUGTYPE,"DRUGTYPE","o"),
  Value(DRUGTYPE,"additive"),
LABITEMS(L),
Childƒ(L, ITEMID, "ITEMID","o"),
Childƒ(L,CATEGORY, "CATEGORY","o"),
  Value(CATEGORY,"blood");
```

Figure A.6: Relational encoding of QBT^{XM} query Q_1.

```
FOR SJ:{V1/query?q=REPORT:'coronary artery'&
        fl=PID1:PATIENTID,AID1:ADMISSIONID},
    PR:{SELECT patientID     AS PID2,
               admissionID   AS AID2,
               admissionLoc  AS ALOC,
               admissionTime AS ATIME
        FROM   V2},
    PJ:{SELECT v3->'PATIENTID'   AS PID3,
               v3->'ADMISSIONID' AS AID3,
               D->'DRUG'         AS DRUG
        FROM V3 v3,
        jsonb_array_elements(v3->'PRESCRIPTIONS') D
        WHERE D->>'DRUGTYPE'='additive'}
WHERE PID1=PID2 AND
        AID1=AID2 AND
        PID2=PID3 AND
        AID2=AID2
RETURN PID1, AID1, ALOC, ALOC, DRUG
```

Figure A.7: Decoding of rewriting RWQ_1.

Appendix B

Appendix: HADAD

B.1 LA Operators Encoding Relations

Table B.1: The $\mathcal{VREM}$ schema (part 1).

Operator	Input(s)	Output	Relational Encoding
Multiplication	Matrices M and N	Matrix R	$\texttt{multi}_M(M,N,R)$
Hadamard Product	Matrices M and N	Matrix R	$\texttt{multi}_E(M,N,R)$
Matrix-Scalar Multiplication	Scalar s and Matrix M	Matrix R	$\texttt{multi}_{MS}(s,M,R)$
Addition	Matrices M and N	Matrix R	$\texttt{add}_M(M,N,R)$
Scalar Addition	Scalars s_1 and s_2	Scalar r	$\texttt{add}_S(s_1,s_2,r)$
Division	Matrices M and N	Matrix R	$\texttt{div}_M(M,N,R)$
Transposition	Matrix M	Matrices R	$\texttt{tr}(M,R)$
Inversion	Matrix M	Matrices R	$\texttt{inv}_M(M,R)$
Scalar Inversion	Scalar s	Scalar r	$\texttt{inv}_S(s,r)$
Trace	Matrix M	Scalar s	$\texttt{trace}(M,s)$
Matrix Sum	Matrix M	Scalar s	$\texttt{sum}(M,s)$
Matrix mean	Matrix M	Scalar s	$\texttt{mean}(M,s)$
Matrix max	Matrix M	Scalar s	$\texttt{max}(M,s)$
Matrix min	Matrix M	Scalar s	$\texttt{min}(M,s)$
Row sum	Matrix M	Vector R	$\texttt{rowSums}(M,R)$
Column sum	Matrix M	Vector R	$\texttt{colSums}(M,R)$
Row Min	Matrix M	Vector R	$\texttt{rowMin}(M,R)$

Table B.2: The $\mathcal{VREM}$ schema (part 2).

Operator	Input(s)	Output	Relational Encoding
Column Min	Matrix M	Vector R	$\texttt{colMin}(M,R)$
Row Max	Matrix M	Vector R	$\texttt{rowMax}(M,R)$
Column Max	Matrix M	Vector R	$\texttt{colMax}(M,R)$
Row Mean	Matrix M	Vector R	$\texttt{rowMean}(M,R)$
Column Mean	Matrix M	Vector R	$\texttt{colMean}(M,R)$
Row Variance	Matrix M	Vector R	$\texttt{rowVar}(M,R)$
Column Variance	Matrix M	Vector R	$\texttt{colVar}(M,R)$
Determinant	Matrix M	Scalar s	$\texttt{det}(M,s)$
Exponential	Matrix M	Matrix R	$\texttt{exp}(M,R)$
Direct Product	Matrices M and N	Matrix R	$\texttt{product}_D(M,N,R)$
Direct Sum	Matrices M and N	Matrix R	$\texttt{sum}_D(M,N,R)$
Adjoints	Matrix M	Matrix R	$\texttt{adj}(M,R)$
Diagonal	Matrix M	Matrix R	$\texttt{diag}(M,R)$
Cholesky Decomposition	Matrix M	Matrix L	$\texttt{CHO}(M,L)$
QR Decomposition	Matrix M	Matrices Q and R	$\texttt{QR}(M,Q,R)$
LU Decomposition	Matrix M	Matrices L and U	$\texttt{LU}(M,L,U)$
Pvioted LU Decomposition	Matrix M	Matrices L, U and P	$\texttt{LUP}(M,L,U,P)$

B.2 Key Constraints of LA Encoding Relations

Table B.3: Key constraints of LA operators relations (part 1).

LA Encoding Relation	Key Constraint
$name(M,n)$	$\forall n,M_1,M_2\ name(M_1,n) \wedge name(M_2,n) \rightarrow M_1 = M_2$
$size(M,r,c)$	$\forall M,r_1,r_2,c_1,r_1\ size(M,r_1,c_1) \wedge size(M,r_2,c_2) \rightarrow r_1 = r_2 \wedge c_1 = c_2$
$\texttt{multi}_M(M,N,R)$	$\forall M,N,R_1,R_2\ \texttt{multi}_M(M,N,R_1) \wedge \texttt{multi}_M(M,N,R_2) \rightarrow R_1 = R_2$
$\texttt{multi}_E(M,N,R)$	$\forall M,N,R_1,R_2\ \texttt{multi}_E(M,N,R_1) \wedge \texttt{multi}_E(M,N,R_2) \rightarrow R_1 = R_2$
$\texttt{multi}_{MS}(s,M,R)$	$\forall s,M,R_1,R_2\ \texttt{multi}_{MS}(s,M,R_1) \wedge \texttt{multi}_{MS}(s,M,R_2) \rightarrow R_1 = R_2$
$\texttt{add}_M(M,N,R)$	$\forall M,N,R_1,R_2\ \texttt{add}_M(M,N,R_1) \wedge \texttt{add}_M(M,N,R_2) \rightarrow R_1 = R_2$
$\texttt{add}_S(s_1,s_2,r)$	$\forall s_1,s_2,r_1,r_2\ \texttt{add}_S(s_1,s_2,r_1) \wedge \texttt{add}_S(s_1,s_2,r_2) \rightarrow r_1 = r_2$
$\texttt{div}_M(M,N,R)$	$\forall M,N,R_1,R_2\ \texttt{div}_M(M,N,R_1) \wedge \texttt{div}_M(M,N,R_2) \rightarrow R_1 = R_2$
$\texttt{tr}(M,R)$	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{tr}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{inv}_M(M,R)$	$\forall M,R_1,R_2\ \texttt{inv}_M(M,R_1) \wedge \texttt{inv}_M(M,R_2) \rightarrow R_1 = R_2$
$\texttt{inv}_S(s,r)$	$\forall s,r_1,r_2\ \texttt{inv}_S(s,r_1) \wedge \texttt{inv}_S(s,r_2) \rightarrow r_1 = r_2$
$\texttt{trace}(M,s)$	$\forall M,s_1,s_2\ \texttt{trace}(M,s_1) \wedge \texttt{trace}(M,s_2) \rightarrow s_1 = s_2$
$\texttt{sum}(M,s)$	$\forall M,s_1,s_2\ \texttt{sum}(M,s_1) \wedge \texttt{sum}(M,s_2) \rightarrow s_1 = s_2$
$\texttt{mean}(M,s)$	$\forall M,s_1,s_2\ \texttt{mean}(M,s_1) \wedge \texttt{mean}(M,s_2) \rightarrow s_1 = s_2$
$\texttt{max}(M,s)$	$\forall M,s_1,s_2\ \texttt{max}(M,s_1) \wedge \texttt{max}(M,s_2) \rightarrow s_1 = s_2$
$\texttt{min}(M,s)$	$\forall M,s_1,s_2\ \texttt{min}(M,s_1) \wedge \texttt{min}(M,s_2) \rightarrow s_1 = s_2$
$\texttt{rowSums}(M,R)$	$\forall M,R_1,R_2\ \texttt{rowSums}(M,R_1) \wedge \texttt{rowSums}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{colSums}(M,R)$	$\forall M,R_1,R_2\ \texttt{colSums}(M,R_1) \wedge \texttt{colSums}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{rowMin}(M,R)$	$\forall M,R_1,R_2\ \texttt{rowMin}(M,R_1) \wedge \texttt{rowMin}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{colMin}(M,R)$	$\forall M,R_1,R_2\ \texttt{colMin}(M,R_1) \wedge \texttt{colMin}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{rowMax}(M,R)$	$\forall M,R_1,R_2\ \texttt{rowMax}(M,R_1) \wedge \texttt{rowMax}(M,R_2) \rightarrow R_1 = R_2$

Table B.4: Key constraints of LA encoding relations (part 2).

LA Encoding Relation	Key Constraint
$\texttt{colMax}(M,R)$	$\forall M,R_1,R_2\ \texttt{colMax}(M,R_1) \wedge \texttt{colMax}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{rowMean}(M,R)$	$\forall M,R_1,R_2\ \texttt{rowMean}(M,R_1) \wedge \texttt{rowMean}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{colMean}(M,R)$	$\forall M,R_1,R_2\ \texttt{colMean}(M,R_1) \wedge \texttt{colMean}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{rowVar}(M,R)$	$\forall M,R_1,R_2\ \texttt{rowVar}(M,R_1) \wedge \texttt{rowVar}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{colVar}(M,R)$	$\forall M,R_1,R_2\ \texttt{colVar}(M,R_1) \wedge \texttt{colVar}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{det}(M,s)$	$\forall M,s_1,s_2\ \texttt{diag}(M,s_1) \wedge \texttt{diag}(M,s_2) \rightarrow s_1 = s_2$
$\texttt{exp}(M,R)$	$\forall M,R_1,R_2\ \texttt{exp}(M,R_1) \wedge \texttt{exp}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{product}_D(M,N,R)$	$\forall M,N,R_1,R_2\ \texttt{product}_D(M,N,R_1) \wedge \texttt{product}_D(M,N,R_2) \rightarrow R_1 = R_2$
$\texttt{sum}_D(M,N,R)$	$\forall M,N,R_1,R_2\ \texttt{sum}_D(M,N,R_1) \wedge \texttt{sum}_D(M,N,R_2) \rightarrow R_1 = R_2$
$\texttt{adj}(M,R)$	$\forall M,R_1,R_2\ \texttt{adj}(M,R_1) \wedge \texttt{adj}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{diag}(M,R)$	$\forall M,R_1,R_2\ \texttt{diag}(M,R_1) \wedge \texttt{diag}(M,R_2) \rightarrow R_1 = R_2$
$\texttt{CHO}(M,L)$	$\forall M,L_1,L_2\ \texttt{CHO}(M,L_1) \wedge \texttt{CHO}(M,L_2) \rightarrow L_1 = L_2$
$\texttt{CHO}(M,L)$	$\forall M,L_1,L_2\ \texttt{CHO}(M,L_1) \wedge \texttt{CHO}(M,L_2) \rightarrow L_1 = L_2$
$\texttt{QR}(M,Q,R)$	$\forall M,Q_1,Q_2,R_1,R_2\ \texttt{QR}(M,Q_1,R_1) \wedge \texttt{QR}(M,Q_2,R_2) \rightarrow Q_1 = Q_2 \wedge R_1 = R_2$
$\texttt{LU}(M,L,U)$	$\forall M,L_1,L_2,U_1,U_2\ \texttt{LU}(M,L_1,U_1) \wedge \texttt{LU}(M,L_2,U_2) \rightarrow L_1 = L_2 \wedge U_1 = U_2$
$\texttt{LUP}(M,L,U,P)$	$\forall M,L_1,L_2,U_1,U_2,P_1,P_2\ \texttt{LU}(M,L_1,U_1,P_1) \wedge \texttt{LU}(M,L_2,U_2,P_2) \rightarrow L_1 = L_2 \wedge U_1 = U_2 \wedge P_1 = P_2$

B.3 Properties of LA Operations Encoded as Constraints

Table B.5: Properties of addition of matrices encoded as integrity constraints.

LA Property	Integrity Constraint
Addition of Matrices	
$M+N=N+M$	$\forall M,N,R\ \texttt{add}_M(M,N,R) \rightarrow \texttt{add}_M(N,M,R)$
$(M+N)+D=M+(N+D)$	$\forall M,N,D,R_1,R_2\quad \texttt{add}_M(M,N,R_1) \wedge \texttt{add}_M(R_1,D,R_2)$ $\rightarrow \exists R_3\ \texttt{add}_M(N,D,R_3) \wedge \texttt{add}_M(M,R_3,R_2)$
$c(M+N)=cM+cN$	$\forall c,M,N,R_1,R_2\quad \texttt{add}_M(M,N,R_1) \wedge \texttt{multi}_{MS}(c,R_1,R_2)$ $\rightarrow \exists R_3,R_4 \texttt{multi}_{MS}(c,M,R_3) \wedge \texttt{multi}_{MS}(c,N,R_4) \wedge$ $\texttt{add}_M(R_3,R_4,R_2)$
$(c+d)M=cM+dM$	$\forall c,d,s,M,R_1 add_S(c,d,s) \wedge \texttt{multi}_{MS}(s,M,R_1)$ $\rightarrow \exists R_2,R_3 \texttt{multi}_{MS}(c,M,R_2) \wedge \texttt{multi}_{MS}(d,M,R_3) \wedge$ $\texttt{add}_M(R_2,R_3,R_1)$
$M+0=M$	$\rightarrow \exists Zero(O)$ $\forall M,n,O\quad name(M,n) \wedge Zero(O) \rightarrow \texttt{add}_M(M,O,M)$ $\forall O\quad Zero(O) \rightarrow \texttt{add}_M(O,O,O)$

Table B.6: Properties of addition and transposition of matrices encoded as integrity constraints.

LA Property	Integrity Constraint
Multiplication of Matrices	
$(MN)D = M(ND)$	$\forall M,N,D,R_1,R_2\ \texttt{multi}_M(M,N,R_1) \wedge$ $\texttt{multi}_M(R_1,D,R_2)$ $\rightarrow \exists R_3 \texttt{multi}_M(N,D,R_3) \wedge \texttt{multi}_M(M,R_3,R_2)$
$M(N+D) = MN+MD$	$\forall M,N,D,R_1,R_2\ \ \texttt{add}_M(N,D,R_1) \wedge$ $\texttt{multi}_M(M,R_1,R_2)$ $\rightarrow \exists R_3,R_4\ \texttt{multi}_M(M,N,R_3) \wedge \texttt{multi}_M(M,D,R_4) \wedge$ $\texttt{add}_M(R_3,R_4,R_2)$
$(M+N)D = MD+MD$	$\forall M,N,D,R_1,R_2\ \texttt{add}_M(M,N,R_1) \wedge$ $\texttt{multi}_M(R_1,D,R_2) \rightarrow$ $\exists R_3,R_4 \texttt{multi}_M(M,D,R_3) \wedge \texttt{multi}_M(N,D,R_4) \wedge$ $\texttt{add}_M(R_3,R_4,R_2)$
$d(MN) = (dM)N$	$\forall d,M,N,R_1,R_2\ \ \texttt{multi}_M(M,N,R_1) \wedge$ $\texttt{multi}_{MS}(d,R_1,R_2) \rightarrow$ $\exists R_3 \texttt{multi}_{MS}(d,M,R_3) \wedge \texttt{multi}_M(R_3,N,R_2)$
$c(dM) = (cd)M$	$\forall c,dM,R_1,R_2$ $\texttt{multi}_{MS}(d,M,R_1) \wedge \texttt{multi}_{MS}(c,R_1,R_2) \rightarrow$ $\exists s\ multi_S(c,d,s) \wedge \texttt{multi}_{MS}(s,M,R_2)$
$I_k M = M = MI_z$	$\forall M,n,k,z\ name(M,n) \wedge size(M,k,z) \rightarrow$ $\exists I\ Identity(I) \wedge size(I,k,k)$ $\forall M,n,k,z\ name(M,n) \wedge size(M,k,z) \rightarrow$ $\exists I_1\ Identity(I_1) \wedge size(I,z,z)$ $\forall M,I_1,n,k,z\ name(M,n) \wedge size(M,k,z) \wedge$ $Identity(I) \wedge size(I_1,k,k) \rightarrow$ $\texttt{multi}_M(I,M,M)$ $\forall M,I_1,n,k,z\ name(M,n), \wedge size(M,k,z) \wedge$ $Identity(I) \wedge size(I_1,z,z) \rightarrow$ $\texttt{multi}_M(M,I,M)$
Transposition of Matrices	
$(MN)^T = N^T M^T$	$\forall M,N,R_1,R_2 \texttt{multi}_M(M,N,R_1) \wedge \texttt{tr}(R_1,R_2) \rightarrow$ $\exists R_3,R_4 \texttt{tr}(M,R_3) \wedge \texttt{tr}(N,R_4) \wedge \texttt{multi}_M(R_4,R_3,R_2)$
$(M+N)^T = M^T + N^T$	$\forall M,N,R_1,R_2\ \texttt{add}_M(M,N,R_1) \wedge \texttt{tr}(R_1,R_2)$ $\rightarrow \exists R_3,R_4 \texttt{tr}(M,R_3) \wedge \texttt{tr}(N,R_4) \wedge \texttt{add}_M(R_3,R_4,R_2)$
$(cM)^T = c(M)^T$	$\forall c,M,R_1,R_2\ \texttt{multi}_{MS}(c,M,R_1) \wedge \texttt{tr}(R_1,R_2)$ $\rightarrow \exists R_3 \texttt{tr}(M,R_3) \wedge \texttt{multi}_{MS}(c,R_3,R_2)$
$((M)^T)^T = M$	$\forall n,M\ name(M,n) \rightarrow \exists R_1 \texttt{tr}(M,R_1) \wedge \texttt{tr}(R_1,M)$

Table B.7: Properties of inverse, determinant and trace of matrices encoded as integrity constraints.

