GraphflowDB: Scalable Query Processing on Graph-Structured Relations

by

Amine Mhedhbi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2023

© Amine Mhedhbi 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Peter Boncz
Professor, Chair of Large-Scale Analytical Data Management,
Vrije Universiteit Amsterdam
Research Scientist,
Centrum Wiskunde & Informatica

Supervisor(s): Semih Salihoğlu
Associate Professor,
School of Computer Science
University of Waterloo

Internal Members: Tamer Özsu
University Professor,
School of Computer Science
University of Waterloo

Ken Salem
Professor
School of Computer Science
University of Waterloo

Internal-External Member: Lukasz Golab
Professor, Canada Research Chair in Data for Good,
Department of Management Sciences
Faculty of Engineering
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I am the primary contributor of the work I present in this thesis. Some portions of this thesis are based on peer-reviewed joint work with Semih Salihoglu, Chathura Kankanamge, and Pranjal Gupta [75, 118, 119]. The design of the query processor plans presented in Chapter 5 overlaps with the query processor design presented by Xiyang Feng in his Master's thesis [61]. I have contributed to the design in that thesis as stated in its statement of contributions. The overlap with Xiyan's thesis is that it also has a `DGroup-by`-like operator called `F-Group` and DAG-style plans. The rest of the design, including the data structure for the cached intermediate results, how plans add, implement, and optimize caching and reuse, and how to handle partial tuples that do no produce output, is different. The master's thesis lacked an optimizer and only included hand-written plans.

Abstract

Finding patterns over graph-structured datasets is ubiquitous and integral to a wide range of analytical applications, *e.g.*, recommendation and fraud detection. When expressed in the high-level query languages of database management systems (DBMSs), these patterns correspond to many-to-many join computations, which generate very large intermediate relations during query processing and degrade the performance of existing systems.

This thesis argues that modern query processors need to adopt two novel techniques to be efficient on growing many-to-many joins: (i) *worst-case optimal join* algorithms; and (ii) *factorized representations*. Traditional query processors generate join plans that use binary joins, which in iteration take two relations, base or intermediate, to join and produce a new relation. The theory of worst-case optimal joins have shown that this style of join processing can be provably suboptimal and hence generate unnecessarily large intermediate results. This can be avoided on cyclic join queries if the join is performed in a multi-way fashion a join-attribute-at-a-time. As its first contribution, this thesis proposes the design and implementation of a query processor and optimizer that can generate plans that mix worst-case optimal joins, *i.e.*, attribute-at-a-time joins and binary joins, *i.e.*, table-at-a-time joins. In contrast to prior approaches with novel join optimizers that require solving hard computational problems, such as computing low-width hypertree decompositions of queries, our join optimizer is cost-based and uses a traditional dynamic programming approach with a new cost metric.

On acyclic queries, or acyclic parts of queries, sometimes the generation of large intermediate results cannot be avoided. Yet, the theory of factorization has shown that often such intermediate results can be highly compressible if they contain multi-valued dependencies between join attributes. Factorization proposes two relation representation schemes, called *f- and d-representations*, to represent the large intermediate results generated under many-to-many joins in a compressed format. Existing proposals to adopt factorized representations require designing processing on fully materialized general tries and novel operators that operate on entire tries, which are not easy to adopt in existing systems. As a second contribution, we describe the implementation of a novel query processing approach we call *factorized vector execution* that adopts f-representations. Factorized vector execution extends the traditional vectorized query processors to use multiple blocks of vectors instead of a single block allowing us to factorize intermediate results and delay or even avoid Cartesian products. Importantly, our design ensures that every core operator in the system still performs computations on vectors. As a third contribution, we further describe how to extend our factorized vector execution model with novel operators to adopt d-representations, which extend f-representations with cached and reused sub-relations. Our design here is based on using nested hash tables that can point to sub-relations instead of copying them and on directed acyclic graph-based query plans.

All of our techniques are implemented in the GraphflowDB system, which was developed throughout the years to facilitate the research in this thesis. We demonstrate that GraphflowDB's query processor can outperform existing approaches and systems by orders of magnitude on both micro-benchmarks and end-to-end benchmarks. The designs proposed in this thesis adopt common-wisdom query processing techniques of pipelining, vector-based execution, and morsel-driven parallelism to ensure easy adoption in existing systems. We believe the design can serve as a blueprint for how to adopt these techniques in existing DBMSs to make them more efficient on workloads with many-to-many joins.

Acknowledgements

I would like to thank my advisor Semih Salihoğlu for teaching me by example and setting a high standard for research and systems building. He introduced me to the world of data management and with his charisma, effortlessly made me want to work with him. Proud to be one of the disciples! Semih gave me ample guidance while ensuring I had the freedom to pursue many of my ideas. He supported me in the pursuit of a research career and cheered me on throughout the ups and downs of the Ph.D. program. He has seen it all. I will always vividly remember his advice and outlook on research from a deep focus on a single problem at a time, to the importance of clarity in communication, asking simple yet deep questions, and his insistence on doing great work. I am immensely grateful for the mentorship and friendship.

I would like to thank professors Tamer Özsu, Ken Salem, Jimmy Lin, and Ihab Ilyas for their support over the years. They gave me great feedback on various projects and presentations and were the perfect audience to refine my work with. They were generous with their time and shared lots of their wisdom and old time stories. The Data Systems Group (DSG) at the University of Waterloo has been an incredible home and I feel lucky to have joined when I did and getting the chance to see it grow.

I would also like to thank my examining committee members Peter Boncz and Lukasz Golab for reviewing my thesis and their helpful comments and discussions. A special thanks to Peter Boncz and Gábor Szárnyas for hosting me at CWI. I truly enjoyed the experience of sharing my work with your inspiring group. I want to extend many thanks to Phil Bernstein, Spyros Blanas, Christian König, Yi Shan and Vivek Narasayya for being incredible mentors and collaborators while at Microsoft Research and for introducing me to new research directions on transactional processing and cloud resource management.

I would be remiss to not mention the impact that professors William E. Lynch, Glenn Cowan, and Jelena Trajkovic had on me. They introduced me to research and inspired my sense of curiosity and passion during my undergraduate studies. While I was still early on in my bachelor's, they took many of my idiotic ideas seriously and shown me interesting applied problems in signal processing and optical network-on-chip design.

I would like to thank Siddhartha Sahu, Chathura Kankanamge, Pranjal Gupta, Shahid Khaliq, Xiyang Feng, Annie Zhou, and Guodong Jin for their contributions to GraphflowDB as part of project collaborations or within their own projects. With Siddhartha and Chathura, I will always remember the excitement of building out the first version of the system. With Shahid and Pranjal, we tackled the storage component which made the subsequent research possible, and with Xiyang and Annie, we had a wonderful summer working on various aspects of factorization and query processing together. I learned a lot from Guodong who was always cheerful and available for

technical discussions and sharing many of his insights. Major thanks to Lori D. Paniak and Harsh Kirit Roghelia from the computing facility who always provided excellent and timely IT support.

I am grateful for all the friendships I got to make and to the colleagues I met that made graduate studies at Waterloo exceptional. You hold a special place in my heart. I hope our paths continue to cross in the future.

I am deeply grateful to my parents who communicated the importance of education and nurtured my passion in mathematics and science early on. You have always supported me and my siblings in our endeavours. Your sacrifices gave us the chance to carry on and be able to focus on what mattered to us. Big thank you to my siblings for their encouragement and for tolerating my absence at times. My family's support has been invaluable and definitely got me to where I am today. Finally, I would like to thank my wife Abir for her unwavering support throughout my last year of Ph.D. as we navigate our major life changes. She propelled me into finishing and helped turn this document into an ex-thesis.

This is the last section I wrote within this thesis and as I did, I reflected on how hard it was to truly express in words my gratitude and how hard it was to acknowledge everyone properly. Sincerely, thank you for the impact you had on me as a person, as a researcher, and on my work. I now shall shelve this thesis, marking the end of my formal education journey and the start of a new one in academic research; a journey I am absolutely ecstatic to embark on.

Dedication

To my parents for inspiring me and instilling in me tenacity and the pursuit of excellence.

Table of Contents

List of Figures xiv

List of Tables xvii

1 Introduction 1

1.1 Thesis Statement . . . 3

1.2 Thesis Contributions . . . 4

1.2.1 End-to-end Adoption of Worst-case Optimal Joins . . . 4

1.2.2 Factorized Vector Execution . . . 5

1.2.3 Caching and Reuse of Intermediate Results . . . 7

1.3 Thesis Outline . . . 8

2 Preliminaries and Background 10

2.1 Queries and Notation . . . 10

2.2 GraphflowDB Overview . . . 11

3 Adopting Worst-case Optimal Joins 14

3.1 Existing Approaches and Overview of Contributions . . . 15

3.1.1 Our Contributions . . . 17

3.2 Preliminaries: Generic Join Algorithm . . . 19

3.3 Optimizing WCOJ Plans . . . 20

3.3.1 WCOJ Plans and INLJ/IMJ Operators 20
3.3.2 Effects of Join Attribute Orderings 22
3.3.3 Cost Metric for WCOJ Plans 25
3.4 Full Plan Space & Dynamic Programming Optimizer 26
3.4.1 Hybrid Plans and Hash Join (HJ) Operator 26
3.4.2 Cost Metric for General Plans 29
3.4.3 Dynamic Programming Optimizer 29
3.5 Cost & Cardinality Estimation 31
3.5.1 Catalogue Construction 32
3.5.2 Cost Estimations 32
3.5.3 Limitations 33
3.6 Adaptive WCOJ Plan Evaluation 34
3.6.1 Adaptive Plans 34
3.6.2 Adaptive Operators 35
3.7 Evaluation 36
3.7.1 Setup 36
3.7.2 Evaluation Overview 37
3.7.3 Plan Suitability For Different Queries and Optimizer Evaluation 38
3.7.4 Adaptive WCOJ Plan Evaluation 41
3.7.5 EmptyHeaded Comparison 43
3.7.6 Cache Trie Join (CTJ) Comparisons 45
3.7.7 CFL Comparison 49
3.7.8 Scalability Experiments 51
3.7.9 Catalogue Experiments 51

4 Factorized Vector Execution **54**

4.1 Existing Approaches and Overview of Contributions 56

4.1.1 Thesis Contributions 56

4.2 Preliminaries 57

4.2.1 Factorized Representations 57

4.2.2 GraphflowDB Storage 61

4.3 Mixing Factorization and Vectorized Execution 61

4.3.1 Intermediate Tuple Set Representation 63

4.3.2 Execution Engines and Operators 63

4.3.3 Benefits and Limitations 66

4.4 Evaluation 68

4.4.1 Setup 69

4.4.2 Microbenchmarks 70

4.4.3 Baseline System Comparisons 71

5 Caching and Reuse of Intermediate Results **77**

5.1 Overview of Contributions 78

5.2 Preliminaries 80

5.2.1 D-representations: F-representations using Definitions 80

5.2.2 D-trees 82

5.2.3 Worst-case Size Bounds for F- and D-representations 83

5.3 Adopting D-representations 83

5.3.1 Overview 84

5.3.2 Query Processor 84

5.3.3 Query Optimization 90

5.4 D-Representation Implementation 92

5.5 Experimental Evaluation 93

5.5.1 Setup 94

5.6 Baseline System Comparison . . . 95
5.6.1 Discussion . . . 95
5.6.2 Impact on WCOJ Plan Space . . . 99
5.6.3 Microbenchmarks . . . 100

6 Related Work **104**
6.1 WCOJ Algorithms . . . 104
6.1.1 Algorithmic Work on WCOJ Algorithms . . . 105
6.1.2 Other Systems Implementations of WCOJ Algorithms . . . 107
6.1.3 Cardinality Estimation . . . 110
6.2 Factorized Representations . . . 111
6.3 Subgraph Matching Algorithms . . . 113
6.4 Other Systems Approaches for Efficient Graph Query Processing in DBMSs . . . 114

7 Conclusion **117**
7.1 Contributions . . . 117
7.2 Future Work . . . 119

References **121**

List of Figures

1.1 Example of a *'Twitter'* sub-graph. 2

1.2 Example Relation $R(from, to)$ highlighting a single output tuple $(1, 0,$ $n{+}1$, $3n{+}1$, $2n{+}3$) of query $R(a, b), R(b, c), R(c, d), R(d, e)$. 6

2.1 Graph Notation for Diamond query. 11

3.1 Example Queries. 15

3.2 Example of two binary join trees, a WCOJ tree, and a hybrid join tree evaluating the diamond-X query. 16

3.3 Generic Join with JAO $a_1a_2a_3a_4$ for diamond query in Figure 2.1. 20

3.4 Queries used to demonstrate the effects of JAOs. 23

3.5 Example join tree not in EmptyHeaded's GHD plan space for the 6-cycle query: $E(a_1, a_2), E(a_2, a_3), E(a_3, a_4), E(a_4, a_5), E(a_5, a_1)$. 27

3.6 Two plans where $P1$ joins two subqueries that share a query edge and $P2$ joins two subqueries that do not. Shows also operator pipelines of $P1$. 28

3.7 Input graph for adaptive JAO example. 34

3.8 Example adaptive WCOJ plan. Shows both the logical and operator pipelines. . . 35

3.9 Query Examples. 38

3.10 Run time (secs) of GraphflowDB plan spectrum for $Q1-8$. 'x' specifies the plan GraphflowDB chooses. 40

3.11 Run time (secs) of GraphflowDB plan spectrum for $Q11 - 13$. 'x' specifies the plan picked by GraphflowDB. 41

3.12 Run time (secs) of adaptive plans enumerated by GraphflowDB for queries $Q2-6$ and $Q10$. 'x' specifies the plan picked by GraphflowDB. 42
3.13 Plan with seamless mixing of intersections and binary joins on Q9. 46
3.14 Example of CTJ's tree decompositions (TDs) for Q2. 47
3.15 Scalability experiments. 51

4.1 Example input graph G and flat representation of $Q_{2H}{=}R(a,b),R(a,c)$ on G. . . 55
4.2 F-representations for Q_{2H} following F-trees $\mathcal{T}_1$ and $\mathcal{T}_2$. 59
4.3 Left-deep plan example for Q_{FFP}. 62
4.4 Example of intermediate chunk and its equivalent logical relation for Q_{FFP}. The first two vector groups are flattened to single tuples, while the last represents k_2 many tuples. 64
4.5 Input dataset and output example of Q_{FFP2}. 66
4.6 Left-deep plan example for Q_{FFP2}. 67
4.7 Example of Intermediate Chunk for $\mathtt{Q}_{2P_C}$. 67
4.8 F-trees supported by factorized vector execution. 68
4.9 Relative speedup/slowdown of the different systems in comparison to GF-V on LDBC10. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles. . . . 73

5.1 Example relation R and output results for $R(a_1,a_2),R(a_2,a_3),R(a_3,a_4),R(a_4,a_5)$ as an f-representation F_1 and as a d-representation Fd_1 following $\mathcal{T}_1$. 78
5.2 Output results for $R(a_1,a_2),R(a_2,a_3),R(a_3,a_4),R(a_4,a_5)$, where R is the relation in Figure 5.1a, as an f-representation F_2 and as a d-representation Fd_2 following $\mathcal{T}_2$. 80
5.3 Examples of three d-trees. $\mathcal{D}_1$ and $\mathcal{D}_2$ for the four-hop query $a_1{\rightarrow}a_2{\rightarrow}a_3{\rightarrow}a_4{\rightarrow}a_5$ and $\mathcal{D}_1$ for the triangle query $a_1{\rightarrow}a_2{\rightarrow}a_3,a_1{\rightarrow}a_2$. 82
5.4 Example of a WCOJ pipeline P and a Dataflow DF that adds reuse to P following $\mathcal{D}_{3H}$ in Figure 5.4a, where $\mathcal{M}_{a_3}(a_4) = R(a_3{=}v_{a_3},a_4,ts_2),P(ts_2)$ and $\mathcal{M}_{a_2}(a_4) = R(a_2{=}v_{a_2},a_3,-),R(a_3,a_4,ts_2),P(ts_2)$. 86
5.5 A query Q_c and two valid d-trees $\mathcal{D}_{c1}$ and $\mathcal{D}_{c2}$ 88

5.6 Example of a WCOJ pipeline P_{c1} and a Dataflow DF_{c1} that adds reuse to P_{c1} following $\mathcal{D}_{c1}$ in Figure 5.5b, where $\mathcal{M}_{a_2}(a_3) = R(a_2{=}v_{a_2}, a_3)$ and $\mathcal{M}_{a_1}(a_2, a_3, a_6, a_4, a_5) = R(a_1{=}v_{a_1}, a_2), R(a_2, a_3), R(a_1, a_4), R(a_4, a_5), R(a_5, a_1)$. 89

5.7 Example of a plan P as two pipelines and three extracted subpipelines P_s to which we add caching and reuse. 91

5.8 Example of internals of d-representation following $\mathcal{D}_{c1}$ in Figure 5.5b. 93

5.9 Plan spectrum for QC$_c$02 showing the run time of WCOJ plans (W) and dataflows with subquery caching and reuse (W$_c$). This is shown on different SF and selectivity (S) configurations. 100

5.10 A WCOJ plan P and a dataflow P_f which adds d-representation usage to P evaluating $Q_{4H}(a_1, a_4) = E(a_0, a_1), E(a_1, a_2), E(a_2, a_3), E(a_3, a_4)$. 101

5.11 Speedup over factorized vector execution when we reuse expressions as as we prune a percentage of edges from three datasets: 1) Amazon(Am); 2) Epinions(Ep); and 3) Google(Go). Query evaluated is: $R(a_0, a_1), R(a_1, a_2), R(a_2, a_3), R(a_3, a_4)$. 103

List of Tables

1.1 Example of *'user'* relation. 2
1.2 Example of *'tweet'* relation. 2
1.3 Example of *'follow'* relation. 2
1.4 Example of *'like'* relation. 2

3.1 Comparison against DBMSs using worst-case optimal joins. Q and D refer to the input query and dataset, respectively. 18
3.2 WCOJ plan run times in seconds with and without intersection cache for diamond-X on the Amazon ($|V|$=$403K$,E=$3.38M$) dataset from Stanford's network analysis project [112]. 22
3.3 Run time (secs), size of intermediate results (S_i), and i-cost of different JAOs for the asymmetric triangle query. 23
3.4 Run time (secs), size of intermediate results (S_i), and i-cost of different JAOs for the tailed triangle query. 24
3.5 Run time (secs), size of intermediate results (S_i), and i-cost of some JAOs for the symmetric diamond-X query. 24
3.6 A subquery catalogue. A is a set of adjacency list descriptors; μ is selectivity. . . 31
3.7 Datasets used. 36
3.8 Output size for queries in Figure 3.9 on the Amazon, Google, and Epinions datasets. 38
3.9 Run time (secs) of GraphflowDB (GF) and EmptyHeaded with good JAOs ($\texttt{EH}_g$) and bad JAOs ($\texttt{EH}_b$). TL indicates a timeout after 48 hrs. Mm indicates running out of memory. 45
3.10 Run time (secs) of GraphflowDB (GF) and CTJ. TL indicates the query did not finish in 48 hrs. 49

3.11 Average run time (secs) of GraphflowDB (GF) and CFL on large queries. Q*i(s/d)* is a query set of 100 randomly generated queries where *i* is the number of query vertices in the graph pattern and *s* and *d* specify sparse and dense queries, respectively. . . . 50

3.12 Q-error and catalogue creation time (CT) in secs for GraphflowDB for different z values. . . . 52

3.13 Postgres and GraphflowDB Q-error and number of catalogue entries (|R|) for GF for different h values. . . . 52

4.1 Datasets used for factorized vector execution evaluation. . . . 69

4.2 Run time in ms of GF−V and GF−F plans. . . . 71

4.3 Run time in ms for running the LDBC Queries on 5 systems: (i) GF−F; (ii) GF−V; (iii) VER for VERTICA; (iv) MON for MONET; and (v) NEO for NEO4J. . . . 75

4.4 Run time in ms for running the JOB Benchmark on 5 systems: (i) GF−F; (ii) GF−V; (iii) VER for VERTICA; (iv) MON for MONET; and (v) NEO for NEO4J. . . . 76

5.1 Datasets used. . . . 94

5.2 Run time (msec) for IC LDBC Queries (SFs 10 and 30) for GF−F and GF−D. . . . 97

5.3 Run time (msec) for IC LDBC Queries (SF 100) for GF−F and GF−D. . . . 98

5.4 Run time (msec) for IC LDBC Queries (SF 100) for GF−F and GF−D. . . . 98

5.5 Number of WCOJ plans on IC$_c$01 per speedup when using subquery caching and reuse buckets based on order of magnitude on: (a) SF10, selectivity 0.1% and 1% and SF30, selectivity 0.1%; and (b) SF30, selectivity 1%. . . . 100

5.6 Run time (secs) comparing GF−F and GF−D when evaluating $Q_{4H}(a_1, a_4) = E(a_0, a_1)$,$E(a_1, a_2)$, $E(a_2, a_3)$, $E(a_3, a_4)$ on: 1) Epinions (Ep); 2) Amazon (Am); and 3) Google (Go). . . . 102

5.7 Evaluating $Q_{4H}(a_1, a_4) = E(a_0, a_1)$,$E(a_1, a_2)$, $E(a_2, a_3)$, $E(a_3, a_4)$, with a WCOJ with JAO $(a_1,)$, we report the total number of joins (#TJ) with no caching at each depth. We also report the % and number of saved joins (#SJ) by caches in prior depths. Finally, we report the number of cache misses (#M) & hits (#H) at each depth. Evaluation on: 1) Epinions (Ep); 2) Amazon (Am); and 3) Google (Go). . . 102

Chapter 1

Introduction

Querying graph-structured data, or graph data for short, is integral to a wide range of analytical applications such as recommendation and fraud detection. Graph data typically refers to highly connected datasets, *i.e.*, ones with a prevalence of many-to-many (n-m) relationships. Examples of such datasets appear in social networks such as the who-follows-whom Twitter graph [7] in which a user follows many users, and payment services like AliPay [1] where an account can transfer money to many others. Often, querying these datasets entails finding instances of a graph pattern in a much larger input dataset. For example, diamond patterns in the who-follows-whom Twitter graph can be used for recommendations [73, 74]. Complex cyclic patterns in the transaction network of AliPay can be used to detect potential fraud [150]. In this thesis, we refer to the combination of highly connected datasets and pattern-finding queries as *graph workloads*.

Pattern-finding queries can be represented as conjunctive queries over edge/relationship tables in a graph, where the relationship table contains at least two attributes representing the source and destination IDs of the edges though it can contain other attributes. Throughout this thesis, we will use a Datalog notation to represent these queries. Figure 1.1 shows an example of a subgraph from Twitter data. Tables 1.1 to 1.4 show the subgraph in its relational representation, with entity relations *user'* and *tweet'* and relationship relations *follow'* and *like'*. The diamond-pattern query used by Gupta *et al.* [74] to make recommendations on the who-follows-whom Twitter graph can be represented as follows:

$$Q(a, d) = follow(a, b), follow(b, d), follow(a, c), follow(c, d)$$

where d could be a good recommendation for a.

Existing database management systems (DBMSs) tend to not perform well on the conjunctive queries above when the underlying relations model highly connected datasets, *i.e.*, when the

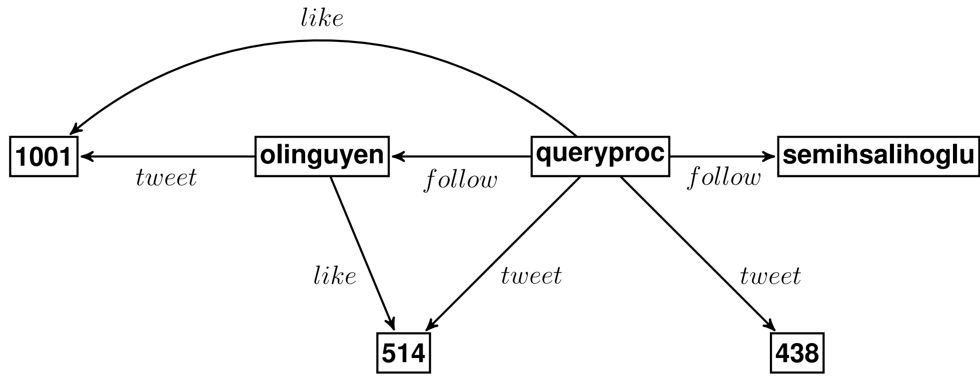


Figure 1.1: Example of a *'Twitter'* sub-graph.

username	**registration_date**
queryproc	Oct. 2013
olinguyen	Jul. 2009
semihsalihoglu	Apr. 2009
...	...

Table 1.1: Example of *'user'* relation.

id	**tweet_text**	**author**
438	'writing a thesis ...'	queryproc
514	'hoping to join a ...'	queryproc
1001	'Japan is awesome!'	olinguyen
...	...	...

Table 1.2: Example of *'tweet'* relation.

follower	**followee**
queryproc	olinguyen
queryproc	semihsalihoglu
...	...

Table 1.3: Example of *'follow'* relation.

username	**tweet_id**
queryproc	1001
olinguyen	514
...	...

Table 1.4: Example of *'like'* relation.

queries contain n-m joins. The evaluation of such queries in existing DBMSs uses binary join algorithms, which iteratively take two relations and joins them until all relations are joined. This process results in intermediate relations that explode in size and can contain a lot of repetition that leads to redundant computation. As a result, existing query processors face serious performance and scalability challenges, resulting in slow run times and in many cases timeouts. Industry users

in a recent survey [159, 161] have also identified scalability as the most pressing challenge for existing DBMSs evaluating graph workloads.

This challenge serves as the core motivation for the research conducted in this thesis. Specifically, this thesis aims to address the problem of efficiently evaluating n-m join-heavy queries that are prevalent in graph workloads by integrating novel query processing techniques into DBMSs.

1.1 Thesis Statement

The central thesis argument is that in order to mitigate the explosion in intermediate results under m-n join-heavy queries, existing DBMSs must adopt two novel techniques:

1. *Worst-case optimal joins:* Recent theoretical results [24, 137] showed that binary joins, can be suboptimal when finding cyclic patterns. Binary joins have asymptotically worse run-times than the worst-case, *i.e.*, maximum, output sizes of these queries. A set of new worst-case optimal joins (WCOJs) have been developed to address the suboptimality of binary joins and thus avoid producing unnecessary results that are guaranteed to not be part of the final output.
2. *Factorized representations:* Factorized representations [145] are a succinct and lossless representation for relations and can be used to compress intermediate and output query results. In factorized representations, relations are represented as unions of Cartesian products which factors out common values across sets of tuples. This technique is particularly effective for representing intermediate relations that are constructed when evaluating large acyclic n-m join queries.

Since graph workloads that motivate this thesis consist of read-only pattern queries, we take as a foundation the architecture of existing read-optimized query processors that adopt several common wisdom principles including pipelined and vector-based query execution [34, 167], or vector execution for short, morsel-driven parallelism [106], and dynamic-programming-based join optimizers [37]. However, directly adopting WCOJs and factorization as done in early implementations prior to this thesis work heavily deviate from this traditional architecture [22, 26, 137, 145]. For example, these papers and implementations assume a non-pipelined execution where operators consume entire relations and output entire relations. They also ignore important details such as the query optimizer, its cost model, the cardinality estimator, and in some work descriptions of physical operators. An efficient and practical adoption of these techniques requires rethinking these core DBMS components. These are the technical challenges that are addressed in this thesis.

1.2 Thesis Contributions

We propose three different query processing techniques for analytical DBMSs. All of these techniques are integrated in GraphflowDB [3, 92], which is an in-memory analytical DBMS that was built from scratch to facilitate the research of this thesis. The code developed as part of this thesis is publicly available on Github [70, 71, 72]. Next, we give an overview of the three techniques. For each, we give a broad overview, including considerations and challenges we tackle, and the specific contributions made within query execution and optimization, which we cover in detail in later chapters.

1.2.1 End-to-end Adoption of Worst-case Optimal Joins

Overview:

The size of the intermediate results of binary joins on cyclic queries can be asymptotically larger than the maximum possible final output size of the query. This maximum output size is known as the AGM bound of a query [24]. Given the sizes of a set of relations $|R_1|$, ..., $|R_n|$ and a conjunctive query Q on these relations, the AGM bound is the maximum output size of Q under all possible database instances with these relation sizes. For example, the AGM bound of the triangle query $R(a, b), R(b, c), R(c, a)$, where $|R| = N$, is $N^{\frac{3}{2}}$. Binary join plans would first evaluate the open wedge $R(x, y), R(y, z)$ before closing the cycle. On certain database instances, binary join plans would generate N^2 intermediate results, which is larger than the AGM bound of the triangle query.

Worst-case optimal joins (WCOJs) correct for the suboptimality of binary joins and do so by introducing a novel evaluation approach. Specifically, binary joins rely on a *table-at-a-time evaluation* approach, while WCOJs introduce an *attribute-at-a-time evaluation* approach. Table-at-a-time evaluation executes a sequence of binary joins between base and intermediate relations to evaluate the conjunctive query Q. Each binary join output effectively matches a larger subset of the relations of Q in the input dataset until Q is fully matched. An attribute-at-a-time evaluation chooses an ordering on join attributes in Q, and uses an n-way join operator to match join attributes in order. In our work, we rely on an n-way join operator that performs multiway intersections. In graph terms, this computation intersects one or more neighbourhoods of vertices to extend partial matches to one more join attribute.

Contributions:

The new attribute-at-a-time evaluation approach that the theory of WCOJs introduces, is asymptotically optimal irrespective of the join attribute ordering (JAO) chosen. The theory however

does not give advice on:

1. *How should a system choose a JAO?* In practice, different JAOs lead to very different performances.

2. *When should binary join algorithms be used solely or in conjunction with WCOJ algorithms?* Decades of research and engineering has shown that binary joins can be efficient on many queries. Further, we will demonstrate in this thesis that on many queries, using plans that mix binary and WCOJ algorithms is superior to binary join-only or WCOJ-only plans. This also indicates that we need an optimizer capable of discriminating between these plans as both evaluation approaches differ significantly. Therefore a related research question is: *How should a system optimize and generate such plans?*

In Chapter 3 of this thesis, we broadly tackle these challenges and present an end-to-end adoption of WCOJs in GraphflowDB. Our approach is based on a dynamic programming-based optimizer with a new cost model, called *i-cost*, for intersection cost, that can generate plans with good JAOs and generate plans that seamlessly mix binary join operators with a new n-way join operator that we introduced. We show that the plan space of our optimizer subsumes tree decomposition-based query optimizers that was introduced in prior work to integrate WCOJ algorithms in DBMSs [11]. We further show how to choose JAOs adaptively in some plans. Finally, we present extensive "plan space" experiments that plot the performance of each plan of a large suite of queries. These experiments aim to answer the question of "Which type of plans among WCOJ, binary join, or hybrid, are best for which queries?" based on how-cyclic the queries are.

1.2.2 Factorized Vector Execution

Overview:

The sizes of intermediate results using WCOJs on many queries can still be very large. Furthermore, they may contain a lot of redundancy *i.e.*, repetition that leads to a lot of redundant computation. For example, consider the following four-hop query:
$Q(a, b, c, d, e) = R(a, b), R(b, c), R(c, d), R(d, e)$, R is the binary relation shown in Figure 1.2
The output query size is n^4 (same as the AGM bound) and is as follows:

$$OUT_Q = \{ (1, 0, n{+}1, 3n{+}1, 2n{+}1), ..., (1, 0, n{+}1, 3n{+}1, 3n), ..., \\ (n, 0, n{+}1, 3n{+}1, 2n{+}1), ..., (n, 0, n{+}1, 3n{+}1, 3n), ..., \\ (n, 0, \quad 2n, 3n{+}1, 2n{+}1), ..., (n, 0, \quad 2n, 3n{+}1, 3n) \}$$

Factorized representations are lossless compressed representations of relations that use algebraic factorization over flat tuples *i.e.*, factoring out common terms, to reduce the overall size of

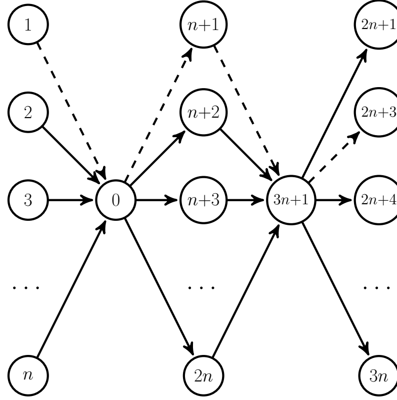


Figure 1.2: Example Relation $R(from, to)$ highlighting a single output tuple $(1, 0, n{+}1, 3n{+}1, 2n{+}3)$ of query $R(a, b), R(b, c), R(c, d), R(d, e)$.

the query results. For example, a more succinct representation of $Q(a, b, c, d, e)$ can be:

$$
\begin{aligned}
OUT_Q = \{\ & \{(a{:}i) \mid i \in [1, n]\ \} \times (b{:}0) \times (c{:}n{+}1) \times (d{:}3n{+}1) \times \{(e{:}j) \mid j \in [2n{+}1, 3n]\ \}, \\
& \{(a{:}i) \mid i \in [1, n]\ \} \times (b{:}0) \times (c{:}n{+}1) \times (d{:}3n{+}1) \times \{(e{:}j) \mid j \in [2n{+}1, 3n]\ \}, \\
& \ldots \\
& \{(a{:}i) \mid i \in [1, n]\ \} \times (b{:}0) \times (c{:}n{+}1) \times (d{:}3n{+}1) \times \{(e{:}j) \mid j \in [2n{+}1, 3n]\ \}\ \}
\end{aligned}
$$

This representation has $(2n + 3) \times n = 2n^2 + 3n$ fields in total. This is less than the AGM bound of $4n^2$ fields. The factorization scheme is possible because attributes a and e are independent conditioned on the (b, c, d) values ***i.e.***, given a fixed (b, c, d) values, the a values do not change the e values and vice versa.

The representation above is called an *f-representation* in the theory of factorization. The theory and our example above shows that such representations can even be strictly smaller than the AGM bound of queries. More importantly, when queries contain m-n joins, a system can find good factorized representations of query results during compilation time, simply by inspecting the dependencies between the attributes in the query [145].

Factorized representations however tend to have a very different physical representation from that of flat tuples used in existing DBMSs. They are suitable for being represented as tries, as done in some prototype implementations and envisioned in the original papers on the foundations of factorized representations [26, 145]. Processing over trie-based intermediate results however deviates significantly from existing query execution architecture, for which processing on top of flat tuples is essential. These implementations also are not pipelined and they materialize entire intermediate outputs between operators. This makes it harder for existing DBMSs to adopt factorization and to abandon the core principles of vector execution.

Contributions:

The main research question we ask in Chapter 4 of this thesis is: *How can existing pipelined and vector-based query processors benefit from f-representations?* Our query processor design, which we call *factorized vector execution*, is based on using f-representations. Although processing arbitrary factorized representations seems to indeed require adopting tries as intermediate relations, our design finds a sweet spot by generating a restricted set of *factorization schemes*, *i.e.*, structure of Cartesian products, without abandoning vector execution. Yet, as we show this design provides significant benefits over many common queries in graph workloads. Importantly, factorized vector execution requires minimal changes to the physical operators of query executors and is easy to integrate in modern query processors. The combination of factorized vectors and worst-case optimal joins gives upwards of three orders of magnitude speedups over traditional analytical DBMSs. Factorized vector execution introduces an order of magnitude speedup over only worst-case optimal joins.

1.2.3 Caching and Reuse of Intermediate Results

Overview:
The f-representations used for intermediate relations in our factorized vector execution approach can still contain redundancy on many queries. Consider for example the f-representation introduced above for the four-hop query:

$$\begin{aligned} OUT_Q = \{ & \{(a\!:\!i) \mid i \in [1,n]\} \times (b\!:\!0) \times (c\!:\!n{+}1) \times (d\!:\!3n{+}1) \times \{(e\!:\!j) \mid j \in [2n{+}1, 3n]\}, \\ & \ldots \\ & \{(a\!:\!i) \mid i \in [1,n]\} \times (b\!:\!0) \times (c\!:\!n{+}1) \times (d\!:\quad 2n) \times \{(e\!:\!j) \mid j \in [2n{+}1, 3n]\} \} \end{aligned}$$

For each (b, c, d) prefix, the same set of a values $\{i \mid i \in [1, n]\}$ and e values $\{i \mid i \in [1, n]\}$ are repeated. The theory of factorization introduces even more succinct representation schemes *by using definitions*. A definition is a representation of the subquery results that are cached and reused *e.g.*, $D_1 \leftarrow \{(a : i) \mid i \in [1, n]\}$ and $D_2 \leftarrow \{(e : j) \mid j \in [2n + 1, 3n]\}$. The definitions D_1 and D_2 can be reused in the representation above as follows:

$$\begin{aligned} OUT_Q = \{ & \{ D_1 \times (b\!:\!0) \times (c\!:\!n{+}1) \times (d\!:\!3n{+}1) \times D_2 \}, \\ & \ldots \\ & \{ D_1 \times (b\!:\!0) \times (c\!:\!n{+}1) \times (d\!:\quad 2n) \times D_2 \} \} \end{aligned}$$

F-representations with definitions are called *d-representations* in the theory of factorization. The theory shows that d-representations can be strictly smaller than the size of f-representations, which can also be strictly smaller than the AGM bound. Further, similar to f-representations, it

is possible to identify viable definitions based on the query attribute dependencies at query compilation time. Prior to our work and at the time of writing this thesis, no d-representation-based query processor implementation has been proposed in the literature. The envisioned implementation in the original papers on the foundations of factorized representations is based on tries. However, as we previously argued, it is challenging to adopt tries in pipelined and vector-based query processors, which require processing on flat-tuples.

Contributions:
The main research question we ask in Chapter 5 is: *How can existing pipelined and vector-based query processors benefit from d-representations?* Our approach generates plans that comprise of multiple pipelines of operators that are executed consecutively. Within each pipeline we identify a sub-pipeline *i.e.*, a subset of contiguous operators, whose results are dependent on a single input attribute value. The results of these sub-pipeline are materialized per unique attribute value. The sub-pipeline executes if the input attribute value has not been encountered and is skipped otherwise. When skipped, the result has been previously materialized and can be reused. To make this possible, we introduce novel physical operators and dataflows constructed as directed acyclic graph (DAG) of operators. These dataflows rely on the foundation of our factorized vector executor. Specifically, these dataflows process data on top of the vectors used by the factorized vector executor while caching intermediate results as d-representations.

Our approach requires the introduction of some novel operators and use techniques inspired by traditional BFS and DFS graph traversals *e.g.*, frontiers. The enumeration of these plans is not straightforward and as such, we generate them in two steps: 1) use a cost-based optimizer to choose a factorized vector execution plan, *i.e.*, a plan that will generate f-representations; then 2) use a rule-based optimizer to turn the plan into a DAG dataflow capable of caching and reusing intermediate results. We cache the results into nested hash tables, which we then also use for a hash join operator. We further implement necessary iterators over these hash tables. Finally, we study when is results caching and reuse most effective. On traditional benchmark, caching and reuse leads to benefits of up to 60x over factorized vector execution.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the queries we optimize for. The chapter also provides an overview of GraphflowDB, an in-memory DBMS developed to facilitate this research. We give overall context in which we make our contributions. Chapters 3, 4, and 5 present the query processing techniques we propose. They respectively

cover: (i) end-to-end adoption of a worst-case optimal join algorithm; (ii) factorized vector execution; and (iii) caching and reuse of intermediate results. We give the necessary background on worst-case optimal joins and factorized representations as needed in each of the chapters. Related work is presented in Chapter 6. We present the conclusion of this thesis and directions for future research in Chapter 7.

Chapter 2

Preliminaries and Background

In this chapter, we introduce the queries we consider in this thesis, our query notation, and GraphflowDB [3, 92], the DBMS in which we integrate our query processing techniques. For the queries, we also introduce a simplified query graph notation that we use in later chapters.

2.1 Queries and Notation

Each conjunctive query can be seen as having two parts: 1) graph pattern finding; and 2) attribute filtering. The graph pattern finding is equivalent to multi-way equi-joins between relations ***e.g.***, the diamond query $follow(a,b), follow(b,d), follow(a,c), follow(c,d)$ is a 4-way equi-join between four "copies" of the $follow$ relation. Attribute filtering extends pattern finding queries by adding predicates ***e.g.***, $a{=}olinguyen$ which on the diamond query translates to diamonds starting from user $olinguyen$, or $b.registration_date \leq c.registration_date$ which eliminates symmetry in the output instances. Adding these two predicates to the diamond query leads to the following query:

$$\begin{aligned} Q(d) :- \ & follow(a,b), follow(b,d), follow(a,c), follow(c,d), a{=}olinguyen, \\ & b.registration_date \leq c.registration_date \end{aligned}$$

Graph pattern finding within a conjunctive query can be rewritten using a graph notation. We assume that relationships are directed from left-to-right ***e.g.***, each edge record $(v_i,v_i) \in follow(a,b)$ implies v_i follows v_j ***i.e.***, $v_i{\rightarrow}v_j$. As such, the diamond query can be presented graphically as shown in Figure 2.1. Entity and relationship relations are depicted as query vertices and query edges in this graph notation. In text, the query is $Q(V_Q, E_Q)$ where V_Q and E_Q

are the query vertices and edges, respectively. From the diamond example, a_1, $a_2 \in V_Q$ and $a_i \rightarrow a_j \in E_Q$. In later chapters, we will use this notation at times, both in text and figures, to describe join algorithms as well as our query plans.

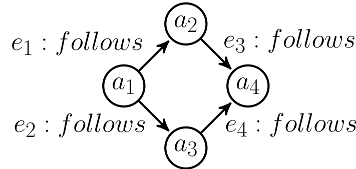


Figure 2.1: Graph Notation for Diamond query.

2.2 GraphflowDB Overview

GraphflowDB [3, 92] is an in-memory DBMS that adopts the property graph data model [21]. In the property graph data model, application data is modelled by: (i) nodes, representing entities; (ii) edges, representing relationships; and (iii) arbitrary key-value properties on nodes and edges. The data stored in GraphflowDB is queried with Cypher [146], a declarative language where users describe graph patterns similar to the graph notation above to search for in their graphs. Cypher further allows the use of traditional relational operations *e.g.*, filtering, projections, orderings, and aggregations for further processing.

Cypher is a SQL-like language and its semantics can be understood in relational algebra that is extended with recursion. Overall the data model and query language of GraphflowDB is general enough to support a very wide range of database applications and offer an alternative to the relational data model and SQL. This has been observed in a recent survey that found a very wide range of applications built on top of GDBMSs [160]. These applications process data, often associated with GDBMSs, such as social and protein networks, as well as data often associated with RDBMSs, such as orders, products, and transactions.

GraphflowDB is optimized for graph pattern finding *i.e.*, performing joins between node records along predefined edge records. As such, GraphflowDB relies on pointer-based joins *i.e.*, uses system-level dense integer IDs of nodes, which serve as pointers to look up neighbours. This contrasts with and can be more efficient than value-based joins on arbitrary attributes in RDBMSs. To implement pointer-based joins, GraphflowDB relies on three components:

- System-level dense integer node IDs: node records are given consecutive IDs *i.e.*, surrogate keys, starting from 0 to $|V|$ for an input data graph $G(V, E)$.

- Adjacency list indexes: the edge records are indexed using the system-level node IDs providing constant-time access to all in-coming/outgoing "neighbours" of a node.

- `Index Nested Loop Join` (`INLJ`): This operator uses adjacency list indexes to "extend" a node record to its "neighbours", which is equivalent to joining a node record with other node records using the edges in the graph. Such join operators are common in other GDBMSs *e.g.*, `Expand` operator in Neo4j. A variant of this operator, called `Index Multiway Join` operator will be introduced in Chapter 3 when we describe our implementation of WCOJ algorithms.

In addition to the `INLJ` operator, GraphflowDB also supports a `Scan` and `Hash Join` (`HJ`) operators:

- `Scan`: The operator scans the forward or backward adjacency lists for the specific edge relation and outputs each matched edge $u \rightarrow v$ as a (u,v) tuple.

- `Hash Join` : This operator is used to generate bushy/hybrid plans that use binary join and WCOJ-style joins. We use the classic hash join operator which split into multiple operator pipelines and `HJ` is implemented as separate `Build` and `Probe Hash Table` operators. This first creates a hash table of all of the tuples of Q_{c1} on the common query vertices between Q_{c1} and Q_{c2}. The table is then probed for each tuple of Q_{c2}.

Parallel Query Execution: GraphflowDB implements a work-stealing-based, *morsel-driven* [106] parallelization technique. Let w be the number of threads in the system. We give a copy of a physical plan P to each worker and workers steal work from a single queue to start scanning ranges of edges in the SCAN operators. Threads can perform extensions in the `INLJ`/`IMJ` operators without any coordination. Hash tables used in `Hash Join` are partitioned into $d >> w$ many hash table ranges. When constructing a hash table, workers grab locks to access each partition but setting $d >> w$ decreases the possibility of contention. Probing does not require coordination and is done independently.

GraphflowDB evolved and improved throughout several years during which the research of this thesis was conducted. As a result, the design of the system's storage of node and edge properties and adjacency lists was different throughout the development of the different query processing techniques. For example, initially we used an in-memory row storage (Chapter 3) and later moved to an in-memory columnar one (Chapters 4 and 5) [75]. Overall, these modifications will not be important for understanding our query processing techniques. When relevant for the techniques we introduce in this thesis, we will explicitly give more details of the storage. Similarly, the query processor and optimizer of the system have also changed throughout the

thesis but since each chapter is about query processing and optimization, these changes form the technical content of each chapter. However, the above core operators have remained similar to their above descriptions. The primary differences have been in the representation of the intermediate relations that these operators take as input and output. When necessary, we will provide some details on these in different parts of this thesis.

Chapter 3

Adopting Worst-case Optimal Joins

This chapter focuses on how to mitigate the large intermediate relations that are generated when evaluating cyclic join queries over many-to-many relationships. For cyclic queries, the recent theory of worst-case optimal join (WCOJ) algorithms [137] has observed that traditional binary join plans of existing systems generate provably large intermediate results. As we mentioned in the Chapter 1, the way this argument is made is to compare the size of the intermediate results generated by binary join plans to the maximum number of possible results there can be for a query, which is known as the AGM bound of the query [24]. For example, consider the triangle query $R(a_1, a_2), R(a_2, a_3), R(a_3, a_1)$ shown in Figure 3.1a. The AGM bound of this query on a relation R with N tuples (or a graph with N edges) is $N^{\frac{3}{2}}$. Yet, one can show that there are input relations R (or graphs) on which any binary join plan, which iteratively perform the join two-relations at a time, will produce $\Omega(N^2)$ many intermediate problems.

The core of the problem is that any binary join plan will perform 2 joins in this query: (i) the first join will join two of the relations, say $R(a, b)$ and $R(b, c)$, which in graph terms finds open triangles in the input graph; and (ii) join the result of step (i) with the third relation, in this case $R(c, a)$, which closes the triangles. Note that the second join is effectively a filter, so can only decrease the number of intermediate results generated in the first join. Because the first step computes open triangles and there can be much more open triangles ($\Omega(N^2)$) than closed triangles ($O(N^{\frac{3}{2}})$), this computation can be quite inefficient. The theory of WCOJ algorithms addresses this problem by using multiway-join algorithms that perform the join one attribute at a time. As we will review in Section 3.2, the core algorithmic operation of certain WCOJ algorithms is to perform intersections of sets of values. In graph terms, these correspond to intersecting adjacency lists.

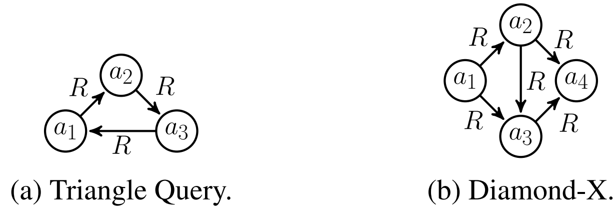


Figure 3.1: Example Queries.

This theory, however, has two shortcomings. First, when using WCOJ algorithms, one needs to pick an attribute ordering and the theory gives no advice as to how to pick a good attribute ordering for WCOJ plans. Specifically, the theory shows that irrespective of the attribute ordering picked, WCOJ algorithms are guaranteed to never generate more intermediate results than the AGM bound of any query. The problem of picking an attribute ordering is very important in practice and similar to how binary join plans can be optimized by picking an order of the relations to join. Second, the theory ignores plans with binary joins, which have been shown to be efficient on many queries by decades-long research in databases. That is, the theory does not give an advice on when to use binary join plans. These two shortcomings motivate the work presented in this chapter. Specifically, in addition to describing a complete end-to-end implementation of WCOJ algorithms in a pipelined query processor, we study how to optimize multiway joins, *i.e.*, , pick good attribute orderings when using WCOJ algorithms, and generate efficient plans that use both worst-case optimal multiway joins and binary joins.

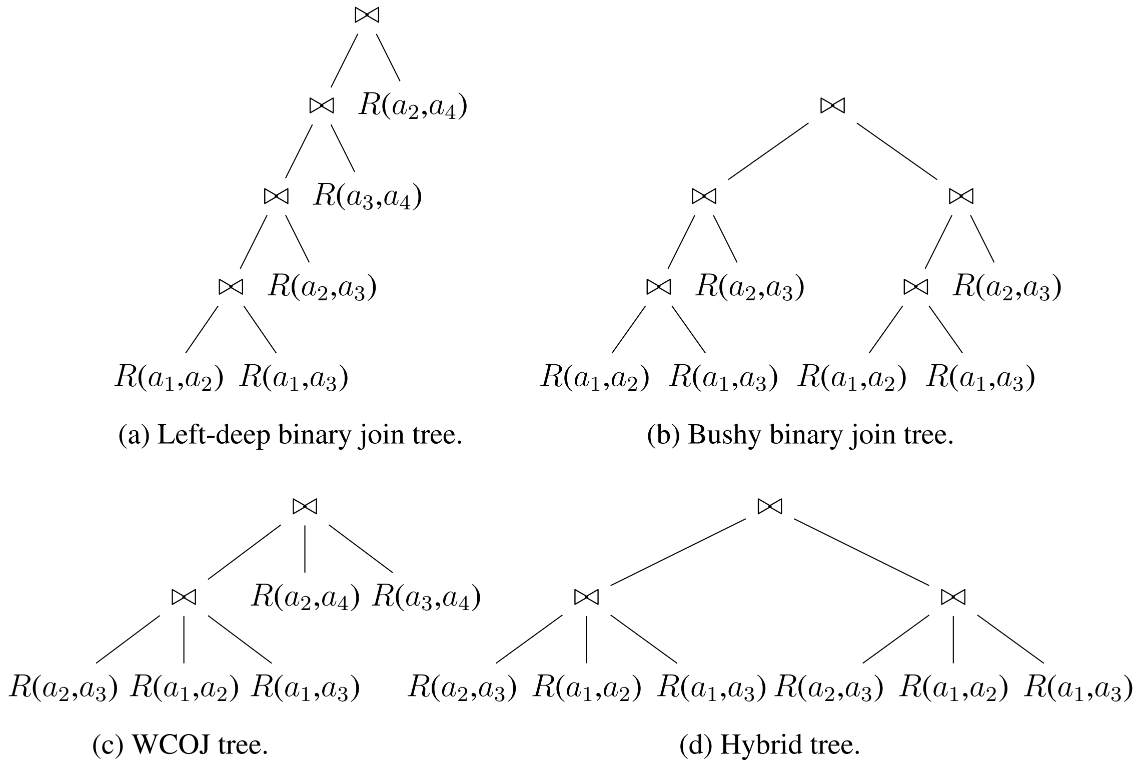
Throughout the chapter we will distinguish between three types of plans: a) *binary join plans*, both left-deep and bushy; b) *WCOJ* plans using multi-way joins; and c) *hybrid* plans containing both evaluation approaches, which are bushy plans with WCOJ subplans. Figures 3.2a, 3.2b, 3.2c, and 3.2d show an example of each plan evaluating the *diamond-X* query, Q_{DX}:

$$Q_{DX} = R(a_1, a_2), R(a_1, a_3), R(a_2, a_3), R(a_2, a_4), R(a_3, a_4)$$

where $R(from, to)$ is an Edge relation. Figure 3.1b shows the query pictorially. The plans in Figures 3.2a-3.2d are logical plans. We cover the details of the physical implementation of these plans in Section 3.2.

3.1 Existing Approaches and Overview of Contributions

To position our approach to end-to-end adoption of WCOJ algorithms and our contributions in the context of prior work, we first briefly review the related work and then describe our contributions. More detailed coverage of related work is in the related work chapter of this thesis (Chapter 6).



(a) Left-deep binary join tree.

(b) Bushy binary join tree.

(c) WCOJ tree.

(d) Hybrid tree.

Figure 3.2: Example of two binary join trees, a WCOJ tree, and a hybrid join tree evaluating the diamond-X query.

Perhaps the most common approach adopted by GDBMSs (*e.g.*, Neo4j), RDBMSs (*e.g.*, SAP HANA), and RDF systems (*e.g.*, RDF-3X [133]), is to evaluate conjunctive queries with binary join plans. As observed in prior work [135], binary join plans are inefficient on highly-cyclic queries like cliques.

Several prior solutions, such as BiGJoin [20], and the LogicBlox system [22] have studied evaluating queries relying solely on WCOJs. We demonstrate empirically in our evaluation (Section 3.7) that only using WCOJs is inefficient on large cyclic queries. In addition, these solutions use simple heuristics to select join attribute orderings or do so arbitrarily.

CTJ is another system only using a WCOJ algorithm [91]. An important advantage of WCOJ algorithms is their small memory footprint. For example, when executed in a purely pipelined fashion, such algorithms do not require memory to keep large intermediate results. CTJ authors observe that by keeping a cache of certain intermediate results and reusing these results, the performance of WCOJ algorithms can be improved.

The EmptyHeaded system [11], which is the closest to our work, is the only system we are aware of that mixes worst-case optimal joins with binary joins. EmptyHeaded plans are based on *generalized hypertree decompositions* (GHDs) of the input query Q. A GHD is effectively a join tree T of Q, where each node of T contains a subquery of Q. EmptyHeaded evaluates each subquery using a WCOJ plan, *i.e.*, using only multi-way join intersections, and then uses a sequence of binary joins to join the results of these subqueries. As a cost metric, EmptyHeaded uses the *generalized hypertree widths* of GHDs and chooses a minimum-width GHD. This approach has three shortcomings: (i) if the GHD contains a single subquery, EmptyHeaded arbitrarily chooses the join attribute ordering for that query, otherwise it chooses the orderings for the subqueries using a simple heuristic; (ii) the width cost metric depends only the input query Q, so when running Q on different datasets, EmptyHeaded always chooses the same plan; and (iii) the GHD plan space does not allow plans that can perform multi-way joins after binary joins. As we demonstrate, there are efficient plans for some queries that *seamlessly mix binary joins and multi-way joins* and do not correspond to any GHD-based plan of EmptyHeaded.

3.1.1 Our Contributions

Table 3.1 summarizes how our approach compares against prior solutions. Our main contribution is a dynamic programming optimizer that generates plans with both binary joins (`HJ` or `INLJ`) and multi-way joins. GraphflowDB implements multi-way joins as a single `Index Multiway Join` (`IMJ`) operator. This operator does two or more adjacency list lookups and intersects these lists. Let Q be the query whose graph pattern contains m query vertices. Our dynamic programming optimizer enumerates plans in a bottom-up fashion to evaluate each subquery of Q with k vertices. For $k = 1$, each Q_1 subquery contains one of the m query vertices in Q evaluated with a scan operator. For $k = 2, ..., m$, the optimizer has two choices: (i) joining the relations of two subqueries Q_{s1} and Q_{s2} using `HJ`; or (ii) extending a subquery $Q_{k\text{-}1}$ with one query vertex into Q_k using the `INLJ`/`IMJ` operators. This generates all possible WCOJ plans for the query as well as a large space of hybrid plans which are not in EmptyHeaded's plan space.

To rank WCOJ plans, our optimizer uses a new cost metric called *intersection cost* (i-cost). I-cost represents the amount of intersection work that a plan P does based on the sizes of the adjacency lists intersected throughout P. For ranking hybrid plans, we combine i-cost with the cost of binary joins. Our cost metrics account for the properties of the input graph, such as the distributions of the forward and backward neighbourhood sizes and output size of different

[1] LogicBlox is not open-source. Two publications describe how the system chooses join attribute orderings; a heuristics-based [141] and a cost-based [22] technique that uses sampling.

	QVO	Binary Joins?
BiGJoin	Arbitrarily	No
LogicBlox	Heuristics or Cost-based[1]	No
CTJ	Heuristic + Cost-Based (uses caching)	No
EmptyHeaded	Arbitrarily	Cost-based: depends on Q
GraphflowDB	Cost-based & Adaptive	Cost-based: depends on Q and D

Table 3.1: Comparison against DBMSs using worst-case optimal joins.
Q and D refer to the input query and dataset, respectively.

subqueries computed as part of a plan. Unlike EmptyHeaded, this allows our optimizer to choose different plans for the same query on different input graphs.

To estimate a plan's cost, we propose a new cardinality estimator that uses a *subquery catalogue* to estimate the intersection work and the number of intermediate results. The catalogue contains estimates for: (i) the distribution of adjacency list sizes, where the list is obtained by extending a vertex of a small graph pattern. (ii) selectivity of subqueries, where the subquery is obtained by extending a smaller subquery by one query vertex using a multiway join.

Our second contribution is an *adaptive technique* for choosing the join attribute ordering of WCOJ subplans during query execution. Consider a WCOJ subplan that extends output tuples of subquery Q_i into a larger subquery Q_k. Suppose there are r possible join attribute orderings, $\sigma_1, ..., \sigma_r$, to perform these extensions. Our optimizer tries to choose the ordering σ^* with the lowest cumulative i-cost when extending all partial output tuples of Q_i. However, for any specific tuple t of Q_i, there may be another σ_j that is more efficient than σ^*. Our adaptive executor re-evaluates the cost of each σ_j for t based on the actual sizes of the adjacency lists of the vertices in t, and chooses a new ordering.

We incorporate our optimizer into GraphflowDB and evaluate it across a large class of queries and datasets. First, we show that our optimizer is capable of choosing close to optimal plans across many queries. Second, we show that some plans that are not in EmptyHeaded's plan space, are up to 68x more efficient than EmptyHeaded's plans. Finally, we also show that our adaptive technique improves the run time of some plans by up to 4.3x. In some queries, adaptivity improves the run time of every plan but more generally, it makes GraphflowDB more robust against bad join orderings.

3.2 Preliminaries: Generic Join Algorithm

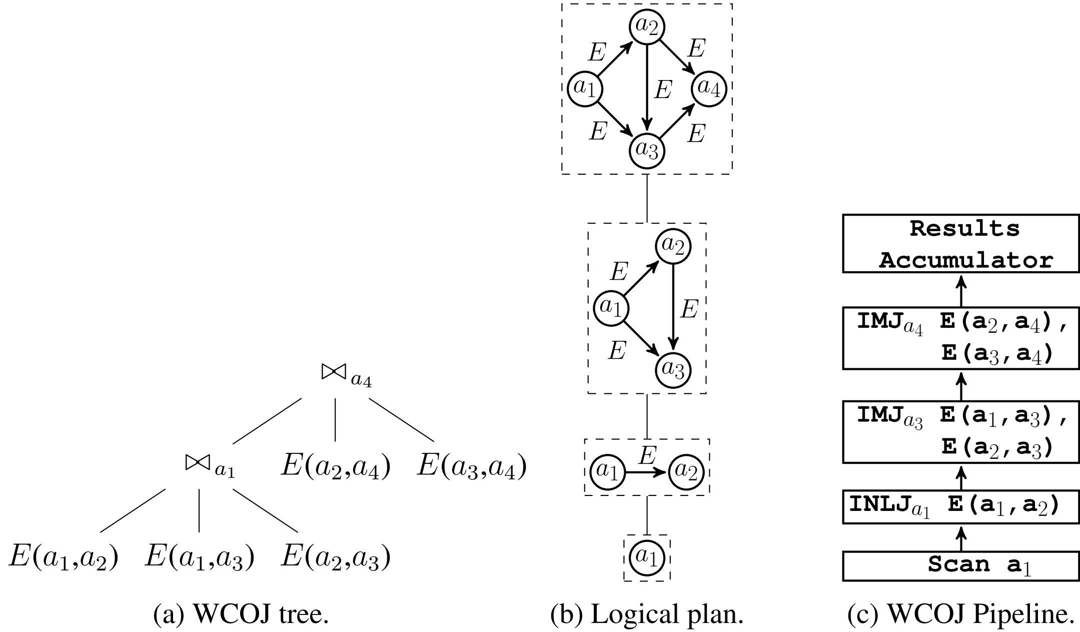
In this section, we introduce Generic Join [135], the WCOJ algorithm we adopt in GraphflowDB. Note that the queries we consider in this chapter contain only equi-joins on edge relations with no further predicates. In the literature, these queries are called *subgraph queries* [169].

Generic Join evaluates queries one attribute at a time. We describe the algorithm in graph terms when evaluating $Q(V_Q, E_Q)$; Ngo *et al.* [135] and Freitag *et al.* [65] give equivalent relational descriptions. Using the graph notation from Section 2.1, the algorithm evaluates queries by extending one query vertex at a time with two main steps:

- **Join Attribute Ordering (JAO):** Generic Join first chooses a JAO *i.e.*, an ordering on the query vertices $V_Q = \{a_1, a_2, ..., a_m\}$, which is a tuple such as $\sigma = (a_3, a_1, ..., , a_m)$. σ specifies the ordering to evaluate m different subqueries $Q_1, Q_2, ..., Q_m$. Each $Q_i(V_i, E_i)$ contains the query vertices $V_i = \{\sigma[1], ..., \sigma[i]\} \subseteq V_Q$ *e.g.*, $V_2 = \{\sigma[1], \sigma[2]\} = \{a_3, a_1\}$. The query edges E_i are the edges between vertices V_i in Q. We say Q_k is the projection of Q into the first k query vertices of σ. For simplicity, we assume each Q_k to be connected. Otherwise, Generic Join computes expensive Cartesian products to produce intermediate results, which we will omit in our description.

- **Iterative Subquery Evaluation:** Generic Join iteratively produces output tuples for each Q_k using Q_{k-1}'s tuples as input. Given $|\sigma| = m$, Generic join has m iterations. Each iteration produces values for $a_k = \sigma[k]$. Specifically, an iteration k produces a set of *k-matches*, *i.e.*, output tuples of size k where each tuple 'matches' Q_k in the input graph. Let $t[i]$ denote the ith element in a tuple t and is the vertex ID that attribute a_i is bound to.
  - i) $k = 1$ *iteration*: To evaluate Q_1, Generic Join produces 1-matches, with each $t[0]$ matching $\sigma[1]$ to a different vertex ID in the input graph.
  - ii) $k > 1$ *iteration*: To evaluate Q_k, Generic Join performs the following computation for each (k-1)-match t of $Q_{k\text{-}1}$. First, the algorithm takes the forward adjacency list of $t[i]$ for each $\sigma[i] \rightarrow a_k \in E_Q$ and the backward adjacency list of $t[i]$ for each $\sigma[i] \leftarrow a_k \in E_Q$, where $i \leq k$-1 and intersects these lists. The result of the intersection is the set $S = \{s_1, ..., s_\ell\}$ of possible vertex IDs for a_k. Then, for each $s_j \in S$, one k-match $(t[1], ..., t[k\text{-}1], s_j)$ is produced by appending s_j to t. If $S = \{\}$, no output tuples are produced.

Consider again the diamond-X query in Figure 3.1b with a JAO $\sigma = (a_1, a_2, a_3, a_4)$. The 4th iteration takes as input 3-matches of $Q_{DX,3}$ and produces 4-matches for $Q_{DX,4}$ *i.e.*, Q_{DX}. Let t $= (v_1, v_4, v_5)$ be a 3-match, where v_1, v_4, and v_5 match a_1, a_2, and a_3, respectively. In order to compute S, the set of vertex IDs for a_4 given t, Generic Join intersects the forward adjacency



(a) WCOJ tree. (b) Logical plan. (c) WCOJ Pipeline.

Figure 3.3: Generic Join with JAO $a_1a_2a_3a_4$ for diamond query in Figure 2.1.

lists of $(a_2{:}v_4)$ and $(a_3{:}v_5)$. Note that Generic Join uses the forward adjacency lists because a_2 and a_3 are already bound and both have forward edges to a_4 in the query *i.e.*, $a_2{\rightarrow}a_4$ and $a_3{\rightarrow}a_4$. Assume $S = \{v_3, v_{11}\}$ then the set of output 4-matches is $\{(v_1, v_4, v_5, v_3), (v_1, v_4, v_5, v_{11})\}$.

3.3 Optimizing WCOJ Plans

This section demonstrates our WCOJ plans, the effects of different join attribute orderings we have identified, and our i-cost metric for WCOJ plans. Throughout this section we present several experiments for demonstration purposes. The datasets we use in these experiments are described in Table 3.7 in Section 3.7.

3.3.1 WCOJ Plans and `INLJ`/`IMJ` Operators

Each join attribute ordering σ is effectively a different WCOJ plan. Figure 3.3a shows a join tree equivalent to the evaluation of Q_{DX} with a JAO (a_1, a_2, a_3, a_4). The top multi-way join operation

$\bowtie_{a_4}$ in the join tree represents the 4th iteration example above. GraphflowDB implements WCOJ trees as an operator pipeline. We denote the pipeline with a logical plan shown in Figure 3.3b. Data flows in a bottom-up fashion. Each node in the logical plan is an operator labelled by the subquery its output matches. For now, operators pass a single tuple (k-match) using a push-based Volcano-like execution model [68]. In later chapters, we move towards a vectorized execution model. The leaf node covers the 1st iteration and each subsequent parent operation covers the kth iteration for $k > 1$. Figure 3.3c shows the physical plan that the logical plan in Figure 3.3b compiles to. Next, we give details of this logical-physical plan mapping and our core physical operators:

`Scan`: Leaf nodes of logical plans, which match a single query edge, are evaluated with a Scan operator.