LA Property	Integrity Constraint
Inverses of Matrices	
$((M)^{-1})^{-1} = M$	$\forall n, M\ name(M,n)$ $\rightarrow \exists R_1 \texttt{inv}_M(M,R_1) \wedge \texttt{inv}_M(R_1,M)$
$(MN)^{-1} = N^{-1}M^{-1}$	$\forall M,N,R_1,R_2\ \texttt{multi}_M(M,N,R_1) \wedge \texttt{inv}_M(R_1,R_2)$ $\rightarrow \exists R_3,R_4 \texttt{inv}_M(M,R_3) \wedge \texttt{inv}_M(N,R_4) \wedge$ $\texttt{multi}_M(R_4,R_3,R_2)$
$((M)^T)^{-1} = ((M)^{-1})^T$	$\forall M,R_1,R_2 \texttt{tr}(M,R_1) \wedge \texttt{inv}_M(R_1,R_2)$ $\rightarrow \exists R_3 \texttt{inv}_M(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
$((kM))^{-1} = k^{-1}M^{-1}$	$\forall k,M,R_1,R_2\ \texttt{multi}_{MS}(k,M,R_1) \wedge \texttt{inv}_M(R_1,R_2)$ $\rightarrow \exists R_3,s\ inv_S(k,s) \wedge \texttt{inv}_M(M,R_3) \wedge$ $\texttt{multi}_{MS}(s,R_3,R_2)$
$M^{-1}M = I = MM^{-1}$	$\forall M,R_1,R_2\ \texttt{inv}_M(M,R_1) \wedge \texttt{multi}_M(R_1,M,R_2)$ $\rightarrow Identity(R_2)$ $\forall M,R_1,R_2\ \texttt{inv}_M(M,R_1) \wedge \texttt{multi}_M(M,R_1,R_2)$ $\rightarrow Identity(R_2)$
Determinant of Matrices	
$\texttt{det}(MN) = \texttt{det}(M) * \texttt{det}(N)$	$\forall M,N,R_1,d\ \texttt{multi}_M(M,N,R_1) \wedge \texttt{det}(R_1,d)$ $\rightarrow \exists d_1,d_2 \texttt{det}(M,d_1) \wedge \texttt{det}(N,d_2) \wedge$ $multi_S(d_1,d_2,d)$
$\texttt{det}((M)^T) = \texttt{det}(M)$	$\forall M,R_1,d\ \texttt{tr}(M,R_1) \wedge \texttt{det}(R_1,d) \rightarrow \texttt{det}(M,d)$
$\texttt{det}((M)^{-1}) = (\texttt{det}(M))^{-1}$	$\forall M,R_1,d\ \texttt{inv}_M(M,R_1) \wedge \texttt{det}(R_1,d) \rightarrow$ $\exists d_1 \texttt{det}(M,d_1) \wedge inv_S(d_1,d)$
$\texttt{det}((cM)) = c^k \texttt{det}(M)$	$\forall M,c,k,d\ size(M,k,k) \wedge \texttt{multi}_{MS}(c,M,d) \rightarrow$ $\exists s_1,s_2 pow(c,k,s_1) \wedge \texttt{det}(M,s_2) \wedge \texttt{multi}_S(s_1,s_2,d)$
$\texttt{det}((I)) = 1$	$\forall I_1,d\ Identity(I_1) \wedge \texttt{det}(I_1,d) \rightarrow d = 1$
Trace of Matrices	
$\texttt{trace}(M+N) = \texttt{trace}(M)$ $+\texttt{trace}(N)$	$\forall M,N,R_1,s_1\ \texttt{add}_M(M,N,R_1) \wedge$ $\texttt{trace}(R_1,s_1) \rightarrow \exists s_2,s_3\ \texttt{trace}(M,s_2) \wedge$ $\texttt{trace}(N,s_3) \wedge \texttt{add}_S(s_2,s_3,s_1)$
$\texttt{trace}(MN) = \texttt{trace}(NM)$	$\forall M,N,R_1,s_1\ \texttt{multi}_M(M,N,R_1) \wedge$ $\texttt{trace}(R_1,s_1) \rightarrow \exists R_2\ \texttt{multi}_M(N,M,R_2) \wedge$ $\texttt{trace}(R_2,s_1)$
$\texttt{trace}(M^T) = \texttt{trace}(M)$	$\forall M,R_1,s_1\ \ \texttt{tr}(M,R_1) \wedge \texttt{trace}(R_1,s_1) \rightarrow$ $\texttt{trace}(M,s_1)$

Table B.8: Properties of direct sum and exponential of matrices encoded as integrity constraints.

LA Property	Integrity Constraint
Direct Sum	
$(M \oplus N)+(C \oplus D) =$ $(M+C) \oplus (N+D)$	$\forall M,N,R_1,C,D,R_2,R_3\ \texttt{sum}_D(M,N,R_1)$ $\wedge \texttt{sum}_D(C,D,R_2) \wedge \texttt{add}_M(R_1,R_2,R_3) \rightarrow$ $\exists R_4,R_5 \texttt{add}_M(M,C,R_4) \wedge \texttt{add}_M(N,D,R_5)$ $\wedge \texttt{sum}_D(R_5,R_{3)}$
$(M \oplus N)(C \oplus D) = (MC) \oplus (ND)$	$\forall M,N,R_1,C,D,R_2,R_3\ \texttt{sum}_D(M,N,R_1)$ $\wedge \texttt{sum}_D(C,D,R_2) \wedge \texttt{multi}_M(R_1,R_2,R_3) \rightarrow$ $\exists R_4,R_5 \texttt{multi}_M(M,C,R_4) \wedge \texttt{multi}_M(N,D,R_5)$ $\wedge \texttt{sum}_D(R_5,R_{3)}$
Exponential of Matrices	
$\texttt{exp}(0) = I$	$\forall O,R_1\ \ Zero(O) \wedge \texttt{exp}(O,R_1) \rightarrow Identity(R_1)$
$\texttt{exp}(M^T) = \texttt{exp}(M)^T$	$\forall M,R_1,R_2 \texttt{tr}(M,R_1) \wedge \texttt{exp}(R_1,R_2) \rightarrow$ $\exists R_3 \texttt{exp}(M,R_3) \wedge \texttt{exp}(R_3,R_2)$

Table B.9: Matrix decompositions properties captured as integrity constraints (part1).

Decomposition Property	Integrity Constraint
Cholesky Decomposition (CD)	
$\texttt{CHO}(M) = L$ such that $M = LL^T$, where M is SPD	$\forall M\ type(M,\text{“}S\text{”}) \rightarrow \exists\ \ L_1 \exists L_2\ \texttt{CHO}(M,L_1) \wedge$ $type(L_1,\text{“}L\text{”}) \wedge \texttt{tr}(L_1,L_2) \wedge \texttt{multi}_M(L_1,L_2,M)$
QR Decomposition	
$QR(M) = [Q,R]$ such that $M = QR$	$\forall M \forall n \forall k\ name(M,n) \wedge size(M,k,k) \rightarrow\ \exists Q,R$ $QR(M,Q,R) \wedge type(Q,\text{“}O\text{”}) \wedge type(R,\text{“}U\text{”}) \wedge$ $\texttt{multi}_M(Q,R,M)$ $\forall Q\ type(Q,\text{“}O\text{”}) \rightarrow \exists I\ QR(Q,Q,I) \wedge identity(I)$ $\wedge \texttt{multi}_M(Q,I,Q)$ $\forall R\ type(R,\text{“}U\text{”}) \rightarrow \exists I\ QR(R,I,R) \wedge identity(I)$ $\wedge \texttt{multi}_M(I,R,R)$ $\forall I\ identity(I) \rightarrow QR(I,I,I)$

Table B.10: Matrix decompositions properties captured as integrity constraints (part2).

Decomposition Property	Integrity Constraint
LU Decomposition	
$\texttt{LU}(M) = [L,U]$ such that $M = LU$	$\forall M \forall n \forall k\ name(M,n) \wedge size(M,k,k) \rightarrow\ \exists L,U$ $\texttt{LU}(M,L,U) \wedge type(L,\text{“}L\text{”}) \wedge type(U,\text{“}U\text{”}) \wedge \texttt{multi}_M(L,U,M)$ $\forall L\ type(L,\text{“}L\text{”}) \rightarrow \exists I\ \texttt{LU}(L,L,I) \wedge identity(I) \wedge \texttt{multi}_M(L,I,L)$ $\forall U\ type(U,\text{“}U\text{”}) \rightarrow \exists I\ \texttt{LU}(U,I,U) \wedge identity(I)$ $\wedge \texttt{multi}_M(I,U,U)$ $\forall I\ identity(I) \rightarrow \texttt{LU}(I,I,I)$
Pivoted LU Decomposition	
$\texttt{LUP}(M) = [L,U,P]$ such that $PM = LU$, where M is a square matrix	$\forall M \forall n \forall k\ name(M,n) \wedge size(M,k,z) \rightarrow\ \exists L,U,P,R$ $\texttt{LUP}(M,L,U,P) \wedge type(L,\text{“}L\text{”}) \wedge type(U,\text{“}U\text{”}) \wedge type(P,\text{“}P\text{”}) \wedge \texttt{multi}_M(L,U,R) \wedge \texttt{multi}_M(P,M,R)$ $\forall L\ type(L,\text{“}L\text{”}) \wedge size(L,k,z) \rightarrow \exists I\ \texttt{LUP}(L,L,I,I) \wedge identity(I) \wedge \texttt{multi}_M(L,I,L) \wedge \texttt{multi}_M(I,L,L)$ $\forall U\ type(U,\text{“}U\text{”}) \rightarrow \exists I\ \texttt{LUP}(U,I,U,I) \wedge identity(I) \wedge \texttt{multi}_M(I,U,U)$ $\forall I\ identity(I) \rightarrow \texttt{LU}(I,I,I)$

B.4 SystemML Rewrite Rules Encoded as Constraints

Table B.11: UnnecessaryAggregates rewrite rules encoded as integrity constraints.

Rule	Integrity Constraint
sum(tr(M))→ sum(M)	$\forall M, R_1, s\ \texttt{tr}(M, R_1), \texttt{sum}(R_1, s) \rightarrow \texttt{sum}(M, s)$
sum(rowSums(M))→ sum(M)	$\forall M, R_1, s\ \texttt{rowSums}(M, R_1), \texttt{sum}(R_1, s) \rightarrow \texttt{sum}(M, s)$
sum(colSums(M))→ sum(M)	$\forall M, R_1, s\ \texttt{colSums}(M, R_1), \texttt{sum}(R_1, s) \rightarrow \texttt{sum}(M, s)$
min(tr(M))→ min(M)	$\forall M, R_1, s\ \texttt{tr}(M, R_1), \texttt{min}(R_1, s) \rightarrow \texttt{min}(M, s)$
min(rowMin(M))→ min(M)	$\forall M, R_1, s\ \texttt{rowMin}(M, R_1), \texttt{min}(R_1, s) \rightarrow \texttt{min}(M, s)$
min(colMin(M))→ min(M)	$\forall M, R_1, s\ \texttt{colMin}(M, R_1), \texttt{min}(R_1, s) \rightarrow \texttt{min}(M, s)$
max(tr(M))→ max(M)	$\forall M, R_1, s\ \texttt{tr}(M, R_1), \texttt{max}(R_1, s) \rightarrow \texttt{max}(M, s)$
max(colMax(M))→ max(M)	$\forall M, R_1, s\ \texttt{colMax}(M, R_1), \texttt{max}(R_1, s) \rightarrow \texttt{max}(M, s)$
max(rowMax(M))→ max(M)	$\forall M, R_1, s\ \texttt{rowMax}(M, R_1), \texttt{max}(R_1, s) \rightarrow \texttt{max}(M, s)$

Table B.12: PushdownUnaryAggTransposeOp rewrite rules encoded as integrity constraints.

Rule	Integrity Constraint
rowSums(tr(M))→tr(colSums(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{rowSums}(R_1,R_2) \rightarrow \exists R_3\ \texttt{colSums}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
colSums(tr(M))→tr(rowSums(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{colSums}(R_1,R_2 \rightarrow \exists R_3\ \texttt{rowSums}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
rowMean(tr(M))→tr(colMean(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{colMean}(R_1,R_2) \rightarrow \exists R_3\ \texttt{rowMean}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
colMean(tr(M))→tr(rowMean(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{rowMean}(R_1,R_2) \rightarrow \exists R_3\ \texttt{colMean}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
rowVar(tr(M))→tr(colVar(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{rowVar}(R_1,R_2) \rightarrow \exists R_3\ \texttt{colVar}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
colVar(tr(X))→tr(rowVar(X))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{colVar}(R_1,R_2) \rightarrow \exists R_3\ \texttt{rowVar}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
rowMax(tr(M))→tr(colMax(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{rowMax}(R_1,R_2) \rightarrow \exists R_3\ \texttt{colMax}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
colMax(tr(M))→tr(rowMax(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{colMax}(R_1,R_2) \rightarrow \exists R_3\ \texttt{rowMax}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
rowMin(tr(M))→tr(colMin(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{rowMin}(R_1,R_2) \rightarrow \exists R_3\ \texttt{colMin}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$
colMin(tr(M))→tr(rowMin(M))	$\forall M,R_1,R_2\ \texttt{tr}(M,R_1) \wedge \texttt{colMin}(R_1,R_2) \rightarrow \exists R_3\ \texttt{rowMin}(M,R_3) \wedge \texttt{tr}(R_3,R_2)$

Table B.13: SimplifyTraceMatrixMult rewrite rules encoded as integrity constraint.

Rule	Integrity Constraint
trace(MN)→sum(M⊙tr(N))	$\forall M,N,R_1,r\ \texttt{multi}_M(M,N,R_1) \wedge \texttt{trace}(R_1,r) \rightarrow \exists R_3,R_4 \texttt{tr}(N,R_3) \wedge \texttt{multi}_E(M,R_3,R_4) \wedge \texttt{sum}(R_4,r)$

Table B.14: SimplifySumMatrixMult rewrite rules encoded as integrity constraints.

Rule	Integrity Constraint
sum(MN) → sum(tr(colSums(M))⊙rowSums(N))	$\forall M,N,R_1,r\ \texttt{multi}_M(M,N,R_1) \wedge \texttt{sum}(R_1,r) \rightarrow$ $\exists R_2,R_3,R_4,R_5\ \texttt{colSums}(M,R_2) \wedge \texttt{tr}(R_2,R_3) \wedge$ $\texttt{rowSums}(N,R_4) \wedge \texttt{multi}_E(R_3,R_4,R_5)$ $\wedge \texttt{sum}(R_5,r)$
colSums(MN) → colSums(M)N	$\forall M,N,R_1,R_2\ \texttt{multi}_M(M,N,R_1) \wedge$ $\texttt{colSums}(R_1,R_2) \rightarrow$ $\exists R_3\ \texttt{colSums}(M,R_3) \wedge \texttt{multi}_M(R_3,N,R_2)$
rowSums(MN) → MrowSums(N)	$\forall M,N,R_1,R_2\ \texttt{multi}_M(M,N,R_1) \wedge$ $\texttt{rowSums}(R_1,R_2) \rightarrow$ $\exists R_3\ \texttt{rowSums}(N,R_3) \wedge \texttt{multi}_M(M,R_3,R_2)$

Table B.15: SimplifyColWiseAgg rewrite rules encoded as integrity constraints.

Rule	Integrity Constraint
colSums(M)→M if x is row vector	$\forall M,n,j\ name(M,n) \wedge size(M,\text{“1”},j)$ $\rightarrow \texttt{colSums}(M,M)$
colMean(M)→M if x is row vector	$\forall M,n,j\ name(M,n) \wedge size(M,\text{“1”},j)$ $\rightarrow \texttt{colSums}(M,M)$
colVar(M)→M if x is row vector	$\forall M,n,j\ name(M,n) \wedge size(M,\text{“1”},j)$ $\rightarrow \texttt{colVar}(M,M)$
colMax(M)→M if x is row vector	$\forall M,n,j\ name(M,n) \wedge size(M,\text{“1”},j)$ $\rightarrow \texttt{colMax}(M,M)$
colMin(M)→M if x is row vector	$\forall M,n,j\ name(M,n) \wedge size(M,\text{“1”},j)$ $\rightarrow \texttt{colMin}(M,M)$
colSums(M)→sum(M) if x is col vector	$\forall M,i,R_1\ \texttt{colSums}(M,R_1) \wedge size(M,i,\text{“1”})$ $\rightarrow \texttt{sum}(M,R_1)$
colMean(M)→mean(M) if x is col vector	$\forall M,i,R_1\ \texttt{colMean}(M,R_1) \wedge size(M,i,\text{“1”})$ $\rightarrow \texttt{mean}(M,R_1)$
colMax(X)→max(M) if x is col vector	$\forall M,i,R_1\ \texttt{colMax}(M,R_1) \wedge size(M,i,\text{“1”})$ $\rightarrow \texttt{max}(M,R_1)$
colMin(M)→min(X) if x is col vector	$\forall M,i,R_1\ \texttt{colMin}(M,R_1) \wedge size(M,i,\text{“1”})$ $\rightarrow \texttt{min}(M,R_1)$
colVar(M)→var(M) if x is col vector	$\forall M,i,R_1\ \texttt{colVar}(M,R_1) \wedge size(M,i,\text{“1”})$ $\rightarrow \texttt{var}(M,R_1)$

Table B.16: SimplifyRowWiseAgg rewrite rules encoded as integrity constraints.

Rule	Integrity Constraint
rowSums(M)→M, if x is col vector	$\forall M,n,i\ name(M,n) \wedge size(M,i,\text{“1”}) \rightarrow \texttt{rowSums}(M,M)$
rowMean(M)→M, if x is col vector	$\forall M,n,i\ name(M,n) \wedge size(M,i,\text{“1”}) \rightarrow \texttt{rowSums}(M,M)$
rowVar(M)→M, if x is col vector	$\forall M,n,i\ name(M,n) \wedge size(M,i,\text{“1”}) \rightarrow \texttt{rowVar}(M,M)$
rowMax(M)→M, if x is col vector	$\forall M,n,i\ name(M,n) \wedge size(M,i,\text{“1”}) \rightarrow \texttt{rowMax}(M,M)$
rowMin(M)→M, if x is col vector	$\forall M,n,i\ name(M,n) \wedge size(M,i,\text{“1”}) \rightarrow \texttt{rowMin}(M,M)$
rowSums(M)→sum(M), if x is row vector	$\forall M,j,R_1\ \texttt{rowSums}(M,R_1) \wedge size(M,\text{“1”},j) \rightarrow \texttt{sum}(M,R_1)$
rowMean(M)→mean(M), if x is row vector	$\forall M,j,R_1\ \texttt{rowMean}(M,R_1) \wedge size(M,\text{“1”},j) \rightarrow \texttt{mean}(M,R_1)$
rowMax(X)→max(M), if x is row vector	$\forall M,j,R_1\ \texttt{rowMax}(M,R_1) \wedge size(M,\text{“1”},j) \rightarrow \texttt{max}(M,R_1)$
rowMin(M)→min(X), if x is row vector	$\forall M,j,R_1\ \texttt{rowMin}(M,R_1) \wedge size(M,\text{“1”},j) \rightarrow \texttt{min}(M,R_1)$
rowVar(M)→var(M), if x is row vector	$\forall M,j,R_1\ \texttt{rowVar}(M,R_1) \wedge size(M,\text{“1”},j) \rightarrow \texttt{var}(M,R_1)$

Table B.17: PushdownSumOnAdd rewrite rules encoded as integrity constraints.

Rule	Integrity Constraint
$\texttt{sum}(M+N) \rightarrow \texttt{sum}(A)+\texttt{sum}(B)$	$\forall M,N,s\ \texttt{add}_M M,N,s_1) \wedge \texttt{sum}(M,s_1) \rightarrow \exists s_2,s_3\ \texttt{sum}(M,s_2) \wedge \texttt{sum}(N,s_3) \wedge add_s(s_2,s_3,s_1)$

Table B.18: ColSums/rowSumsMVMult rewrite rules encoded as integrity constraints.

Rule	Integrity Constraint
$\texttt{colSums}(M*N) \rightarrow \texttt{tr}(N)M$, if N is col vector	$\forall M,N,R_1,R_2,i\ size(N,i,\text{“1”}) \wedge \texttt{multi}_E(M,N,R_1) \wedge \texttt{colSums}(R_1,R_2) \rightarrow \exists R_3 \texttt{tr}(N,R_3) \wedge \texttt{multi}_M(R_3,M,R_2)$
$\texttt{rowSums}(M*N) \rightarrow M\texttt{tr}(N)$, if N is row vector	$\forall M,N,R_1,R_2,j\ size(N,\text{“1”},j) \wedge \texttt{multi}_E(M,N,R_1) \wedge \texttt{rowSums}(R_1,R_2) \rightarrow \exists R_3 \texttt{tr}(N,R_3) \wedge \texttt{multi}_M(M,R_3,R_2)$

B.5 Additional Results: $\mathcal{P}^{\neg Opt}$ and $\mathcal{P}^{Views}$ Pipelines Rewrites

Table B.19: $\mathcal{P}^{\neg Opt}$ pipelines rewrites (part 1).