`INLJ/IMJ`: Internal nodes of logical plans labelled $Q_k(V_k, E_k)$ that have a child labelled $Q_{k\text{-}1}(V_{k\text{-}1}, E_{k\text{-}1})$ are evaluated with `INLJ/IMJ` operators. The `INLJ/IMJ` operators takes as input (k–1)-matches and extends each tuple t to a set of k-matches. The operators are configured with one or more *adjacency list descriptors* (descriptors for short), which indicate the adjacency lists of appropriate edge relations that the operator needs to use when performing intersections. Each descriptor is an (`i`, `dir`, r_e) triple, where `i` is the position of a vertex in t, `dir` is `forward` or `backward`, and r_e is the edge relation. For each (k–1)-match t, `IMJ` first computes the extension set $S = \{s_1, ..., s_\ell\}$ of t by intersecting the adjacency lists described by its descriptors. `IMJ` intersects the adjacency lists using iterative 2-way in tandem intersections and then produces one k-match, $(t[1], ..., t[k\text{–}1], s_i)$, for each $s_i \in S$. When there is a single descriptor, `INLJ` is used and S is the vertices in the adjacency list described by the descriptor.

Multi-way Intersection Optimization: When extending a single (k–1)-match t, we append each $s_i \in S$ to t, where $|S| = \ell$, leading to ℓ many k-matches. Since all ℓ k-matches were obtained from t, they are identical on the first k–1 values matched. Therefore later `IMJ` operators in the WCOJ pipeline might use the adjacency lists of these k–1 vertices to perform repeated intersections. In such cases, our `IMJ` operator caches and reuses all or a subset of the intersections they make for the last tuple they extend. We next explain this optimization through two examples. Consider the diamond-X query Q_{DX} (recall Figure 3.1b) with a JAO $\sigma = (a_2, a_3, a_1, a_4)$. Let o_3 and o_4 be the `IMJ` operators extending the 2-matches to 3-matches and 3-matches to 4-matches, respectively. Let $t = (v_1, v_2)$ be a 2-match, where v_1 and v_2 match a_2 and a_3, respectively. o_3, when taking t as input, computes an extension set $S = \{s_1, ..., s_\ell\}$ and passes each output 3-match (v_1, v_2, s_i) to o_4 consecutively. Therefore, o_4 would intersect the forward adjacency lists of v_1 and v_2 ℓ consecutive times. Instead, o_4 can compute this intersection for (v_1, v_2, s_1) once and reuse it for the following $\ell{-}1$ (v_1, v_2, s_i) tuples it receives. Similarly, consider a 4-clique query, which is the same as Q_{DX} with an added edge $a_1 \rightarrow a_4$. o_4, given the same input, would now intersect the forward adjacency lists of v_1, v_2, and s_i. In this example, the intersections that

	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8
Cache On	2.4	2.9	3.2	3.3	3.3	3.4	4.4	6.5
Cache Off	3.8	3.2	3.2	3.3	3.3	3.4	8.5	10.7

Table 3.2: WCOJ plan run times in seconds with and without intersection cache for diamond-X on the Amazon ($|V|$=$403K$,E=$3.38M$) dataset from Stanford's network analysis project [112].

o_4 needs to perform to extend each of the ℓ tuples is different. However, if we order our 2-way in tandem intersections to start with the forward adjacency lists of v_1 and v_2, they would all perform this *partial intersection*, which we can cache and reuse in each of the ℓ extensions, *i.e.*, in each extension, we intersect this partial intersection's result with the forward adjacency list of s_i.

Caching and reusing the last full or partial intersection overall improves the performance of WCOJ plans as it reduces the amount of repetitive work in the `IMJ` operators. This optimization also has a very small memory footprint, since we only store at most one full or one partial intersection at each `IMJ` operator at any point in time during query execution. As a demonstrative example, Table 3.2 shows the run time of all WCOJ plans for the diamond-X query with caching enabled and disabled on the Amazon dataset from the *Stanford Network Analysis Project* (SNAP) [112]. The orderings in the table are omitted. Four of the total eight plans use the intersection cache and improve their run time *e.g.*, σ_7 improves by 1.9x.

3.3.2 Effects of Join Attribute Orderings

The work done by a WCOJ plan is commensurate with the "amount of intersections" it performs. Three main factors affect intersection work and therefore the run time of a WCOJ plan P:

i) directions of the adjacency lists P intersects;
ii) the size of intermediate results P generates; and
iii) how much P uses the intersection cache.

We discuss each effect next.

Directions of Intersected Adjacency Lists:

Perhaps surprisingly, there are WCOJ plans that have very different run times *only because* they compute their intersections using different directions of the adjacency lists. The simplest example of this is the triangle query $E(a_1, a_2), E(a_2, a_3), E(a_1, a_3)$ *i.e.*, the graph pattern $a_1 \rightarrow a_2$,

(a) Diamond-X with symmetric triangle. (b) Tailed triangle.

Figure 3.4: Queries used to demonstrate the effects of JAOs.

	BerkStan			Live Journal		
JAO	run time (sec)	S_i	i-cost	run time (sec)	S_i	i-cost
(a_1, a_2, a_3)	2.6	8M	490M	64.4	69M	13.1B
(a_2, a_3, a_1)	15.2	8M	55,8B	75.2	69M	15.9B
(a_1, a_3, a_2)	31.6	8M	55,9B	79.1	69M	17.3B

Table 3.3: Run time (secs), size of intermediate results (S_i), and i-cost of different JAOs for the asymmetric triangle query.

$a_2 \rightarrow a_3$, $a_1 \rightarrow a_3$. This query has 3 JAOs, all of which have the same SCAN operator, which scans each $u \rightarrow v$ edge in the input graph as a 2-match, followed by a 2-way intersection. The direction of the intersections for each JAO are different and none can use the intersection cache:

- σ_1 (a_1, a_2, a_3): given $(a_1{:}u, a_2{:}v)$, intersects both u and v's forward lists.
- σ_2 (a_2, a_3, a_1): given $(a_2{:}u, a_3{:}v)$, intersects both u and v's backward lists.
- σ_3 (a_1, a_3, a_2): given $(a_1{:}u, a_3{:}v)$, intersects u's forward and v's backward lists.

Table 3.3 shows a demonstrative experiment studying the performance of each plan on the BerkStan and LiveJournal datasets also from SNAP (the i-cost column in the table will be discussed in Section 3.3.3 momentarily). For example, σ_1 is 12.1x faster than σ_2 on the BerkStan graph. Which combination of adjacency list directions is more efficient depends on the structural properties of the input graph, *e.g.*, forward and backward adjacency list distributions.

Size of Intermediate Results:

Different WCOJ plans generate different partial matches with different sizes and intersection work. Consider the *tailed triangle* query in Figure 3.4b, which can be evaluated by two categories of WCOJ plans:

- EDGE-WEDGE: Some plans, such as JAO (a_1, a_2, a_4, a_3), extend each scanned edge $u \rightarrow v$ *i.e.*, $(a_1{:}u, a_2{:}v)$ match to wedges $(u \rightarrow v \leftarrow w)$ *i.e.*, $(a_1{:}u, a_2{:}v, a_3{:}w)$ matches, and then closes the cycle of triangle from one of the 2 edges in the wedge.

	Amazon			Epinions		
JAO	run time (secs)	S_i	i-cost	run time (secs)	S_i	i-cost
(a_1, a_2, a_3, a_4)	0.9	15M	176M	0.9	4M	0.9B
(a_1, a_3, a_2, a_4)	1.4	15M	267M	1.0	4M	0.9B
(a_2, a_3, a_1, a_4)	2.4	15M	267M	1.7	4M	1.0B
(a_1, a_4, a_2, a_3)	4.3	35M	640M	56.5	55M	32.5B
(a_1, a_4, a_3, a_2)	4.6	35M	1.4B	72.0	55M	36.5B

Table 3.4: Run time (secs), size of intermediate results (S_i), and i-cost of different JAOs for the tailed triangle query.

	Amazon			Epinions		
JAO	run time (sec)	S_i	i-cost	run time (sec)	S_i	i-cost
(a_2, a_3, a_1, a_4)	1.0	11M	0.1B	0.9	2M	0.1B
(a_1, a_2, a_3, a_4)	3.0	11M	0.3B	4.0	2M	1.0B

Table 3.5: Run time (secs), size of intermediate results (S_i), and i-cost of some JAOs for the symmetric diamond-X query.

- EDGE-TRIANGLE: Another group of plans, such as JAO (a_1, a_2, a_3, a_4), extend scanned edges to triangles and then extend the triangles by one edge.

Let $|E|$, $|W|$, and $|\triangle|$ denote the number of edges, wedges, and triangles. Ignoring the directions of extensions and intersections, the EDGE-WEDGE plans do $|E|$ many extensions plus $|W|$ many intersections, whereas the EDGE-TRIANGLE plans do $|E|$ many intersections and $|\triangle|$ many extensions. Table 3.4 shows the run times of the different plans on Amazon and Epinions datasets with intersection caching disabled (again the i-cost column will be discussed momentarily). The first 3 rows are the EDGE-TRIANGLE plans. EDGE-TRIANGLE plans are significantly faster than EDGE-WEDGE plans because when evaluated on a single edge relation, $|W|$ is always at least $|\triangle|$ and often much larger. Which JAOs will generate fewer intermediate matches depends on several factors: i) the structure of the query; ii) the selectivity of joining different edge relations; and iii) the structural properties of the input graph, *e.g.*, graphs with low clustering coefficient generate fewer intermediate triangles than those with a high clustering coefficient.

Intersection Cache Hits

The intersection cache of our IMJ operator is used more if the JAO extends (k–1)-matches to a_k using adjacency lists with indices from $a_1...a_{k-2}$. Intersections that access the (k–1)th index cannot be reused because a_{k-1} is the result of an extension by the last INLJ/IMJ operators and will match to different vertex IDs. Instead, those accessing indices $a_1...a_{k-2}$ can potentially be reused. We demonstrate that some plans perform significantly better than others only because they can use the intersection cache. Consider a variant of the diamond-X query in Figure 3.4a. One type of WCOJ plans for this query extend $u{\rightarrow}v$ edges to (u, v, w) symmetric triangles by intersecting u's backward and v's forward adjacency lists. Then each triangle is extended to complete the query, intersecting again the forward and backward adjacency lists of one of the edges of the triangle. There are two sub-groups of JAOs that fall under this type of plans: (i) (a_2, a_3, a_1, a_4) and (a_2, a_3, a_4, a_1), which are equivalent plans due to symmetries in the query, so will perform exactly the same operations; and (ii) (a_1, a_2, a_3, a_4), (a_3, a_1, a_2, a_4), (a_3, a_4, a_2, a_1), and (a_4, a_2, a_3, a_1), which are also equivalent plans. Importantly, all of these plans cumulatively perform exactly the same intersections but those in group (i) and (ii) have different orders in which these intersections are performed, which lead to different intersection cache use.

Table 3.5 shows the performance of one representative plan from each sub-group: (a_2, a_3, a_1, a_4) and (a_1, a_2, a_3, a_4), on the Amazon and Epinions datasets. The (a_2, a_3, a_1, a_4) plan is 4.4x faster on Epinions and 3x faster on Amazon. This is because when (a_2, a_3, a_1, a_4) extends (a_2, a_3, a_1) triangles to complete the query, it will be accessing a_2 and a_3, so the first two indices in the triangles. For example if (a_2:v_0, a_3:v_1) extended to t triangles (v_0, v_1, v_2),...,(v_0, v_1, v_{t+2}), these partial matches will be fed into the next IMJ operator consecutively, and their extensions to a_4 will all require intersecting v_0 and v_1's backward adjacency lists, so the cache would avoid t–1 intersections. Instead, the cache will not be used in the (a_1, a_2, a_3, a_4) plan as each input 3-match in the last intersection has a different ID for a_3.

3.3.3 Cost Metric for WCOJ Plans

We introduce a new cost metric called *intersection cost* (i-cost), which we define as the size of adjacency lists that will be accessed and intersected within a WCOJ plan. Consider a WCOJ plan σ that evaluates subqueries Q_2,...,Q_m, respectively, where Q=Q_m. Let t be a (k–1)-match of Q_{k-1} and suppose t is extended to instances of Q_k by intersecting a set of adjacency lists, described with adjacency list descriptors A_{k-1}. Formally, i-cost of σ is:

$$\sum_{Q_{k-1} \in Q_2 ... Q_{m-1}} \sum_{t \in Q_{k-1}} \sum_{\substack{(i,dir) \in A_{k-1} \\ \text{s.t. (i, dir) is accessed}}} |t[i].dir| \quad (3.1)$$

We discuss how we estimate i-costs of plans in Section 3.5. For now, note that Equation 3.1 captures the three effects of JAOs we identified: (i) the $|t.dir|$ quantity captures the sizes of the adjacency lists in different directions; (ii) the second summation is over all intermediate matches, capturing the size of intermediate results; and (iii) the last summation is over all adjacency lists that are accessed, so ignores the lists in the intersections that are cached. For the demonstrative experiments we presented in the previous section, we also report the *actual* i-costs of different plans in Tables 3.3, 3.4, and 3.5. The actual i-costs are measured in a profiled run of each query. Notice that in each experiment, i-costs of plans rank in the correct order of run times of plans.

There are alternative cost metrics from the literature, such as the C_{out} [44] and C_{mm} [107] metrics, that would also do reasonably well in differentiating good and bad WCOJ plans. However, these metrics capture only the effect of the number of intermediate matches. For example, they would not differentiate the plans in the asymmetric triangle query or the symmetric diamond-X query, *i.e.*, the plans in Tables 3.3 and 3.5 have the same actual C_{out} and C_{mm} costs.

3.4 Full Plan Space & Dynamic Programming Optimizer

In this section we describe our full plan space, which contain plans with `HJ` and `INLJ/IMJ`, the costs of these plans, and our dynamic programming optimizer.

3.4.1 Hybrid Plans and `Hash Join` (`HJ`) Operator

In Section 3.3, we introduced our WCOJ plans as operator pipelines with the `Scan` operator as a leaf with subsequent `IMJ` operators. Our full plan space adds to our WCOJ plans, hybrid plans *i.e.*, bushy plans by introducing the joining of intermediate relations with `HJ` which can be seamlessly mixed with WCOJ subplans.

We describe our full plan space in terms of logical plans. A plan is a rooted tree as follows. Below, Q_k refers to a projection of Q onto an arbitrary set of k query vertices.

- Leaf nodes are labelled with a single query edge of Q.
- Root is labelled with Q.
- Each internal node o_k is labelled with Q_k=$\{V_k, E_k\}$, with the *projection constraint* that Q_k is a projection of Q onto a subset of query vertices. o_k has either one child or two children. If o_k has one child o_{k-1} with label Q_{k-1}=$\{V_{k-1}, E_{k-1}\}$, then Q_{k-1} is a subgraph of Q_k missing one query vertex $q_v \in V_k$ and q_v's incident edges in E_k. This represents a WCOJ extension of partial matches of Q_{k-1} by one query vertex to Q_k. If o_k has two children o_{c1} and o_{c2} with

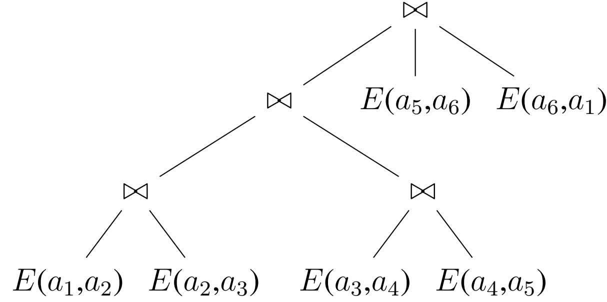


Figure 3.5: Example join tree not in EmptyHeaded's GHD plan space for the 6-cycle query: $E(a_1, a_2), E(a_2, a_3), E(a_3, a_4), E(a_4, a_5), E(a_5, a_1)$.

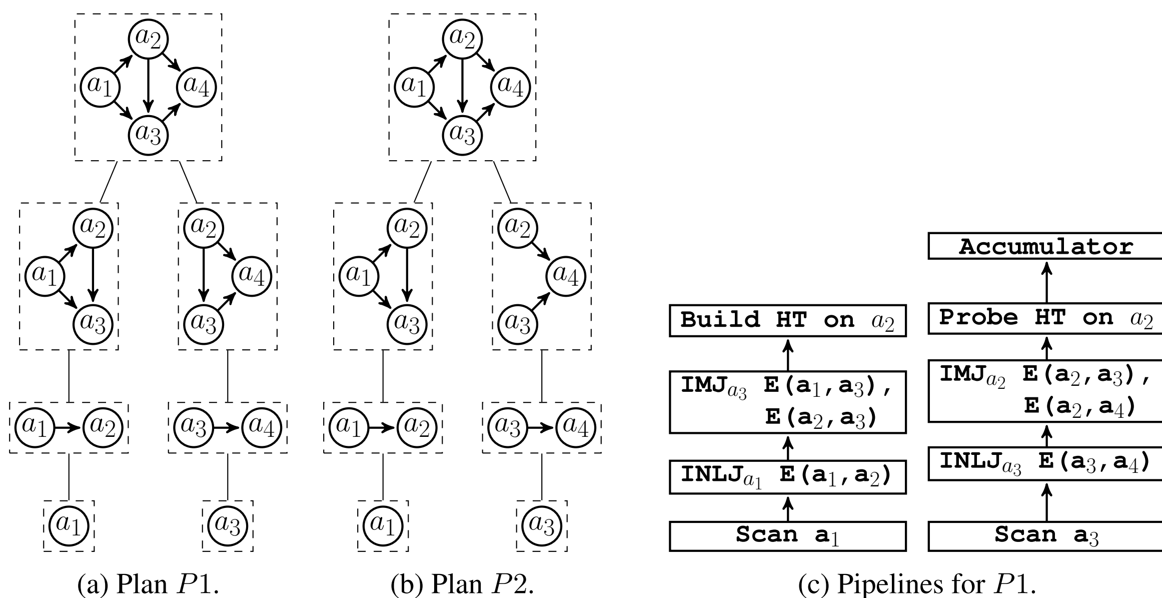
labels Q_{c1} and Q_{c2}, respectively, then $Q_k = Q_{c1} \cup Q_{c2}$ and $Q_k \neq Q_{c1}$ and $Q_k \neq Q_{c2}$. This represents a binary join of matches Q_{c1} and Q_{c2} to compute Q_k.

As before, leaves map to SCAN operators, an internal node o_k with a single child maps to the IMJ operators. If o_k has two children, then it maps to the HJ operator as described in Chapter 2.

Our plans are highly expressive and contain several classes of plans: (1) WCOJ plans from the previous section, in which each internal node has one child; (2) binary join plans, in which each node has two children and satisfy the projection constraint; and (3) hybrid plans that satisfy the projection constraint. We show in our evaluation section that our hybrid plans contain EmptyHeaded's minimum-width GHD-based hybrid plans that satisfy the projection constraint. For example the hybrid plan in Figure 3.2d corresponds to a GHD for the diamond-X query with width 3/2. In addition, our plan space also contains hybrid plans that do not correspond to a GHD-based plan. Figure 3.5 shows an example hybrid plan for the 6-cycle query that is not in EmptyHeaded's plan space. As we show in our evaluations, such plans can be very efficient for some queries.

The projection constraint prunes two classes of plans:

1. Our plan space does not contain binary join plans that first compute open wedges and then close them. Consider a triangle Q_T which is a subquery of a larger query Q. Suppose Q_T is $a_1 \rightarrow a_2 \rightarrow a_3$, $a_1 \rightarrow a_3$. Then due to the projection constraint, we do not enumerate any plan that contains an open wedge Q_{OT} *e.g.*, $a_1 \rightarrow a_2 \rightarrow a_3$, of Q_T, with say a later binary join to close the cycle and add the $a_1 \rightarrow a_3$ edge. This is because Q_{OT} is not a projection of Q, as it does not contain the $a_1 \rightarrow a_3$ edge. Such binary join plans are in the plan spaces of existing optimizers, *e.g.*, Postgres, MySQL, and Neo4j. This is not a disadvantage because for each such plan, there is a more efficient WCOJ plan that computes triangles directly with



(a) Plan $P1$. (b) Plan $P2$. (c) Pipelines for $P1$.

Figure 3.6: Two plans where $P1$ joins two subqueries that share a query edge and $P2$ joins two subqueries that do not. Shows also operator pipelines of $P1$.

an intersection of two already-sorted adjacency lists. Specifically, we force the triangles to be computed by extending edges (which are projections of Q) directly to Q_T using WCOJ multiway join intersections.

2. More generally, some of our hybrid plans contain the same query edge $a_i{\rightarrow}a_j$ in multiple parts of the join tree, which may look redundant because $a_i{\rightarrow}a_j$ is effectively joined multiple times. There can be alternative plans that remove $a_i{\rightarrow}a_j$ from all but one of the sub-trees. For example, consider the two hybrid plans P_1 and P_2 for the diamond-X query in Figures 3.6a and 3.6b, respectively. P_2 is not in our plan space because it does not satisfy the projection constraint because $a_2{\rightarrow}a_3$ is not in the right sub-tree. Omitting such plans is also not a disadvantage because we duplicate $a_i{\rightarrow}a_j$ only if it closes cycles in a sub-tree, which effectively is an additional filter that reduces the partial matches. For example, on the Amazon graph dataset, P_1 takes 14.2 seconds and P_2 takes 56.4 seconds, which is 3.97x slower than P_1.

Finally, Figure 3.6c also shows the `HJ` physical plan pipelines that are generated for the plan P_1 in Figure 3.6a. These pipelines execute in order so the left one executes fully and once done, the second pipeline on the right executes.

3.4.2 Cost Metric for General Plans

A HJ operator performs a very different computation than IMJ operators, so the cost of HJ needs to be normalized with i-cost. This is an approach taken by DBMSs to merge costs of multiple operators, *e.g.*, a scan and a group-by, into a single cost metric. Consider a HJ operator o_k that will join matches of Q_{c1} and Q_{c2} to compute Q_k. Suppose there are n_1 and n_2 instances of Q_{c1} and Q_{c2}, respectively. Then o_k will hash n_1 number of tuples into a table and probe this table n_2 times. We compute two weight constants w_1 and w_2 and calculate the cost of o_k as $w_1 \times n_1$ + $w_2 \times n_2$ i-cost units. These weights can be hard-coded as done in the C_{mm} cost metric [107], but we pick them empirically.

3.4.3 Dynamic Programming Optimizer

Algorithm 1 shows the pseudocode of our optimizer, which takes as input a query $Q(V_Q, E_Q)$. We start by enumerating and computing the cost of all WCOJ plans (line 1). We then initialize the cost of computing 2-vertex subqueries of Q *i.e.*, a query edge, to the number of edges/records matching in the query edge/relation (line 2). If $a_i \xrightarrow{r_e} a_j$ is a query edge, then let the number of edges with label r_e be $\mu(r_e)$, which is the initial cost we give to the subquery of $a_i \xrightarrow{r_e} a_j$. Then starting from $k = 3$ up to $|V_Q|$, for each k-vertex subquery Q_k of Q, we find the lowest cost plan $P^*_{Q_k}$ to compute Q_k in three different ways:

i) $P^*_{Q_k}$ is the lowest-cost WCOJ plan that we enumerated. (line 5).
ii) $P^*_{Q_k}$ extends the best plan $P^*_{Q_{k\text{-}1}}$ for a $Q_{k\text{-}1}$ by an INLJ/IMJ operator. ($Q_{k\text{-}1}$ contains one fewer query vertex than Q_k). (lines 7-10).
iii) $P^*_{Q_k}$ joins the output of two best plans $P^*_{Q_{c1}}$ and $P^*_{Q_{c2}}$ for Q_{c1} and Q_{c2}, respectively, with a HJ . (lines 12-15).

The best plan for each Q_k is stored in a *subquery map*. We enumerate all WCOJ plans because the best WCOJ plan $P^*_{Q_k}$ for Q_k is not necessarily an extension of the best WCOJ plan $P^*_{Q_{k\text{-}1}}$ for a $Q_{k\text{-}1}$ by one query vertex. That is because $P^*_{Q_k}$ may be extending a worse plan $P^{bad}_{Q_{k\text{-}1}}$ for $Q_{k\text{-}1}$ if the last extension has a good intersection cache utilization. Strictly speaking, this problem can arise when enumerating hybrid plans too, if an INLJ/IMJ operator in case ii) above follows a HJ. A full plan space enumeration would avoid this problem completely but we adopt dynamic programming to make our optimization time efficient, *i.e.*, to make our optimizer efficient, we are potentially sacrificing picking the plan with the lowest estimated-cost.

Finally, our optimizer omits plans that contain a Hash Join that can be converted to an INLJ/IMJ operator. Consider the $a_1 \rightarrow a_2 \rightarrow a_3$ query. Instead of using a HASH-JOIN to mate-

rialize the $a_2 \to a_3$ edges and then probe a scan of $a_1 \to a_2$ edges, it is more efficient to use an `INLJ`/`IMJ` to extend $a_1 \to a_2$ edges to a_3 using a_2's existing forward adjacency list.

Algorithm 1 DP Optimization Algorithm

Input: $Q(V_Q, E_Q)$

1: WCOJP = enumerateAllWCOJPlans(Q) *// WCOJ plans*
2: QPMap: *init each* $a_i \xrightarrow{r_e} a_j$*'s cost to the* $\mu(r_e)$
3: **for** k = 3, ..., $|V_Q|$ **do**
4:     **for** $V_k \subseteq V$ s.t. $|V_k|$=k **do**
5:         $Q_k(V_k, E_k) =$ Projection of Q on V_k; bestP $=$ WCOJP(Q_k); minC $= \infty$
6:         *// Find best plan that extends to* Q_k *by one query vertex*
7:         **for** $v_j \in V_k$ let $Q_{k\text{-}1}(V_{k\text{-}1}, E_{k\text{-}1}) =$ Projection of Q_k on V_k–v_j **do**
8:             P = QPMap($Q_{k\text{-}1}$).extend(Q_k);
9:             **if** cost(P) $<$ minC **then**
10:                 bestPlan = P;
11:         *// Find best plan that generates* Q_i *with a binary join*
12:         **for** $V_{c1}, V_{c2} \subset V_k$: $Q_{c1} =$ Projection of Q_k on V_{c1}, $Q_{c2} =$ Projection of Q_k on V_{c2} **do**
13:             P = join(QPMap(Q_{c1}), QPMap(Q_{c2}));
14:             **if** cost(P) $<$ minC **then**
15:                 bestPlan = P;
16:     QPMap(Q_k) = bestPlan;
17: **return** QPMap(Q);

Plan Generation For Very Large Queries:

Our optimizer can take a very long time to generate a plan for large queries. For example, enumerating only the best WCOJ plan for a 20-clique requires inspecting $20!$ different JAOs, which would be prohibitive. To overcome this, we further prune plans for queries with more than 10 query vertices as follows:

- We avoid enumerating all WCOJ plans. Instead, WCOJ plans get enumerated in the DP part of the optimizer. Therefore, we potentially ignore good WCOJ plans that benefit from the intersection cache.
- At each iteration k, out of the t_k many plans that evaluate a k-vertex subquery of Q we only keep the r lowest cost plans (5 by default). At iteration $k+1$, we will extend these r plans to t_{k+1} many plans that evaluate $(k+1)$-vertex subqueries but we will again keep on the top r, so on and so forth.

$(Q_{k-1}$	A	$\|A\|$	$\mu(Q_k)$
$(1 \xrightarrow{r_x} 2;$	$L_1{:}2\xrightarrow{r_x})$	$\|L_1\|$:4.5	4.5
$(1 \xrightarrow{r_x} 2;$	$L_1{:}2\xrightarrow{r_y})$	$\|L_1\|$:8.0	8.0
$(1 \xrightarrow{r_x} 2;$	$L_1{:}1\xrightarrow{r_x}$, $L_2{:}2\xrightarrow{r_x})$	$\|L_1\|$:4.2, $\|L_2\|$:5.1	1.5
$(1^{l_a} \xrightarrow{r_x} 2;$	$L_1{:}1\xleftarrow{r_x}$, $L_2{:}2\xleftarrow{r_x})$	$\|L_1\|$:9.8, $\|L_2\|$:8.4	2.5
(...;	...)	...	...

Table 3.6: A subquery catalogue. A is a set of adjacency list descriptors; μ is selectivity.

3.5 Cost & Cardinality Estimation

To assign costs to the plans we enumerate, we need to estimate: (1) the size of intermediate results that different plans generate; (2) the i-costs of extending a subquery Q_{k-1} to Q_k by extending or intersecting from a set of adjacency lists in INLJ/IMJ operators; and (3) the costs of HJ operators. We focus on the setting where each subquery Q_k is joining multiple edge relations. In the remainder of this section, we describe how we make these estimations using a data structure called the *subquery catalogue*. However, we emphasize that our optimizer can be used with any estimation technique that can estimate i-cost and size of partial matches of subqueries.

Table 3.6 shows an example catalogue. Each entry contains a key (Q_{k-1}, A, $a_k^{l_k}$), where A is a set of adjacency list descriptors. Let Q_k be the subgraph that extends Q_{k-1} with query edges in A to a query vertex a_k. Each entry contains two estimates for extending a match of a subquery Q_{k-1} to Q_k by intersecting the adjacency lists A describes:

i) |A|: Average sizes of the lists in A that are intersected.

ii) $\mu(Q_k)$: Average number of Q_k that will extend from one Q_{k-1}, *i.e.*, the average number of vertices that: are in the extension set of intersecting the adjacency lists A.

In Table 3.6, the query vertices of the input subgraph Q_{k-1} are shown with canonicalized integers, *e.g.*, 0, 1 or 2, instead of the non-canonicalized a_i notation we used before. This is to note that we account for unique subgraphs, *e.g.*, only one of $0 \rightarrow 1 \rightarrow 2$ and $0 \leftarrow 1 \leftarrow 2$ would be found in the Q_{k-1} column. Note that Q_{k-1} can be extended to Q_k using different adjacency lists in A with different i-costs. The third and fourth entries of Table 3.6, which extend a single edge to an asymmetric triangle, demonstrate this. When canonicalizing a query graph, for a lookup, we rely on isomorphism and finding a mapping from the query vertices of an input graph to the query vertices of Q_{k-1}.

3.5.1 Catalogue Construction

For each input dataset G, we construct a catalogue containing all entries that extend an at most h-vertex subgraph to an (h+1)-vertex subgraph. By default we set h to 3. When generating a catalogue entry for extending $Q_{k\text{–}1}$ to Q_k, we do not find all instances of $Q_{k\text{–}1}$ and extend them to Q_k. Instead, we first sample $Q_{k\text{–}1}$. We take a WCOJ plan that extends $Q_{k\text{–}1}$ to Q_k. We then sample z random edges (1000 by default) uniformly at random from G in the SCAN operator. The last INLJ/IMJ operator of the plan extends each partial match t it receives to Q_k by intersecting the adjacency lists in A. The operator measures the size of the adjacency lists in A and the number of Q_k's this computation produced. These measurements are averaged and stored in the catalogue as |A| and $\mu(Q_k)$ columns.

3.5.2 Cost Estimations

We use the catalogue to do three estimations as follows:

1. Results size of Q_k: To estimate the results size of Q_k, we pick a WCOJ plan P that computes Q_k through a sequence of ($Q_{j\text{–}1}$, A_j) extensions. The estimated cardinality of Q_k is the product of the $\mu(A_j)$ of the ($Q_{j\text{–}1}$, A_j) entries in the catalogue. If the catalogue contains entries with up to h-vertex subgraphs and Q_k contains more than h nodes, some of the entries we need for estimating the cardinality of Q_k will be missing. Suppose for calculating the cardinality of Q_k, we need the $\mu(A_x)$ of an entry ($Q_{x\text{–}1}$, A_x, l_x) that is missing because $Q_{x\text{–}1}$ contains x–1 $>$ h query vertices. Let z=(x–h–1). In this case, we remove each z-size set of query vertices $a_1, ...a_z$ from $Q_{x\text{–}1}$ and Q_x, and the adjacency list descriptors from A_x that include $1, ..., z$ in their indices. Let ($Q_{y\text{–}1}$, A_y, l_y) be the entry we get after a removal. We look at the $\mu(A_y)$ of ($Q_{y\text{–}1}$, A_y, l_y) in the catalogue. Out of all such z set removals, we use the minimum $\mu(A_y)$ we find.

As an example, consider a missing entry for extending $Q_{k\text{–}1}$= 1→2→3 by one query vertex to 4 by intersecting three adjacency lists all pointing to 4 from 1, 2, and 3. For simplicity, assume the extensions are from the same edge relation. The resulting subquery Q_k will have two triangles: (i) an asymmetric triangle touching edge 1→2; and (ii) a symmetric triangle touching 2→3. Suppose entries in the catalogue indicate that an edge on average extends to 10 asymmetric triangles but to 0 symmetric triangles. We estimate that $Q_{k\text{–}1}$ will extend to zero Q_k taking the minimum of our two estimates.

2. I-cost of INLJ/IMJ operators: Consider an INLJ/IMJ operator o_k extending $Q_{k\text{–}1}$ to Q_k using adjacency lists A. We have two cases:

- No intersection cache: We estimate o_k's i-cost as

$$\text{i-cost}(o_k) = \mu(Q_{k\text{-}1}) \times \sum_{L_i \in A} |L_i| \qquad (3.2)$$

  Here, $\mu(Q_{k\text{-}1})$ is the estimated cardinality of $Q_{k\text{-}1}$, and $|L_i|$ is the average size of the adjacency list $L_i \in A$ that are logged in the catalogue for entry ($Q_{k\text{-}1}$, A), *i.e.*, the $|A|$ column.
- Intersection cache utilization: If two or more of the adjacency list in A, say L_i and L_j, access the vertices in a partial match Q_j that is smaller than $Q_{k\text{-}1}$, then we multiply the estimated sizes of L_i and L_j with the estimated cardinality of Q_j instead of $Q_{k\text{-}1}$. This is because we infer that o_k will utilize the intersection cache for intersecting L_i and L_j.

Reasoning about utilization of intersection cache is critical in picking good plans. For example, recall our experiment from Table 3.2 to demonstrate that the intersection cache broadly improves all plans for the diamond-X query. Our optimizer, which is "cache-conscious" picks σ_2 ($a_2a_3a_4a_1$). Instead, if we ignore the cache and make our optimizer "cache-oblivious" by always estimating i-cost with Equation 3.2, it picks the slower σ_4 ($a_1a_2a_3a_4$) plan. Similarly, our cache-conscious optimizer picks (a_2, a_3, a_1, a_4) in our experiment from Table 3.5. Instead, the cache-oblivious optimizer assigns the same estimated i-cost to plans (a_2, a_3, a_1, a_4) and (a_1, a_2, a_3, a_4), so cannot differentiate between these two plans and picks one arbitrarily.

3. **Cost of `Hash Join` operator:** Consider a `HJ` operator joining Q_{c1} and Q_{c2}. The estimated cost of this operator is simply w_1n_1 + w_2n_2 (recall Section 3.4.2), where n_1 and n_2 are now the estimated result sizes of Q_{c1} and Q_{c2}, respectively.

3.5.3 Limitations

Similar to Markov tables [13] and MD- and Pattern-tree summaries [115], our catalogue is an estimation technique that is based on storing information about small size subgraphs and extending them to make estimates about larger subgraphs. We review these techniques in detail and discuss our differences in Chapter 6. Here, we discuss several limitations that are inherent in such techniques. We emphasize again that our optimizer can be used with more advanced cardinality estimation techniques.

First, as expected our estimates (both for i-cost and results size) get worse as the size of the subgraphs for which we make estimates increase beyond h. Equivalently, as h increases, our estimates for fixed-size large queries get better. At the same time, the size of the catalogue increases significantly as h increases. Similarly, the size of the catalogue increases as graphs get more heterogeneous, *i.e.*, contain more edge relations. Second, using larger sample sizes, *i.e.*, larger z

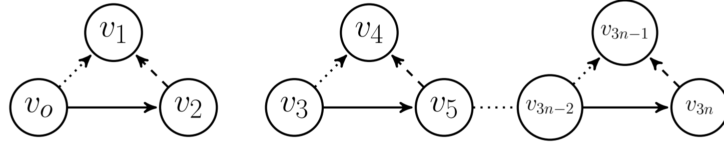


Figure 3.7: Input graph for adaptive JAO example.

values, increase the accuracy of our estimates but require more time to construct the catalogue. Therefore h and z respectively trade off catalogue size and creation time with the accuracy of estimates. We provide demonstrative experiments of these tradeoffs in our evaluation.

3.6 Adaptive WCOJ Plan Evaluation

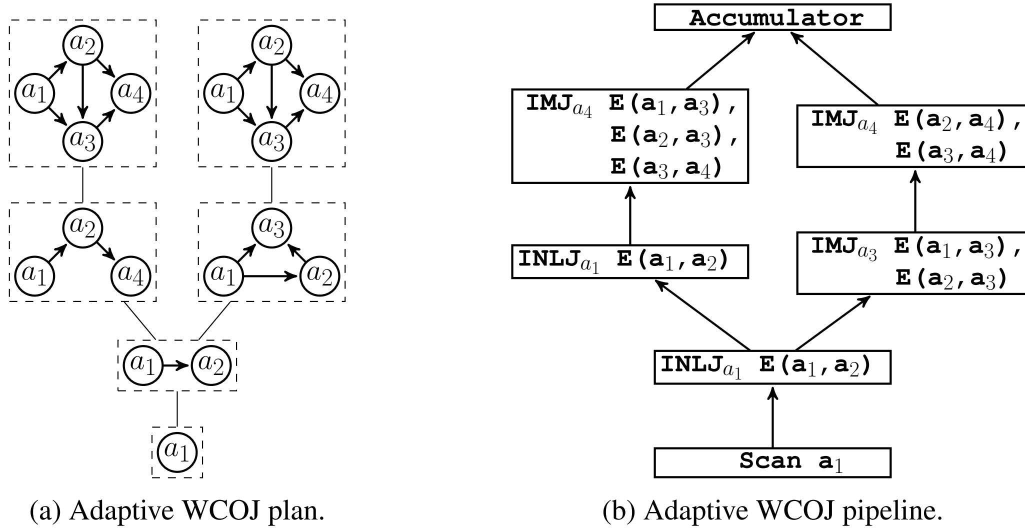
Recall that the $|A|$ and μ statistics stored in a catalogue entry ($Q_{k\text{-}1}$, A), are estimates of the adjacency list sizes (and selectivity) for matches of $Q_{k\text{-}1}$. These are *estimates* based on *averages* over many sampled matches of $Q_{k\text{-}1}$. In practice, *actual* adjacency list sizes and selectivity of *individual* matches of $Q_{k\text{-}1}$ can be very different. Let us refer to parts of plans that are chains of one or more `INLJ/IMJ` operators as ***WCOJ subplans***. Consider a WCOJ subplan of a *fixed* plan P that has a JAO σ^* and extends partial matches of a subquery Q_i to matches of Q_k. Our optimizer picks σ^* based on the estimates of the average statistics in the catalogue. Our adaptive evaluator updates our estimates for individual matches of Q_i (and other subqueries in this part of the plan) based on actual statistics observed during evaluation and possibly changes σ^* to another JAO for each individual match of Q_i.

Example:

Consider the input graph G shown in Figure 3.7. G contains 3n edges. Consider the diamond-X query and the WCOJ plan P with $\sigma = (a_2, a_3, a_4, a_1)$. Readers can verify that this plan will have an i-cost of $3n$: $2n$ from extending solid edges, n from extending dotted edges, and 0 from extending dashed edges. Now consider the following *adaptive* plan that picks σ for the dotted and dashed edges as before but $\sigma' = (a_2, a_3, a_1, a_4)$ for the solid edges. For the solid edges, σ' incurs an i-cost of 0, reducing the i-cost to n.

3.6.1 Adaptive Plans

We optimize queries as follows. First, we get a *fixed* plan P from our dynamic programming optimizer. If P contains a chain of two or more `INLJ/IMJ` operators $o_i, o_{i+1} ..., o_k$, we replace



(a) Adaptive WCOJ plan. (b) Adaptive WCOJ pipeline.

Figure 3.8: Example adaptive WCOJ plan. Shows both the logical and operator pipelines.

it with an *adaptive* WCOJ plan. The adaptive plan extends the first partial matches Q_i that o_i takes as input in all possible connected ways to Q_k. We fix the first two query vertices in a JAO and pick the rest adaptively. Figure 3.8 shows the adaptive version of the fixed plan for the diamond-X query. Figure 3.8b shows the adaptive version of the fixed plan for the diamond-X query. More generally, we adapt each operator pipeline and therefore can adapt hybrid plans.

3.6.2 Adaptive Operators

Unlike the operators in fixed plans, our adaptive operators can feed their outputs to multiple operators. An adaptive operator o_i is configured with a function f that takes a partial match t of Q_i and decides which of the next operators should be given t. f consists of two high-level steps: (1) For each possible σ_j that can extend Q_i to Q_k, f re-evaluates the estimated i-cost of σ_j by re-calculating the cost of plans using *updated cost estimates* (explained momentarily). o_i gives t to the next `INLJ/IMJ` operator of σ_j^* that has the lowest re-calculated cost. The cost of σ_j is re-evaluated by changing the estimated adjacency list sizes that were used in results size and i-cost estimations with actual adjacency list sizes we obtain from t.

Example:

Consider the diamond-X query from Figure 3.1b and suppose we have an adaptive plan in which the SCAN operator matches edges to (a_2, a_3), so for each edge needs to decide whether to pick

the ordering $\sigma_1 = (a_2, a_3, a_4, a_1)$ or $\sigma_2 = (a_2, a_3, a_1, a_4)$. Suppose the catalogue estimates the sizes of $|a_2 \rightarrow|$ and $|a_3 \rightarrow|$ as 100 and 2000, respectively. So we estimate the i-cost of extending an (a_2, a_3) edge to (a_2, a_3, a_4) as 2100. Suppose the selectivity μ_j of the number of triangles this intersection will generate is 10. Suppose we read an edge $u \rightarrow v$ where u's forward adjacency list size is 50 and v's backward adjacency list size is 200. Then we update our i-cost estimate directly to 250 and μ_j to $10 \times (50/100) \times 200/2000 = 0.5$.

As we show in our evaluations, adaptive JAO selection improves the performance of many WCOJ plans but more importantly guards our optimizer from picking bad JAOs.

3.7 Evaluation

In this section, we demonstrate the efficiency of the plans that our query optimizer generates. Before presenting the experiments, we describe the hardware, datasets, and queries we use.

3.7.1 Setup

Hardware: We use a single machine that has two Intel E5-2670 @2.6GHz CPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical cores. All code runs on openjdk-17. We set the maximum size of the JVM heap to 500 GB and keep the minimum heap size of the JVM as is. We use only one physical core except for our scalability experiments in Section 3.7.8.

Domain	Name	Nodes	Edges
Social	Epinions (Ep)	76K	509K
	LiveJournal (LJ)	4.8M	69M
	Twitter (Tw)	41.6M	1.46B
Web	BerkStan (BS)	685K	7.6M
	Google (Go)	876K	5.1M
Product	Amazon (Am)	403K	3.5M
Citation	Patent (Pa)	3.7M	16.5M

Table 3.7: Datasets used.

Datasets: The datasets we use are in Table 3.7. They are obtained from the Stanford Large Network Dataset Collection [112] except for the Twitter graph, prepared by Kwak *et al.* [102]. Our datasets differ in several structural properties: (1) size; (2) skew distribution of forward and

backward adjacency lists; and (3) average clustering coefficients, which is the ratio of number of triangles to the total possible in the ego-graph of a node. The datasets also come from a variety of application domains *e.g.*, social networks, the web, and product co-purchasing. Each dataset catalogue was generated with $z = 1000$ and $h = 3$ except for Twitter, where we set $h = 2$.

Queries: All datasets are made of a single edge relation. In some cases, we split them into multiple edge relations randomly as done in prior work [28, 76]. For a query Q, we use the notation Q_i to refer to evaluating Q when the dataset is randomly split into edge relations $\{R_1, R_2, \ldots, R_i\}$. For example, evaluating Q_2 on Amazon indicates that each edge in Amazon and each edge relation in Q is assigned to one of two edge relations R_1 or R_2. If a query is made of self-joins on a single edge relation, we simply refer to it as Q.

Execution: We ran each experiment twice, a first time to warm-up the system, and a second time to record the measurement.

3.7.2 Evaluation Overview

In these experiments, we aim to answer five questions: (1) How good are the plans our optimizer chooses? (2) Which type of plans work better for which queries? (3) How much benefit do we get from adapting JAOs at run time? (4) How do our plans and execution engine compare against EmptyHeaded (EH), which is the closest to our work and the most performant baseline we are aware of? (5) How do our plans compare against prior work titled "Flexible Caching in Trie Joins", or CTJ for short [91], which is another algorithm that extends the WCOJ algorithm LFTJ with caching [179]?

We tested the scalability of our single-threaded and parallel implementation on our largest graphs LiveJournal and Twitter. We compare our plans on big queries against the subgraph matching algorithm CFL [29] which optimizes for much larger queries. We will give detailed overviews of EH, CTJ, and CFL in later sections when we present experiments that use them. Finally, for the completeness of our study, we compare our cardinality estimator against that of a more traditional RDBMS, that of Postgres.

We used the 14 queries shown in Figure 3.9, which contain both acyclic and cyclic queries with dense and sparse connectivity with up to 7 query vertices and 21 query edges. To give a sense of the scale, we report in Table 3.8 the output query size when evaluated with each dataset as a single edge relation. In some experiments, we use "labelled" versions of these queries. In these versions, we split the datasets among multiple relations that represent different edge labels and each query edge gets one of these labels. In such cases, the queries have smaller output size. The majority of these queries are obtained from real applications and from the literature *e.g.*, $Q1$

and $Q2$ are used by Aberger *et al.* [11] and Ammar *et al.* [20], $Q2-7$ are used by Lai *et al.* in [114], and $Q12$ is used by Qiu *et al.* [151].

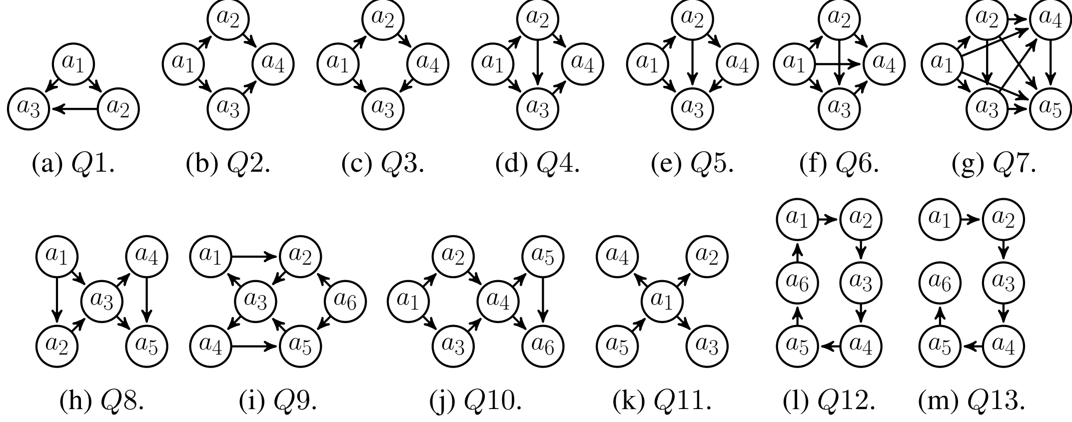


Figure 3.9: Query Examples.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**
Amazon	11.6M	118.2M	110.0M	59.0M	64.8M	37.8M	118.9M
Google	28.2M	375.3M	358.4M	239.9M	295.8M	217.0M	2.0B
Epinions	3.6M	326.3M	305.0M	87.0M	100.3M	32.0M	320.6M
	Q8	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**
Amazon	558.7M	3.1B	5.8B	3.1B	4.5B	30.2B	907.3M
Google	3.9B	42.5B	61.5B	173.1B	34.2B	266.4B	256.6B
Epinions	12.5B	262.1B	1125.2B	7865.1B	1544.5B	39502.0B	32.9B

Table 3.8: Output size for queries in Figure 3.9 on the Amazon, Google, and Epinions datasets.

3.7.3 Plan Suitability For Different Queries and Optimizer Evaluation

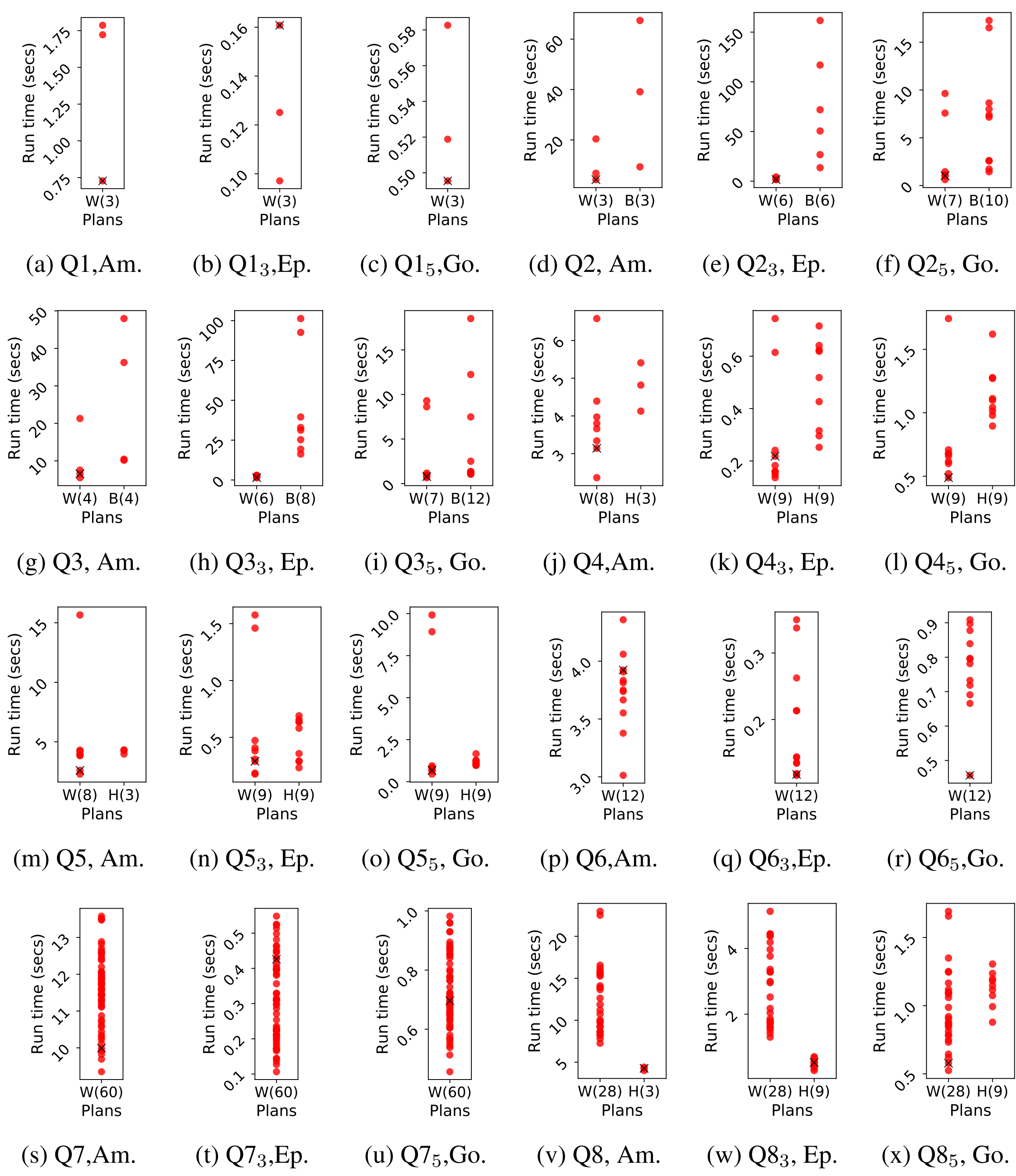
In order to evaluate how good the plans our optimizer generates are, we compare the run time of its chosen plan against the *plan spectrum of the query i.e.*, the run time of all plans GraphflowDB

enumerates for that query on a particular dataset. This also allows us to study which types of plans are suitable for which queries. We generated the plan spectrums of $Q1-8$ and $Q11-13$ on Amazon as a single edge relation, Epinions split into 3 edge relations, and Google split into 5 edge relations. The spectrums of $Q9-10$ on the three datasets and $Q12-13$ on Epinions took a prohibitively long time to generate and are omitted. Figures 3.10 and 3.11 show the GraphflowDB plan spectrums for queries $Q1-8$ and $Q11-13$, respectively. Each dot in the figures is the run time of a plan and '$\times$' indicates the plan our optimizer chose. Throughout these experiments, we use the term "optimal plan" to refer to the executed plan with the lowest run time, *i.e.*, the plan corresponding to the lowest dot in our plan spectrum charts.

We first observe that different types of plans are more suitable for different queries. The main structural properties of a query that govern which types of plans will perform well are how large and how cyclic the query is. For clique-like queries, such as $Q5$, and small cycle queries, such as $Q3$, the best plans are the WCOJ ones. On acyclic queries, such as $Q11$ and $Q13$, binary join plans are best on some datasets and WCOJ plans on others. On acyclic queries WCOJ plans are equivalent to left deep binary join plans that only use `INLJ`, which are worse than bushy binary join plans on some datasets. Finally, hybrid plans are best plans for queries that contain small cyclic structure that do no share edges *e.g.*, $Q8$.

The most interesting query is $Q12$, which is a 6-cycle query. $Q12$ can be evaluated efficiently with both WCOJ and hybrid plans (and reasonably well with some binary join plans). The hybrid plans first perform binary joins to compute 4-paths, and then extend 4-paths into 6-cycles with a multi-way join intersection. Figure 3.5 from Section 3.4.1 shows an example of such hybrid plans. These plans do not correspond to the generalized hypertree decompositions in the plan space of EmptyHeaded. On the Amazon dataset, one of these hybrid plans is optimal and our optimizer chooses that plan. On the Google dataset, our optimizer chooses an efficient binary join plan although the optimal plan is a WCOJ one.

The plans of our optimizer were broadly close to optimal across our experiments. Specifically, our optimizer's plan was optimal in 15 of our 31 spectrums, was within 1.4x of the optimal in 21 spectrum and within 2x in 28 spectrums. In 2 of the 3 cases we were more than 2x of the optimal, the absolute run time difference was in sub-seconds. Ignoring queries whose plans generally ran in sub-second latency, there was only one experiment in which our plan was not close to the optimal plan, which is shown in Figure 3.11b. Observe that our optimizer chooses different types of plans across different types of queries. In addition, as we demonstrated with $Q12$ above, we can pick different plans for the same query on different datasets ($Q8$ and $Q13$ are other examples). Finally, our optimizer generated a plan within 331ms in all of our experiments except for $Q7_5$ on Google which took 1.4 secs.



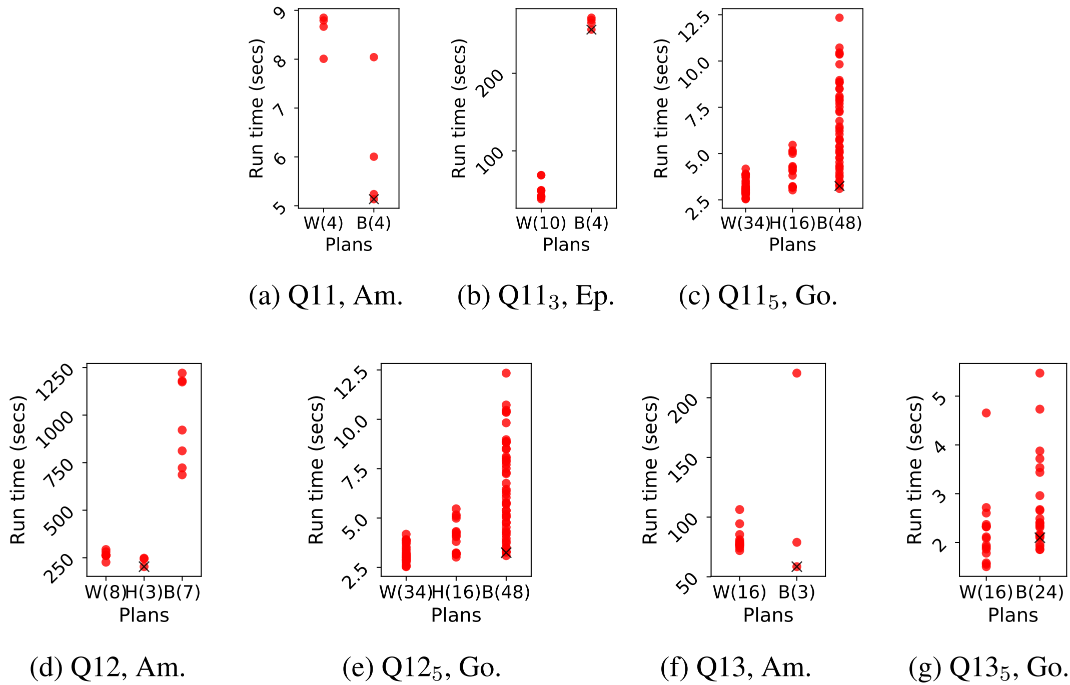
(a) Q1,Am. (b) $Q1_3$,Ep. (c) $Q1_5$,Go. (d) Q2, Am. (e) $Q2_3$, Ep. (f) $Q2_5$, Go.

(g) Q3, Am. (h) $Q3_3$, Ep. (i) $Q3_5$, Go. (j) Q4,Am. (k) $Q4_3$, Ep. (l) $Q4_5$, Go.

(m) Q5, Am. (n) $Q5_3$, Ep. (o) $Q5_5$, Go. (p) Q6,Am. (q) $Q6_3$,Ep. (r) $Q6_5$,Go.

(s) Q7,Am. (t) $Q7_3$,Ep. (u) $Q7_5$,Go. (v) Q8, Am. (w) $Q8_3$, Ep. (x) $Q8_5$, Go.

Figure 3.10: Run time (secs) of GraphflowDB plan spectrum for $Q1-8$.
'x' specifies the plan GraphflowDB chooses.



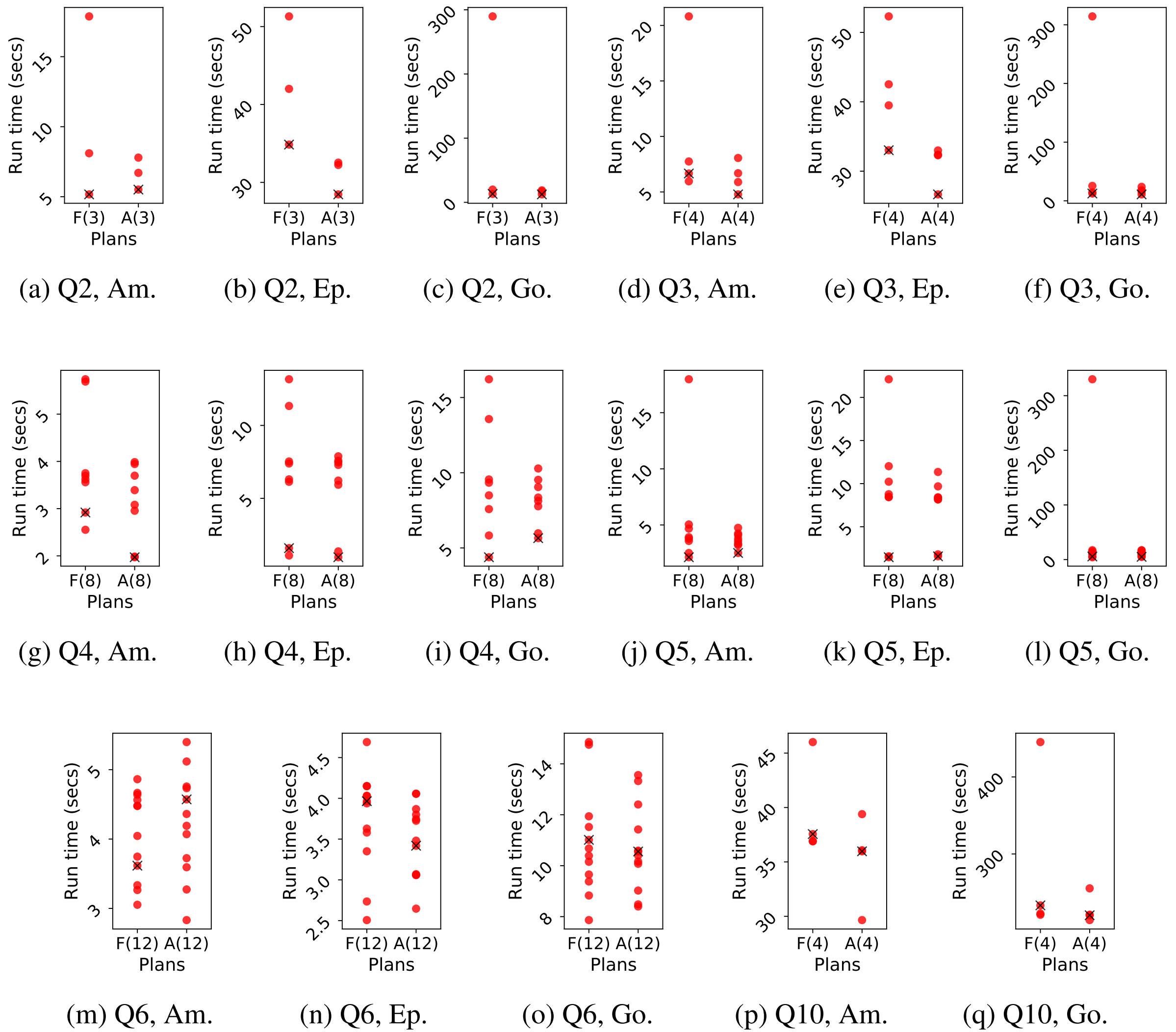
(a) Q11, Am. (b) $Q11_3$, Ep. (c) $Q11_5$, Go.

(d) Q12, Am. (e) $Q12_5$, Go. (f) Q13, Am. (g) $Q13_5$, Go.

Figure 3.11: Run time (secs) of GraphflowDB plan spectrum for $Q11 - 13$. 'x' specifies the plan picked by GraphflowDB.

3.7.4 Adaptive WCOJ Plan Evaluation

In order to understand the benefits we get by adaptively choosing JAOs, we studied the spectrums of WCOJ plans of $Q2$, $Q3$, $Q4$, $Q5$, and $Q6$, and hybrid plans for $Q10$ on Epinions, Amazon and Google datasets. These are the queries in which the fixed plans of our dynamic programming (DP) optimizer contained a chain of two or more `IMJ` operators so we could adapt the ordering over the chain. The spectrum of $Q10$ on Epinions took a prohibitively long time to generate and is omitted. Figure 3.12 shows the 17 spectrums we generated. In the case of $Q2$, $Q3$, and $Q4$, selecting JAOs adaptively overall improves the performance of every fixed plan. For example, the fixed plan our DP optimizer chooses for $Q3$ on Epinions improves by 1.2x but other plans improve by up to 1.6x. The spectrum of $Q10$ for hybrid plans is similar to that of $Q3$ and $Q4$. Each hybrid plan of $Q10$ computes the diamonds on the left and triangles on the right and joins on a_4. Here, we can adaptively compute the diamonds but not the triangles. Each fixed hybrid plan improves by adapting and some improve by up to 2.1x. On $Q5$, the run time of most plans remain similar but one WCOJ plan improves by 4.3x. The main benefit of adapting is that it makes our optimizer more robust against choosing bad JAOs. Specifically, the deviation between



(a) Q2, Am. (b) Q2, Ep. (c) Q2, Go. (d) Q3, Am. (e) Q3, Ep. (f) Q3, Go.

(g) Q4, Am. (h) Q4, Ep. (i) Q4, Go. (j) Q5, Am. (k) Q5, Ep. (l) Q5, Go.

(m) Q6, Am. (n) Q6, Ep. (o) Q6, Go. (p) Q10, Am. (q) Q10, Go.

Figure 3.12: Run time (secs) of adaptive plans enumerated by GraphflowDB for queries $Q2-6$ and $Q10$. 'x' specifies the plan picked by GraphflowDB.

the best and worst plans are smaller in adaptive plans than fixed plans.

The only exception to these observations is $Q6$, where the performance of several plans gets worse, even though the deviation between good and bad plans still became smaller. We observed that for cliques, the overheads of adaptively choosing JAOs is higher than other queries. This is because: (i) cost re-evaluation accesses many actual adjacency list sizes, so the overheads are high; and (ii) the JAOs of cliques have similar behaviours, *i.e.*, each one extends edges to triangles, then 4-cliques, *etc.* , so the benefits are low.

3.7.5 EmptyHeaded Comparison

EmptyHeaded (EH) is one of the most efficient systems for conjunctive queries within graph workloads and its plans are the closest to ours. Recall from Section 3.1 that EH has a cost-based optimizer that chooses a generalized hypertree decomposition (GHD) with the minimum width, *i.e.*, chooses a GHD with the lowest AGM bound across all of its subqueries. The details of choosing GHDs by EH are introduced by Aberger *et al.* [11]. Briefly, A GHD D of Q is a decomposition of Q where each GHD node i is labelled with a subquery Q_i of Q. The interpretation of a GHD D as a join plan is as follows: each subquery is evaluated using Generic Join first and materialized into an intermediate table. Then, starting from the leaves, each table is joined into its parent in an arbitrary order.

This approach allows EH to often choose good decompositions. However EH: (1) does not optimize the choice of JAOs for computing its subqueries; and (2) cannot choose plans that have multi-way join intersections after a binary join, as such plans do not correspond to GHDs. In particular, the JAO that EH chooses for a query Q is the lexicographic order of the attributes used when a user issues the query in Datalog. EH's only heuristic is that JAOs of two subqueries that are joined start with attributes on which the join will happen. Therefore by issuing the same query with different variables, users can make EH choose a good or a bad JAO. This shortcoming has the advantage that by making EH choose good JAOs, we can show that our orderings also improve EH. The important point is that EH does not optimize for JAOs. We therefore report EH's performance with both "bad" JAOs ($\texttt{EH}_b$) and "good" JAOs ($\texttt{EH}_g$). For good JAOs we use the JAO that GraphflowDB chooses. For bad orderings, we generated the spectrum of plans in EH (explained momentarily) and chose the worst-performing JAO for the GHD EH chooses. For our experiments we ran $Q3$, $Q5$, $Q7$, $Q8$, $Q9$, $Q12$, and $Q13$ on Amazon, Google, and Epinions. We first explain how we generated the EH spectrums and then present our results.