No.	Rewrite	No.	Rewrite	No.	Rewrite
P1.1	$N^T M^T$	P1.2	$(A+B)^T$	P1.3	$(DC)^{-1}$
P1.4	$Av_1 + Bv_1$	P1.5	D	P1.6	$s_1\texttt{trace}(D)$
P1.7	A	P1.8	$(s_1+s_2)A$	P1.9	$\texttt{det}(D)$
P1.10	$\texttt{colSums}(A)^T$	P1.11	$\texttt{colSums}(A+B)^T$	P1.12	$\texttt{colSums}(M)N$
P1.13	$\texttt{sum}(\texttt{colSums}(M)^T * \texttt{rowSums}(N))$	P1.14	$\texttt{sum}(\texttt{colSums}(M)^T * \texttt{rowSums}(N))$	P1.15	$M(NM)$
P1.16	$\texttt{sum}(A)$	P1.17	$\texttt{det}(C) * \texttt{det}(D) * \texttt{det}(C)$	P1.18	$\texttt{sum}(A)$
P1.25	$M \odot (N^T/(M(NN^T)))$				

Table B.20: $\mathcal{P}^{\neg Opt}$ pipelines rewrites (part 2).

No.	Rewrite	No.	Rewrite	No.	Rewrite
P2.1	$\texttt{trace}(C) + \texttt{trace}(D)$	P2.2	$1/\texttt{det}(D)$	P2.3	$\texttt{trace}(D)$
P2.4	$s_1(A+B)$	P2.5	$1/\texttt{det}((C+D))$	P2.6	$(D^{-1}C)^T$
P2.7	C	P2.8	$\texttt{det}(C) * \texttt{det}(D)$	P2.9	$\texttt{trace}(DC) + \texttt{trace}(D)$
P2.10	$MrowSums(N)$	P2.11	$\texttt{sum}(A) + \texttt{sum}(B)$	P2.12	$\texttt{sum}(\texttt{colSums}(M)^T * \texttt{rowSums}(N))$
P2.13	$(M(NM))^T$	P2.14	$(M(NM))N$	P2.15	$\texttt{sum}(A)$
P2.16	$\texttt{trace}((DC)^{-1}) + \texttt{trace}(D)$	P2.17	$((((C+D)^{-1})^T)D$	P2.18	$\texttt{rowSums}(A+B)^T$
P2.25	$u_1 v_2^T v_2 - Xv_2$				

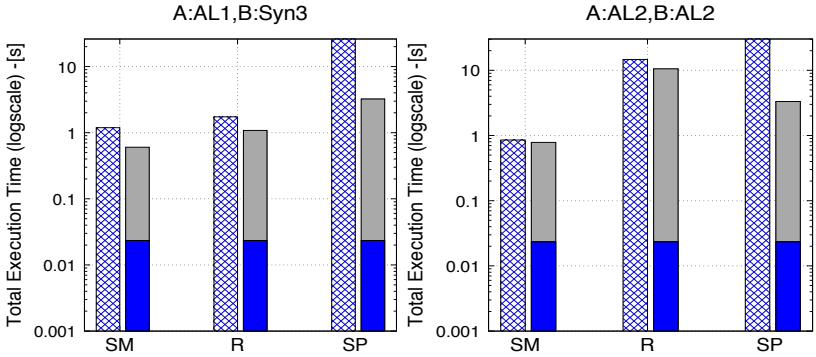
Table B.21: The set of views V_{exp}.

No.	Expression	No.	Expression	No.	Expression
V_1	$(D)^{-1}$	V_2	$(C^T)^{-1}$	V_3	NM
V_4	$u_1 v_2^T$	V_5	DC	V_6	$A+B$
V_7	C^{-1}	V_8	$C^T D$	V_9	$(D+C)^{-1}$
V_{10}	$\texttt{det}(CD)$	V_{11}	$\texttt{det}(DC)$	V_{12}	$(DC)^T$

Table B.22: $\mathcal{P}^{Views}$ pipelines rewrites.

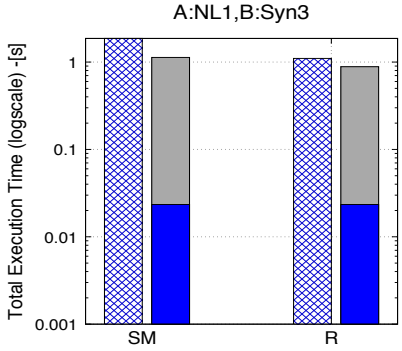
No.	Rewrite	No.	Rewrite	No.	Rewrite
P1.2	$(V_6)^T$	P1.3	V_7V_1	P1.4	$(V_6)v_1$
P1.11	$\texttt{colSums}(V_6)^T$	P1.15	$M(V_3)$	P1.17	$V_{10} * \texttt{det}(C)$
P1.19	V_2	P1.20	$\texttt{trace}(V_7)$	P1.21	$(C+V_1)^T$
P1.22	$\texttt{trace}(V_9)$	P1.24	$\texttt{trace}(V_1V_7)+$ $\texttt{trace}(D)$	P1.29	V_5CCC
P1.30	$V_3 \odot V_3R^T$	P2.2	$\texttt{det}(V_1)$	P2.4	$s_1(V6)$
P2.5	$\texttt{det}(V_9)$	P2.6	$(V_1C)^T$	P2.9	$\texttt{trace}(V_{12})+$ $\texttt{trace}(D)$
P2.11	$\texttt{sum}(V_6)$	P2.13	$(MV_3)^T$	P2.14	MV_3N
P2.16	$\texttt{trace}(V_7V_1)$ $+traceD)$	P2.17	$(V_9^T)D$	P2.18	$\texttt{rowSums}(V_6)^T$
P2.20	$(MV_3)^T$	P2.21	$V_1(V_1^T(D^Tv_1))$	P2.25	$V_4v_1 - Xv_1$
P1.23	$\texttt{det}((V_7V_1)+D)$	P2.26	$exp(V_9)$	P2.27	$V_9^TV_5$

B.6 Additional Results: $\mathcal{P}^{\neg Opt}$ Pipelines: Naïve Cost Model



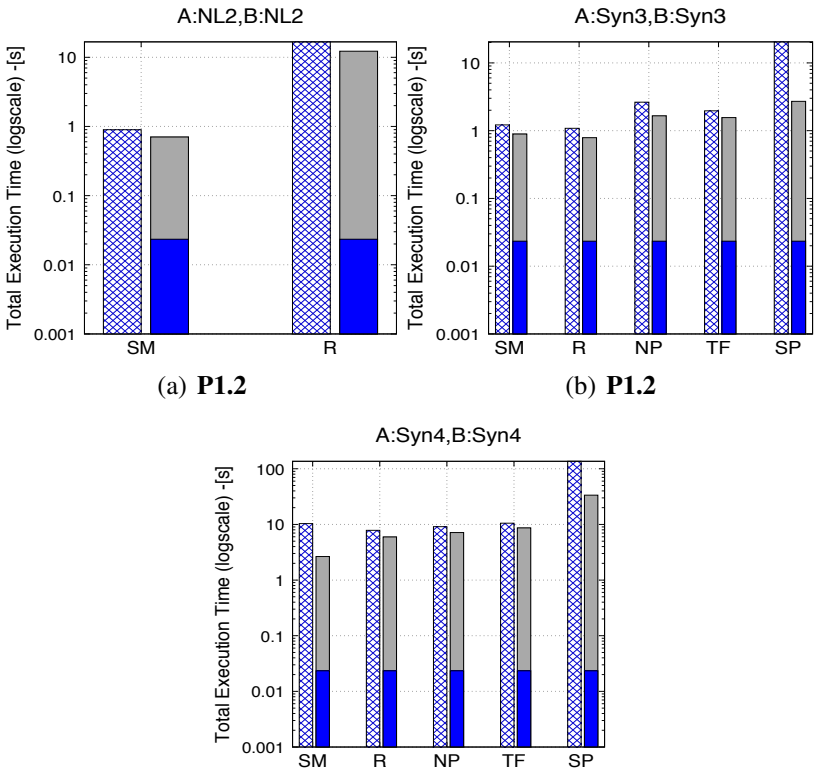
(a) **P1.2**

(b) **P1.2**



(c) **P1.2**

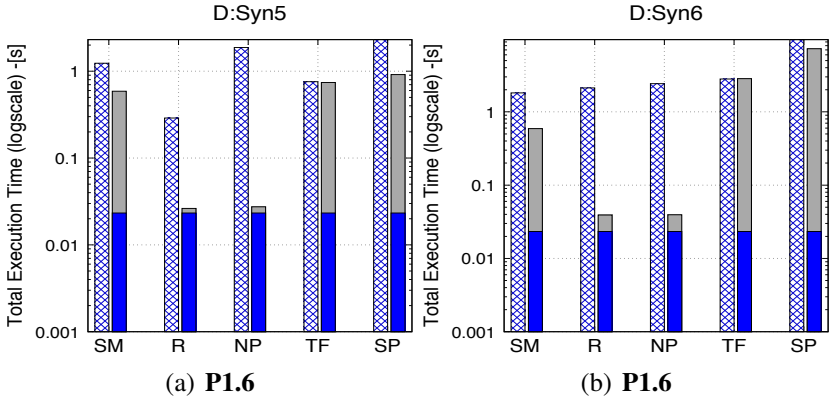
Figure B.1: P1.2 evaluation before and after rewriting.



(a) **P1.2** (b) **P1.2**

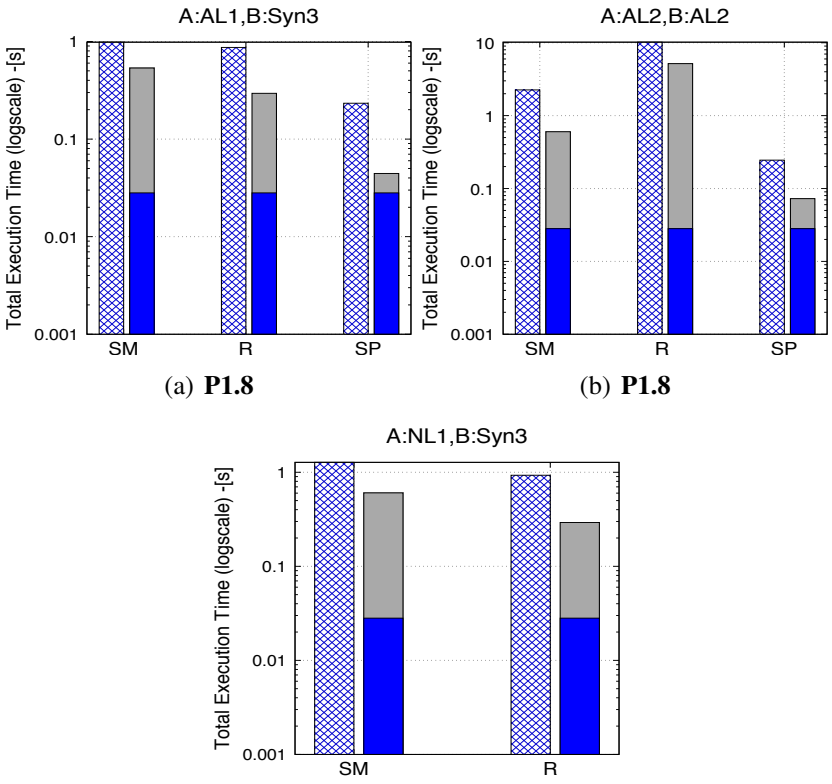
(c) **P1.2**

Figure B.2: P1.2 evaluation before and after rewriting.



(a) **P1.6** (b) **P1.6**

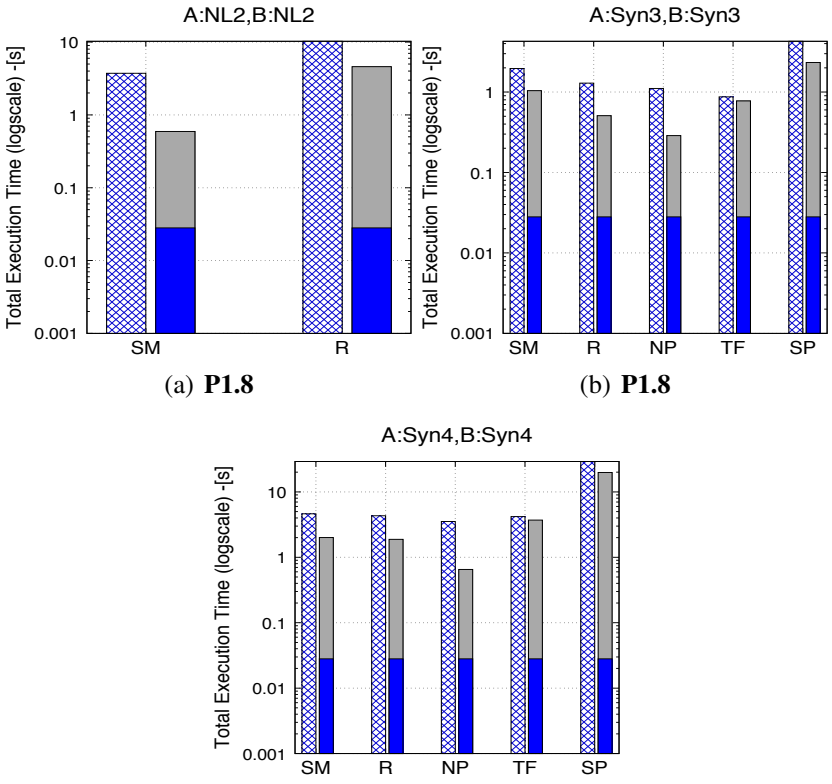
Figure B.3: P1.6 evaluation before and after rewriting.



(a) **P1.8** (b) **P1.8**

(c) **P1.8**

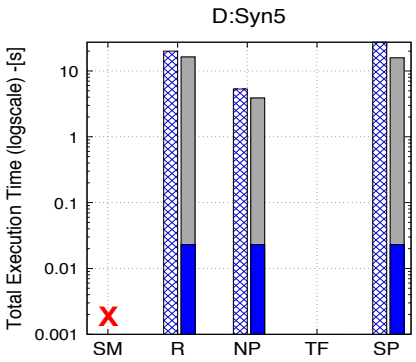
Figure B.4: P1.8 evaluation before and after rewriting.



(a) **P1.8** (b) **P1.8**

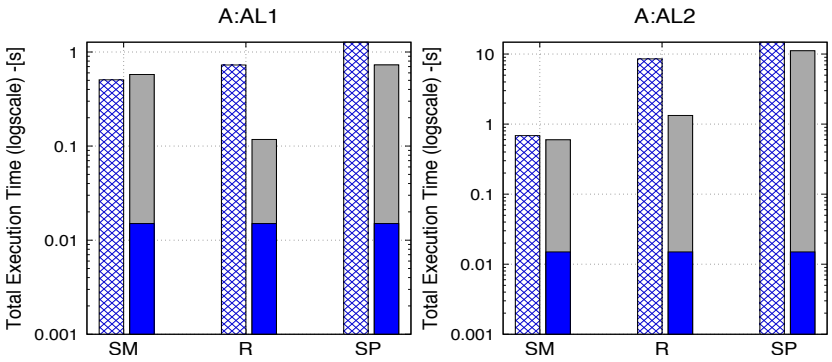
(c) **P1.8**

Figure B.5: P1.8 evaluation before and after rewriting.

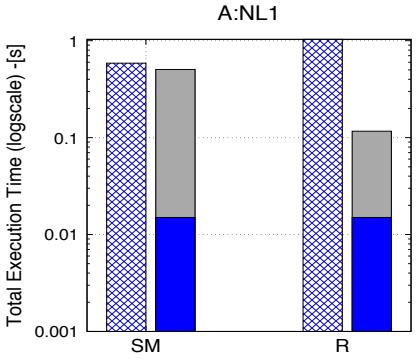


(a) **P1.9**

Figure B.6: P1.9 evaluation before and after rewriting.

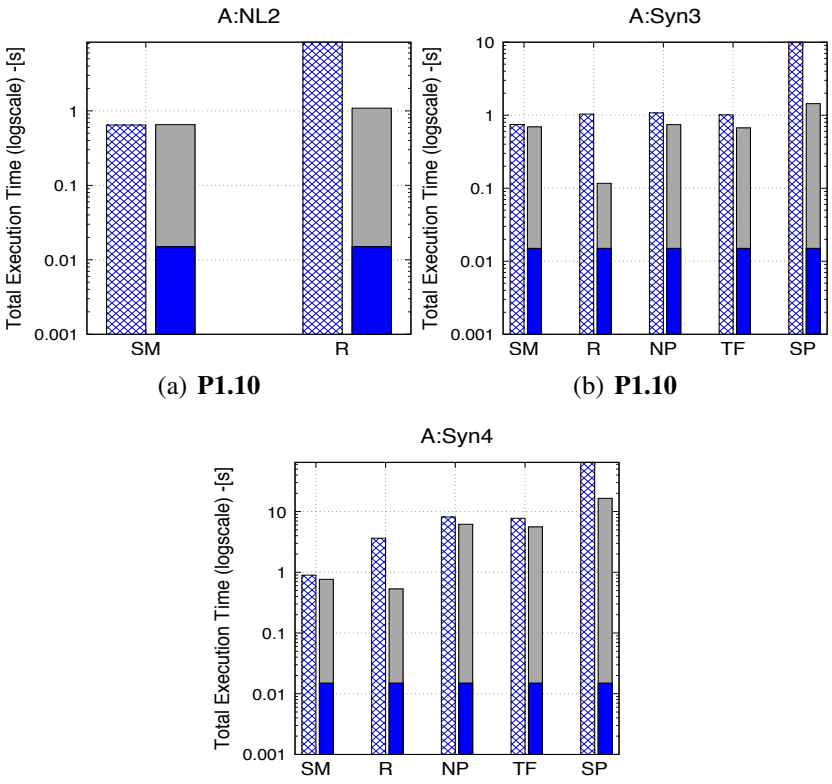


(a) **P1.10** (b) **P1.10**



(c) **P1.10**

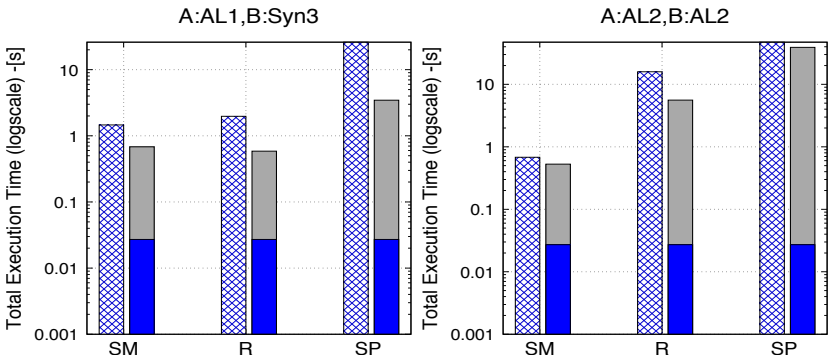
Figure B.7: P1.10 evaluation before and after rewriting.



(a) **P1.10** (b) **P1.10**

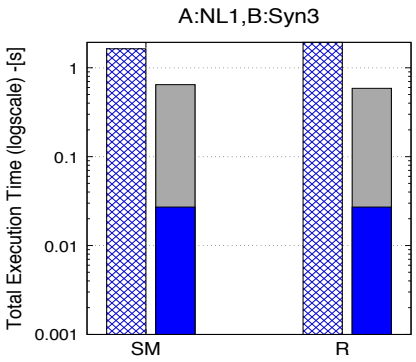
(c) **P1.10**

Figure B.8: P1.10 evaluation before and after rewriting.