Subsumed EmptyHeaded Plans: We show that our plan space contains the GHD plans of EmptyHeaded (EH) that satisfy the projection constraint. A GHD can easily be turned into a join plan

T in our notation (from Section 3.4) by "expanding" each subquery Q_i into a WCOJ subplan according to the chosen JAO that EH picks for Q_i and adding intermediate nodes in T that are the results of the joins that EH performs. Given Q, EH chooses the GHD D^* for Q as follows. First, EH loops over each GHD D of Q, and computes the worst-case size of the subqueries, which are the AGM bounds of these queries *i.e.*, the minimum *fractional edge covers* of subqueries [24]. The maximum size of the subqueries is the width of GHD and EH chooses the GHD with the minimum width. This effectively implies that one of these GHDs satisfy our projection constraint. This is because adding a relation to a query Q_i while keeping the same join attributes decreases its fractional edge cover. In graph terms, the added relation closes a cycle in the graph pattern. To see this consider Q_i', which contains the same set of join attributes as Q_i but has some of missing relations of Q_i. Let E_i and E_i' be the query edges associated with the graph patterns for Q_i and Q_i', respectively. Some missing relations while keeping the same join attributes is equivalent to missing query edges in the graph pattern with the same query vertices. Any fractional edge cover for Q_i is a fractional edge cover for Q_i' (by giving weight 0 to $E_i'{-}E_i$ in the cover), so the minimum fractional edge cover of Q_i' is at most that of Q_i, proving that D^* is in our plan space.

GraphflowDB vs EmptyHeaded Run Time Comparisons: We ran our queries on GraphflowDB with adapting off. To compare, we ran two EH plans with good and bad JAOs for $Q3$, $Q5$, $Q7$, and $Q8$ (recall no EH plan ran within our time limit for $Q9$, $Q12$, and $Q13$). We repeated the experiments with edges in a single relation and split randomly across two relations. Table 3.9 shows our results. GraphflowDB is always faster than $\texttt{EH}_b$, except for $Q1$ on Google and $Q8_2$ on Amazon where the difference is only 500ms and 200ms, respectively. The run time of EH is as high as 68x that of GraphflowDB in one instance. The most performance difference is on $Q5$ and Google, for which both our system and EH use a WCOJ plan. When we force EH to choose a good JAO, on smaller size queries EH can be more efficient than our plans. For example, although GraphflowDB is 32x faster than $\texttt{EH}_b$ on $Q5$ Google, it is 1.2x slower than $\texttt{EH}_g$. Importantly $\texttt{EH}_g$ is always faster than $\texttt{EH}_b$, showing that our JAOs improve run times consistently in a completely independent system that implements WCOJs.

We next discuss $Q9$, which demonstrates again the benefit we get by seamlessly mixing intersections with binary joins. Figure 3.13 shows the plan our optimizer chooses on $Q9$ on all of our datasets. Our plan separately computes two triangles, joins them, and finally performs a 2-way join intersection. This execution does not correspond to the GHD plans of EH, so is not in its plan space. Instead, EH considers two GHDs for this query and both timed out.

We verified that for every query from Figure 3.9, the plans EH chooses satisfy the projection constraint. However, there are minimum-width GHDs that do not satisfy this constraint. For example, for $Q10$, EH finds two minimum-width GHDs: (i) one that joins a diamond *i.e.*,

		Q1	Q3	Q3$_2$	Q5	Q5$_2$	Q7	Q7$_2$	Q8	Q8$_2$	Q9	Q9$_2$	Q12	Q12$_2$	Q13	Q13$_2$
	EH$_b$	1.0	19.0	3.4	47.1	9.2	91.4	11.6	22.2	1.8	Mm	Mm	Mm	Mm	Mm	Mm
Am	EH$_g$	0.6	5.4	1.3	3.3	1.5	21.2	1.7	10.6	1.4	Mm	Mm	Mm	Mm	Mm	Mm
	GF	0.6	5.5	2.1	1.9	0.8	9.5	0.9	5.1	2.0	24.7	2.4	209.2	14.8	48.0	11.3
	EH$_b$	1.9	444.5	42.6	401.1	77.6	1.04K	23.4	66.6	16.0	Mm	Mm	Mm	Mm	Mm	Mm
Go	EH$_g$	1.4	12.0	2.1	11.3	2.3	107.3	4.8	35.8	3.0	Mm	Mm	Mm	Mm	Mm	Mm
	GF	2.6	14.0	4.0	5.9	2.1	48.8	3.3	17.0	4.5	236.2	6.9	510.6	73.8	1.44K	70.1
	EH$_b$	0.5	42.7	6.5	64.5	11.4	560.7	2.9	1.01K	22.0	Mm	Mm	Mm	Mm	Mm	Mm
Ep	EH$_g$	0.2	26.6	1.7	3.5	0.9	45.7	0.8	117.2	7.0	Mm	Mm	Mm	Mm	Mm	Mm
	GF	0.4	28.1	4.6	1.5	0.6	23.7	1.2	37.5	5.4	865.3	26.1	TL	TL	95.0k	2.35k

Table 3.9: Run time (secs) of GraphflowDB (GF) and EmptyHeaded with good JAOs (EH$_g$) and bad JAOs (EH$_b$). TL indicates a timeout after 48 hrs. Mm indicates running out of memory.

$(a_1, a_2), (a_2, a_4), (a_1, a_3), (a_3, a_4)$ and a triangle *i.e.*, $(a_4, a_5), (a_4, a_6), (a_5, a_6)$ (width 2); and (ii) one that joins a three-path *i.e.*, $(a_1, a_2), (a_1, a_3), (a_3, a_4)$ with a triangle with an extended edge *i.e.*, $(a_2, a_4), (a_4, a_5), (a_4, a_6), (a_5, a_6)$ (also width 2). The first GHD satisfies the projection constraint, while the second one does not. EH arbitrarily chooses the first GHD. As we argued in Section 3.4.1, satisfying the projection constraint is not a disadvantage, as it makes the plans generate fewer intermediate tuples. For example, on a Gnutella peer-to-peer graph [112] (neither GHD finished in a reasonable amount of time on our datasets from Table 3.7), the first GHD for $Q10$ takes around 150ms, while the second one does not finish within 30 minutes.

3.7.6 Cache Trie Join (CTJ) Comparisons

Similar to Generic Join, LFTJ [179] is a WCOJ algorithm that evaluates join queries one attribute at a time using intersections. This is all that is necessary to know about LFTJ to understand CTJ's approach. We explain the algorithm in detail in our related work Chapter 6. Therefore the same optimization problem of choosing a good JAO arises when using LFTJ. An important advantage of these algorithms is their small memory footprints. For example, when executed in a purely pipelined fashion, LFTJ does not require memory to keep large intermediate results. Kalinsky *et al.* observe that by keeping a cache of certain intermediate results and reusing these results, LFTJ's performance can be improved [91]. For example, consider evaluating the 'two-triangle' query $Q8$ and using the JAO $(a_1, a_2, a_3, a_4, a_5)$. Note that for each a_3 value, irrespective of the

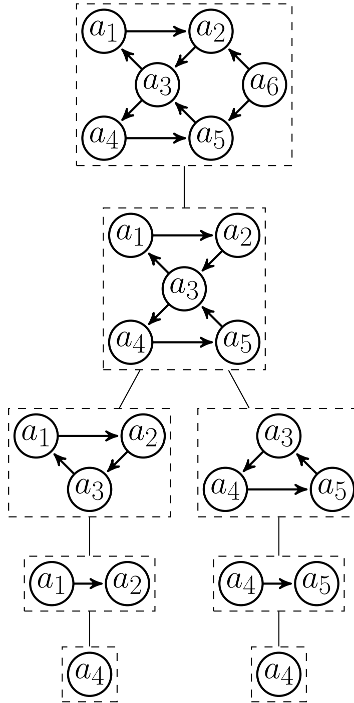
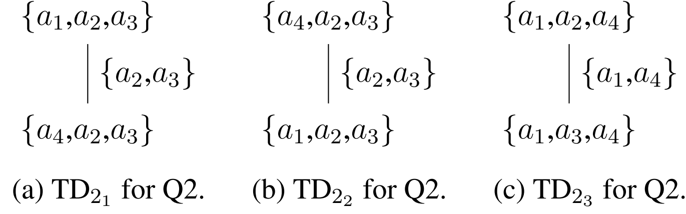


Figure 3.13: Plan with seamless mixing of intersections and binary joins on Q9.

previous a_1 and a_2 values, the same a_4a_5 values would be matched. Therefore, if LFTJ keeps a cache of a_3 to a_4a_5 matches as it extends a_3's, it can save and reuse computation. CTJ is an algorithm that extends LFTJ with caching [91]. This is a more advanced cache than our cache within multiway intersections and in some queries gives LFTJ benefits that are similar to using the `Hash Join` operator in binary or hybrid join plans. For example, consider a hybrid plan for $Q8$ that uses a `HJ` to evaluate (a_3, a_4, a_5) triangles on the one side, hashes these on a_3, and then probes this hash table with (a_1, a_2, a_3) triangles. The hash table here is similar to CTJ's cache and reuses the computation that was done to compute (a_3, a_4, a_5) triangles for different a_3 values.

CTJ generates plans as follows. First CTJ enumerates a set of *ordered tree decompositions* (TDs), which are rooted TDs, whose bags have a particular *preorder* [91]. The adhesion of two parent-child bags is the number of common attributes they have. Then using a set of heuristics, CTJ chooses one of these TDs. Specifically, CTJ chooses a TD with the minimum value for its largest adhesions, breaking ties with maximizing the number of bags, and then minimizing the sum of adhesions. One of these TDs is chosen arbitrarily (say TD T). Then for T, CTJ defines: 1) a set of *compatible JAOs*; and 2) a caching scheme. Finally, from the compatible JAOs, one is



(a) TD_{2_1} for Q2. (b) TD_{2_2} for Q2. (c) TD_{2_3} for Q2.

Figure 3.14: Example of CTJ's tree decompositions (TDs) for Q2.

chosen using heuristics from another reference, Tributary Join [43]. We explain with an example.

Example:
Consider the Q2 diamond query from Figure 3.9b. For this query, there are several TDs that CTJ can choose according to its heuristics. Three of these are shown in Figure 3.14a, 3.14b, and 3.14c as they have the same adhesion sizes (which is minimum) and the other tie-break metrics. Suppose CTJ chooses TD_{1_2}. A preorder traversal on TD_{1_2} orders the bags as follows: 1) $\{a_1,a_2,a_3\}$; and 2) $\{a_4,a_2,a_3\}$. Next, CTJ removes from each bag the query vertices in the adhesions found in the root-to-bag path, which yields the ordering: 1) $\{a_1,a_2,a_3\}$; and 2) $\{a_4\}$. These are the variables *owned* by each bag. The compatible JAOs are those that order the JAO for each bag and concatenating these orderings from root to the leaf. Out of these, CTJ uses another cost called Tributary Join's [43] cost model to choose the JAO in each bag. We explain Tributary Join momentarily. Suppose the algorithm chooses the JAO (a_1, a_2, a_3, a_4). For each non-root bag B in TD, CTJ adds a cache to LFTJ. Suppose the parent of B is p in TD. The cache has: (i) as key the query vertices in the adhesion of p and B; and (ii) as value the query vertices 'owned' by B. For example, the cache for the JAO (a_1, a_2, a_3, a_4) for TD_{1_2} will be from key:$\{a_2,a_3\}$ to value:a_4.

The focus of CTJ and reference [91] is to control the memory consumption of LFTJ to increase its performance and not on how to choose TDs or optimize the JAO selection. For example, while CTJ can avoid storing the complete joins of subqueries, our binary join and hybrid plans do not have mechanisms to control for memory. In our setting we assume that the HASHJOIN operators have enough memory to create their hash tables. Instead, our work focuses primarily on efficient plan selection for queries. There are several important differences between our optimized plans and the plans CTJ uses:

1. On some queries, the heuristics that CTJ uses to choose a TD cannot distinguish between efficient TDs from inefficient ones. For example, consider the diamond query Q2 from Figure 3.9b. CTJ's heuristics will not be able to tie break between TD_{2_1}, TD_{2_2}, and TD_{2_3} in Figures 3.14a, 3.14b, and 3.14c, respectively and chooses one of these arbitrarily, which yield different JAOs (and caching schemes). In fact, in the code provided by the authors, we

noticed that similar to EH, we can make CTJ choose different TDs and final JAOs, with very different runtimes. For instance, TD_{2_1} and TD_{2_2} on Google lead to run times 86.6 seconds and 806.2 seconds, respectively. The difference in run time is mainly due to a difference in the number of intermediate results, which are 72.94M for TD_{2_1} and 1.38B for TD_{2_2}. Yet CTJ's optimizer does not differentiate between these two TDs and their final JAOs. Instead, our i-cost-based model can differentiate between these JAOs.

2. Once a TD has been chosen, CTJ uses the Tributary Join technique to choose a JAO within each bag[43]. Tributary Join studies choosing the JAO for LFTJ algorithm in the context of joining multiple relational tables and chooses the JAO based on the distinct values in the attributes of the relations. This heuristic however is not designed for self-join queries as in graph workloads, where attributes will have the same number of distinct values, specifically $|V|$ (assuming every vertex in an input graph has an incoming and outgoing edge). Recall that in conjunctive queries in graph workloads, each binary `E(`a_i`,`a_j`)` relation is a replica of the edges of the input graph $G(V, E)$. Note that in our evaluations we either use a single edge relation or split the edges into multiple relations, but any differences in the distinct values across when splitting would be due to random assignment.
3. On some queries CTJ's plans do not benefit from caching results of subqueries larger than a single query edge, due to the heuristics CTJ uses to choose TDs. For example, for a path query, say $Q13$, CTJ considers TD's in which each bag consists of a single query edge. Since CTJ caches the results of a single bag, only results of a single query edge, so adjacency lists can be cached. This contrasts with traditional binary join plans that can cache subqueries within bushy plans.

We compared GraphflowDB to CTJ on queries $Q1$ to $Q14$ on Amazon, Google, and Epinions. We obtained the code from the authors of CTJ [91]. Recall that CTJ's main focus is in controlling the cache size. We observed that we obtain the best runtime numbers when we run CTJ with an unbounded cache size, which implies that CTJ caches every key-value between each bag. Table 3.10 shows our results. As we explained above, CTJ can choose between multiple different TDs and JAOs depending on how the query is written. In Table 3.10, we report the best run time for CTJ for each query after writing the attributes of the query in every lexicographic order. As shown in the table, GraphflowDB outperforms the implementation we obtained for CTJ across these queries, varying from competitive performances to differences that are two orders of magnitude in runtime. We note that it is not possible to do a very controlled comparison here because the implementations of GraphflowDB plans and CTJ are very different, *e.g.*, the implementations use different programming languages and data organization. However, the differences we discussed above contribute to these runtime differences. For example, the plan that CTJ uses for $Q13$, which is a path query and where CTJ does not benefit from caching, generates 3.43B intermediate tuples on Amazon. In contrast, GraphflowDB's plan hashes on a_4 and

		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**
	CTJ	5.1	38.9	41.5	22.6	21.0	18.6	61.1
Am	GF	0.6	4.7	5.5	2.0	1.9	3.3	9.5
		(8.5x)	**(8.3x)**	**(7.6x)**	**(11.3x)**	**(11.1x)**	**(5.6x)**	**(6.4x)**
	CTJ	15.3	82.7	86.6	59.4	55.7	64.0	464.1
Go	GF	2.6	12.3	12.0	4.9	5.9	8.6	48.8
		(5.9x)	**(6.7x)**	**(7.2x)**	**(12.1x)**	**(9.4x)**	**(7.4x)**	**(9.5x)**
	CTJ	2.3	88.4	94.7	10.5	9.5	27.5	329.2
Ep	GF	0.4	31.5	26.6	1.5	1.5	3.3	23.7
		(5.8x)	**(2.8x)**	**(3.6x)**	**(7.2x)**	**(6.3x)**	**(8.3x)**	**(13.9x)**
		Q8	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**
	CTJ	94.8	142.1	2256.5	184.2	878.5	456.0	639.6
Am	GF	5.1	56.3	20.8	6.8	209.2	48.0	125.0
		(18.6x)	**(2.5x)**	**(108.5x)**	**(27.1x)**	**(4.2x)**	**(9.5x)**	**(5.12x)**
	CTJ	606.3	574.4	94084.1	8055.1	3048.5	2165.4	67049.9
Go	GF	17.0	303.9	135.9	214.6	510.6	1440.0	5348.7
		(35.7x)	**(1.9x)**	**(692.3x)**	**(37.5x)**	**(6.0x)**	**(1.5x)**	**(1.5x)**
	CTJ	3251.1	1618.8	158274.2	TL	TL	145K	95027.4
EP	GF	37.5	**(1.5x)**	1908.7	12852.5	TL	95027.4	3373.1
		(86.7x)	2384.8	**(82.9x)**			**(1.5x)**	**(13.0)**

Table 3.10: Run time (secs) of GraphflowDB (GF) and CTJ.
TL indicates the query did not finish in 48 hrs.

generates only 0.39B intermediate tuples.

3.7.7 CFL Comparison

CFL [29] is one of the state-of-the-art *subgraph matching algorithms*. Subgraph matching is equivalent to conjunctive queries in graph workloads. The main optimization of CFL is "postponing Cartesian products". There are conditionally independent parts of the query that can be matched separately and appear as Cartesian products in the output. CFL decomposes a query into a dense *core* and a *forest*. Broadly, the algorithm first matches the core, where fewer matches are expected and there is less chance of independence between the parts. Then the forest is matched.

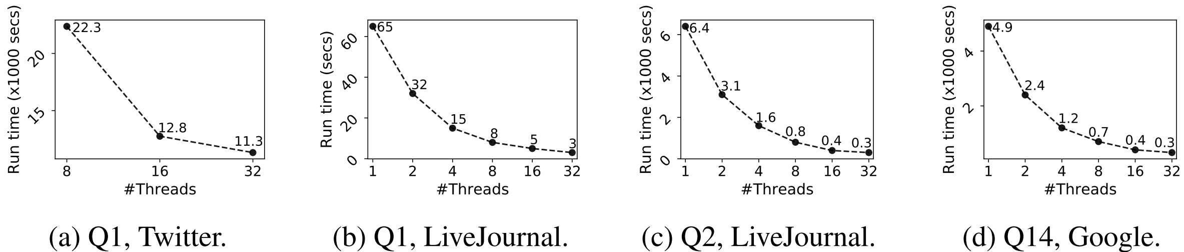
|T|		Q10s	Q15s	Q20s	Q10d	Q15d	Q20d
10^5	GF	7.3	6.0	5.5	29.2	99.8	142.0
	CFL	9.3	17.5	40.5	13.2	389.9	1,140.7
		(1.2x)	**(2.9x)**	**(7.3x)**	**(0.4x)**	**(3.9x)**	**(8.0x)**
10^8	GF	625.6	665.5	797.2	1,159.6	1,906.2	1,556.9
	CFL	4,818.9	5,898.1	7,104.1	7,974.3	11,656.2	19,135.7
		(7.7x)	**(8.8x)**	**(8.9x)**	**(6.8x)**	**(6.1x)**	**(12.2x)**

Table 3.11: Average run time (secs) of GraphflowDB (`GF`) and `CFL` on large queries. Q*i(s/d)* is a query set of 100 randomly generated queries where *i* is the number of query vertices in the graph pattern and *s* and *d* specify sparse and dense queries, respectively.

In both parts, any detected Cartesian products are postponed and evaluated independently. This reduces the number of intermediate results the algorithm generates.[2]

CFL also builds an index called CPI, which is used to quickly enumerate matches of paths in the query during evaluation. We follow CFL's prior experimental setting [29]. We obtained the CFL code and 6 different query sets from the authors [29]. Each query set contains 100 randomly generated queries that are either sparse (average query vertex degree ≤ 3) or dense (average query vertex degree > 3). We used three sparse query sets $Q10s$, $Q15s$, and $Q20s$ containing queries with 10, 15, and 20 query vertices, respectively. Similarly, we used three dense query sets $Q10d$, $Q15d$, and $Q20d$. To be close to CFL's setup, we use the `human` dataset from their original experimental setup. The dataset contains 86282 edges, 4674 vertices, with edges split into 44 distinct relations. Table 3.11 compares the run time of GraphflowDB and CFL. We report the average run time per query for each query set when we limit the output to 10^5 and 10^8 matches. Except for one of our experiments, on $Q10d$ with 10^5 output size limit, GraphflowDB's run times are faster (between 1.2x to 12.2x) than CFL. We note that although our run time results are faster than CFL on average, readers should not interpret these results as one approach being superior to another. For example, postponing of Cartesian products optimization and a CPI index can improve our approach. However, one major advantage of our approach is that we do flat tuple-based processing using standard operators, so our techniques can easily be integrated into existing DBMSs. It is less clear how to decompose CFL-like processing into database operators.

[2] CFL, as a "graph matching" algorithm does not describe its computation as relational joins. The theory of factorization, covered in Chapters 4 and 5, capture the similar observation in a formal way to show how one can exploit such Cartesian products when performing joins.



(a) Q1, Twitter. (b) Q1, LiveJournal. (c) Q2, LiveJournal. (d) Q14, Google.

Figure 3.15: Scalability experiments.

3.7.8 Scalability Experiments

We next demonstrate the scalability of GraphflowDB on larger datasets and across a larger number of physical cores. The goal of our experiments is to demonstrate that when more cores are available, our approach can utilize them efficiently. We evaluated $Q1$ on LiveJournal and Twitter graphs, $Q2$ on LiveJournal, and $Q14$, which is a very difficult 7-clique query, on Google. We repeated each query with 1, 2, 4, 8, 16, and 32 cores, except we use 8, 16, and 32 cores on the Twitter graph. Figure 3.15 shows our results. Our plans scale linearly until 16 cores with a slight slow down when moving 32 cores which is the maximum number of cores in our hardware. For example, going from 1 core to 16 cores, our run time is reduced by 13x for $Q1$ on LiveJournal, 16x for $Q2$ on LiveJournal and 12.3x for $Q14$ on Google.

3.7.9 Catalogue Experiments

We present preliminary experiments to show two tradeoffs: (1) the space vs estimation quality tradeoff that parameter h determines; and (2) construction time vs estimation quality tradeoff that parameter z determines. For estimation quality we evaluate cardinality estimation and omit the estimation of adjacency list sizes, *i.e.*, the $|A|$ column, that we use in our i-cost estimates. We first generated all 5-vertex size unlabeled queries. This gives us 535 queries. For each query, we assign labels at random given the number of labels in the dataset (we consider Amazon with 1 label, Google with 3 labels). Then for each dataset, we construct two sets of catalogues: (1) we fix z to 1000, and construct a catalogue with $h = 2$, $h = 3$, and $h = 4$ and record the number of entries in the catalogue; (2) we fix h to 3 and construct a catalogue with $z = 100$, $z = 500$, $z = 1000$, and $z = 5000$ and record the construction time. Then, for each labeled query Q, we first compute its actual cardinality, $|Q_{true}|$, and record the estimated cardinality of Q, Q_{est} for each catalogue we constructed. Using these estimation we record the q-error of the estimation, which is max($|Q_{est}|$ / $|Q_{true}|$, $|Q_{true}|$ / $|Q_{est}|$). This is an error metric used in prior cardinality estimation

	z	CT	≤2	≤3	≤3	≤5	≤10	>20
Am	100	0.1	318	445	510	526	529	535
	500	0.3	384	486	520	527	530	535
	1,000	0.5	383	481	519	529	532	535
	5,000	1.5	384	475	518	529	532	535
Go$_3$	100	3.1	166	276	356	415	461	535
	500	9.3	214	310	371	430	477	535
	1,000	17.0	222	315	371	430	475	535
	5,000	66.1	219	322	373	432	473	535

Table 3.12: Q-error and catalogue creation time (CT) in secs for GraphflowDB for different z values.

		h	|R|	≤2	≤3	≤3	≤5	≤10	>20
Am	GF	2	8	348	464	512	523	527	535
		3	138	381	482	512	524	527	535
		4	2858	498	510	518	524	527	535
	PG	-	-	15	15	23	23	25	535
Go$_3$	GF	2	144	181	289	375	447	492	535
		3	20.3K	222	315	371	430	475	535
		4	11.9M	441	497	515	524	529	535
	PG	-	-	0	0	0	0	0	535

Table 3.13: Postgres and GraphflowDB Q-error and number of catalogue entries (|R|) for `GF` for different h values.

work [110] that is at least 1, where 1 indicates completely accurate estimation. As a very basic baseline, we also compared our catalogues to the cardinality estimator of PostgreSQL. For each dataset, we created an Edge relation E(from, to). We create two composite indexes on the table on (from, to) and (to, from) which are equivalent to our forward and backward adjacency lists. We collected stats on each table through the ANALYZE command. We obtain PostgreSQL's estimate by writing each query in an equivalent SQL select-join query and running EXPLAIN on the SQL query.

Our results are shown in Tables 3.12 and 3.13 as cumulative distributions as follows: for different q-error bounds τ, we show the number of queries that a particular catalogue estimated

with q-error at most τ. As expected, larger h and larger z values lead to less q-error, while respectively yielding larger catalogue sizes and longer construction times. The biggest q-error differences are obtained when moving from $h = 3$ to $h = 4$ and $z = 100$ to $z = 500$. There are a few exception τ values when the larger h or z values lead to very minor decreases in the number of queries within the τ bound but the trend holds broadly.

Chapter 4

Factorized Vector Execution

GraphflowDB's execution engine for our hybrid plans that seamlessly mix worst-case optimal joins (WCOJs) and binary joins is a push-based tuple-at-a-time engine. This execution engine has two shortcomings. First, the tuple-at-a-time approach, which follows the Volcano execution model, works well when disk is the primary bottleneck but incurs a high cost for in-memory DBMSs like GraphflowDB. However, Volcano execution model has a high 'interpretation cost'. This is the cost of the high number of *virtual* function calls incurred throughout the execution of a pipeline. Furthermore, the model is not efficient on modern CPUs. Two broad approaches are known when it comes to improving on the Volcano model: i) *Code generation execution model*: pioneered by HyPer [128]; and ii) *Vectorized execution model*: pioneered by Vectorwise [194]. We focus on moving GraphflowDB towards a vectorized execution engine as it is easier to manage from the perspective of code complexity. Within this model, instead of passing a single tuple, we pass a block of tuples in the form of vectors on which primitive operations run. This would reduce the 'interpretation cost'.[1]

Second, GraphflowDB's executor so far represents intermediate relations that are generated using flat tuples. The theory of factorization [145] has shown that it is possible to compress intermediate results using *factorized representations*. To motivate this shortcoming, consider the 2-hop query $Q_{2H} = R(a, b), R(a, c)$ as a running example and the input graph shown in Figure 4.1. The GraphflowDB executor described in Chapter 3 so far would generate, in a pipelined manner, the flat representation shown in Figure 4.1b that contains $2n^2$ many tuples. However, this relation is highly compressible if for each $b = v_1$, we "factor out" the set of incoming neighbors

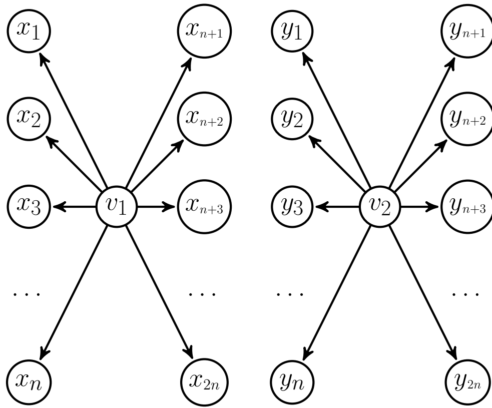
[1] Note that vectorization refers to the block of tuples being a set of vectors of data and does not directly relate to the use of vectorized/SIMD instructions.

of v_1 and outgoing neighbors of v_1 (and similarly for $b = v_2$). So in an alternative representation we could represent the same set of n^2 many tuples as

$$Q_{2H:G_{f_1}} = (<a{:}v_1> \times \cup_{i=1}^{i=2n} <b{:}x_i> \times \cup_{i=1}^{i=2n} <c{:}x_i>) \cup$$

$$(<a{:}v_2> \times \cup_{i=1}^{i=2n} <b{:}y_i> \times \cup_{i=1}^{i=2n} <c{:}y_i>)$$

using 2 tuples and $8n + 2$ values.



(a) Example input graph G.

$R(a, b), R(a, c)$			
a_1	a_2	a_3	
v_1	x_1	x_1	
$\vdots$	$\vdots$	$\vdots$	$Neighbours(v_1)$
v_1	x_1	x_{2n}	
$\vdots$	$\vdots$	$\vdots$	
v_1	x_{2n}	x_1	
$\vdots$	$\vdots$	$\vdots$	$Neighbours(v_1)$
v_1	x_{2n}	x_{2n}	
v_2	y_1	y_1	
$\vdots$	$\vdots$	$\vdots$	$Neighbours(v_2)$
v_2	y_1	y_{2n}	
$\vdots$	$\vdots$	$\vdots$	
v_2	y_{2n}	y_1	
$\vdots$	$\vdots$	$\vdots$	$Neighbours(v_2)$
v_2	y_{2n}	y_{2n}	

(b) Flat Representation for $Q_{2H:G}$.

Figure 4.1: Example input graph G and flat representation of $Q_{2H}{=}R(a, b), R(a, c)$ on G.

Factorized representations can be strictly smaller than the AGM bound of queries, *i.e.*, can lead to plans generating smaller intermediate results than our hybrid plans. Choosing a good factorization can be done statically at compilation time by inspecting the dependencies between the query's attributes. Adopting factorized representations however is not straightforward as general factorized representations are represented as tries and make primitive operations deviate heavily from those operating on traditional flat tuple representations that vectorized execution engines rely on.

The goal of this chapter is to develop a vectorized query processes that also adopts factorization representations. These two techniques are at odds because processing arbitrary factorized

representations seems to indeed require adopting tries as intermediate relations as shown by prior work [26]. We find a novel design by which we obtain the full benefit of a vectorized execution model and some of the benefits of factorization by restricting the set of factorization schemes possible. We call our approach *Factorized Vector Execution*. We cover the necessary changes to the intermediate tuple representation as well as changes to the physical operators of our hybrid plans. Our approach is easy to adopt by modern analytical DBMSs.

Next, we give an overview of traditional vector execution and existing approaches to adopting factorized representation as well as our contributions.

4.1 Existing Approaches and Overview of Contributions

Analytical DBMSs such as MonetDB [34], DuckDB [152], and VectorWise [194] use a vectorized query execution model. Within this model, vectors store a subset of the tuples of intermediate relations. Each vector stores values of the same attribute and hence of the same datatype. Furthermore, a vector has a predefined maximum size, usually 1024. The vectors are passed between operators in contrast to prior tuple-at-a-time execution models such as Volcano leading to a reduction in the cost of interpretation. In all of these systems, many of the primitive operations are done in tight-loops reducing the overhead of traditional Volcano-style execution.

The general query processor design of these systems however would not seem to be suitable for adopting factorized representations that require processing tries. The FDB system is the first and only proposed factorized query processor [26]. The query processor of FDB adopts representations called f-representations and is based on operators that perform transformations over entire tries. Specifically, FDB indexes all input relations as tries and its plans consist of a linear sequence of operators that take as input tries and output and materialize tries, starting with tries that represent the input relations. This approach would be hard to adopt as it deviates significantly from vectorized execution model, which processes sets of flat values in a pipelined manner.

4.1.1 Thesis Contributions

Our primary contribution is the design and implementation of vectorized query processor that adopts a limited form of f-representations that are particularly suitable for the type of queries GraphflowDB optimizes for, *i.e.*, conjunctive queries in graph workloads. Our design modifies traditional vectorized executors in two ways: (i) Instead of representing the intermediate tuples processed by operators as a single group of equal-sized vectors, we represent them as multiple

factorized groups of vector. We call these vector groups. This allows us to avoid or delay Cartesian products in the intermediate results, keeping them in compressed form. (ii) Instead of using a single group of fixed-length vectors, we use multiple variable-length vectors that take the lengths of adjacency lists that are represented in the intermediate tuples due to the nature of m-n joins in our queries. Because GraphflowDB stores adjacency lists in memory consecutively, this allows us to simply 'point' to the list and avoid materializing adjacency lists during join processing, further improving query performance.

We present extensive experiments that demonstrate the scalability and performance benefits of factorized vector execution model both on microbenchmarks and end-to-end benchmarks. The code, queries, and datasets related to the contents of this chapter available on Github.[2].

4.2 Preliminaries

In this section, we give an overview of a type of factorized representations called f-representations. We also cover the extensions we have made to the storage layer of GraphflowDB from Chapter 3. The storage layer of GraphflowDB in the Chapter 3 did not store node and edge table attributes. The storage layer is extended in this chapter to allow us to expand the queries we can support to those with multi-way joins containing simple and complex predicates on node and relationship attributes.

4.2.1 Factorized Representations

Factorized representations (f-representations) are a succinct and lossless representation for relations introduced by Olteanu *et al.* [145]. The outline of this section is as follows. First, we introduce f-representations and contrast them with *flat representations*, the common representation used by traditional DBMSs. Second, we also introduce *factorized trees (f-trees)*, a formalism to describe the structure of an f-representation. Finally, we introduce the size bounds of f-representations on query results.

To introduce f-representations, we use the 2-hop query $Q_{2H} = R(a, b), R(a, c)$ as a running example. DBMSs use flat representations for relations. Figure 4.1 shows an example input graph $G(V, E)$ and the corresponding flat representation of the output of Q_{2H} on G, which we denote as $Q_{2H:G}$. The flat representation $Q_{2H:G}$ can be seen as an algebraic expression of a union over the product of unary $<a>$, $<b>$, and $<c>$ values as follows:

[2]github.com/queryproc/columnar-storage-and-list-based-processing-for-graph-dbms

$$Q_{2H:G} = (<a{:}v_1> \times <b{:}x_1> \times <c{:}x_1>) \cup$$
$$\dots$$
$$(<a{:}v_1> \times <b{:}x_1> \times <c{:}x_{2n}>) \cup$$
$$\dots$$
$$(<a{:}v_2> \times <b{:}y_1> \times <c{:}y_1>) \cup$$
$$\dots$$
$$(<a{:}v_2> \times <b{:}y_{2n}> \times <c{:}y_{2n}>)$$

Let the size of a representation be the number of unary values it contains. For example, the size of the flat representation of $Q_{2H:G}$ from Figure 4.1b is $24{\times}n^2$ because the representation is a union over $(2n)^2$ tuples of size 3 (n^2 tuples have $a{:}v_1$, and another n^2 tuples have $a{:}v_1$). Notice that the flat representation contains a lot of repetition shown in boxes in Figure 4.1b denoted by $Neighbours(v_i)$. F-representations are compressed representations of relations that factor out such repetitions. A possible f-representation for $Q_{2H:G}$ is as follows (shown in Figure 4.2b):

$$Q_{2H:G_{f_1}} = (<a{:}v_1> \times \cup_{i=1}^{i=2n}<b{:}x_i> \times \cup_{i=1}^{i=2n}<c{:}x_i>) \cup$$
$$(<a{:}v_2> \times \cup_{i=1}^{i=2n}<b{:}y_i> \times \cup_{i=1}^{i=2n}<c{:}y_i>)$$

Observe that the size of the f-representation above is $8n + 2$.

Given an arbitrary relation $R(a, b, ...)$, finding the most compact F-representation of R is NP-hard. However, if R is the result of a query, then the tuples in R might have attribute-level conditional independences, *i.e.*, , multi-valued dependencies [56], which can be exploited to obtain compact f-representations. For example, in Q_{2H} for any fixed value of a, we can infer from the query that the sets of b and c values are independent. In other words, attributes b and c in the query results are independent conditioned on a, which is how the f-representation above was obtained. This conditional independence is described using *factorized trees*, which we introduce next.

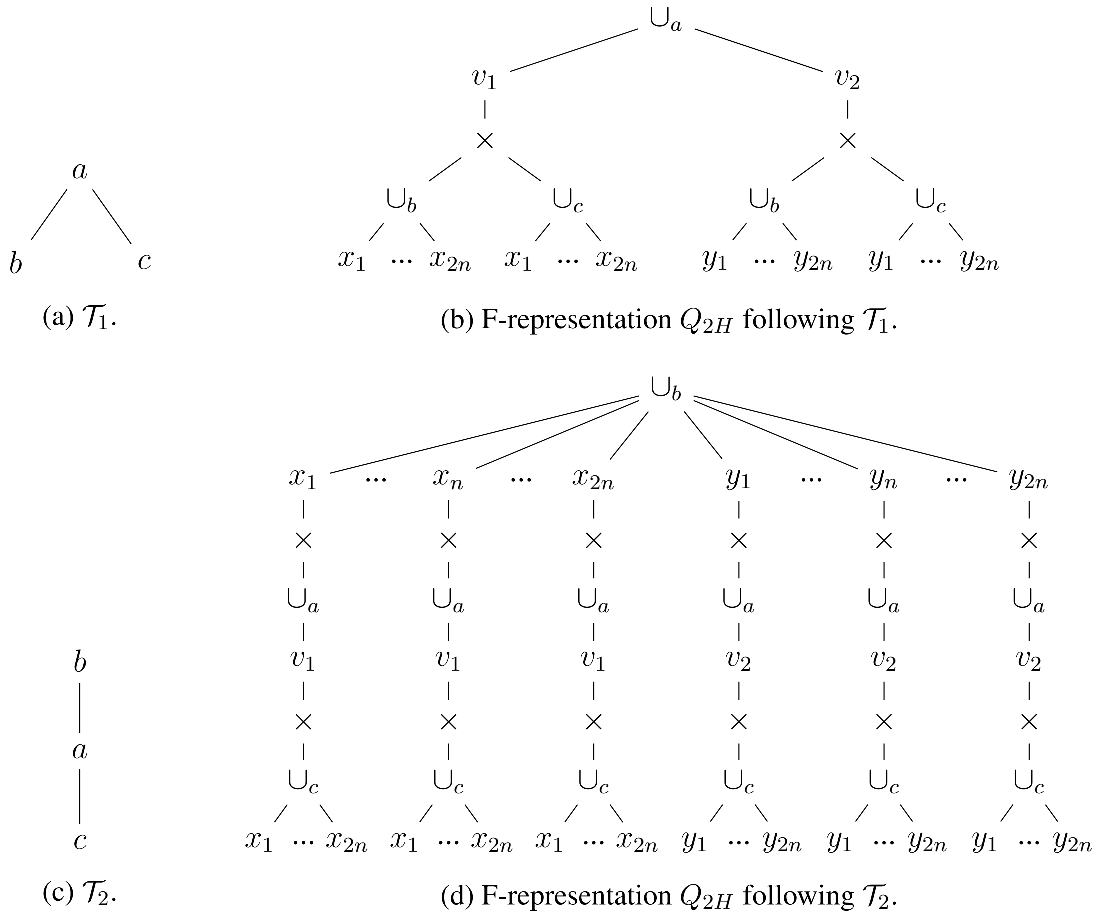


Figure 4.2: F-representations for Q_{2H} following F-trees $\mathcal{T}_1$ and $\mathcal{T}_2$.

4.2.1.1 Factorized Trees

F-trees are a formalism to describe conditional dependencies between attributes of a relation R. An f-tree $\mathcal{T}$ can be used to describe the structure of an f-representation. As such, an f-representation of R follows the structure of $\mathcal{T}$. More broadly, an f-tree $\mathcal{T}$ is a rooted tree such that each node n_i of T is labelled by an attribute $v_i \in V$. The union of all vertices labelling the nodes in $\mathcal{T}$ is V. The shape of $\mathcal{T}$ provides a hierarchy of attributes by which we group the tuples of the represented relation: we group the tuples by the values of the attributes labelling the root, factor out the common values, and then continue recursively on each group using the attributes lower in the f-tree. For instance, the f-representation in Figure 4.2b is described by the f-tree T_1 in Figure 4.2a. At each node in T_1 except the root, a list of attribute dependencies on the

ancestors is shown in brackets. T_1 points to a grouping on a, *i.e.*, we group separately the sets of values of b and c for each a value. In general, an f-tree over attributes $A(a, b, ...)$ cannot factorize an arbitrary relation R over A. However, for relations that are results of a join query $Q(V, E)$, from the shape of Q, we can find f-trees that will factorize the outputs of Q. This is done by analyzing the dependencies between the attributes in Q. We introduce the notion of dependency in graph terms focusing on our context of conjunctive queries in graph workloads. A general definition for general queries are given in reference [145].

Definition 4.2.1 *(Attribute dependency) Two attributes a_i and a_j are dependent if $a_i \rightarrow a_j \in Q$ or $a_j \rightarrow a_i \in Q$. If there are no projections in the query, this is the only constraint. If there are projections in the output of the conjunctive query, then two a_i and a_j are also dependent if there is a path in the query that start with a_i and end with a_j, such that every attribute in the path is projected out, i.e., , not in the projection list of the query.*

Olteanu *et al.* [145] has shown that any f-tree that satisfies the following *path constraint* is an f-tree correctly factorizing the results of Q:

Definition 4.2.2 *(Path constraint) Given a query Q and an f-tree $\mathcal{T}$, $\mathcal{T}$ satisfies the path constraint if any two dependent attributes are on the same root-to-leaf path.*

For Q_{2H}, figures 4.2a and 4.2c show two f-trees, $\mathcal{T}_1$ and $\mathcal{T}_2$, that correctly factorizes Q_{2H}. Figures 4.2b and 4.2d show the corresponding f-representations when factorizing Q_{2H} using $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. Readers can verify that both f-trees satisfy the path constraint. As demonstrated in this simple example, a query results may be factorized under different f-trees, resulting in different sizes. In our example, $\mathcal{T}_1$ leads to a significantly more compact F-representation than $\mathcal{T}_2$. Observe also that $\mathcal{T}_2$ is effectively a flat representation and requires the same size as a flat representation.

4.2.1.2 Worst-case Size Bounds for F-representations

The AGM bound of a query Q over a database D, is denoted by $|D|^{\rho^*(Q)}$, where $|D|^{\rho^*(Q)}$ is the worst-case *i.e.*, maximum output size for a given query on any database instance. $|D|^{\rho^*(Q)}$ would be the worst-case size of the query results for flat representations. Olteanu *et al.* [145] define $|D|^{s(Q)}$ as the minimal worst-case output sizes for f-representations.

Definition 4.2.3 ***F-representation minimal worst-case output sizes -*** *Given Q, all f-trees of Q, $|D|^s(Q)$ is the size of the f-representation following f-tree T*, which is the one describing the f-representation with the lowest maximum output size across all database instances.*

We note that $|D|^{\rho^*(Q)}$ is the size of f-representations following f-trees that are paths in which each node has at most a single child. Furthermore, $s(Q)$ is usually smaller than $\rho^*(Q)$ such that $s(Q) \leq \rho^*(Q)$. We show this for the 2-hop query $Q_{2H} = R(a,b), R(b,c)$ as an example for possible input edge relations of size N. The AGM bound of Q_{2H} is N^2 hence $\rho^*(Q_{2H}) = 2$. Meanwhile the worst-case output size of the f-representations described by the f-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ in Figures 4.2a and 4.2c are $\Theta(N)$ and $\Theta(N^2)$, respectively. $\mathcal{T}_1$ and $\mathcal{T}_2$ are representative of all f-trees of Q_{2H} and therefore $s(Q_{2H}) = 1$. To summarize, for Q_{2H}, $\rho^*(Q) = 2$ and $s(Q) = 1$. As such, f-trees for the 2-hop query lead to representations with sizes of either N and N^2.

4.2.2 GraphflowDB Storage

GraphflowDB follows a columnar storage for its properties. The node/entity attributes are stored as in-memory columns while the edges/relationships and their attributes are stored in a compressed CSR structure that represents adjacency lists. These are indexed in the forward and backward direction. The details are covered by the work of Pranjal *et al.* [75]. We use what is referred to in the design as single-directional property pages which avoid duplicating properties on both edge directions. It achieves good locality when reading properties of edges in one direction and still guarantees random access in the other.

4.3 Mixing Factorization and Vectorized Execution

We next motivate factorized vector execution by discussing limitations of traditional Volcano-style tuple-at-a-time processors and existing vector-based processors of columnar RDBMSs when processing m-n joins. Consider the following query:
$Q_{FFP}(a,b,c,d) = P(a, age, -), F(a,b), F(b,c), P(c, -, d), age > 50$, where F and P are the $Follows(follower, followee)$ and $Person(ID, age, lives_at)$ relations, respectively. Assume that $Follows$ is an m-n relationship while $lives_at$ is an n-1 relationship.

Consider a simple plan for this query shown in Figure 4.3, which is a left-deep plan with JAO (a,b,c,d) made of `Scan`, `Filter`, and `INLJ` operators. Volcano-style tuple-at-a-time processing [68], which some GDBMSs adopt [117, 126], is efficient in terms of how much data is copied to the intermediate tuple. Suppose the scan matches a to a_1 and a_1 extends to k_1 many b's, b_1 ... b_{k1} , and each b_i extends to k_2 many c's. For now we ignore the d extension. This generates $k_1 \times k_2$ tuples. Depending on the implementation, the number of times values are copied can be optimized, *e.g.*, , the Volcano-style GraphflowDB processor from Chapter 3 would copy a_1 values only once, and then each of the b values once, and each of the c values

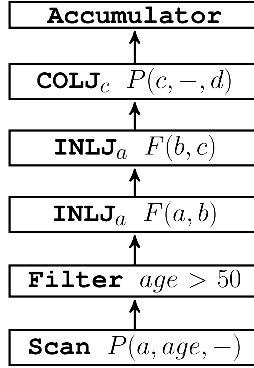


Figure 4.3: Left-deep plan example for Q_{FFP}.

k_2 times. However, the number of function calls to operators to process these tuples would be commensurate with $k_1 \times k_2$. For this reason, Volcano-style processors are known to achieve low CPU utilization as well as low cache utilization as processing is intermixed with many function calls. These shortcomings especially surface in Volcano-style processors when processing m-n joins.

Column-oriented RDBMSs instead adopt vector-based processors [33, 83], which process an entire block of vectors at a time in operators. Block sizes are fixed length, *e.g.*, 1024 tuples. While processing a block of vectors, operators read consecutive memory locations, achieving good cache locality, and perform computations inside loops over vectors which is efficient on modern CPUs. However, traditional vector-based processors have two shortcomings for graph workloads: (1) For m-n joins, vector-based processing requires often high memory-to-memory data copies. Suppose for simplicity a block size of k_2 and $k_1 < k_2$. In our example, the scan would output a vector $a : [a_1]$, the first join would output $a : [a_1, ..., a_1]$, $b : [b_1, ..., b_{k_1}]$ vectors, and the second join would output $a : [a_1, ..., a_1]$, $b : [b_1, ..., b_1]$, $c : [c_1, ..., c_{k_2}]$ vectors, where for example the value a_1 gets copied k_2 times into intermediate vectors[3] . (2) Traditional vector-based processors do not exploit the list-based data organization of adjacency lists. Specifically, adjacency lists that are used as join indexes by operators are already stored consecutively in memory, which can be exploited to avoid materializing and copying these lists into vectors.

We developed a new vectorized execution model we call factorized vector execution, which we next describe. We use factorized representation of intermediate tuples [145] to address the data copying problem and uses vectors with sizes set to the lengths of adjacency lists in the database up to a maximum, to exploit the list-based data storage in GraphflowDB.

[3]In some systems, such as DuckDB [152], instead of the a_1 value, some offset values into some other vector could be repeated.

4.3.1 Intermediate Tuple Set Representation

Traditional vectorized executors represent intermediate data as a set of *flat* tuples in a single group of multiple vectors. In our example within the first three extensions, we had three variables a, b, and c corresponding to three vectors. The values at position i of all vectors form a single tuple. Therefore to represent the tuples that are produced by m-n joins, repetitions of values are necessary. To address these repetitions we adopt a *factorized tuple set representation* scheme [145]. Instead of flat tuples, factorized representation systems represent tuples as unions of Cartesian products. For example, the k_2 flat tuples $[(a_1, b_1, c_1) \cup (a_1, b_1, c_2) \cup ... \cup (a_1, b_1, c_{k_2})]$ from above can be represented more succinctly in a factorized form as: $[(a_1) \times (b_1) \times (c_1 \cup ... \cup c_{k_2})]$.

To adopt factorization in a vectorized executor, we instead use multiple groups of vectors, which we call *vector groups*, to represent intermediate data. Each vector group has a `pos` field making it in one of two states:

- **Flat**: If `pos` ≥ 0, the vector group is *flattened* and represents a single tuple that consists of the `pos`'th values in the vectors.
- **Unflat**: If `pos` $=$ -1, the vector groups represent a list of tuples with as many as the size of element in each vector as vectors within the same group have the same size.

We call the union of vector groups an *intermediate chunk*, which represents a set of intermediate tuples as the Cartesian product of each tuple that each vector group represents.

In addition, we are align to the lengths of the vectors to those of the adjacency lists in the database. As we shortly explain when introducing the changes to our operators, this allows us to avoid materializing adjacency lists and copy them into the vectors.

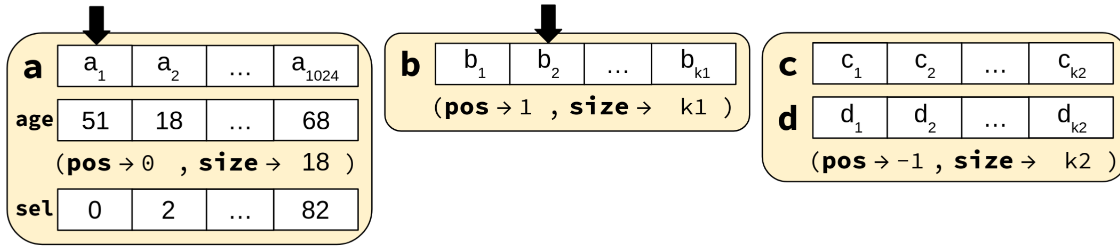
Example:

Figure 4.4a shows an intermediate chunk, that consists of three vector groups. The first two groups are flattened and the last is unflat. In its current state, the intermediate chunk represents k_2 intermediate tuples of Q as: $(a_1, 51) \times (b_2) \times ((c_1, d_1) \cup ... \cup (c_{k2}, d_{k2}))$. The equivalent logical intermediate relation stored in the intermediate tuples is shown in Figure 4.4b.

4.3.2 Execution Engines and Operators

We next give a description of the main relational operator changes to process intermediate chunks in a our factorized vector execution.

`Scan:` Scans are the same as before and read a fixed size of IDs; 1024 by default. IDs are added into a vector within a new vector. Since we read consecutive IDs in a fixed size chunks, we



(a) Example of an intermediate chunk for Q_{FFP}.

a	age	b	c	d
a_1	51	b_2	c_1	d_1
a_1	51	b_2	c_2	d_2
...	...	...	...	...
a_1	51	b_2	c_{k2}	d_{k2}

(b) Equivalent logical intermediate relation stored in the intermediate chunk.

Figure 4.4: Example of intermediate chunk and its equivalent logical relation for Q_{FFP}. The first two vector groups are flattened to single tuples, while the last represents k_2 many tuples.

use a special vector group implementation that also contains a `startOffset` value. When the operator starts executing, initially it pushes a vector with `startOffset` set to 0 and `size` set to 1024, then it pushes the same vector with `startOffset` set to 1024 and so on and so forth. This leads to avoiding a lot of unnecessary memory writes within the vector. Any other attributes of interest are scanned and added as separate vectors to the same vector group.

`INLJ/IMJ` : are used to perform joins from a node, say, a to nodes b over 1-n or m-n edges e. The input vector group VG_a that holds the vector of a values can be flat or unflat. If VG_a is not flat, `INLJ/IMJ` first flattens it, *i.e.*, sets the `pos` field of the vector group to 0. It then loops through each a value, say, a_ℓ, and extends it to the set of b and e values using a_ℓ's adjacency list Adj_{a_ℓ}. The vectors holding b and e values are put in a new vector group, VG_b. This allows factoring out a vector of b and e values for a single a value. The lengths of all vectors in VG_b, including those storing b and e as well as vectors that may be added later, will be equal to the length of Adj_{a_ℓ}. In addition, we exploit that Adj_{a_ℓ} already stores b and e values as vectors, and do not copy these to the intermediate chunk. Instead, the b and e vectors simply point to Adj_{a_ℓ}.

`Column Join` (**`COLJ`**): is used to perform 1-1 or n-1 joins. Suppose now that each a can extend to at most one b nodes. `COLJ` expects a vector of unflat a values, though they can be flat due to prior operators. That is, it expects VG_a to be unflat and adds a new vector into VG_a to store b.

The vector is the same length as a's vector (so unlike COLJ does not create a new vector group). Inside a for loop, COLJ copies the matching b of each a to a vector. Note that because each a value has a single b value, these values do not need to be factored out.

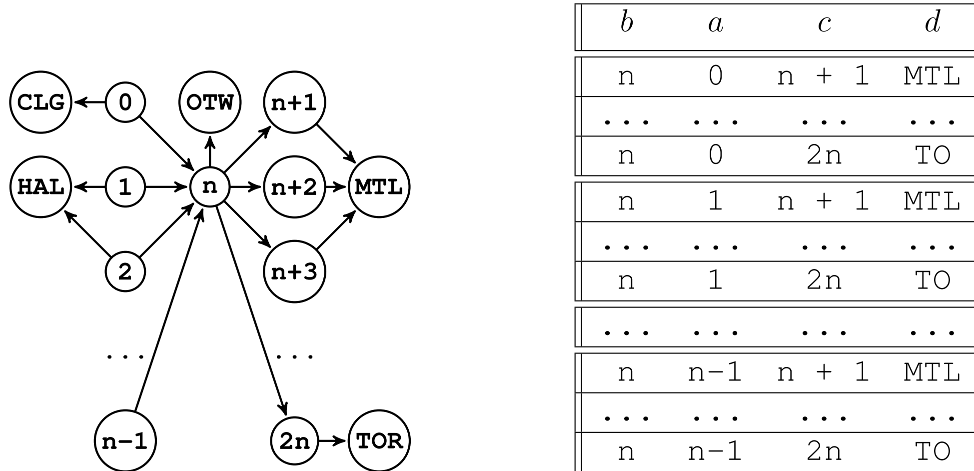
Hash Join (HJ): Our HJ stores in its Hash Table all of the values from flattened vector groups similar to how tuples were stored in a regular Hash Table implementation. Furthermore, it stores lists as if they were variable-length data types instead of flattening them. When probing, all flattened values are added to the same group with a single tuple and all lists are stored in different vector groups, *i.e.*, we keep the factorized representation across

Filter: We require a more complex filter operator than those in traditional vector-based processors. In particular, in traditional vector-based processors, binary expressions, such as a comparison expression, can always assume that their inputs are two vectors of values. Instead, now binary expressions need to operate on three possible value combinations: two flat, two lists or one list and one flat, because any of the two vectors can now be in a flattened vector group. The filter adds a selection array containing the position of the elements that passed the filter such as the one in the first vector group in Figure 4.4a.

Group By And Aggregate: Briefly, similar to Filter, Group By And Aggregate needs to consider whether the values it should group by or aggregate are flat or not, and performs a group by and aggregation on possibly multiple factorized tuples. Factorization allows us to sometimes perform fast group by and aggregations, similar to prior techniques that compute aggregations on compressed data [9, 176]. For example, count(*) simply multiplies the sizes of each vector group to compute the number of tuples represented by each intermediate chunk it receives.

Example:

Continuing our example, the three vector groups in Figure 4.4 are an example intermediate chunk output by the COLJ operator in the plan from Figure 4.3. In this, the initial Scan and Filter have filled the 1024-size a and age vectors in VG_1. The first INLJ has: (i) flattened VG_1 to tuple $(a_1, 51)$; and (ii) filled a vector of k_1 b values in a new vector group VG_2. The second INLJ has (i) flattened VG_2 and iterated over it once, so its pos field is 1, and VG_2 now represents the tuple (b_2); and (ii) has filled a vector of k_2 c values in a new vector group VG_3. Finally, the last COLJ fills a vector of k_2 d values, also in VG_3, by extending each c_j value to one d_j value through the join with $Person$ to obtain the d: $lives_at$ attributes.



b	a	c	d
n	0	n + 1	MTL
...	...	...	...
n	0	2n	TO
n	1	n + 1	MTL
...	...	...	...
n	1	2n	TO
...	...	...	...
n	n-1	n + 1	MTL
...	...	...	...
n	n-1	2n	TO

(a) Input graph with $follows$ and $lives_at$ edges. (b) Output as a table of $\mathtt{Q}_{2P_C}$.

Figure 4.5: Input dataset and output example of Q_{FFP2}.

4.3.3 Benefits and Limitations

We show the benefits and limitations of our approach with an example. Consider the input data in Figure 4.5a and the following query: $Q_{PFFP}(a,b,c,d) = P(b,-,k)$,$F(a,b)$,$F(b,c)$, $P(c,-,d)$, where F and P are the $Follows(follower, followee)$ and $Person(ID, age, lives_at)$ relations, respectively. We consider the left deep plan with JAO (b,a,c,d) evaluating Q_{PFFP} shown in Figure 4.6. The plan contains `Scan`, `INLJ`, and `COLJ` operators.

The output has three vector groups in Figure 4.7a are an example intermediate chunk output by the `COLJ` operator in the plan from Figure 4.6. In this example, we showcase the internals of the vector implementations. * refers to a pointer to an array instead of a copy. In this, the initial `Scan` have filled the 1024-size b and k vectors in VG_1. The first `INLJ` has: (i) flattened VG_1 to tuple (n, OTW); and (ii) filled a vector of n a values in a new vector group VG_2. The second `INLJ` (i) gets a flat VG_1 and filled a vector of n c values in a new vector group. Finally, the last `COLJ` fills a vector of n d values, also in VG_3, by extending each c_j value to one d_j value through the join with $Person$ to obtain the d: $lives_at$ attributes. Note that the Cartesian product of these tuples would produce n^2 tuples meanwhile the intermediate chunk stores it as $2n + 1$ fields. Note that the intermediate chunk if stored within a `Hash Table` would be stored as shown in the Figure 4.7b.

The benefits of factorized vector execution are: a) completely avoiding the enumeration of intermediate results when plans have heavy projections and aggregations; b) even when results

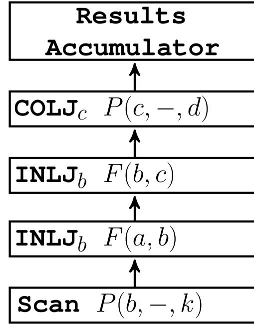
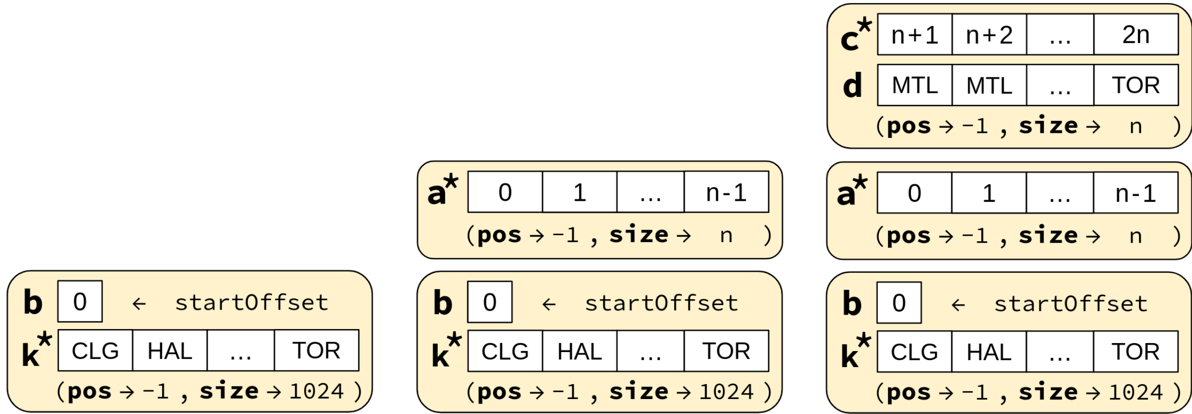
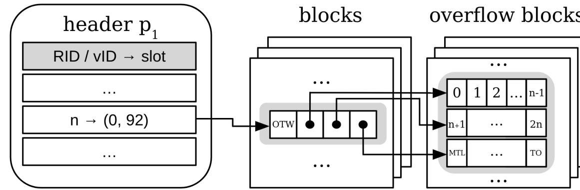


Figure 4.6: Left-deep plan example for Q_{FFP2}.



(a) Example of vector groups changing as they evaluate $\mathtt{Q}_{PFFP}$.



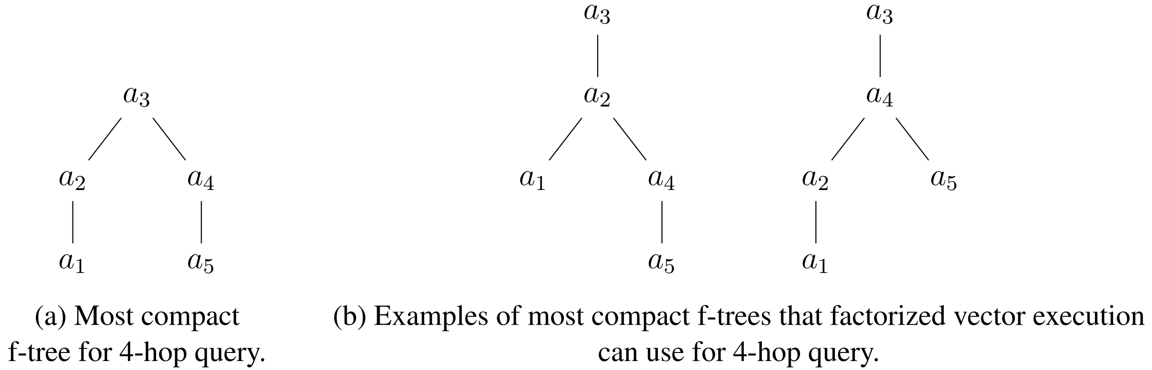
(b) Example of `Hash Table` storing the intermediate chunk.

Figure 4.7: Example of Intermediate Chunk for $\mathtt{Q}_{2P_C}$.

are enumerated at the end with no asymptotic difference on the output size, redundant computation is avoided. An example is the final join with $P(c,, d)$ in Q_{FFP2}. The join would have been evaluated n times if we apply the Cartesian product of traditional systems to the a vector turning it into a FLAT vector unnecessarily; c) minimizing further the overhead of interpretation; and d) leading in some cases to smaller `HashTable`(s).

The integration of factorized vector execution does not require major changes to a DBMS with a vectorized execution model. In order to implement factorized vector executions, the operators have to be configured to decide if the Cartesian product is necessary and the cost model in the optimizer has to take the factorized format into account.

The major limitation however, is that factorized vector execution has a limited form of factorization making it impossible to obtain certain smaller f-representations. Generally, the only allowed representations are ones following f-trees where each node has at most 1 non-leaf non child node. Consider for example the 4-hop query $R(a_1, a_2), R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$. The best possible representation would follow the f-tree in Figure 4.8a with a worst-case size N^2. However since our factorized vector executor operates on vectors following the structure described, we only obtain factorizations of the form in Figure 4.8b with a worst-case size N^3.



(a) Most compact f-tree for 4-hop query.

(b) Examples of most compact f-trees that factorized vector execution can use for 4-hop query.

Figure 4.8: F-trees supported by factorized vector execution.

4.4 Evaluation

In this section, we demonstrate the effectiveness of our factorized vector execution model. We refer to our GraphflowDB with factorized vector execution as `GF-F`. We refer to the older version from Chapter 3 with a Volcano execution model is `GF-V`. To ensure that our experiments only test differences due to query processing techniques, the same columnar storage and compression techniques introduced by Gupta *et al.* [75] are integrated into `GF-F` and `GF-V`. We present microbenchmark experiments comparing `GF-V` and `GF-F` and baseline experiments against `Neo4j`, `MonetDB`, and `Vertica` using the *Social Network Benchmark* SNB by the *Linked Data Benchmark Council* (LDBC) [55] and the *Join Order Benchmark* (JOB) [108]. Before presenting the experiments, we describe the hardware and datasets.

4.4.1 Setup

Hardware: Similar to the prior chapter, we use a single machine that has two Intel E5-2670 @2.6GHz CPUs and 512 GB of RAM. All code runs on openjdk-17. We set the maximum size of the JVM heap to 500 GB and keep the minimum heap size of the JVM as is. We use only one physical core.

Domain	Name	Nodes	Edges
Social	LDBC10	30M	176.6M
	LDBC100	300M	1.77B
	Flickr	2.3M	33.1M
Wiki	German	2.1M	86.3M

Table 4.1: Datasets used for factorized vector execution evaluation.

Datasets: The datasets we use are summarized in Table 4.1. Our factorized execution model is designed to yield benefits under join queries over 1-n and m-n relationships. The technique is not designed for datasets that do not depict structure, *e.g.*, a highly heterogeneous knowledge graph, such as DBPedia. As such, we choose the following datasets and queries:

- *LDBC*: We generated the LDBC social network data [55] using scale factors 10 and 100, which we refer to as LDBC10 and LDBC100, respectively. In LDBC, all of the edges as well as edge and vertex attributes are structured but several attributes and edges are very sparse. LDBC10 contains 30M vertices and 176.6M edges while LDBC100 contains 1.77B edges and 300M vertices. Both datasets contain 8 entity tables, 15 relationship tables and 34 attributes (29 for entities, 5 for relationships).
- *JOB*: We used the IMDb movie database and the JOB benchmark [108]. Although the workload has originally been created to study optimizing join order selection, the dataset contains several m-n, 1-n, and 1-1 relationships between entities, like actors, movies, and companies, and structured attributes, some of which are `NULL`. JOB contains join queries over m-n relationships, making it suitable to demonstrate benefits of factorized vector execution. We created a property graph version of this database and workload for GraphflowDB as follows. IMDb contains three groups of tables: (i) *entity tables* representing entities, such as actors (*e.g.*, `name` table), movies, and companies; (ii) *relationship tables* representing n-n relationships between the entities (*e.g.*, the `movie_companies` table represents relationships between movies and companies); and (iii) *type tables*, which denormalize the entity or relationship tables to indicate the types of entities or relationships. We converted each row of an entity table to a vertex. Let u and v be vertices representing, respectively, rows r_u and r_v

from tables T_u an T_v. We added two sets of edges between u and v: (i) a *foreign key edge* from u to v if the primary key of row r_u is a foreign key in row r_v; (ii) a *relationship edge* between u to v if a row r_ℓ in a relationship table T_ℓ connects row r_u and r_v. The final dataset can be found on Github.[4]

- *Flickr* and *Wiki*: To enhance our microbenchmarks further, we use two additional datasets from the popular Konect graph sets [100] covering two application domains: a Flickr social network [121] and a Wikipedia hyperlink graph between articles of the German Wikipedia (`WIKI`) [101]. Flickr has 2.3M nodes and 33.1M edges while Wikipedia has 2.1M nodes and 86.3M edges. Both datasets have timestamps as relationship attributes.