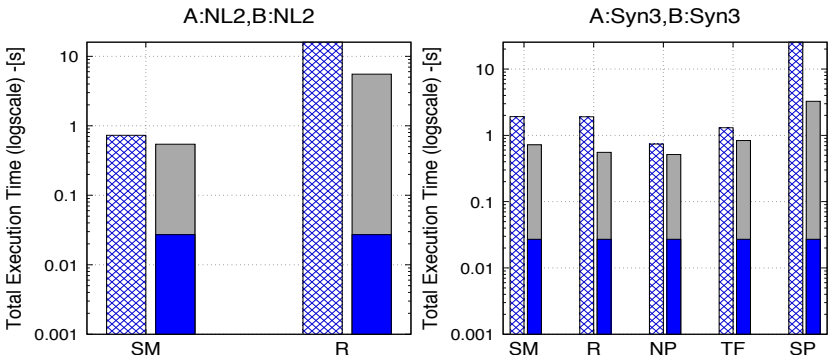
(a) **P1.11**

(b) **P1.11**

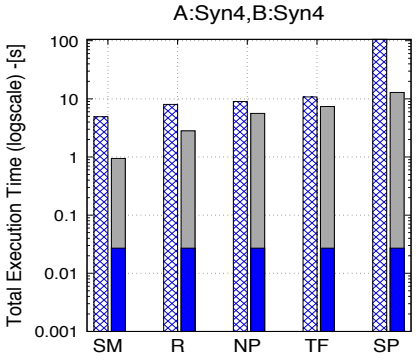


(c) **P1.11**

Figure B.9: P1.11 evaluation before and after rewriting.

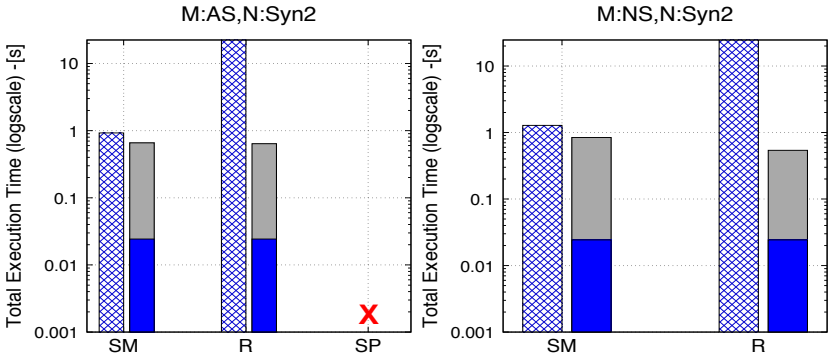


(a) **P1.11** (b) **P1.11**



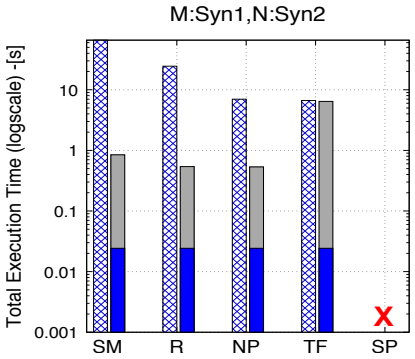
(c) **P1.11**

Figure B.10: P1.11 evaluation before and after rewriting.



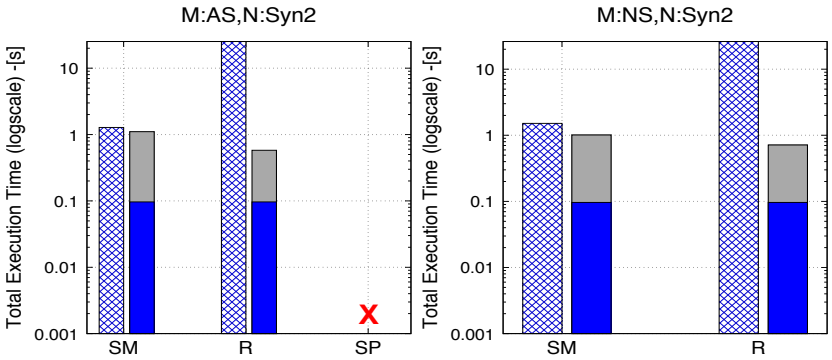
(a) **P1.12**

(b) **P1.12**



(c) **P1.12**

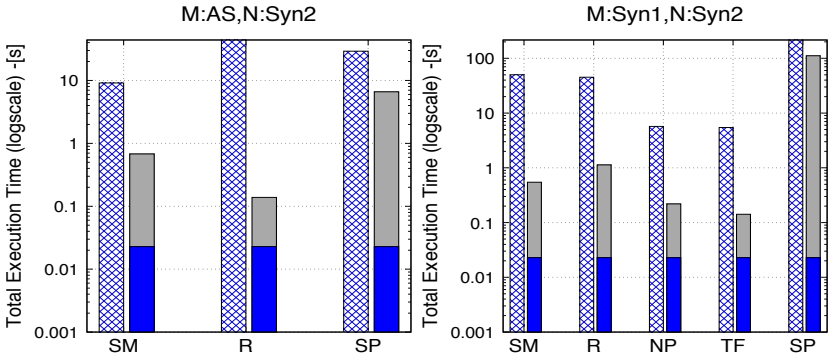
Figure B.11: P1.12 evaluation before and after rewriting.



(a) **P1.14**

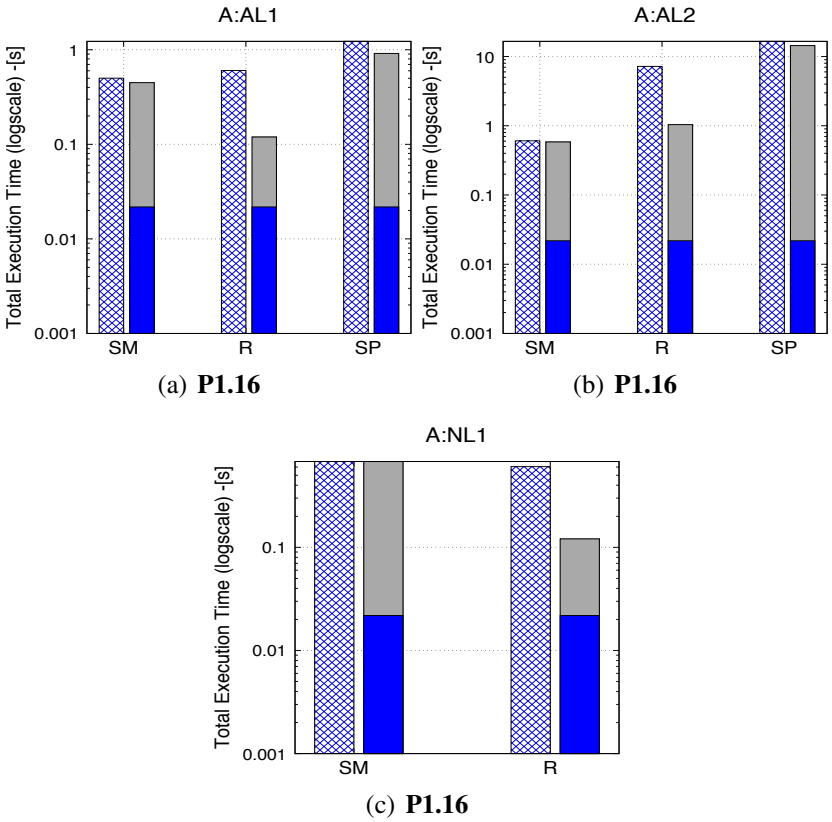
(b) **P1.14**

Figure B.12: P1.14 evaluation before and after rewriting.



(a) **P1.15** (b) **P1.15**

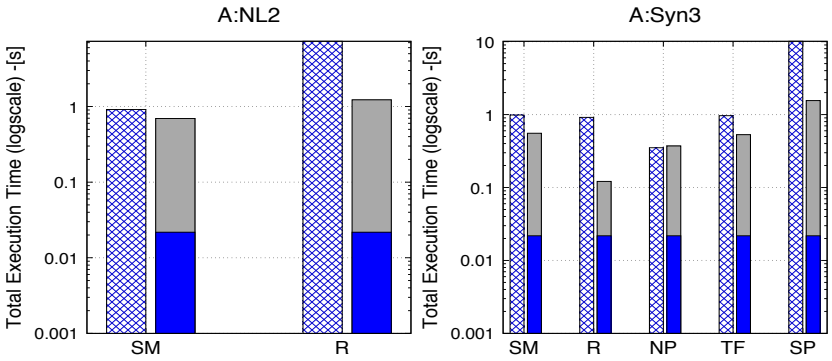
Figure B.13: P1.15 evaluation before and after rewriting.



(a) **P1.16** (b) **P1.16**

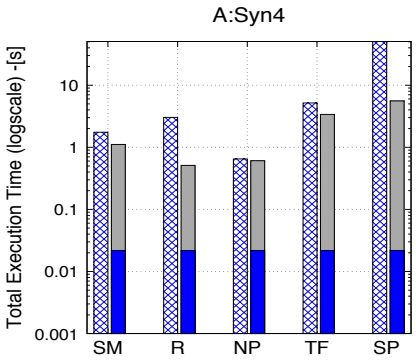
(c) **P1.16**

Figure B.14: P1.16 evaluation before and after rewriting.



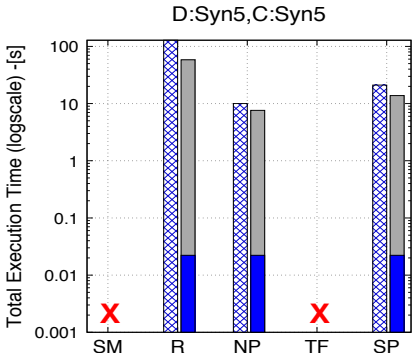
(a) **P1.16**

(b) **P1.16**



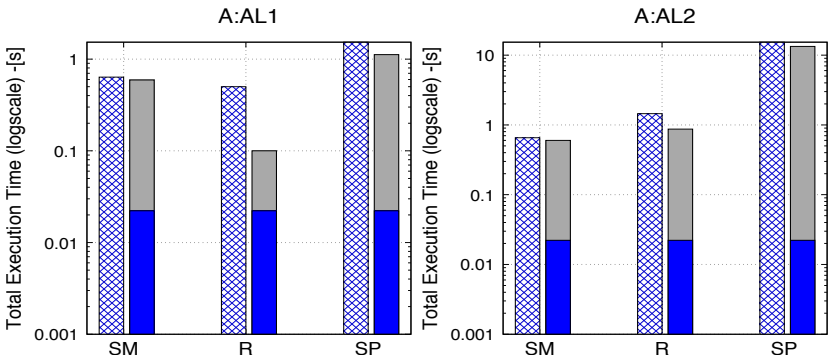
(c) **P1.16**

Figure B.15: P1.16 evaluation before and after rewriting.



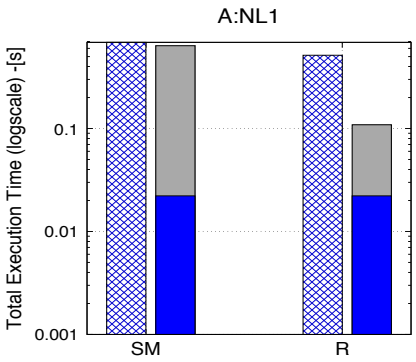
(a) **P1.17**

Figure B.16: P1.17 evaluation before and after rewriting.



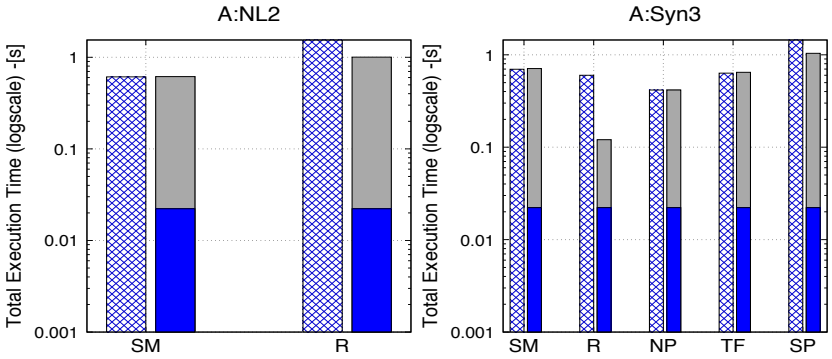
(a) **P1.18**

(b) **P1.18**



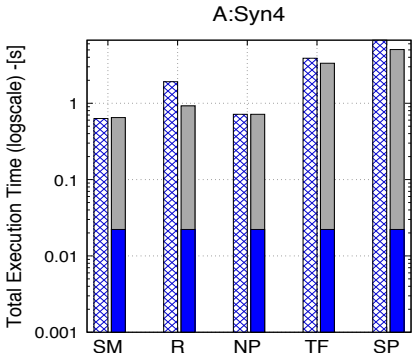
(c) **P1.18**

Figure B.17: P1.18 evaluation before and after rewriting.



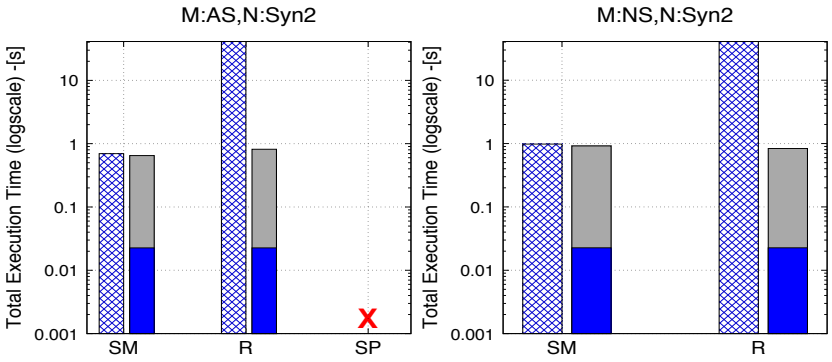
(a) **P1.18**

(b) **P1.18**



(c) **P1.18**

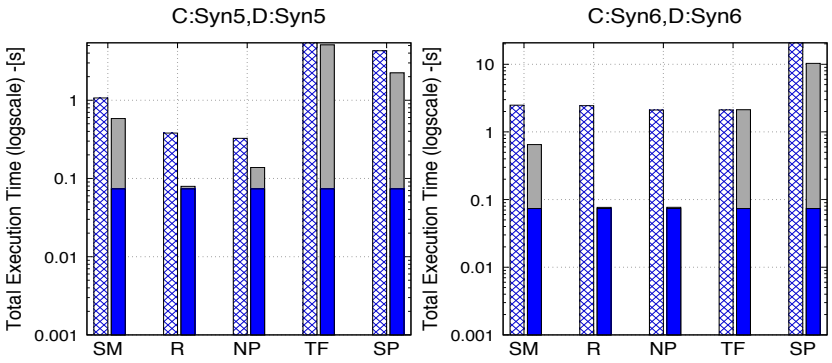
Figure B.18: P1.18 evaluation before and after rewriting.



(a) **P1.25**

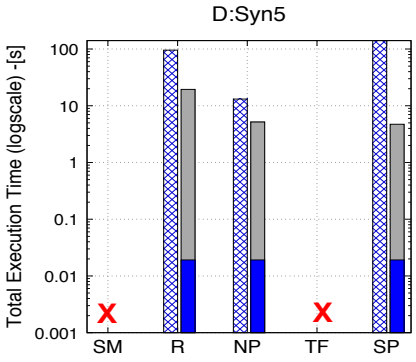
(b) **P1.25**

Figure B.19: P1.25 evaluation before and after rewriting.



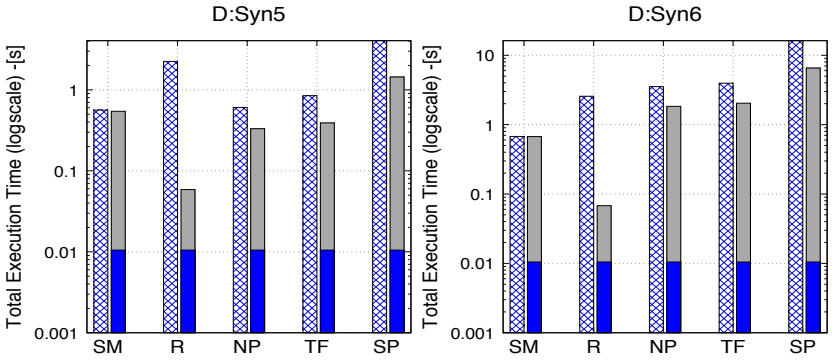
(a) **P2.1** (b) **P2.1**

Figure B.20: P2.1 evaluation before and after rewriting.



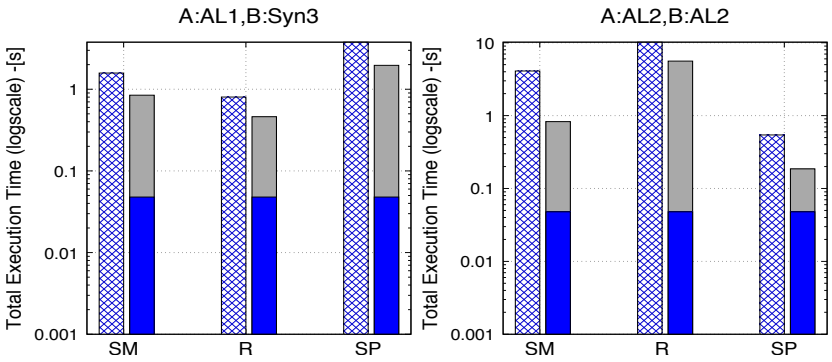
(a) **P2.2**

Figure B.21: P2.2 evaluation before and after rewriting.

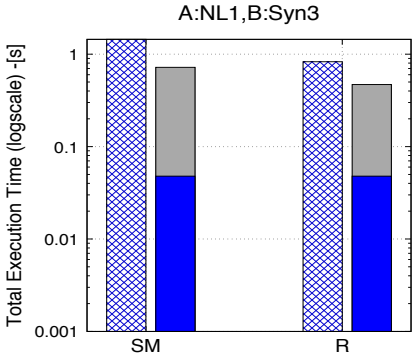


(a) **P2.3** (b) **P2.3**

Figure B.22: P2.3 evaluation before and after rewriting.

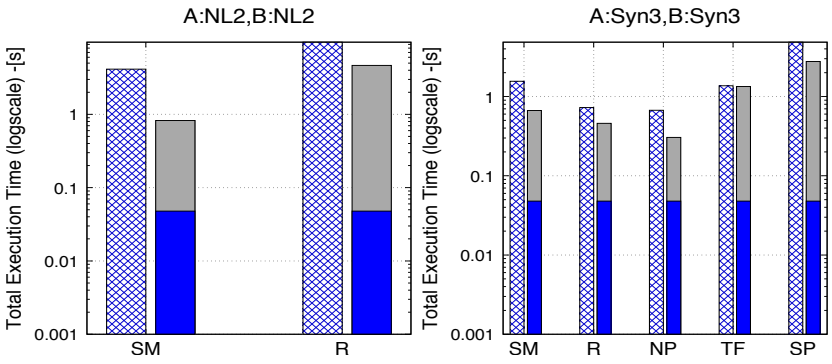


(a) **P2.4** (b) **P2.4**

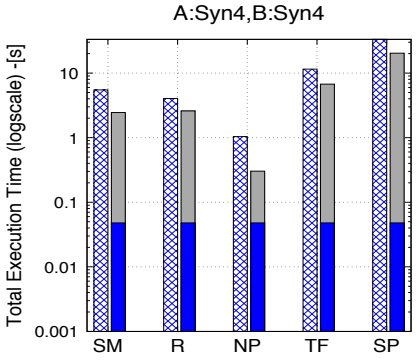


(c) **P2.4**

Figure B.23: P2.4 evaluation before and after rewriting.