Execution: In each experiment, we ran our queries 5 times consecutively and report the average of the last 3 runs. We did not observe large variances in these experiments. Across all of the LDBC and JOB benchmark queries we report, the median difference between the minimum and maximum of the last 3 runs was 1.02% and the largest was 25%, which was a query in which the maximum run was 24ms while the minimum was 19ms.

4.4.2 Microbenchmarks

We next present experiments demonstrating the performance benefits of factorized vector execution against our prior traditional push-based Volcano-like processor. Tuple-at-a-time processing like Volcano's is adopted in existing systems, like `Neo4j` [126] and `MemGraph` [117]. Factorized vector execution has three advantages over traditional tuple-at-a-time processors: (1) all primitive computations over data, *i.e.*, physical operators within pipelines and expression evaluation, happen in tight loops similar to vector execution engines; (2) the join operators can avoid copies of edge ID-neighbour ID pairs into intermediate tuples, exploiting the list-based storage; and (3) we can perform group-by and aggregation operations directly on compressed data. We present two separate sets of experiments that demonstrate the benefits from these three factors.

We use the LDBC100, Wiki, and Flickr datasets. In our first experiment, we take 3 queries over a relationship table $R(from, to, p)$ where p is a relationship attribute (*e.g.*, date for LDBC): i) $R(a, b, p)$ (1-hop); ii) $R(a, b, -), R(b, c, p)$ (2-hop); and iii) $R(a, b, -), R(b, c, -), R(c, d, p)$ (3-hop). Each query as a predicate $p > c$ where c is a constant. For both `GF-V` and `GF-F`, we consider the standard plan that scans the left most node a, extends right to match the entire path, and a final `Filter` operator on the p property of the last extended edge. A major part of the work in these plans happen at the final join and filter operation, therefore these plans allow us to measure the performance benefits of performing computations inside tight loops and avoiding

[4] `github.com/queryproc/columnar-storage-and-list-based-processing-for-graph-dbms`

			1-hop	**2-hop**	**3-hop**
LDBC100	FILTER	GF-V	24.6	1470.5	40252.4
		GF-F	7.7 **3.2x**	116.2 **12.7x**	2647.3 **15.2x**
	COUNT(*)	GF-V	13.4	241.9	6947.3
		GF-F	4.2 **3.2x**	18.9 **12.8x**	357.9 **19.4x**
FLICKR	FILTER	GF-V	32.6	1300.0	14864.0
		GF-F	12.2 **2.7x**	95.3 **13.7x**	1194.7 **12.4x**
	COUNT(*)	GF-V	35.3	519.2	4162.5
		GF-F	16.9 **2.1x**	23.4 **21.4x**	51.7 **80.6x**
WIKI	FILTER	GF-V	35.8	4500.2	236930.2
		GF-F	11.9 **2.9x**	1192.5 **3.8x**	20329.3 **11.7x**
	COUNT(*)	GF-V	32.7	1745.2	109000.2
		GF-F	19.0 **1.7x**	27.6 **63.2x**	120.4 **905.1x**

Table 4.2: Run time in ms of GF-V and GF-F plans.

data copying in joins. Our results are shown in the FILTER rows of Table 4.2. We see that GF-F outperforms GF-CV by large margins, between 2.7x and 15.2x.

In our second experiment, we demonstrate the benefits of performing fast aggregations over compressed intermediate results. We modify the previous queries by removing the predicate and instead add a return value of COUNT(*). We use the same plans as before except we change the last Filter operator with a GroupBy operator. Our results are shown in the COUNT(*) rows of Table 4.2. Observe that the improvements are much more significant now, up to close to three orders of magnitude on Wiki (by 905.1x). The primary advantage of GF-F is now that the counting happens on compressed intermediate results.

4.4.3 Baseline System Comparisons

In our second experiment, we compare the query performance of GF-F against GF-V, Neo4j, which is a row-oriented and Volcano style GDBMS, and two columnar analytical RDBMSs,

MonetDB and Vertica, which are not tailored for m-n joins. Our primary goal is to verify that GF−F is faster than GF−V also on an independent end-to-end benchmark. We also aim to verify that GF−V on which we base our work is already competitive with or outperforms other baseline systems on workloads containing m-n joins. We used the SNB on LDBC10 and JOB, both of which contain m-n join queries.

We used the community version v4.2 of Neo4j GDBMS [126], the community version 10.0 of Vertica [181] and MonetDB 5 server 11.37.11 [124]. We note that our experiments should not be interpreted as one system being more efficient than another. It is difficult to meaningfully compare completely separate systems, *e.g.*, all baseline systems have many tunable parameters, and some have more efficient enterprise versions. For all baseline systems, we map their storage to an in-memory filesystem, set number of CPUs to 1 and disable spilling intermediate files to disk. We maintain 2 copies of edge tables for Vertica and MonetDB, sorted by the source and destination vertex IDs, respectively. This gives the systems the option to add further plans that can perform fast merge joins without sorting for example. For GF−V and GF−F, we use the best left-deep plan we could manually pick, which was obvious in most cases. For example, LDBC path queries start from a particular vertex ID, so the best join orders start from that vertex and iteratively extend in the same direction. For Vertica, MonetDB, and Neo4j, we use the better of two plans: i) the systems' default plans; and ii) the left-deep that is equivalent to the one we use in GF−V and GF−F.

4.4.3.1 LDBC

We use the LDBC10 dataset. GraphflowDB implements parts of the Cypher language, so lacks several features that LDBC queries exercise. The system has support for select-project-join queries and a limited form of aggregations, where joins are expressed as fixed-length subgraph patterns in the MATCH clause. We modified the Interactive Complex Reads (IC) and Interactive Short Reads (IS) queries from LDBC [55] in order to be able to run them. Specifically GraphflowDB does not support variable length queries that search for joins between a minimum and maximum length, which we set to the maximum length to make them fixed-length instead, and shortest path queries, which we removed from the benchmark. We also removed predicates that check the existence or non-existence of edges between nodes and the ORDER BY clauses. Our final workload contains variants of 18 of the 21 queries in the IS and IC benchmarks and are provided as part of the open-source codebase [71].

Figure 4.9a shows the relative speedup/slowdown of the different systems in comparison to GF−V. Tables 4.3a and 4.3b, show the individual run time numbers of each IS and IC query, respectively. As expected, GF−F is broadly more performant than GF−V on LDBC with a median query improvement factor of 2.6x. With the exception of one query, which slows a bit, the

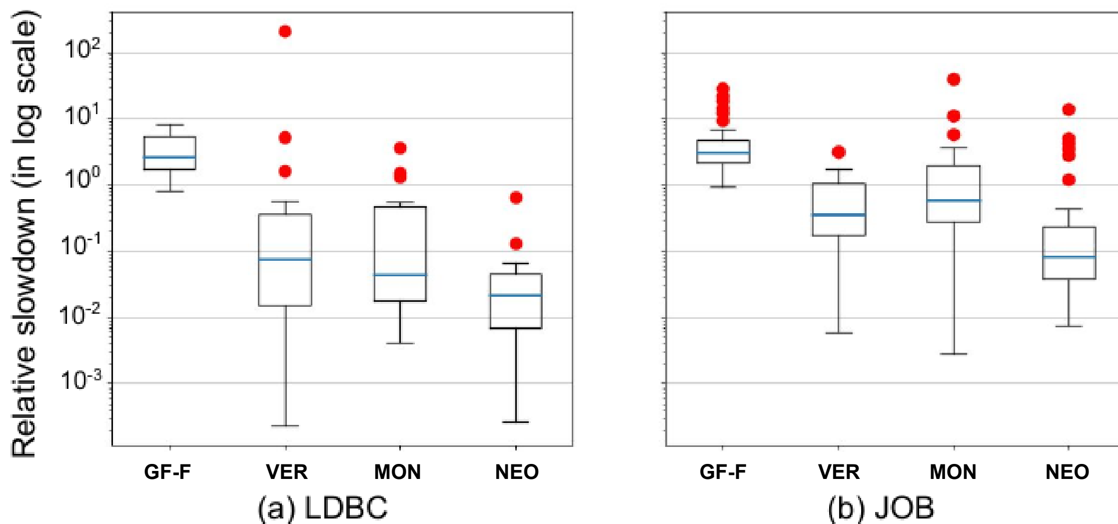


Figure 4.9: Relative speedup/slowdown of the different systems in comparison to `GF-V` on LDBC10. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.

performance of *every query* improves between 1.3x to 8.3x. We observed large improvements on queries that produce large intermediate results and perform filters, such as IC05. On IC05, `GF-V` took 8.9s while `GF-F` took 1.6s. IC05 has 4 m-n joins starting from a node and extending in the forward direction and a predicate on the edges of the third join. `GF-F` has several advantages that become visible here. First, `GF-F`'s factorized vector execution does not copy any edge and neighbour IDs to intermediate tuples and performs filters in tight-loops when compared with `GF-V`.

As we expected, we also found other baseline systems to not be as performant as `GF-F` or `GF-V`. In particular `Vertica`, `MonetDB`, and `Neo4j` have median slowdown factors of 13.1x, 22.8x, and 46.1x, respectively, when compared to `GF-V`. Although `Neo4j` performed slightly worse than other baselines, we also observed that there were some queries in which it outperformed `Vertica` and `MonetDB` (but not `GF-V` or `GF-F`) by a large margin. These were queries that started from a single node, had several m-n joins, but did not generate large intermediate results, like IS02 or IC06. On such queries, GDBMSs, both GraphflowDB and `Neo4j`, have the advantage of using join operators that use the adjacency list indices to extend a set of partial matches. This can be highly efficient if the partial matches that are extended are small in number. For example the first join of IC06 extends a single `Person` node, say p_i, to its two-degree friends. In SQL, this is implemented as joining a `Person` table with a `Knows`

table with a predicate on the Person table to select p_i. In Vertica or MonetDB this join is performed using merge or hash joins, which requires scanning both Person and Knows tables. Instead, Neo4j and GraphflowDB only scan the Person table to find p_i and then extend p_i to its neighbours, without scanning all Knows edges. For this, GF-V, GF-F, and Neo4j took 333ms, 113ms, and 515ms, while Vertica and MonetDB take 4.7s and 2.7s, respectively. We also found that all baseline systems, including Neo4j, degrade in performance on queries with many m-n joins that generate large intermediate results. For example, on IC05 that we reviewed above, Vertica took 1 minute, MonetDB took 3.25 minutes, while Neo4j took over 10 minutes.

4.4.3.2 JOB

JOB queries come in four variants and we used their first variant (a). We converted the JOB queries to their Cypher equivalent following our conversion of the dataset. Many of the JOB queries returned aggregations on strings, such as min(name), where name is a string column. Since GraphflowDB supports aggregations only on numeric types, we removed these aggregations. Our final queries are provided as part of the open-source codebase [71].

Figure 4.9b shows the relative performance of different systems in comparison to GF-V. Table 4.3b shows individual run time numbers of each query. Similar to our LDBC results, we see GF-F to improve the performance, now by 3.1x. Again similar to LDBC, with the exception of one query, we see consistent speed ups across all queries between 1.5x and 28.8x. Different from LDBC, we also see queries on which the improvement factors are much larger, *i.e.*, ¿20x. In LDBC, the largest improvement factor was 8.3x. This is expected as most of the queries in JOB perform star joins while LDBC queries contained path queries that start from a node with a selective filter. On path queries, our plans start from a single node and extend in one direction, in which case only the last extension can truly be factorized, so be in unflat form. This is because each *i.e.*, operator that we use first flattens the previously extended node. Whereas on star queries, multiple extensions from the center node can remain unflattened. Therefore GF-F's plans can benefit more from factorized vector execution as they can compress their intermediate tuples more. We also see that similar to LDBC, GF-V is more performant than the columnar RDBMSs. However, these systems are now more competitive. We noticed that one reason for this is that on star queries, these systems's default plans are often bushy plans (27 out of 33 for MonetDB and 26 out of 33 for Vertica), which produce fewer intermediate tuples than GF-V, which does not benefit from factorization and uses left-deep plans. So these systems now benefit from bushy plans which they did not in LDBC. In contrast, on LDBC, these systems would also primarily use left-deep plans (only 2 out of 18 for MonetDB and 4 out of 18 for Vertica were bushy) because on these path queries, it is better to start from a single highly filtered node table and join iteratively in a left-deep plan to match the entire path. Finally, similar to LDBC, Neo4j is again least competitive of these baselines.

	IS01	**IS02**	**IS03**	**IS04**	**IS05**	**IS06**	**IS07**
`GF-F`	2.7	3.0	2.2	36.9	40.6	69.3	38.3
`GF-V`	2.2 **0.8x**	3.9 **1.3x**	3.9 **1.8x**	307.3 **8.3**	236.6 **5.8x**	423.0 **6.1x**	307.9 **8.0x**
`VER`	6.1 **2.2x**	16728.2 **5574.2x**	7.0 **3.2x**	1.5 **0.04x**	45.2 **1.1x**	259.2 **3.7x**	24818.9 **647.7x**
`MON`	112.3 **40.9x**	282.2 **94.0x**	8.3 **3.8x**	84.1 **2.3x**	516.4 **12.7x**	323.0 **4.7x**	206.3 **5.4x**
`NEO`	103.1 **37.5x**	117.4 **39.1x**	86.1 **39.1**	12418.9 **336.6**	11665.9 **287.4**	67390.3 **972.3**	12095.2 **315.6**

(a) Interactive Short Queries.

	IC01	**IC02**	**IC03**	**IC04**	**IC05**	**IC06**
`GF-F`	36.7	32.4	409.4	13.1	1565.2	113.0
`GF-V`	88.4 **2.4x**	45.2 **1.4x**	1521.8 **3.7**	57.3 **4.4x**	8925.0 **5.7x**	333.1 **3.0x**
`VER`	257.2 **7.0x**	3063.8 **94.5x**	18610.3 **45.5x**	1711.6 **130.5x**	59351.0 **37.9x**	4715.7 **41.7x**
`MON`	160.3 **4.4x**	323.2 **10.0x**	187330.9 **457.6x**	13955.1 **1064.3x**	165273.0 **105.6x**	2783.1 **24.6x**
`NEO`	669.3 **18.3x**	170722.9 **5264.2x**	86231.9 **210.6x**	75254.9 **5739.4x**	*TLE* **-**	515.4 **4.6x**
	IC07	**IC08**	**IC09**	**IC11**	**IC12**	
`GF-F`	3.0	2.6	1519.8	11.1	34.2	
`GF-V`	6.3 **2.1x**	7.0 **2.7x**	2098.1 **1.4x**	19.2 **1.7x**	84.9 **2.5x**	
`VER`	4092.2 **1348.8x**	2837.2 **1094.2x**	17276.2 **11.4x**	672.9 **60.9x**	5028.1 **147.1x**	
`MON`	206.3 **68.0x**	920.5 **354.8x**	121943.2 **80.0x**	572.0 **51.7x**	3251.9 **95.1x**	
`NEO`	95.6 **31.5x**	108.3 **41.8x**	219425.5 **144.4x**	2804.1 **253.6x**	34043.0 **996.0x**	

(b) Interactive Complex Queries.

Table 4.3: Run time in ms for running the LDBC Queries on 5 systems: (i) `GF-F`; (ii) `GF-V`; (iii) `VER` for `VERTICA`; (iv) `MON` for `MONET`; and (v) `NEO` for `NEO4J`.

	1.a	**2.a**	**3.a**	**4.a**	**5.a**	**6.a**	**7.a**	**8.a**	**9.a**	**10.a**	**11.a**
GF-F	13.5	59.7	41.4	17.4	124.8	9.8	53.4	298.3	209.7	51.3	23.6
GF-V	50.3 **3.7x**	120.4 **2.0x**	38.0 **0.9x**	29.4 **1.7x**	460.9 **3.7x**	91.4 **9.3x**	77.5 **1.5x**	1245.6 **4.2x**	560.2 **2.7x**	110.4 **2.2x**	487.3 **2.1x**
VER	214.8 **15.9x**	329.8 **5.5x**	3928.3 **94.9x**	798.1 **45.9x**	2591.4 **20.8x**	495.2 **50.6x**	72.8 **1.4x**	1166.2 **3.9x**	442.5 **2.1x**	395.4 **7.7x**	136.9 **5.8x**
MON	36.2 **2.7x**	33.0 **0.6x**	36.2 **0.9x**	120.2 **1.3x**	232.5 **1.9x**	1428.1 **145.9x**	133.6 **2.5x**	112.5 **0.4x**	282.5 **1.4x**	304.1 **5.9x**	92.8 **3.9x**
NEO	3077.7 **227.5x**	895.3 **15.0x**	774.3 **18.7x**	203.4 **11.7**	10727.2 **85.9x**	206.7 **21.1x**	6497.6 **121.8x**	3451.7 **11.6x**	14946.4 **71.3x**	1480.7 **28.9x**	2332.6 **98.8x**

	12.a	**13.a**	**14.a**	**15.a**	**16.a**	**17.a**	**18.a**	**19.a**	**20.a**	**21.a**	**22.a**
GF-F	58.3	70.0	14.6	362.5	15.0	268.5	548.1	207.8	12.8	13.2	28.6
GF-V	253.7 **4.4x**	406.9 **5.8x**	33.3 **2.3x**	6772.3 **18.7x**	34.0 **2.3x**	594.6 **2.2x**	1700.9 **3.1x**	983.0 **4.7x**	208.5 **1.6x**	22.6 **1.7x**	64.4 **2.3x**
VER	870.2 **2.4x**	286.3 **4.1x**	28.2 **1.9x**	2100.5 **5.8x**	1028.8 **68.5x**	2538.5 **9.5x**	1686.0 **3.1x**	4777.2 **23.0x**	982.5 **76.7x**	34.0 **2.6x**	99.0 **3.5x**
MON	56.7 **1.0x**	1148.2 **16.4x**	83.4 **5.7x**	172.0 **0.5x**	224.5 **14.9x**	1304.3 **4.9x**	868.2 **1.6x**	644.0 **3.1x**	7552.3 **590.0x**	60.7 **4.6x**	140.2 **4.9x**
NEO	5079.1 **87.2x**	93.8 **1.3x**	291.9 **20.1x**	2437.4 **6.7x**	4526.6 **301.2x**	167.6 **0.6x**	1414.8 **2.6x**	12047.2 **58.0x**	1849.0 **144.5x**	272.4 **20.7x**	317.8 **11.1x**

	23.a	**24.a**	**25.a**	**26.a**	**27.a**	**28.a**	**29.a**	**30.a**	**31.a**	**32.a**	**33.a**
GF-F	14.5	10.8	107.7	10.5	10.3	26.1	5.6	18.3	112.5	10.0	52.3
GF-V	407.8 **28.8x**	47.9 **4.4x**	1527.8 **14.2x**	19.9 **1.9x**	125.3 **12.2x**	56.7 **2.2x**	18.5 **3.1**	52.7 **2.9x**	775.9 **6.9x**	24.1 **2.4x**	201.6 **3.9x**
VER	698.5 **49.4x**	518.1 **47.8x**	496.2 **4.6x**	1239.9 **118.7x**	231.0 **22.5x**	197.3 **7.6x**	3153.1 **529.9x**	152.6 **8.5x**	2696.3 **23.9x**	193.6 **19.3x**	125.3 **2.4x**
MON	124.2 **8.8x**	993.9 **91.8x**	784.8 **7.3x**	1736.1 **166.1x**	75.9 **7.4x**	323.8 **12.4x**	1012.3 **170.1x**	1940.2 **107.5x**	848.1 **7.5x**	87.7 **8.8x**	88.1 **1.7x**
NEO	2497.1 **176.5x**	3505.4 **323.7x**	108.6 **1.0x**	694.1 **66.3x**	1276.7 **124.4x**	1573.7 **60.3x**	648.2 **108.9x**	326.1 **18.1x**	152.7 **1.4x**	364.1 **36.4x**	2723.8 **52.0x**

Table 4.4: Run time in ms for running the JOB Benchmark on 5 systems: (i) GF-F; (ii) GF-V; (iii) VER for VERTICA; (iv) MON for MONET; and (v) NEO for NEO4J.

Chapter 5

Caching and Reuse of Intermediate Results

F-representations are the relation representation scheme adopted by the factorized vector executor approach from Chapter 4. This representation can still contain redundancy on many queries for intermediate and output relations. To demonstrate this, consider the 4-hop query $Q_{4H} = R(a_1, a_2), R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$. Consider further running Q_{4H} on the input graph in Figure 5.1a on a query processor capable of producing output following the f-tree shown in Figure 5.1b. The output f-representation is shown in Figure 5.1c. Observe that the two blue boxes shown in Figure 5.1c are exactly the same sub-relation. This sub-relation represents all of the $a_1 \leftarrow a_2$ 1-paths in the input graph that have $a_2 = v_2$ binding. Similarly, the red boxes in the figure are exactly the same sub-relation representing the $a_4 \rightarrow a_5$ 1-paths in the input graph that have $a_5 = v_4$ binding. In fact, omitted from the f-representation in the figure are $k - 2$ other branches where these same sub-relations are repeated. The reason for these repetitions is that regardless of whether a_3 binds to v_{3_1}, v_{3_2},..., or v_{3_k}, the bindings generated for a_2 and a_4 are the same and are $a_2 = v_2$ and $a_4 = v_4$, respectively.

D-representations, *i.e.*, f-representations extended with *definitions*, capture such repetitions to provide further factorization benefits over f-representations. A definition is a reusable subexpression stored as the "actual values" or as a pointer. Figure 5.1d shows a d-representation that represents the output of Q_{4H} in a more succinct way than the f-representation shown in Figure 5.1c. We use the same nodes in dashed boxes to indicate a single shared sub-expression. The sub-relations rooted in $a_2 = v_2$ and $a_4 = v_4$ are stored once as values, *i.e.*, cached, and reused as indicated with the pointers. The theory of factorization shows that worst-case size of d-representations can be polynomially smaller than the worst-case size of f-representations, which as discussed in Chapter 4 can be polynomially smaller than the AGM bound, *i.e.*, worst-case size of flat representations [145]. These worst-case sizes will be made more formal in Section 5.2.

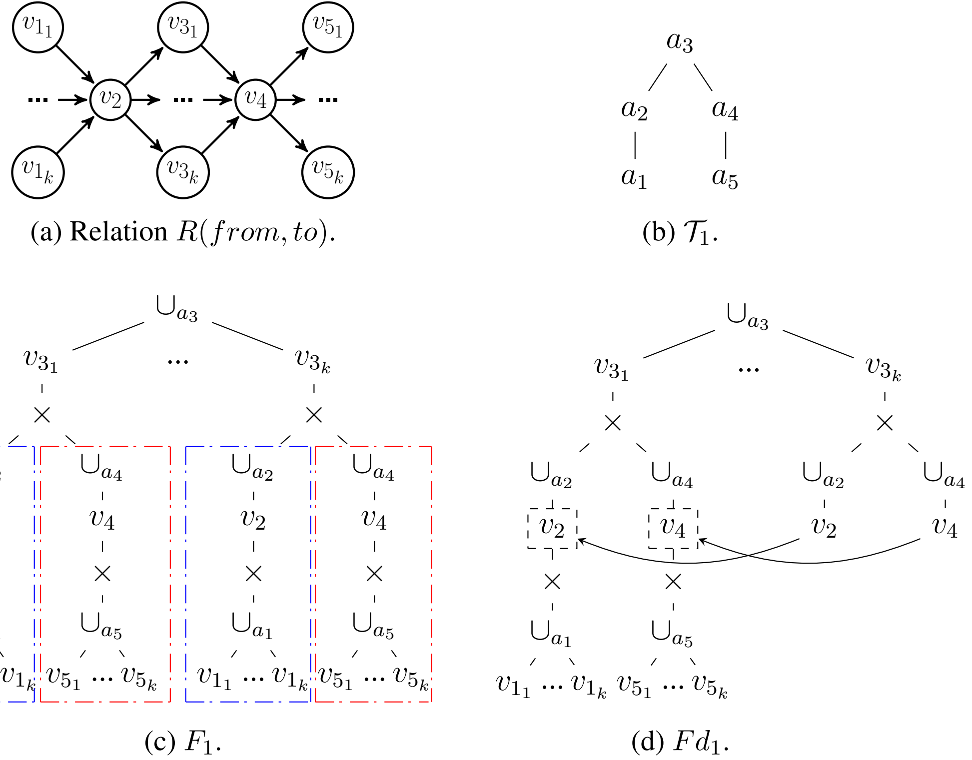


Figure 5.1: Example relation R and output results for $R(a_1, a_2), R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$ as an f-representation F_1 and as a d-representation Fd_1 following $\mathcal{T}_1$.

This chapter focuses on the adoption of *d-representations* in a vectorized query executor. Similar to f-representations, d-representation are tries. Similar to what we previously argued in Chapter 4, it is challenging to adopt tries in pipelined and vector-based query processors, which require processing on flat-tuples. No DBMS has adopted d-representations in its query processor. The chapter proposes the design and implementation of the first general approach to adopting d-representations in analytical and vectorized query processors. We next overview the key components of our design and the contributions of this chapter.

5.1 Overview of Contributions

In this chapter, we build on top of our factorized vector executor from Chapter 4. Recall that the executor's vectors only use f-representations and factorize data on only limited parts of query results. In this chapter, we adopt d-representations meeting specific constraints in our design:

i) use flat tuple-based operators from our factorized vector executor; and ii) keep the standard pipelined query processor architecture. Both of these constraints enable us to rely on traditional optimizations and minimize the necessary changes within analytical DBMSs for wider adoption.

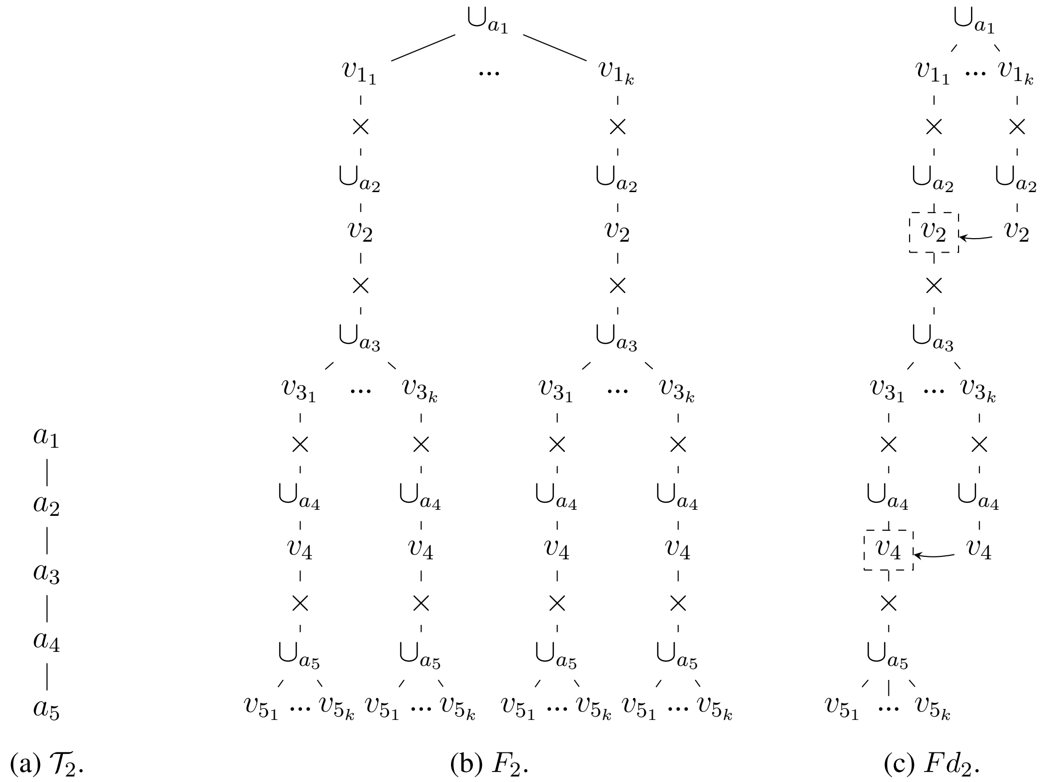
We begin by observing that d-representations provide benefits for two reasons: (i) caching and reusing intermediate relations which are the results of certain subqueries; and (ii) compressing these reusable intermediate relations and the final query results. Note that the standard `Hash Join` (`HJ`) operator from Chapter 4 already achieves some form of (i) and (ii). This is because `HJ` decomposes a query into two subqueries and then materializes the results of one within its build side to reuse. Inside hash tables of `HJ`(s) in our factorized vector executor, for each key value, the sub-relation is stored in factorized vectors as explained in Chapter 4. However, this approach achieves a limited form of f-representations limiting compression benefits and cannot re-use nested parts of repeated sub-relations as done in d-representations with definitions.

With this insight, the first key component of our design consists of an implementation of d-representations as nested hash tables that caches reusable sub-relations in query plans and whose keys can point to other hash tables. We also introduce a blocking operator called `DGroup-by` that builds progressively these nested hash tables by appending multiple `DGroup-by`(s) to a WCOJ subplan.

The second key component is DAG-style plans, in which `INLJ/IMJ` operators share d-representations generated by `DGroup-by`(s) and use them to skip parts of sub-plans if the sub-relation for a particular key, *e.g.*, $a_2 = v_2$ or $a_4 = v_4$ in our example from above, has already been computed. If so, then the computation execution moves to the child of the last `DGroup-by` operator that was used to generate this cached sub-relation. If the sub-relation for a key has not been computed, then the computation moves to the immediate child of the `INLJ/IMJ` operator. This is a minimal change to the `INLJ/IMJ` operator and importantly the rest of the operators, *e.g.*, scan and filter, are not changed and continue processing vectors.

For a full system solution, we describe a rule-based optimization layer that we added to the system to generate plans that perform factorized query processing using definitions. Our rule-based optimizer modifies the hybrid plan generated by the cost-based optimizer we described in Chapter 3 by adding the necessary `DGroup-by` operators.

Our experiments on the LDBC graph benchmark show that our factorized processor can improve the performance of the system's factorized vector executor by upto 60x. In our micro-benchmarks we demonstrate the performance benefits and overheads of our processor under different query and data-specific factors, such as join structures, and demonstrate which factors affect the benefits we can obtain from factorization. We further perform a plan study on a set of queries and show that integrating our approach can make the optimizer of the system more robust by enabling a larger classes of efficient plans on these queries.



(a) $\mathcal{T}_2$. (b) F_2. (c) Fd_2.

Figure 5.2: Output results for $R(a_1, a_2), R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$, where R is the relation in Figure 5.1a, as an f-representation F_2 and as a d-representation Fd_2 following $\mathcal{T}_2$.

5.2 Preliminaries

We begin by giving an overview of d-representations and d-trees and compare its size bound with that of f-representations and the AGM bound. A more detailed and formal coverage of this background is given by Olteanu *et al.* [145].

5.2.1 D-representations: F-representations using Definitions

Consider the input relation R in Figure 5.1a and the earlier Q_{4H}=$R(a_1, a_2)$, $R(a_2, a_3)$, $R(a_3, a_4)$, $R(a_4, a_5)$. We will revisit the f-tree and f-representation we used in the introductory section of this chapter momentarily. We first give the more classic example of an f-representation and f-tree. Consider F_2, an f-representation following f-tree $\mathcal{T}_2$ that is equivalent to a flat representation

shown in Figure 5.2b. In the figure, all sub-expressions rooted in v_2 and v_4 are exactly the same showing that F_2 uses the two sub-expressions multiple times. These sub-expressions can be re-used to compress F_2. Recall that a reusable sub-expression is called a definition. We show this in Figure 5.2c, where we use the same nodes in dashed boxes to indicate a single shared sub-expression. The figure shows a d-representation Fd_2 with two definitions:

- $D_1 = \cup_{i=1}^{i=k} (v_{5_i})$
- $D_2 = \cup_{j=1}^{j=k} (v_{3_j} \times v_4 \times D_1)$

F_2 in Figure 5.2b has as size, *i.e.*, number of node values:

k [for k v_{1_i}s] $\times$ (
  2 [for $v_1.v_2$ tuple] +
  k [for k v_{3_i}s] $\times$ (
    2 [for $v_3.v_4$ tuple] +
    k [for k v_{5_i}s]
)
)
$= k \times (2 + k \times (2 + k))$
$= k^3 + 2.k^2 + 2.k$

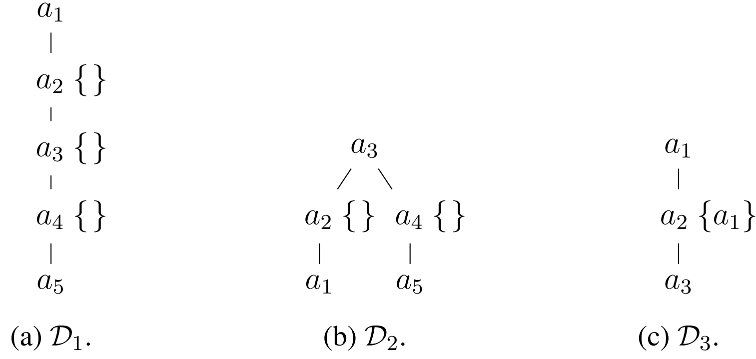
After reuse of the subexpressions D_1 and D_2, the size of the d-representation Fd_2 without accounting for pointers is $2k$ [for $v_{1_i}.v_2$ tuples] $+$ $2k$ [for $v_{3_i}.v_4$ tuples] $+$ k [for v_{5_i} tuples] $= 5k$.