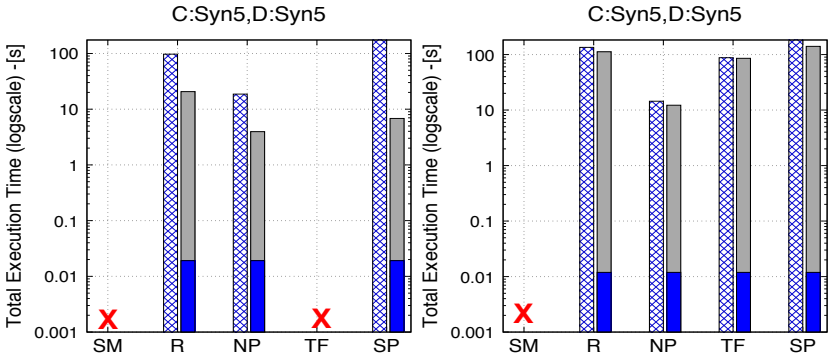


(a) **P2.4** (b) **P2.4**



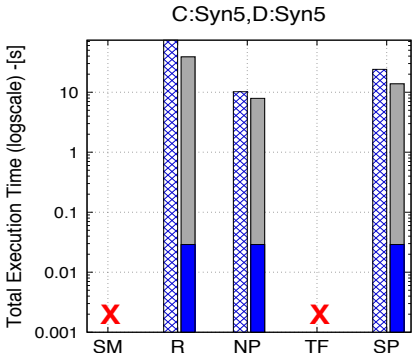
(c) **P2.4**

Figure B.24: P2.4 evaluation before and after rewriting.



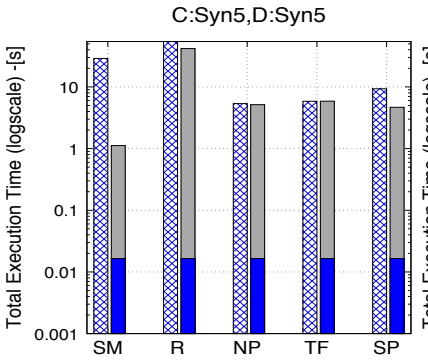
(a) **P2.5**

(b) **P2.6**

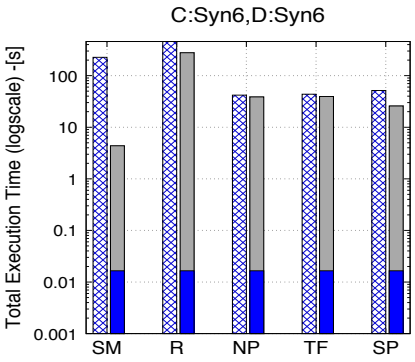


(c) **P2.8**

Figure B.25: P2.5, P2.6 and P2.8 evaluation before and after rewriting.

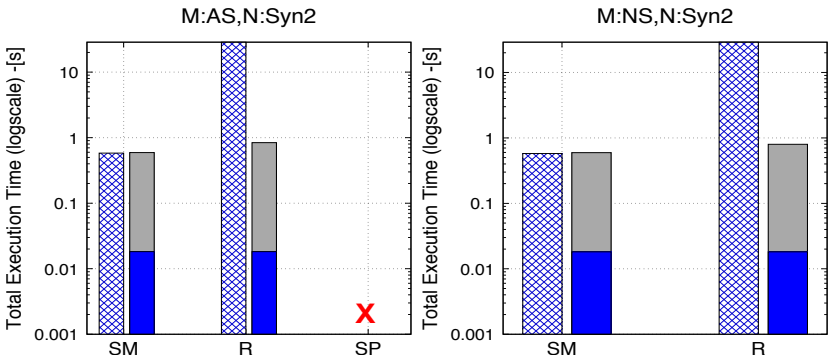


(a) **P2.9**



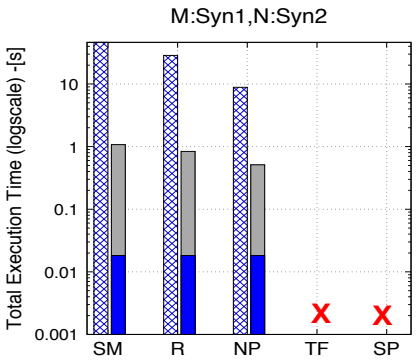
(b) **P2.9S**

Figure B.26: P2.9 evaluation before and after rewriting.



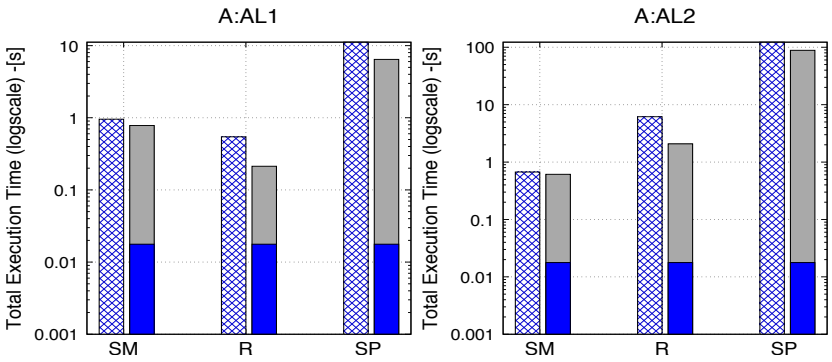
(a) **P2.10**

(b) **P2.10**



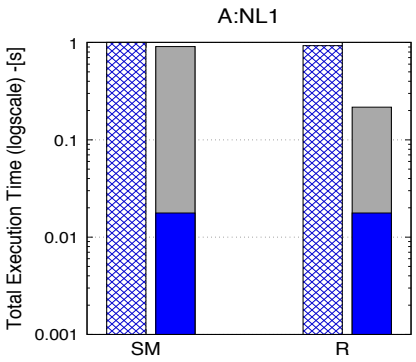
(c) **P2.10**

Figure B.27: P2.10 evaluation before and after rewriting.



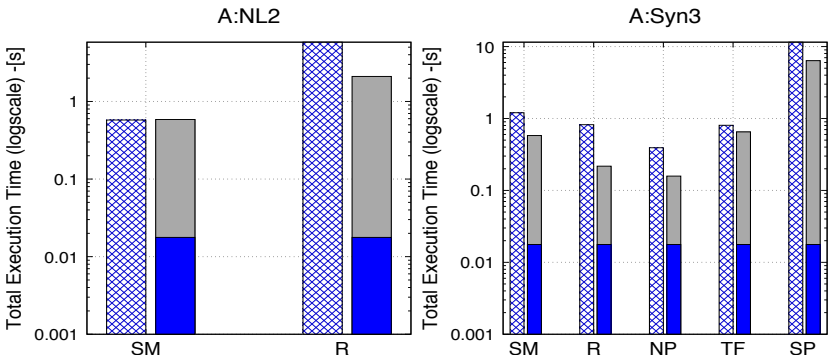
(a) **P2.11**

(b) **P2.11**

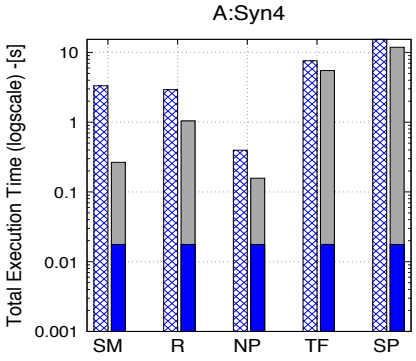


(c) **P2.11**

Figure B.28: P2.11 evaluation before and after rewriting.

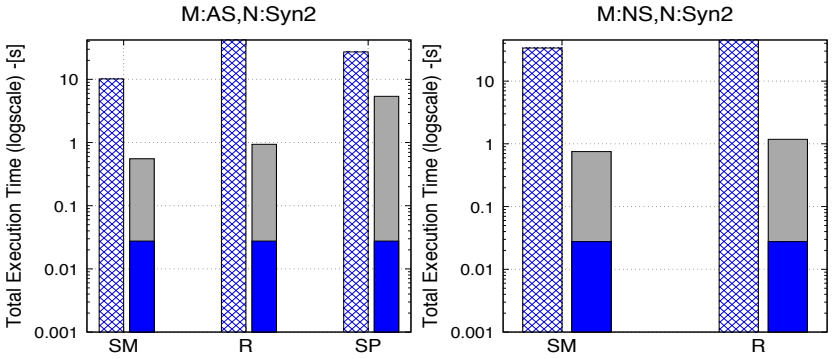


(a) **P2.11** (b) **P2.11**



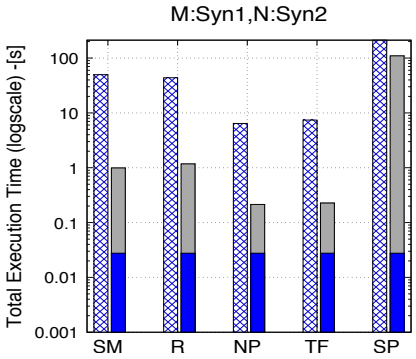
(c) **P2.11**

Figure B.29: P2.11 evaluation before and after rewriting.



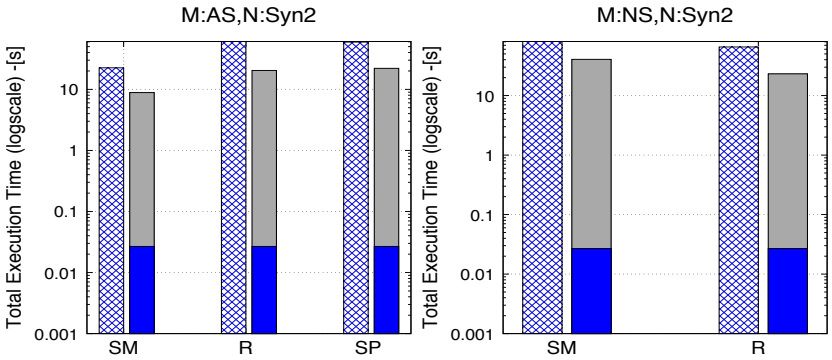
(a) **P2.13**

(b) **P2.13**



(c) **P2.13**

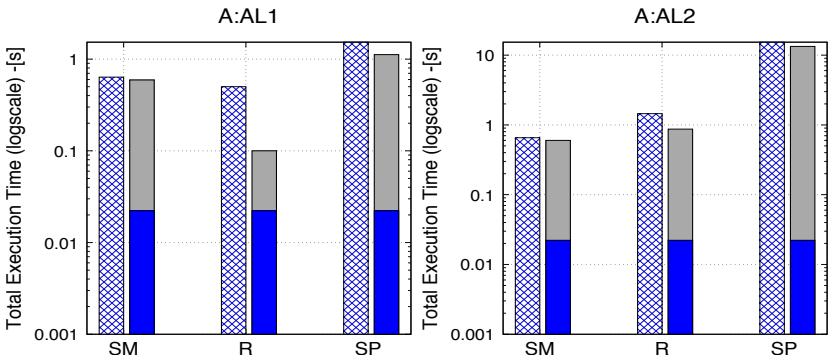
Figure B.30: P2.13 evaluation before and after rewriting.



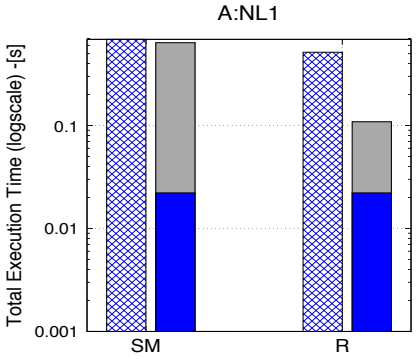
(a) **P2.14**

(b) **P2.14**

Figure B.31: P2.14 evaluation before and after rewriting.

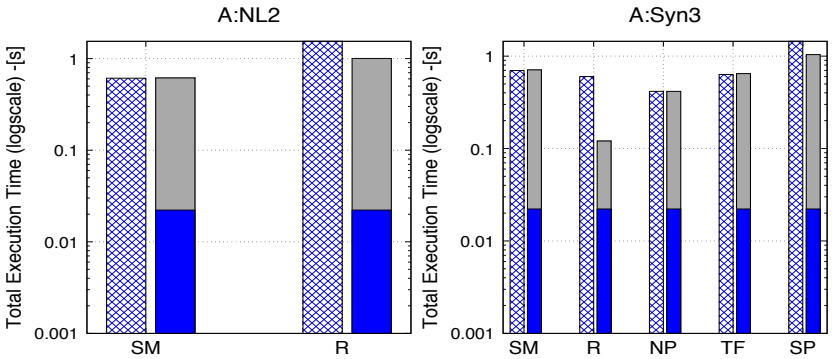


(a) **P2.15** (b) **P2.15**



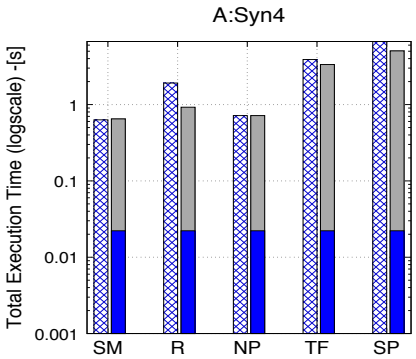
(c) **P2.15**

Figure B.32: P2.15 evaluation before and after rewriting.



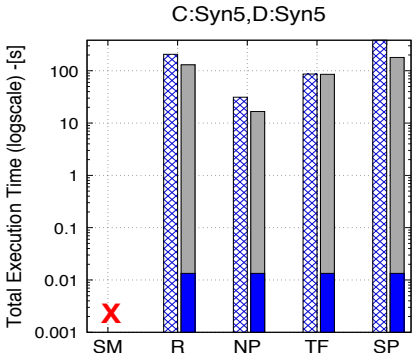
(a) **P2.15**

(b) **P2.15**



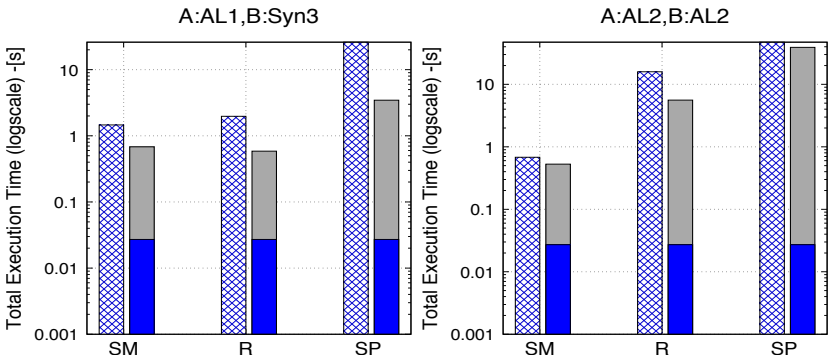
(c) **P2.15**

Figure B.33: P2.15 evaluation before and after rewriting.



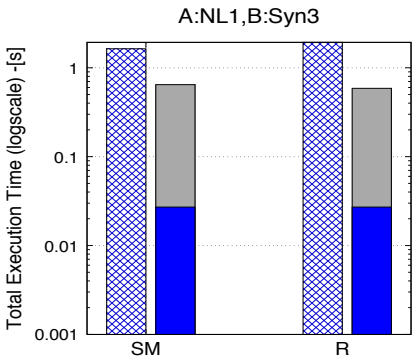
(a) **P2.16**

Figure B.34: P2.16 evaluation before and after rewriting.



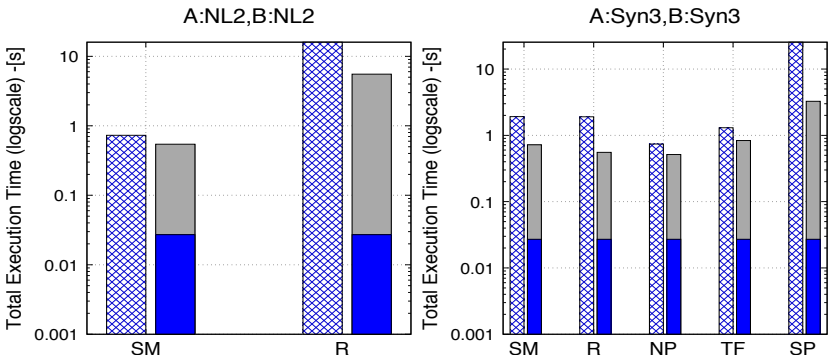
(a) **P2.18**

(b) **P2.18**

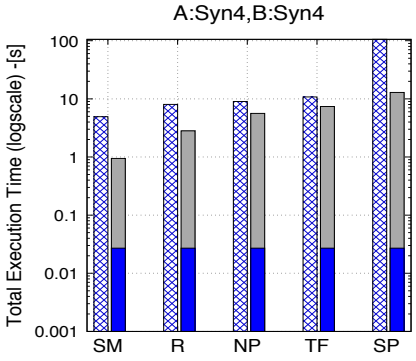


(c) **P2.18**

Figure B.35: P2.18 evaluation before and after rewriting.



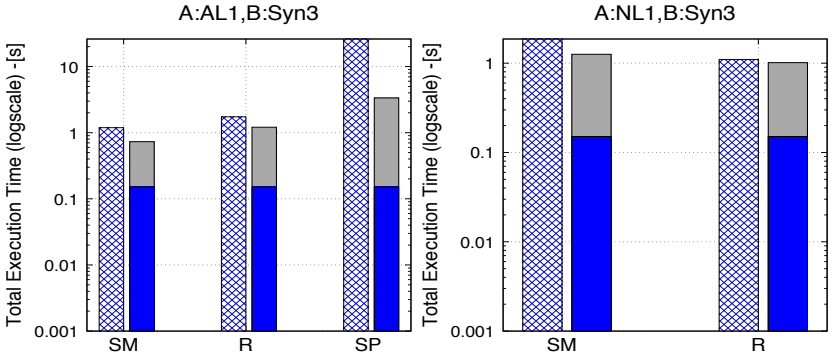
(a) **P2.18** (b) **P2.18**



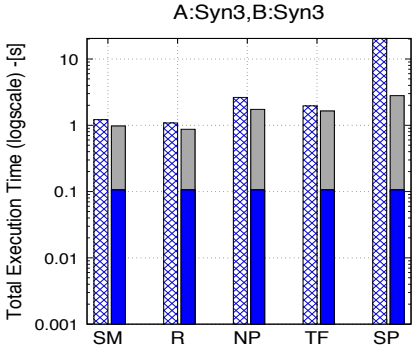
(c) **P2.18**

Figure B.36: P2.18 evaluation before and after rewriting.

B.7 Additional Results: $\mathcal{P}^{\neg Opt}$ Pipelines: MNC Cost Model

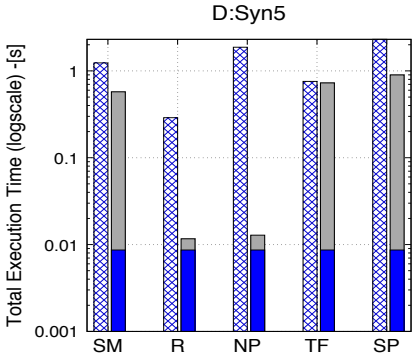


(a) **P1.2** (b) **P1.2**



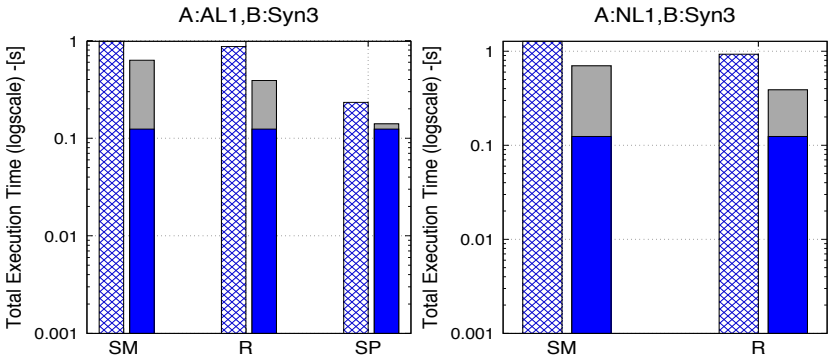
(c) **P1.2**

Figure B.37: P1.2 evaluation before and after rewriting.



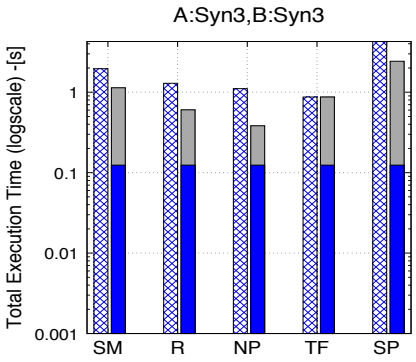
(a) **P1.6**

Figure B.38: P1.6 evaluation before and after rewriting.



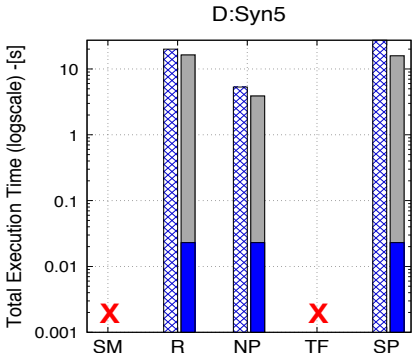
(a) **P1.8**

(b) **P1.8**



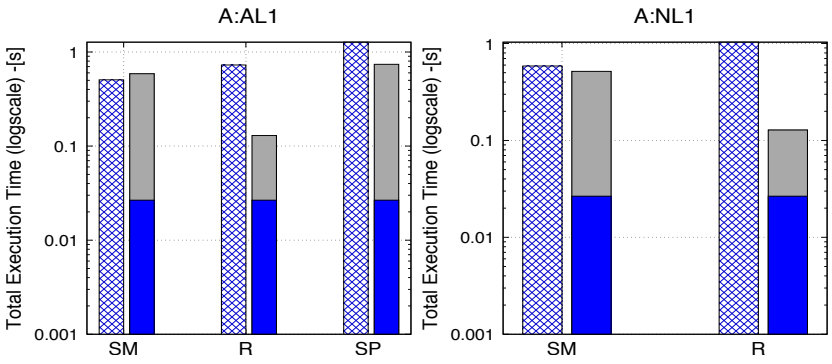
(c) **P1.8**

Figure B.39: P1.8 evaluation before and after rewriting.



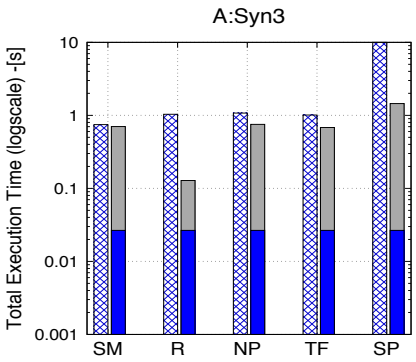
(a) **P1.9**

Figure B.40: P1.9 evaluation before and after rewriting.



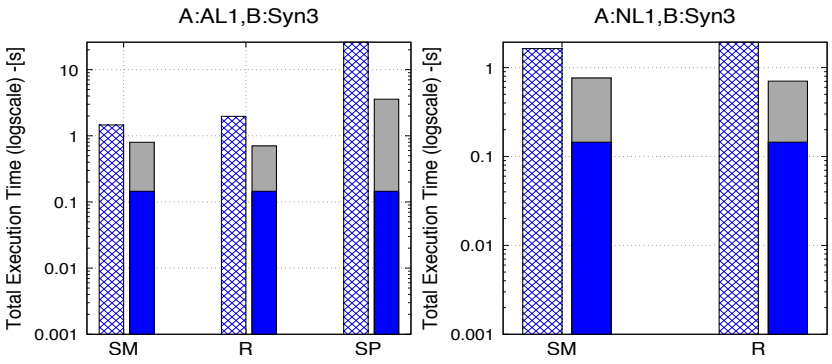
(a) **P1.10**

(b) **P1.10**

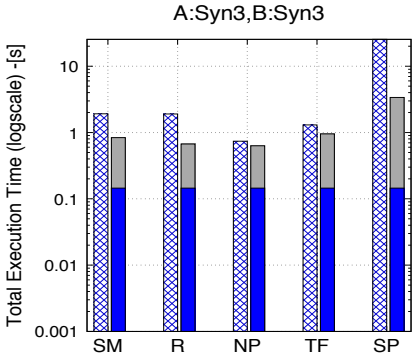


(c) **P1.10**

Figure B.41: P1.10 evaluation before and after rewriting.

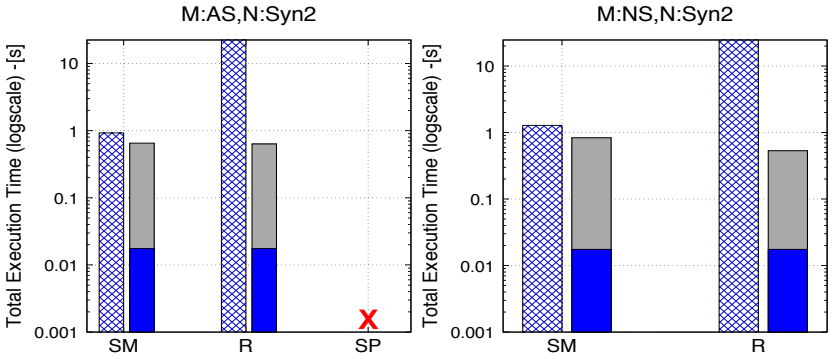


(a) **P1.11** (b) **P1.11**



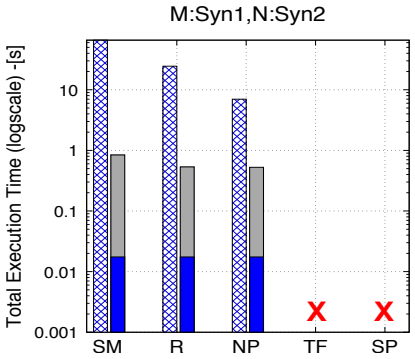
(c) **P1.11**

Figure B.42: P1.11 evaluation before and after rewriting.



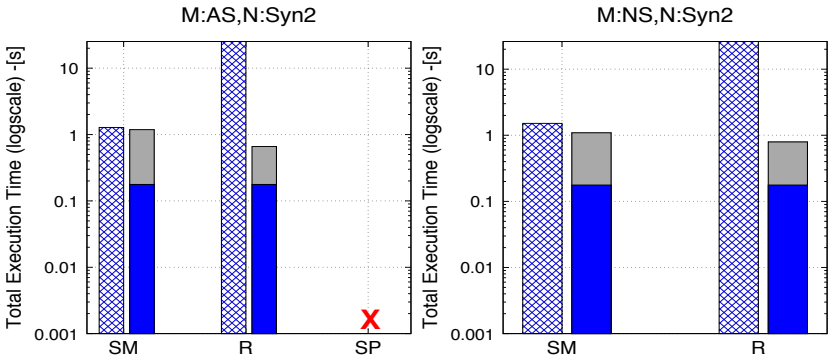
(a) **P1.12**

(b) **P1.12**



(c) **P1.12**

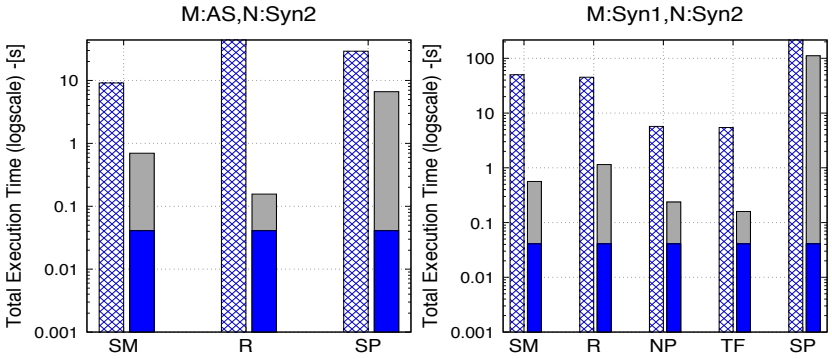
Figure B.43: P1.12 evaluation before and after rewriting.



(a) **P1.14**

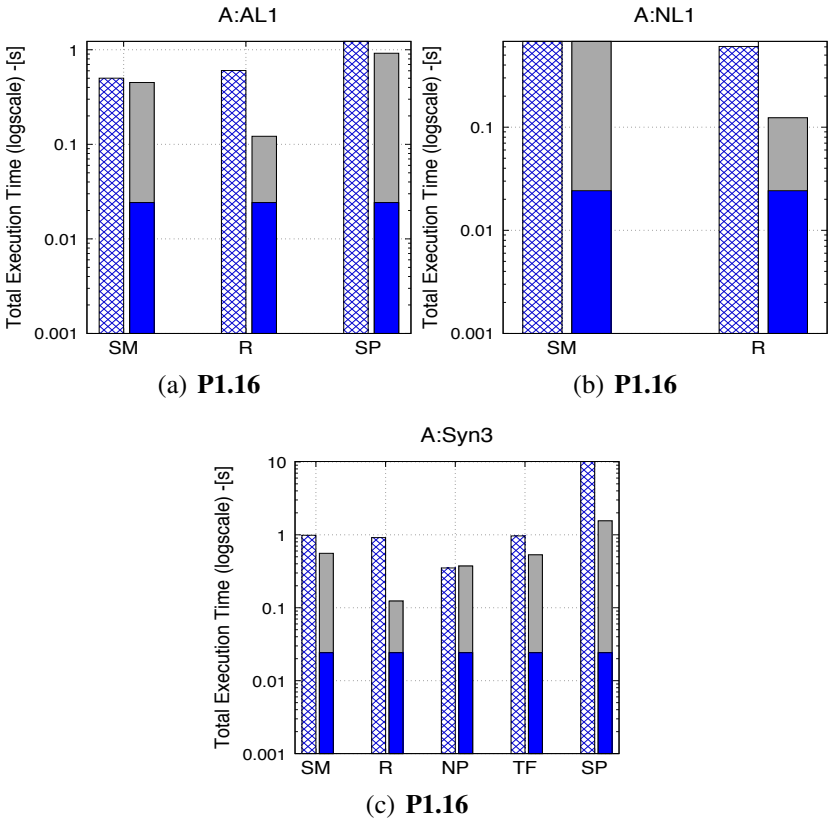
(b) **P1.14**

Figure B.44: P1.14 evaluation before and after rewriting.



(a) **P1.15** (b) **P1.15**

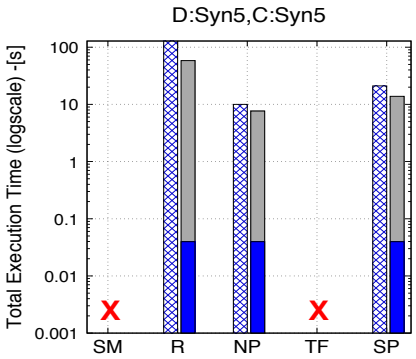
Figure B.45: P1.15 evaluation before and after rewriting.



(a) **P1.16** (b) **P1.16**

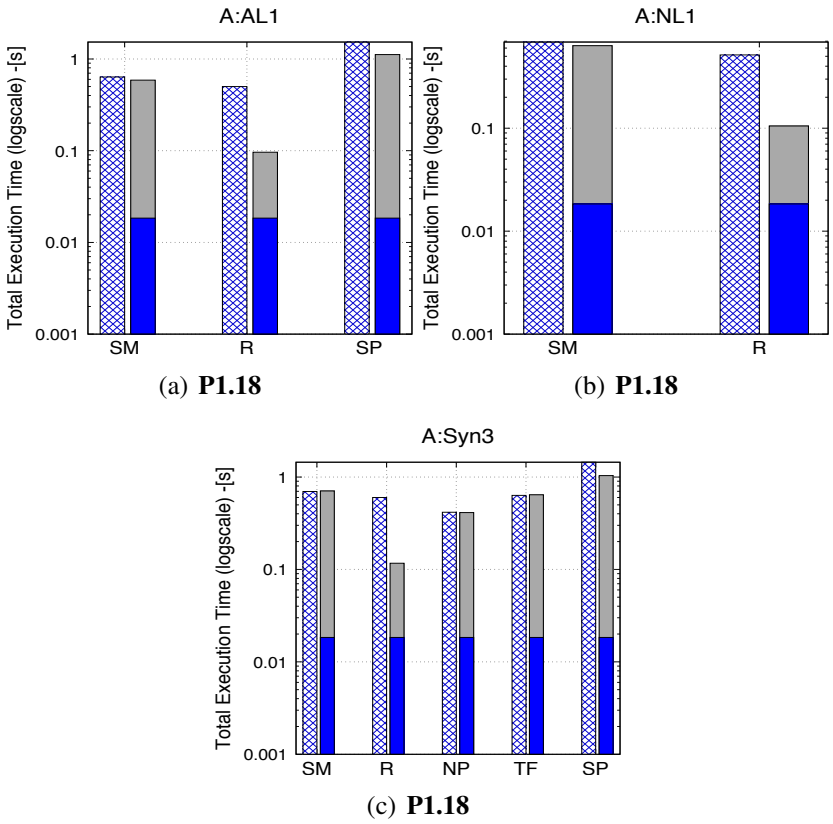
(c) **P1.16**

Figure B.46: P1.16 evaluation before and after rewriting.



(a) **P1.17**

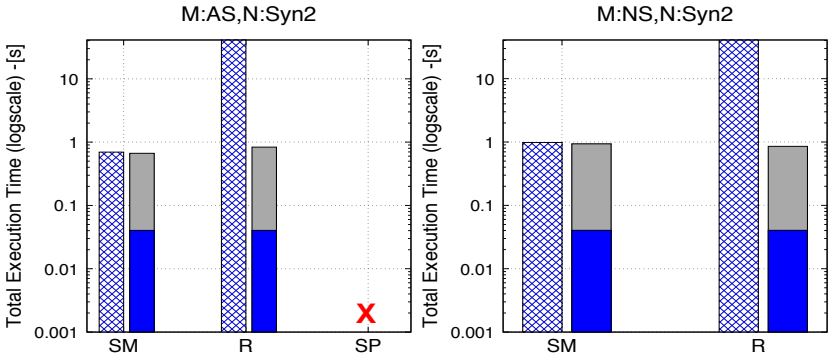
Figure B.47: P1.17 evaluation before and after rewriting.



(a) **P1.18** (b) **P1.18**

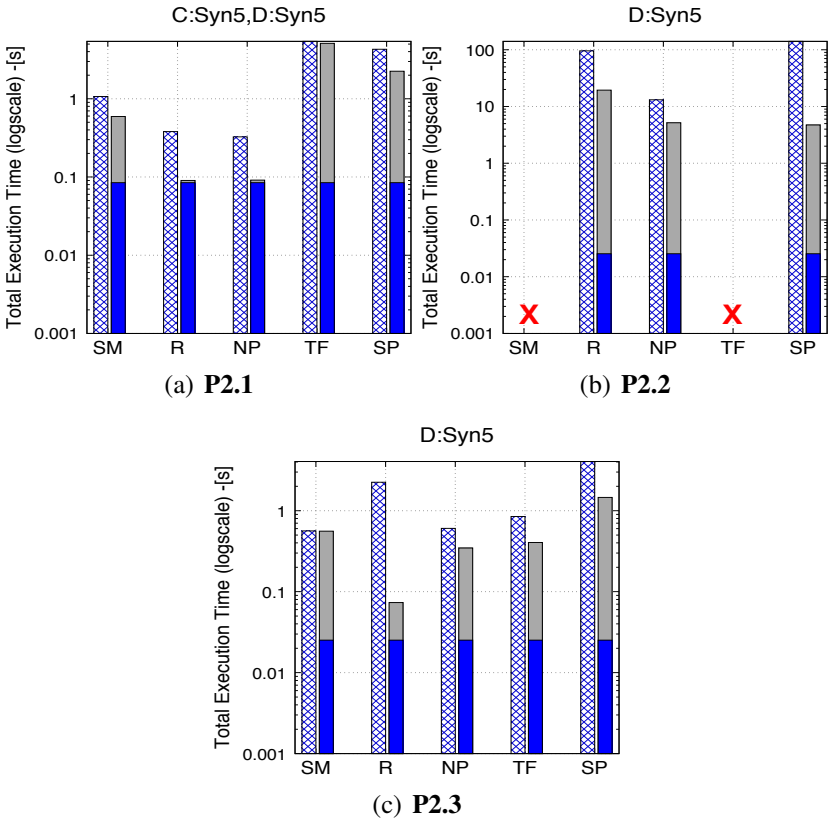
(c) **P1.18**

Figure B.48: P1.18 evaluation before and after rewriting.



(a) **P1.25** (b) **P1.25**

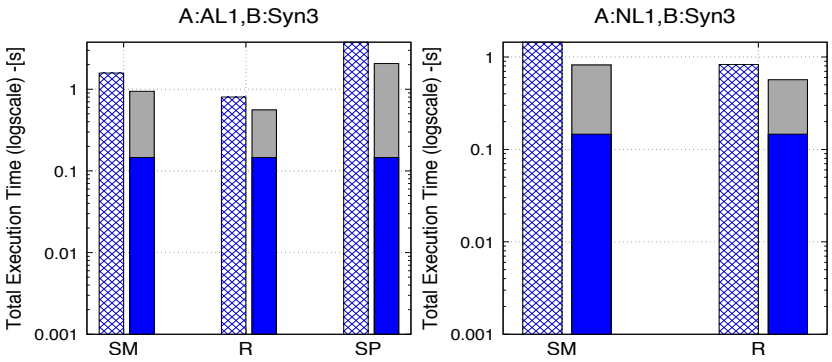
Figure B.49: P1.25 evaluation before and after rewriting.



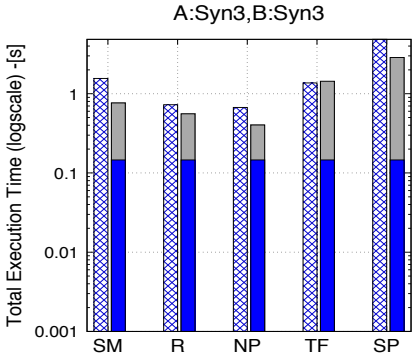
(a) **P2.1** (b) **P2.2**

(c) **P2.3**

Figure B.50: P2.1 evaluation before and after rewriting.

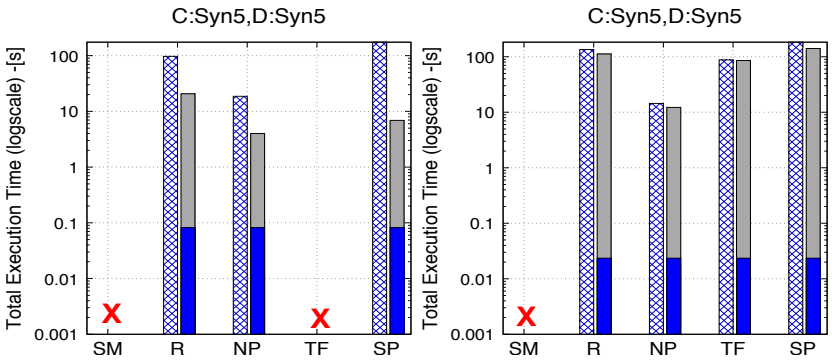


(a) **P2.4** (b) **P2.4**



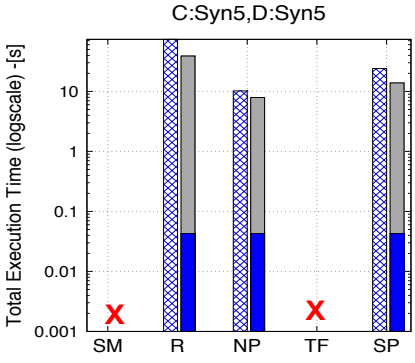
(c) **P2.4**

Figure B.51: P2.4 evaluation before and after rewriting.

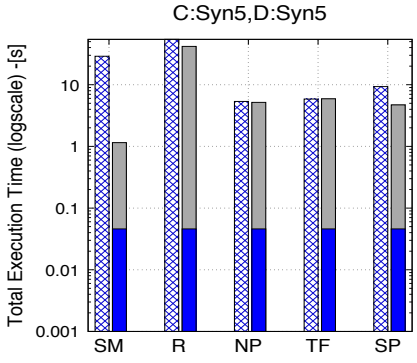


(a) **P2.5**

(b) **P2.6**

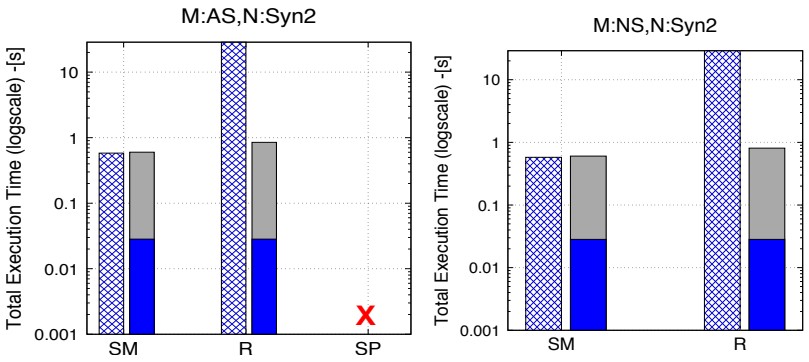


(c) **P2.8**

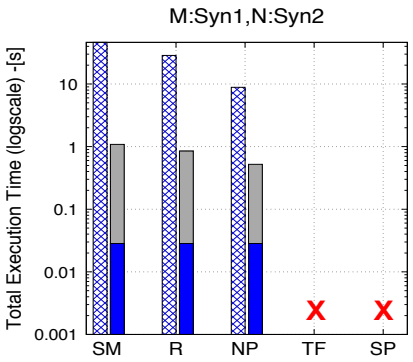


(d) **P2.9**

Figure B.52: P2.5, P2.6, P2.8 and P2.9 evaluation before and after rewriting.

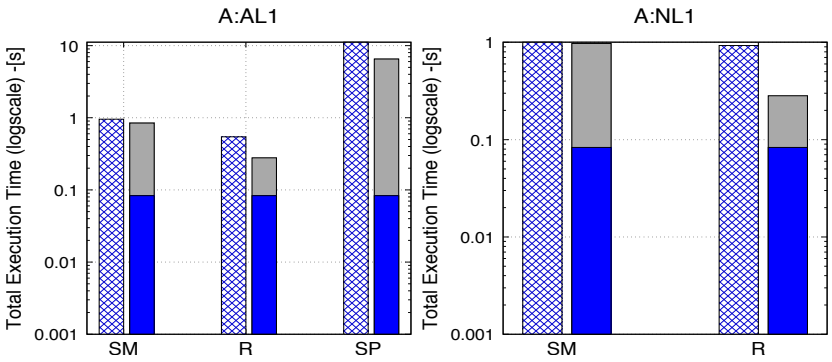


(a) **P2.10** (b) **P2.10**



(c) **P2.10**

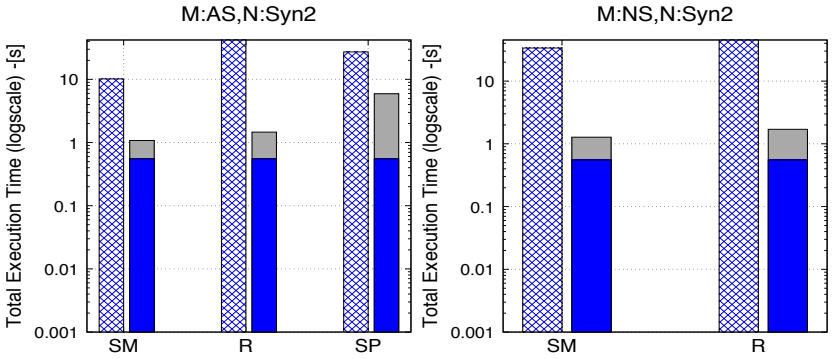
Figure B.53: P2.10 evaluation before and after rewriting.



(a) **P2.11** (b) **P2.11**

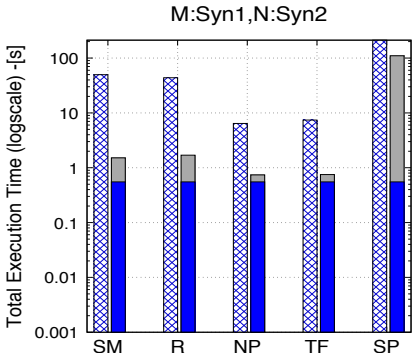
(c) **P2.11**

Figure B.54: P2.11 evaluation before and after rewriting.



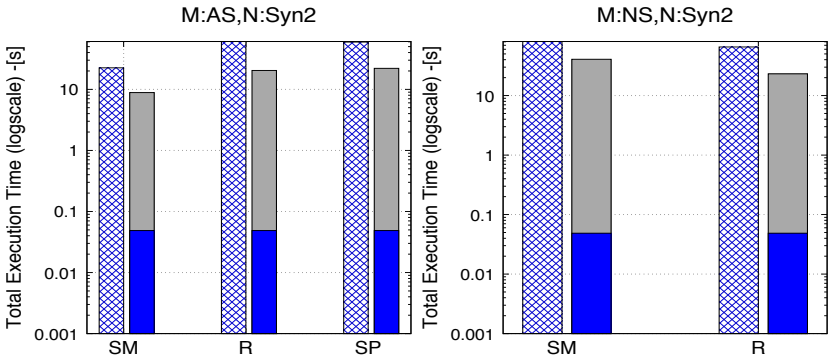
(a) **P2.13**

(b) **P2.13**



(c) **P2.13**

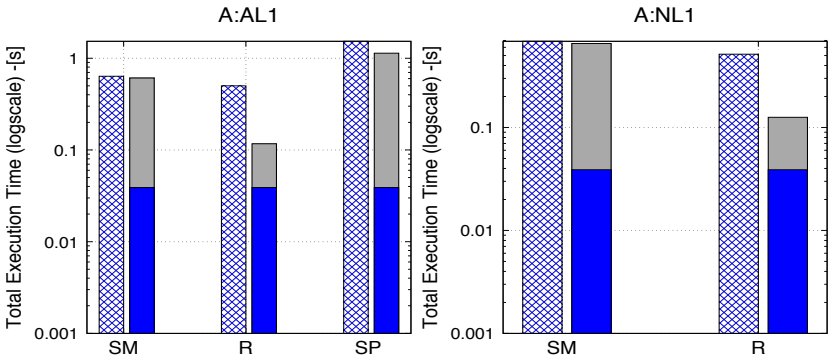
Figure B.55: P2.13 evaluation before and after rewriting.



(a) **P2.14**

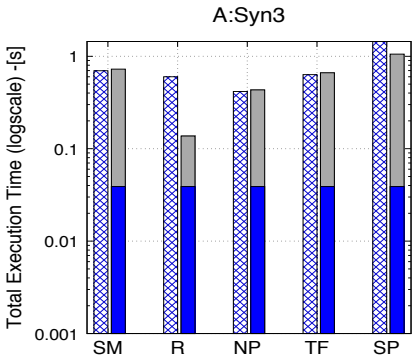
(b) **P2.14**

Figure B.56: P2.14 evaluation before and after rewriting.



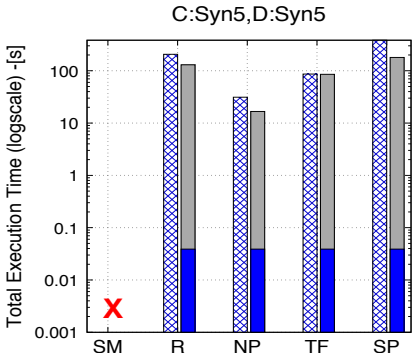
(a) **P2.15**

(b) **P2.15**



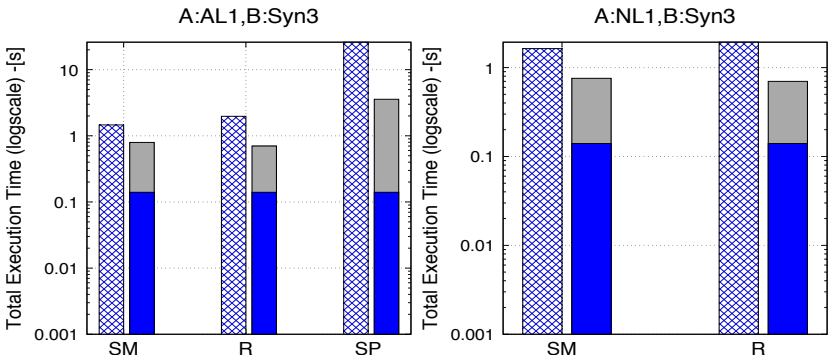
(c) **P2.15**

Figure B.57: P2.15 evaluation before and after rewriting.

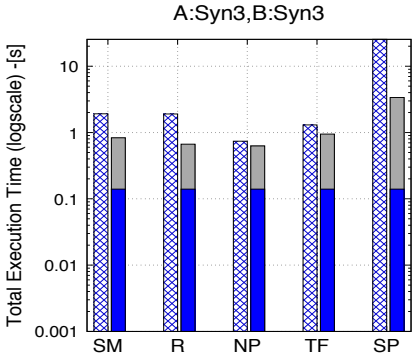


(a) **P2.16**

Figure B.58: P2.16 evaluation before and after rewriting.



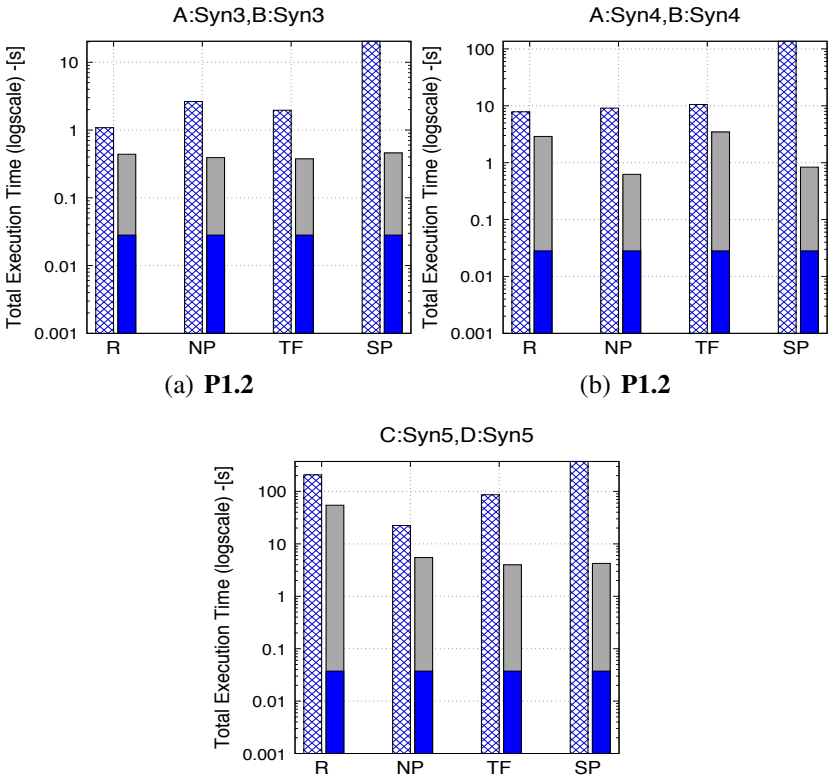
(a) **P2.18** (b) **P2.18**



(c) **P2.18**

Figure B.59: P2.18 evaluation before and after rewriting.

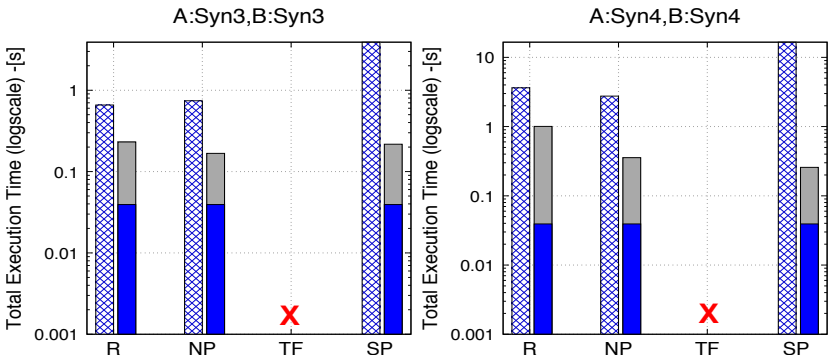
B.8 Additional Results: $\mathcal{P}^{Views}$ Pipelines



(a) **P1.2** (b) **P1.2**

(c) **P1.3**

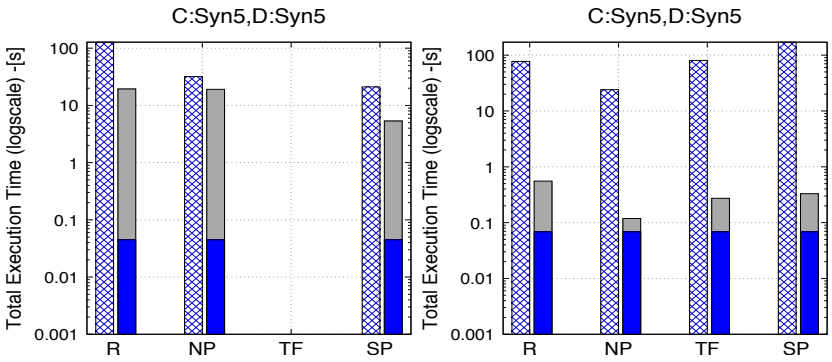
Figure B.60: P1.2 and P1.3 evaluation before and after rewriting.



(a) **P1.4** (b) **P1.4**

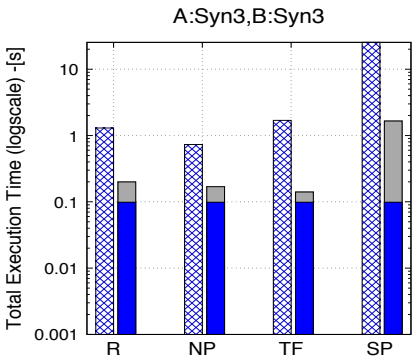
(c) **P1.11** (d) **P1.11**

Figure B.61: P1.4 and P1.11 evaluation before and after rewriting.

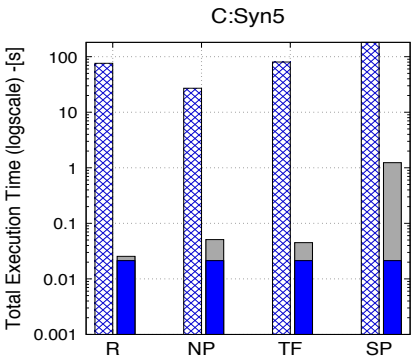


(a) **P1.17**

(b) **P1.19**

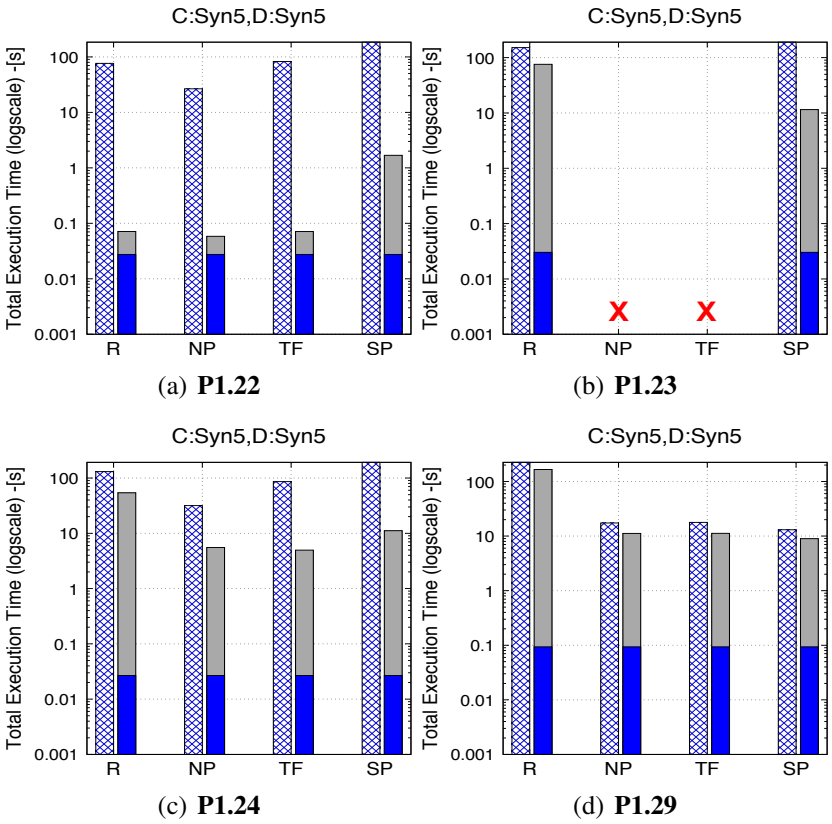


(c) **P1.20**



(d) **P1.21**

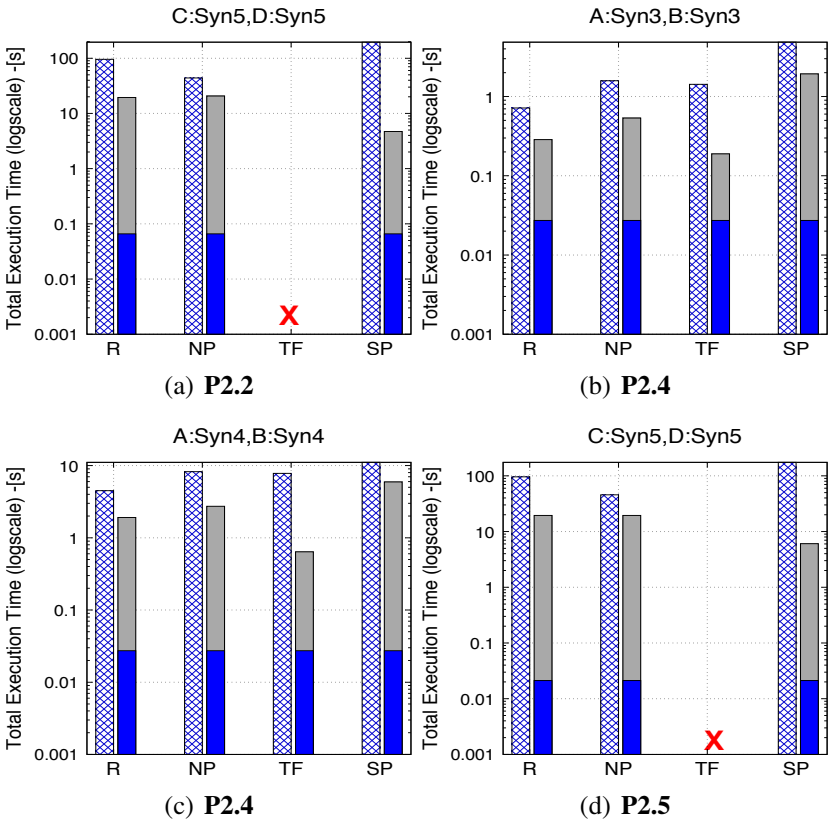
Figure B.62: P1.17, P1.19, P1.20 and P1.21 evaluation before and after rewriting.



(a) **P1.22** (b) **P1.23**

(c) **P1.24** (d) **P1.29**

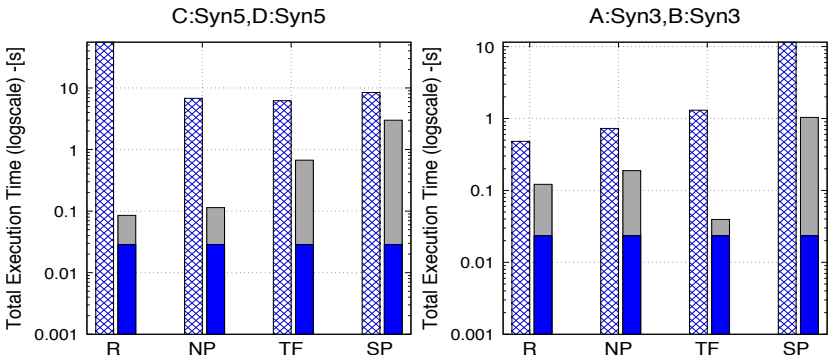
Figure B.63: P1.22, P1.23, P1.24 and P1.29 evaluation before and after rewriting.



(a) **P2.2** (b) **P2.4**

(c) **P2.4** (d) **P2.5**

Figure B.64: P2.2, P2.4 and P2.5 evaluation before and after rewriting.



(a) **P2.9** (b) **P2.11**

(c) **P2.11** (d) **P2.16**

Figure B.65: P2.9, P2.11 and P2.16 evaluation before and after rewriting.

Bibliography

[1] Teaching the Elephant New Tricks. http://www.vertica.com/2012/07/05/teaching-the-elephant-new-tricks, Accessed June, 2018.

[2] Amazon Review Data. https://nijianmo.github.io/amazon/index.html, Accessed August, 2020.

[3] Apache Accumulo. https://accumulo.apache.org, Accessed Feb, 2019.

[4] Apache Asterixdb. https://asterixdb.apache.org/, Accessed Sept, 2018.

[5] Breeze Wiki. https://github.com/scalanlp/breeze/wiki, Accessed June, 2020.

[6] Cox Proportional-Hazards Model. https://github.com/apache/systemds/blob/master/scripts/algorithms/Cox-predict.dml, Accessed Feb, 2021.

[7] High Performance Connectors for Load and Access of Data From Hadoop to Oracle Database. http://developer.teradata.com/connectivity/articles/teradata-connector-for-hadoop-now-available, Accessed June, 2018.

[8] IBM InfoSphere BigInsights. http://pic.dhe.ibm.com/infocenter/bigins/v1r4/index.jsp, Accessed June, 2018.

[9] Kaggle Survey. https://www.kaggle.com/kaggle-survey-2019, Accessed June, 2020.

[10] M. Stonebraker, The Case for Polystores. http://wp.sigmod.org/?p=1629, Accessed June, 2018.

[11] MongoDB. https://www.mongodb.com/, Accessed June, 2018.

[12] Morpheus. https://github.com/lchen001/Morpheus, Accessed December, 2020.

[13] Native Blas in SystemDS. https://apache.github.io/systemds/native-backend, Accessed June, 2020.

[14] Neo4j. https://neo4j.com/, Accessed Feb, 2020.

[15] Netflix Movie Rating. https://www.kaggle.com/netflix-inc/netflix-prize-data, Accessed August, 2020.

[16] NumPy. https://numpy.org/, Accessed June, 2020.

[17] Project R. https://www.r-project.org/other-docs.html, Accessed June, 2020.

[18] Python/NumPy in Monetdb. https://tinyurl.com/11ljy21v, Accessed June, 2020.

[19] Saxon. http://saxon.sourceforge.net/, Accessed June, 2018.

[20] Scidb. https://www.paradigm4.com/, Accessed June, 2018.

[21] SparkMLlib. https://spark.apache.org/mllib, Accessed June, 2020.

[22] Sqoop. http://sqoop.apache.org, Accessed June, 2018.

[23] Teradata Connector for Hadoop. http://developer.teradata.com/connectivity/articles/teradata-connector-for-hadoop-now-available, Accessed June, 2018.

[24] Twitter API. https://developer.twitter.com/en/docs, Accessed January, 2021.

[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A System for Large-Scale Machine Learning. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 265–283, 2016.

[26] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[27] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. K. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, and A. Troudi. RHEEM: Enabling Cross-Platform Data Processing - May the Big Data Be With You! *Proc. VLDB Endow.*, 11(11):1414–1427, 2018.

[28] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous View Maintenance for VLSD Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 179–192, 2009.

[29] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 487–499, 1994.

[30] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-Free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1103–1114, 2014.

[31] R. Alotaibi, D. Bursztyn, A. Deutsch, I. Manolescu, and S. Zampetakis. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1660–1677, 2019.

[32] R. Alotaibi, B. Cautis, A. Deutsch, M. Latrache, I. Manolescu, and Y. Yang. ESTOCADA: Towards Scalable Polystore Systems. *Proc. VLDB Endow.*, 13(12):2949–2952, 2020.

[33] R. Alotaibi, B. Cautis, A. Deutsch, and I. Manolescu. HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries (Extended Version). *CoRR*, abs/2103.12317, 2021.

[34] A. Arion, V. Benzaken, and I. Manolescu. XML Access Modules: Towards Physical Data Independence in XML Databases. In *Proceedings of the International Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>, in cooperation with ACM SIGMOD*, 2005.

[35] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[36] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform Access to NoSQL Systems. *Inf. Syst.*, 43:117–133, 2014.

[37] S. Axler. *Linear Algebra Done Right*. Springer, 2015.

[38] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient Processing of Data Warehousing Queries in a Split Execution Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1165–1176, 2011.

[39] P. Barceló, N. Higuera, J. Pérez, and B. Subercaseaux. On the Expressiveness of LARA: A Unified Language for Linear and Relational Algebra. In *Proceedings of the International Conference on Database Theory, ICDT*, pages 6:1–6:20, 2020.

[40] C. Beeri and M. Y. Vardi. A Proof Procedure for Data Dependencies. *Journal of the ACM (JACM)*, 31:718–741, 1984.

[41] S. Bergamaschi, S. Castano, and M. Vincini. Semantic Integration of Semistructured and Structured Data Sources. *ACM SIGMOD Record*, 28:54–59, 1999.

[42] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. SystemMl: Declarative Machine Learning on Spark. *Proc. VLDB Endow.*, 9(13):1425–1436, 2016.

[43] M. Boehm, A. V. Evfimievski, N. Pansare, and B. Reinwald. Declarative Machine Learning - A Classification of Basic Properties and Types. *CoRR*, abs/1605.05826, 2016.

[44] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.*, 11(12):1755–1768, 2018.

[45] R. Bonaque, T. D. Cao, B. Cautis, F. Goasdoué, J. Letelier, I. Manolescu, O. Mendoza, S. Ribeiro, X. Tannier, and M. Thomazo. Mixed-Instance Querying: A Lightweight Integration Architecture for Data Journalism. *Proc. VLDB Endow.*, 9(13):1513–1516, 2016.

[46] C. Bondiombouy and P. Valduriez. Query Processing in Multistore Systems: an Overview. *International Journal of Cloud Computing*, 5:309–346, 2016.

[47] V. R. Borkar, M. J. Carey, D. Lychagin, T. Westmann, D. Engovatov, and N. Onose. Query Processing in the Aqualogic Data Services Platform. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 1037–1048, 2006.

[48] R. Brijder, F. Geerts, J. V. den Bussche, and T. Weerwag. On the Expressive Power of Query Languages for Matrices. *ACM Trans. Database Syst.*, 44(4):15:1–15:31, 2019.

[49] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proceedings of the International Workshop on Research Issues in Data Engineering-Distributed Object Management, RIDE-DOM*, pages 124–131, 1995.

[50] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive queries in Relational Data Bases. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[51] D. Chen and C. Chan. Viewjoin: Efficient View-Based Evaluation of Tree Pattern Queries. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 816–827, 2010.

[52] L. Chen, A. Kumar, J. F. Naughton, and J. M. Patel. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.*, 10(11):1214–1225, 2017.

[53] E. F. Codd. Relational Completeness of Database Sublanguages. *Research Report / RJ / IBM / San Jose, California*, 1972.

[54] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting Queries with Arbitrary Aggregation Functions Using Views. *ACM Trans. Database Syst.*, pages 672–715, 2006.

[55] S. Dasgupta, K. Coakley, and A. Gupta. Analytics-Driven Data Ingestion and Derivation in the AWESOME Polystore. In *Proceedings of IEEE International Conference on Big Data*, pages 2555–2564. IEEE, 2016.

[56] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A Scalable, Portable, and Interactive Index Advisor for Large workloads. *Proc. VLDB Endow.*, 4(6):362–372, 2011.

[57] A. Deshpande and D. Van Gucht. An Implementation for Nested Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 76–87, 1988.

[58] A. Deutsch. FOL Modeling of Integrity Constraints (Dependencies). In *Encyclopedia of Database Systems, Second Edition*. 2018.

[59] A. Deutsch and R. Hull. Provenance-Directed Chase&Backchase. In *Search of Elegance in the Theory and Practice of Computation*, pages 227–236. Springer, 2013.

[60] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting Queries Using Views with Access Patterns Under Integrity constraints. In *Proceedings of the International Conference on Database Theory, ICDT*, pages 352–367, 2005.

[61] A. Deutsch, A. Nash, and J. B. Remmel. The Chase Revisited. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS*, pages 149–158, 2008.

[62] A. Deutsch, L. Popa, and V. Tannen. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 459–470, 1999.

[63] A. Deutsch, L. Popa, and V. Tannen. Query Reformulation with Constraints. *ACM SIGMOD Record*, 35:65–73, 2006.

[64] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 201–212, 2003.

[65] A. Deutsch and V. Tannen. Reformulation of XML Queries and Constraints. In *Proceedings of the International Conference on Database Theory, ICDT*, pages 225–241, 2003.

[66] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split Query Processing in Polybase. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1255–1266, 2013.

[67] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Elsevier, 2012.

[68] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The Bigdawg Polystore System. *ACM SIGMOD Record.*, 44(2):11–16, 2015.

[69] R. Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems (TODS)*, 2:262–278, 1977.

[70] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *Proceedings of the International Conference on Database Theory, ICDT*, pages 207–224, 2003.

[71] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[72] R. Fagin and M. Y. Vardi. *The Theory of Data Dependencies: A Survey*. IBM Thomas J. Watson Research Division, 1984.

[73] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. Silkroute: A Framework for Publishing Relational Data in XML. *ACM Trans. Database Syst.*, 27(4):438–493, 2002.

[74] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 311–322, 1999.

[75] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational Plans for Data Integration. *AAAI/IAAI*, pages 67–73, 1999.

[76] G. Grahne and A. O. Mendelzon. Tableau Techniques for Querying Information Sources Through Global Schemas. In *Proceedings of the International Conference on Database Theory, ICDT*, pages 332–347, 1999.

[77] G. Grahne and A. Onet. Anatomy of the Chase. *CoRR*, abs/1303.6682, 2013.

[78] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 805–810, 2005.

[79] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 276–285, 1997.

[80] A. Halevy, A. Rajaraman, and J. Ordille. Data Integration: The Teenage Years. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 9–16, 2006.

[81] A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB J.*, 10(4):270–294, 2001.

[82] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research, CIDR*, pages 68–78, 2007.

[83] I. Ileana. *Query Rewriting Using Views : a Theoretical and Practical Perspective*. Theses, Télécom ParisTech, Oct. 2014.

[84] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete Yet Practical Search for Minimal Query Reformulations Under Constraints. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1015–1026, 2014.

[85] ISO/IEC 9075-1:1999. SQL:1999. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26196`. Accessed March 2016.

[86] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of Data Wrehouses*. Springer Science & Business Media, 2002.

[87] A. Jindal, J. Quiané-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. In *Proceedings of the Biennial Conference on Innovative Data Systems Research, CIDR*, 2013.

[88] A. Johnson, J. Pollard, L. Shen, L. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. Celi, and R. Mark. MIMIC-III, A Freely Accessible Critical Care Database. *Scientific Data (2016). DOI: 10.1038/sdata.2016.35. Available at: http://www.nature.com/articles/sdata201635*, 2016.

[89] D. S. Johnson and A. Klug. Testing Containment of Conjunctive Queries Under Functional and Inclusion Dependencies. *Journal of Computer and system Sciences*, 28:167–189, 1984.

[90] P. C. Kanellakis. Elements of Relational Database Theory. In *Formal Models and Semantics*, pages 1073–1156. Elsevier, 1990.

[91] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino. Extending Relational Query Processing with ML Inference. In *Proceedings of the Biennial Conference on Innovative Data Systems Research, CIDR*, 2020.

[92] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *Proceedings of the Biennial Conference on Innovative Data Systems Research, CIDR*, 2015.

[93] A. Katsifodimos, I. Manolescu, and V. Vassalos. Materialized View Selection for XQuery Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 565–576, 2012.

[94] D. Kernert, F. Köhler, and W. Lehner. Bringing Linear Algebra Objects to Life in a Column-Criented in-Memory Database. In *Proceedings of the International Workshop on In Memory Data Management and Analytics, IMDM*, pages 37–49, 2013.

[95] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, and J. Pereira. The CloudMdsQL Multistore System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2113–2116, 2016.

[96] D. Kossmann. The State of The Art in Distributed Query Processing. *ACM Computing Surveys (CSUR)*, 32:422–469, 2000.

[97] A. Kunft, A. Katsifodimos, S. Schelter, S. Breß, T. Rabl, and V. Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *Proc. VLDB Endow.*, 12(11):1553–1567, 2019.

[98] K. Kuttler. *Linear Algebra: Theory and Applications*. The Saylor Foundation, 2012.

[99] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: Souping Up Big Data Query Processing with a Multistore System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1591–1602, 2014.

[100] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic Physical Design for Big Data Analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 851–862. ACM, 2014.

[101] M. Lenzerini. Data integration: A Theoretical Perspective. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS*, pages 233–246, 2002.

[102] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 251–262, 1996.

[103] A. Y. Levy, A. Rajaraman, and J. J. Ordille. The World Wide Web as a Collection of Views: Query Processing in the Information Manifold. In *Workshop on Materialized Views: Techniques and Applications, VIEWS@SIGMOD*, pages 43–55, 1996.

[104] H. Lim, Y. Han, and S. Babu. How to Fit When No One Size Fits. In *Proceedings of the Biennial Conference on Innovative Data Systems Research, CIDR*, 2013.

[105] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine. Scalable Linear Algebra on a Relational Database System. *IEEE Trans. Knowl. Data Eng.*, 31(7):1224–1238, 2019.

[106] R. Machlin. Index-based multidimensional Array Queries: Safety and Equivalence. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems,PODS*, pages 175–184, 2007.

[107] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[108] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems (TODS)*, 4:455–469, 1979.

[109] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 241–250, 2001.

[110] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with Xml and Relational. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 623–626, 2000.

[111] M. Meier, M. Schmidt, and G. Lausen. On Chase Termination Beyond Stratification. *arXiv preprint arXiv:0906.4228*, 2009.

[112] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.*, 17:34:1–34:7, 2016.

[113] M. Milani, S. Hosseinpour, and H. Pehlivan. Rule-based Production of Mathematical Expressions. *Mathematics*, 6:254, 11 2018.

[114] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[115] A. Nash and B. Ludäscher. Processing First-Order Queries Under Limited Access Patterns. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS*, pages 307–318, 2004.

[116] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. K. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, pages 27–33, 2001.

[117] D. Olteanu, J. Huang, and C. Koch. Sprout: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 640–651, 2009.

[118] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting Nested XML Queries Using Nested Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 443–454, 2006.

[119] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*, volume 2. Springer, 1999.

[120] D. Phillips, N. Zhang, I. F. Ilyas, and M. T. Özsu. InterJoin: Exploiting Indexes and Materialized Views in Xpath Evaluation. In *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, pages 13–22, 2006.

[121] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A Chase Too Far? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 273–284, 2000.

[122] L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, Views. In *International Conference on Database Theory*, pages 39–57, 1999.

[123] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering Queries Using Templates with Binding Patterns. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS*, pages 105–112, 1995.

[124] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the International Conference on Database Theory, ICDT*, pages 4–33, 2010.

[125] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[126] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, 1990.

[127] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.*, 6(11):1068–1079, 2013.

[128] J. Sommer, M. Boehm, A. V. Evfimievski, B. Reinwald, and P. J. Haas. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1607–1623, 2019.

[129] A. P. Stolboushkin. Axiomatizable classes of finite models and definability of linear order. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92)*, pages 64–70, 1992.

[130] M. Stonebraker and U. Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 2–11, 2005.

[131] A. Thomas and A. Kumar. A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics. *Proc. VLDB Endow.*, 11(13):2168–2182, 2018.

[132] Y. Tian, F. Özcan, T. Zou, R. Goncalves, and H. Pirahesh. Building a Hybrid Warehouse: Efficient JoinsBetween Data Stored in HDFS and Enterprise Warehouse. *ACM Transactions on Database Systems (TODS)*, 41:1–38, 2016.

[133] Y. Tian, T. Zou, F. Ozcan, R. Goncalves, and H. Pirahesh. Joins for Hybrid warehouses: Exploiting Massive Parallelism in Hadoop and Enterprise Data Warehouses. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 373–384, 2015.

[134] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[135] M. Vardi. Trends in Theoretical Computer Science, Chapter Fundamentals of Dependency Theory, 1987.

[136] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria Big Data Management and Analytics System and Cloud Services. In *Proceedings of the Biennial Conference on Innovative Data Systems Research, CIDR*, 2017.

[137] Y. R. Wang, S. Hutchison, D. Suciu, B. Howe, and J. Leang. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.*, 13(11):1919–1932, 2020.

[138] Y. R. Wang, M. A. Khamis, H. Q. Ngo, R. Pichler, and D. Suciu. Optimizing Recursive Queries with Program Synthesis. *arXiv preprint arXiv:2202.10390*, 2022.

[139] C. Yu and L. Popa. Constraint-based XML Query Rewriting for Data Integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 371–382, 2004.

[140] C. T. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers Inc., 1998.