Next, let us revisit the f-representation F_1 following $\mathcal{T}_1$ in Figure 5.1. F_1 also uses two sub-expressions multiple times. Specifically, the sub-expressions rooted at v_2 and v_4 are exactly the same. These sub-expressions can be re-used to compress F_1. This is shown in the d-representation Fd_1 in Figure 5.1d which has two definitions:

- $D_1 = \cup_{i=1}^{i=k} (v_{1_i})$
- $D_2 = \cup_{i=1}^{i=k} (v_{5_i})$

Given R, F_1 has as size: k [for k v_{3_i}s] $\times$ (
  1 [for v_{3_i} tuple] +
  1 [for v_2 tuple] +
  k [for k v_{1_i}s] +
  1 [for v_4 tuple] +
  k [for k v_{5_i}s]
)
$= k \times (1 + 1 + k + 1 + k)$
$= 2.k^2 + 3.k$

After reuse, the size of the d-representation Fd_1 without accounting for pointers is $3k$ [for k $v_2.v_{3_i}.v_4$ tuples] + $2k$ [for k v_1s and k v_5s] $= 5k$.



(a) $\mathcal{D}_1$. (b) $\mathcal{D}_2$. (c) $\mathcal{D}_3$.

Figure 5.3: Examples of three d-trees. $\mathcal{D}_1$ and $\mathcal{D}_2$ for the four-hop query $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5$ and $\mathcal{D}_1$ for the triangle query $a_1 \rightarrow a_2 \rightarrow a_3$,$a_1 \rightarrow a_2$.

Common sub-expressions for reuse can be inferred by analyzing the shapes of the input query and a given f-tree. Specifically, the rule is the following [145]: *given an f-tree T and a sub-expression rooted at node a_i, if all the nodes in the sub-expression do not depend on any of a_i's ancestors then it is possible to re-use the whole sub-expression for the same value of a_i.*

5.2.2 D-trees

D-trees, which describe the schema of a d-representation, are similar to the f-trees introduced in Section 4.2.1.1. The only addition is that at each internal node o, *i.e.*, non-root and non-leaf node, we add the set of ancestors that the descendants of o depend on. If the descendants do not depend on any of o's ancestors, the ancestor dependency set is empty. This is exactly the condition when we can reuse sub-expressions rooted in different bindings of keys to attribute o.

Figure 5.3 shows examples of three d-trees. $\mathcal{D}_1$ and $\mathcal{D}_2$ in Figure 5.3a and 5.3b, respectively, are two d-trees for Q_{4H} that extend the f-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ in Figure 5.1b and 5.2a, respectively, with subtree dependence attributes. Expression reuse for d-representations following $\mathcal{D}_1$ and $\mathcal{D}_2$ is possible on each non-leaf node as their ancestor dependency sets are empty. Alternatively, $\mathcal{D}_3$ in Figure 5.3c is another d-tree for the triangle query $R(a_1, a_2), R(a_2, a_3), R(a_1, a_3)$. This is an example where we cannot do caching on the only internal node a_2 since its child a_3 depends on a_2's ancestor a_1. As such, the d-representation following $\mathcal{D}_3$ is equivalent to an f-representation.

5.2.3 Worst-case Size Bounds for F- and D-representations

Recall that the AGM bound of a query Q, denoted by $|D|^{\rho^*(Q)}$, where D is a database instance, is the worst-case *i.e.*, maximum output size for a given query on any database instance. Recall as well that we define $|D|^{s(Q)}$ as the minimal worst-case output size for f-representations and that $s(Q) \leq \rho^*(Q)$. We define $|D|^{s^\uparrow(Q)}$ as the minimal worst-case output size for d-representations.

Definition 5.2.1 ***D-representation minimal worst-case output sizes -*** *Given Q and all d-trees of Q, $|D|^{s^\uparrow(Q)}$ is the size of the d-representation following d-tree T*, which is the one describing the d-representation with the lowest maximum output size across all database instances.*

Note that $|D|^{\rho^*(Q)}$ is the size of f-representations following f-trees that are paths, *i.e.*, each node has at most a single child. Furthermore, $s(Q)$ and $s^\uparrow(Q)$ are usually smaller than $\rho^*(Q)$ such that $s^\uparrow(Q) \leq s(Q) \leq \rho^*(Q)$. We show this for the 4-hop query Q_{4H} as an example for possible input edge relations of size N. The AGM bound of Q_{4H} is N^4 hence $\rho^*(Q_{4H}) = 4$. Meanwhile the worst-case output size of the f-representations following f-trees $\mathcal{T}_2$ and $\mathcal{T}_1$ in Figures 5.2a and 5.1b are $\theta(N^4)$ and $\theta(N^2)$, respectively. $\mathcal{T}_1$ leads to the minimal worst-case f-representation size for Q_{4H} and therefore $s(Q_{2H}) = 2$. D-representations described by the d-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are of size $\theta(N)$ therefore $s\uparrow(Q_{2H}) = 1$. To summarize, for Q_{4H}, $\rho^*(Q) = 4$, $s(Q) = 2$ and $s^\uparrow(Q) = 1$. Note that d-trees for the four-hop queries lead to representations of size N^4, N^3 (not shown in our example) and N^2 while all d-trees lead to representations of size N.

5.3 Adopting D-representations

In this section, we present a comprehensive query processor design built on top of GraphflowDB's factorized vector executor. Our goal of adopting d-representations is to find new caching and reuse opportunities of intermediate results while minimizing their size. In our approach, we do this within a single pipeline of a query plan, which in our query processor runs a sequence of `INLJ`/`IMJ` operators, *i.e.*, corresponds to a WCOJ sub-plan. We cache intermediate results within pipelines and materialize them as d-representations. In this section, we give the details of our design and changes to the GraphflowDB query processor and optimizer. We will describe our design for the general setting when plans can contain `HJ` operators and as such are the hybrid plans of Chapter 3 using factorized vector execution of Chapter 4.

5.3.1 Overview

A pipeline is a chain of `Scan`, `INLJ/IMJ`, `Filter`, and `Probe HT` operators ending in a `Sink` operator, ***i.e.***, `Build HT` or `Accumulator`. Note that while it is possible to cache large parts of pipelines that mix WCOJ *subpipelines*, (a subpipeline is a chain of operators within a pipeline), and `Probe HT` operators, this comes at a large complexity cost. As such, within a single pipeline, we cache only subqueries evaluated by WCOJ subpipelines, possibly delimited by `Probe HT` operators.

We use d-trees of subqueries to statically analyze caching opportunities within a WCOJ subpipeline. Given a WCOJ subpipeline that evaluates a subquery Q_s and the valid d-trees for Q_s, we choose one of the d-trees to find the cacheable subqueries of Q_s (how we choose a d-tree is explained in Section 5.3.3). The subqueries are cached and materialized as d-representations following parts of the chosen d-tree. We introduce a new operator `DGroup-by`, multiple of which are chained to materialize the cached subquery. We further introduce changes to the join operator in order to 'jump ahead' to a further away operator in the pipeline skipping a whole subpipeline. The idea is if this subpipline's output is materialized already for a particular input, then it can be reused and we can skip the evaluation for that input. Finally, we introduce new iterator operators that map d-representations into factorized vectors.

Next, we introduce the details of our query processor and query optimizer in details. Within this section, we view d-representations logically as presented in our preliminaries in Section 5.2. Our implementation of d-representations is discussed in details in the next Section 5.4.

5.3.2 Query Processor

In this section, we first cover how we materialize subqueries as d-representations for reuse and the functionality of our `DGroup-by` operator. We start with materialization over a simple d-tree that is a path and then go over handling arbitrary d-trees. We also introduce an optimization to exploit partial caching opportunities. Throughout the section, we cover the necessary changes to our operators.

5.3.2.1 D-Representation Materialization

We decompose the materialization of d-representations into interleaved subpipelines with two different tasks: i) WCOJ subquery evaluation; and ii) subquery materialization as d-representations using a chain of blocking `DGroup-by` operators.

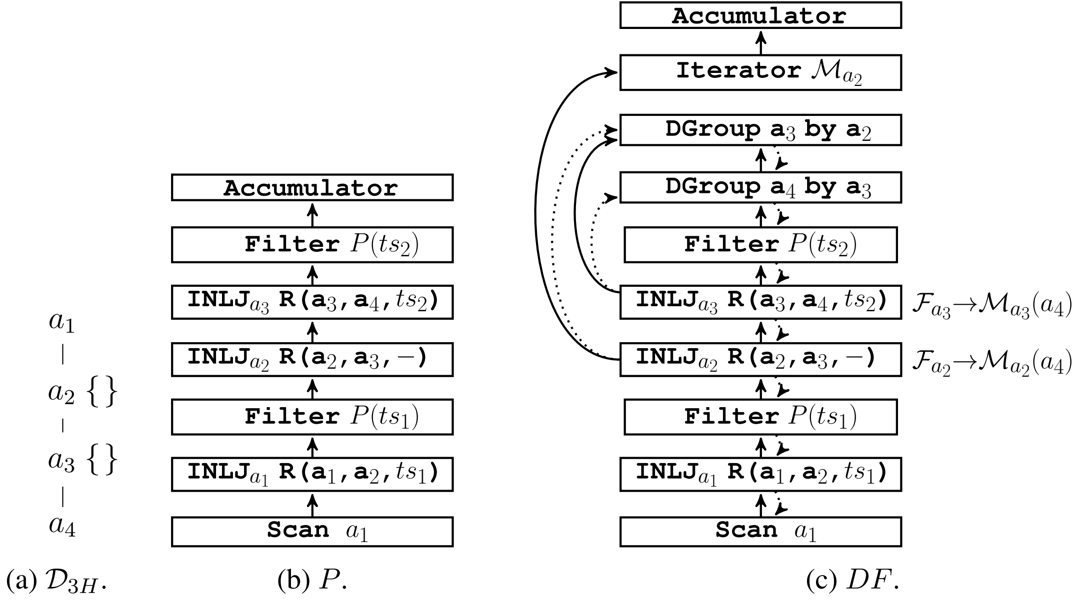
A `DGroup-by` fills a node in a d-representation by grouping a list of attribute values of a_i by an a_j key. For example, the operator `DGroup` a_5 `by` a_4 as part of materializing results following $\mathcal{D}_1$ in Figure 5.3a is used to group a list of a_5 values by a given v_{a_4} key. It does so in three steps. First, `DGroup` a_5 `by` a_4 is informed of the grouping value v_{a_4} and initializes an empty list of values L_{a_5}. Second, successive call to `DGroup-by` append a_5 values to L_{a_5}. Finally, `DGroup-by` is informed that there are no more a_5 values for v_{a_4} and inserts v_{a_4} as an a_4 node with a pointer to L_{a_5}. Other `DGroup-by` operators for $\mathcal{D}_1$ would work in the same way *e.g.*, `DGroup-by` a_4 by a_3 inserts a v_{a_3} as an a_3 node that points to a list of a_4 values, where each value itself points to a_5 lists.

For a given query Q and pipeline, we generate dataflows that replace some of the sub-pipelines. These dataflows exploit reuse opportunities by materializing subqueries of Q following chosen d-trees. In this section, we just inform the reader of the chosen d-tree and explain how we choose it in the next Section 5.3.3 on query optimization. We showcase this with two examples: 1) a path d-tree; and 2) a d-tree with branching.

Example 1. (Path d-tree)
Consider the query $Q_{3H}(a_1, a_4) = R(a_1, a_2, ts_1),\ R(a_2, a_3, -), R(a_3, a_4, ts_2), P(ts_1), P(ts_2)$, where $R(a_i, a_j, timestamp)$ is an m-n relationship $a_i{\rightarrow}a_j$ and $P(ts_i)$ is a simple predicate on the $timestamp$ attribute. Consider further the D-tree $\mathcal{D}_{3H}$ in Figure 5.4a, which is a valid factorization for the output, and the WCOJ plan P with JAO (a_1, a_2, a_3, a_4) in Figure 5.4b.

As P evaluates Q_{3H}, note that $\{\ a_3, a_4\ \}$ are only dependent on a_2 and $\{\ a_4\ \}$ is only dependent on a_3. However both are evaluated using tuples (v_{a_1},v_{a_2}) and (v_{a_1},v_{a_2},v_{a_3}), respectively. $\mathcal{D}_{3H}$ shows these dependencies. When analyzing opportunities for reuse within a subpipeline, we choose a *d-tree that matches the subpipeline's JAO*. We say a d-tree matches a JAO if a preorder traversal on the d-tree outputs the JAO, which is the case for $\mathcal{D}_{3H}$ and JAO (a_1, a_2, a_3, a_4). As such, we can rely on $\mathcal{D}_{3H}$'s structure to decide which subqueries to materialize and reuse. Note there are other possible d-trees to use but $\mathcal{D}_{3H}$ would be the d-tree chosen within our approach as explained in later in Section 5.3.3.

Given $\mathcal{D}_{3H}$, we would map P into the dataflow DF in Figure 5.4c, which we cover momentarily. In this dataflow, we would materialize the subqueries of Q_{3H} that can be evaluated once and reused. We would materialize as a d-representation the subqueries $\mathcal{M}_{a_2}(a_2{=}v_{a_2}, a_4)$ $= R(v_{a_2}, a_3, -), R(a_3, a_4, ts_2), P(ts_2)$ per unique v_{a_2}. $\mathcal{M}_{a_2}(a_4)$ materialization would follow $\mathcal{D}_{3H}$'s sub-d-tree rooted in a_2. Consequently, we would also materialize recursively subqueries of $\mathcal{M}_{a_2}$. Some of which can also rely on reuse during evaluation such as $\mathcal{M}_{a_3}(a_3{=}v_{a_3}, a_4) =$ $R(v_{a_3}, a_4, ts_2), P(ts_2)$, where $M(a_3) \subset M(a_4)$. Note that DF applies projections as necessary. For instance $\mathcal{M}_{a_2}(a_4)$ implies caching a_4 attributes per unique v_{a_2} as a_3 is not part of



(a) $\mathcal{D}_{3H}$. (b) P. (c) DF.

Figure 5.4: Example of a WCOJ pipeline P and a Dataflow DF that adds reuse to P following $\mathcal{D}_{3H}$ in Figure 5.4a, where $\mathcal{M}_{a_3}(a_4) = R(a_3{=}v_{a_3}, a_4, ts_2), P(ts_2)$ and $\mathcal{M}_{a_2}(a_4) = R(a_2{=}v_{a_2}, a_3, -), R(a_3, a_4, ts_2), P(ts_2)$.

$Q_{3H}(a_1, a_4)$'s output. Note further that we do not group a_2 per a_1 values as this is redundant due to P scanning unique v_{a_1}(s) in its `Scan` a_1.

DF aims from a high-level to follow the hand-written code in Algorithm 2. We will refer to specific lines from the algorithm as we describe the functionality of DF's operators. DF is split into two parts. A WCOJ subpipeline from the `Scan` operator to `Filter` $P(ts_2)$ (lines 3, 5, 10, and 15) followed by two `DGroup-by` operators (lines 19 and 23). From the join operators on a_2 and a_3, *i.e.*, INLJ$_{a_2}$ and INLJ$_{a_3}$, respectively, we have solid lines for 'data flow' that point to operators ahead in the pipeline. These are similar to the data flow ones pushing intermediate results to the next operator in P. The jump ahead data flow pointers, indicate the possibility of skipping a subpipeline. A join operator on a_i that points ahead keeps track of the unique v_{a_i} in a set $\mathcal{F}_{a_i}$. If v_{a_i} has been encountered, then the subpipeline between the join operator and the one ahead it points to has materialized the subquery for v_{a_i} and provides fast access to its results given v_{a_i}. For example, INLJ$_{a_2}$ in DF keeps track of encountered v_{a_2}(s) in $\mathcal{F}_{a_2}$. Specifically, $\mathcal{F}_{a_2}$ is a set of pairs (v_{a_2}, reference to $M_{a_2}(a_2{=}v_{a_2}, a_4)$). So, INLJ$_{a_2}$ can jump ahead and skip the evaluation of $\mathcal{M}_{a_2}(a_2{=}v_{a_2}, a_4)$.

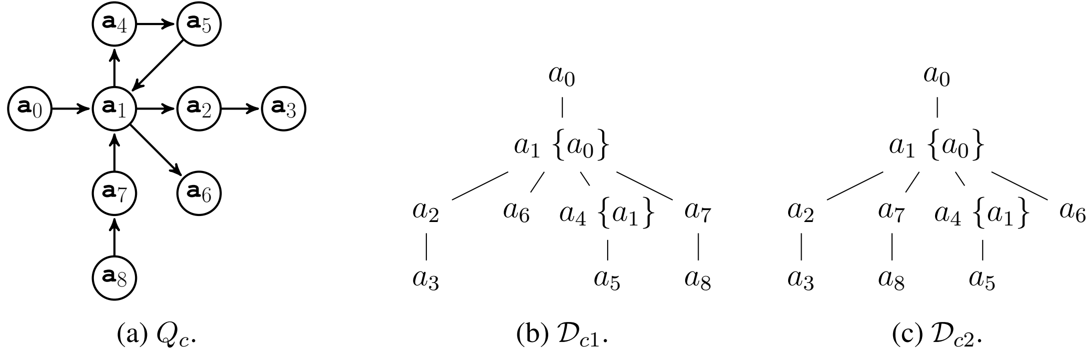
Cached subqueries can be accessed later with a new operator (`iterator` $\mathcal{M}$) that we append latest in the plan. In DF, an `iterator` $\mathcal{M}_{a_2}$ is appended to produce results in a factorized vector format for the `Accumulator`.

Algorithm 2 Materializing a d-representation following $\mathcal{D}_1$ in Figure 5.4a for $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4$.

Input: D-tree $\mathcal{D}$

1: $F_d =$ empty d-representation
2: $P_{a_2}, P_{a_3} = \{\ \}$ # P_{a_i}: set of a_i values encountered that lead to no output.
3: **for** $v_{a_1} \in \texttt{Scan}(a_1)$ **do**
4: $S_{a_2} = \{\ \}$ # S_{a_i}: d-representation node as (v_{a_i}, children Ptrs) tuples.
5: **for** $v_{a_2} \in \texttt{INLJ}_{a_1}$ (v_{a_1},R(a$_1$, a$_2$, ts_1)).Filter($P(ts_1)$) **do**
6: **if** $v_{a_2} \in P_{a_2}$ **then**
7: **continue** # because v_{a_2} encountered and lead to no output.
8: $S_{a_3} = F_d$.get(S_{a_3} given v_{a_2})
9: **if** $S_{a_3} = \{\ \}$ **then** # 1st v_{a_2} encounter
10: **for** $v_{a_3} \in \texttt{INLJ}_{a_2}$ (v_{a_2},R(a$_2$, a$_3$, -)) **do**
11: **if** $v_{a_3} \in P_{a_3}$ **then**
12: **continue** # because v_{a_3} encountered and lead to no output.
13: $S_{a_4} = F_d$.get(S_{a_4} given v_{a_3})
14: **if** $S_{a_4} = \{\ \}$ **then** # 1st v_{a_3} encounter
15: $S_{a_4} = \texttt{INLJ}_{a_3}$ (v_{a_3},R(a$_3$, a$_4$, ts_2)).Filter($P(ts_2)$)
16: **if** $S_{a_4} = \emptyset$ **then**
17: P_{a_3}.insert(v_3)
18: **if** $S_{a_4} \neq \{\ \}$ **then** # either cached in F_d or just evaluated lines (14-16).
19: S_{a_3}.insert((v_{a_3}, S_{a_4}))
20: **if** $S_{a_3} = \emptyset$ **then**
21: P_{a_2}.insert(v_2)
22: **if** $S_{a_3} \neq \{\ \}$ **then** # either cached in F_d or just evaluated lines (9-20).
23: S_{a_2}.insert((v_{a_2}, S_{a_3}))
24: F_d.insert((v_{a_1}, S_{a_2}))
25: results = iterate over F_d;
26: **return** results;

Note that each operator has a further control flow shown as a dashed arrow to the previous operator. It indicates that an operator returns to its previous operator whether the downstream operators led to any output tuples. Join operations capable of jumping ahead keep track of the bindings that led to no final output tuples to avoid evaluating them and as such prune the generation of unnecessary partial tuples. In Algorithm 2, this is equivalent to lines 2, 6-7, 11-12, 16-17, and 20-21.



(a) Q_c. (b) $\mathcal{D}_{c1}$. (c) $\mathcal{D}_{c2}$.

Figure 5.5: A query Q_c and two valid d-trees $\mathcal{D}_{c1}$ and $\mathcal{D}_{c2}$.

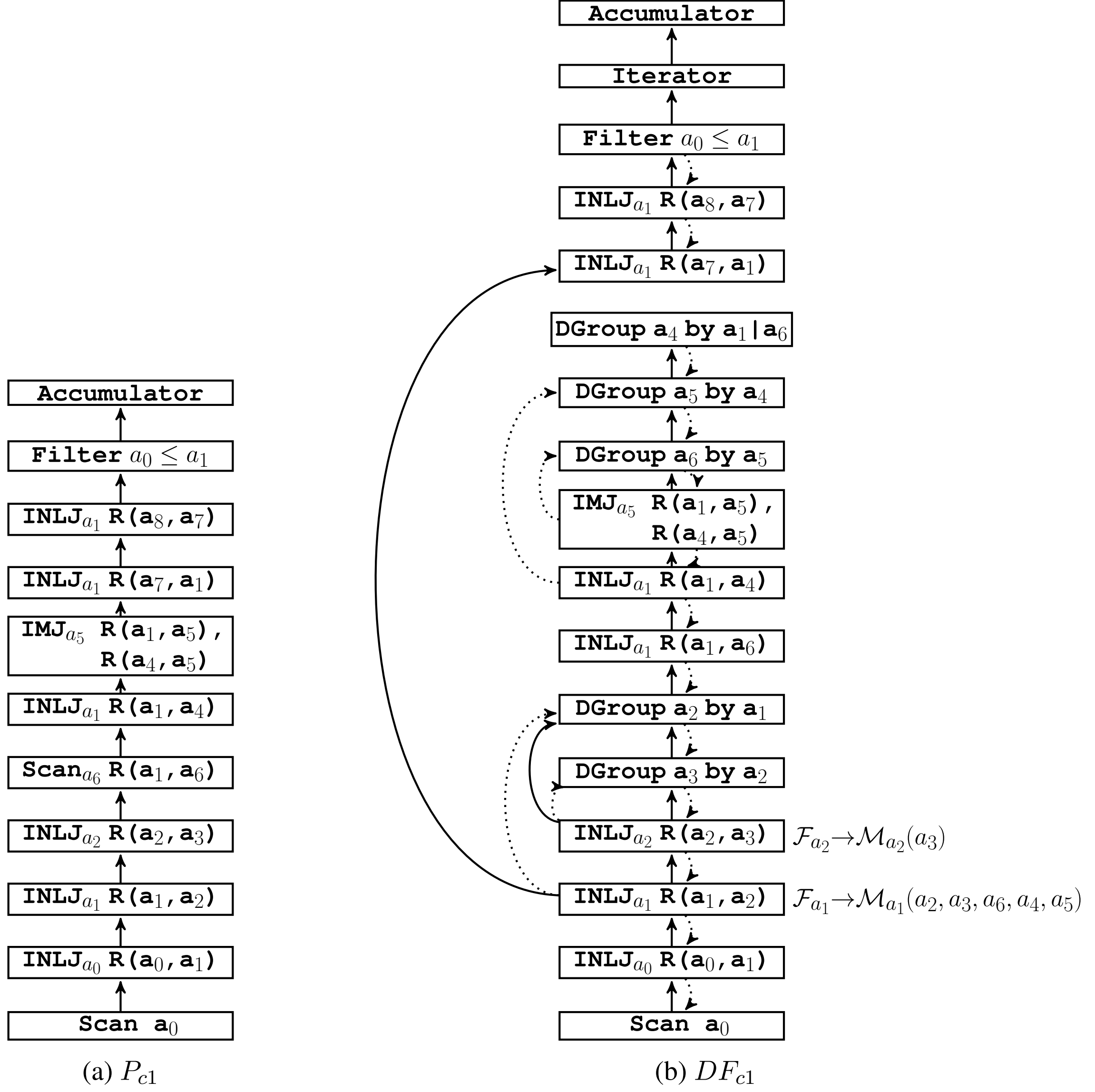
Finally, note that the join operations on a_i have a control flow arrow to a corresponding `DGroup-by` a_i operator. This is the third step described when we introduced the `DGroup-By` operator. It is because tuples are produced in vectors with a maximum value, *e.g.*, 1024 and if a many-to-many join produces more than the maximum, we need to control for this and notify the `DGroup-by` a_i accumulating the grouped values that there are no more incoming tuples with a_i's current binding. The control arrow is equivalent to grouping after `for` loops in Algorithm 2 (lines 19 and 23). Note that the `DGroup-by` applies projections as necessary. For instance, the corresponding `DGroup-by` of $\texttt{INLJ}_{a_2}$ projects a_3 values and points to a_4(s). This is missing from Algorithm 2 for simplicity of presentation. We explain the projection in Section 5.4 when we cover our d-representation implementation.

Example 2. (arbitrary d-tree with branching)
The prior example showed a path d-tree with each node having at most a single child node materialized. In the next example, we show how we build on top of that foundation to use arbitrary d-trees with branching. We also introduce a novel optimization to exploit further partial caching opportunities.

Consider the complex query below whose structure is shown in Figure 5.5a:

$$\begin{aligned} Q_c(a_1, a_3, a_4, a_5, a_6, a_8) = {} & R(a_0, a_1), R(a_1, a_2), R(a_2, a_3), R(a_1, a_4), R(a_4, a_5), R(a_5, a_1), \\ & R_m(a_1, a_6), R(a_7, a_1), R(a_8, a_7),\ a_0 \leq a_8, \text{ where} \\ & R \text{ and } R_m \text{ are m-n and m-1 relations, respectively.} \end{aligned}$$

Consider the WCOJ plan P_{c1} with JAO $(a_0, a_1, a_2, a_3, a_6, a_4, a_5, a_6, a_7)$ in Figure 5.6a. The plan first scans 3 hop subquery (a_0,a_1,a_2,a_3) and then evaluates the triangles (a_1,a_4,a_5), extends to a_6, then the 2-hop (a_1,a_7,a_8) and finally applies the filter between a_0 and a_8. Given P_{c1}, our approach chooses the d-tree $\mathcal{D}_{c1}$ in Figure 5.5b to find possible caching opportunities. Note that



(a) P_{c1} (b) DF_{c1}

Figure 5.6: Example of a WCOJ pipeline P_{c1} and a Dataflow DF_{c1} that adds reuse to P_{c1} following $\mathcal{D}_{c1}$ in Figure 5.5b, where $\mathcal{M}_{a_2}(a_3) = R(a_2{=}v_{a_2}, a_3)$ and $\mathcal{M}_{a_1}(a_2, a_3, a_6, a_4, a_5) = R(a_1{=}v_{a_1}, a_2), R(a_2, a_3), R(a_1, a_4), R(a_4, a_5), R(a_5, a_1)$.

the subtree rooted at a_1 in $\mathcal{D}_{c1}$ depends on a_0 and therefore per the theory of factorization, we cannot benefit from caching and reuse. In our approach however, we exploit caching and reuse even for a subset of the children branches; those that can be cached on a_1. Specifically, in $\mathcal{D}_{c1}$, the first left-most three branches can be cached on a_1 and as such we generate a dataflow plan that materializes the subquery of Q_c containing $(a_1, a_2, a_3, a_6, a_4, a_5)$ following the sub d-tree in $\mathcal{D}_{c1}$ rooted at a_1 while ignoring the right most branch (a_7, a_8).

P_{c1} is mapped to the dataflow DF_{c1} in Figure 5.6b. Branching means that our pipeline interleaves multiple WCOJ subpipelines and d-representation subpipelines as `DGroup-by` operators to materialize a subquery's results as shown in DF_{c1}. For instance, DF_{c1} has a WCOJ subpipeline starting from $\texttt{INLJ}_{a_1}(a_1, a_2)$ followed by `DGroup-by` operators and then a WCOJ subpipeline starting from $\texttt{INLJ}_{a_6}R(a_1, a_6)$ followed again by `DGroup-by` operators.

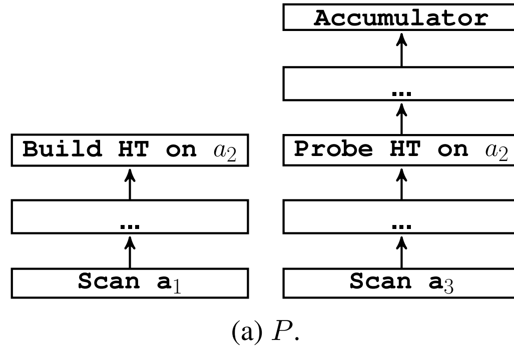
The dataflow materializes the subquery $M_{a_1}(a_1{=}v_{a_1}, a_3, a_6, a_4, a_5) = R(v_{a_1}, a_2)$,$R(a_2, a_3)$, $R(v_{a_1}, a_4)$, $R(a_4, a_5)$,$R(a_5, v_{a_1})$ for reuse. Consequently it also materializes M_{a_1}'s subquery $\mathcal{M}_{a_2}(a_3) = R(a_2 = v_{a_2}, a_3)$ for reuse. For a given v_{a_1}, DF_{c1} materializes first (a_2, a_3) results following $\mathcal{D}_{c1}$, however since we pipeline and evaluate the full subquery for a given v_{a_1} in downstream operators, if we produce no output tuples, this information is propagated back to $\texttt{INLJ}_{a_1}$ using the control flow pointing to previous operators which updates $\mathcal{F}_{a_1}$ to capture that v_{a_1} led to no output results and remove previously materialized parts of M_{a_1}.

The last `DGroup-by` operator in DF_{c1} is `DGroup` a_4 by $a_1|a_6$. In this operator, the a_6 attribute value is stored with a_1 when grouping a_4 values because $R_m(a_1, a_6)$ is an m-1 relationship and hence we fuse both nodes. This is an optimization to minimize the size of d-representations and avoid creating a list and pointing to it. We only do so for 1-m and n-m relationships.

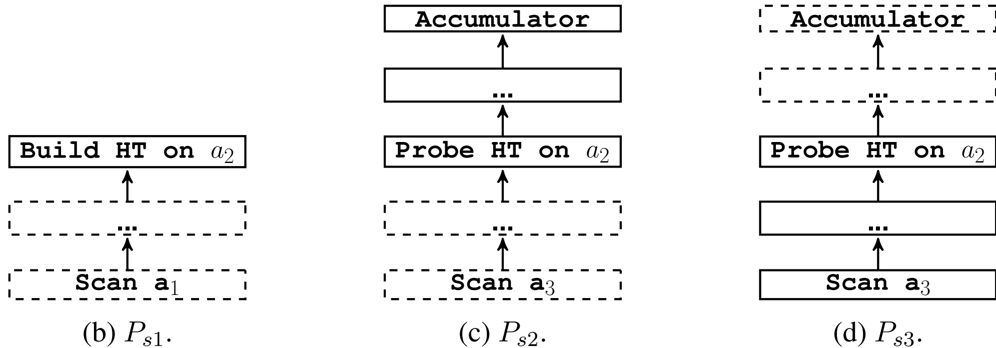
In our optimization of exploiting partial caching, we aim to avoid materializing results that cannot be reused and would be flattened right after the pipeline anyway. Consider another possible plan with JAO $(a_0, a_1, a_2, a_3, a_7, a_8, a_4, a_5, a_6)$ and d-tree D_{c2} in Figure 5.5c. In this case, we would only cache the subquery of Q_c containing attributes (a_1,a_2,a_3) since the second branch cannot be cached and therefore we decide to pipeline starting at that branch and we ignore caching in all subsequent branches.

5.3.3 Query Optimization

We build on top of the query optimizer of Chapter 3. This optimizer chooses a WCOJ or hybrid plan that is executed as a list of pipelines. Factorized vector execution introduced in Chapter 4 is taken into account by parameterizing the physical operators to execute on flat and unflact vectors as necessary. Recall from Section 5.3.1 that our goal is to make it easier to adopt d-



(a) P.



(b) P_{s1}. (c) P_{s2}. (d) P_{s3}.

Figure 5.7: Example of a plan P as two pipelines and three extracted subpipelines P_s to which we add caching and reuse.

representations and as such we introduce a rule-based approach to adding caching and reuse to our existing query optimizer instead of developing a new optimizer from the ground-up.

Given a plan as a list of pipelines, we map each pipeline P to a dataflow separately as follows. For each pipeline we only change subpipelines that include `Scan`, `INLJ/IMJ` , and `Filter` operators, *i.e.*, only WCOJ subplans delimited possibly by `Probe` operators and not including a `Sink`. As an example, consider the plan in Figure 5.7a made of two pipelines in which operators denoted by dots (. . .) represent a chain of `INLJ/IMJ` and `Filter` operators. For both pipelines we consider their WCOJ subpipelines to which we can add caching and reuse and end up with three such subpipelines in Figures 5.7b, 5.7c, 5.7d. For the subpipelines identified to which we consider adding caching and reuse, the operators are in dashed boxes in each figure.

Next for each subpipeline, we obtain the join attribute ordering σ within a subpipeline P_s. Given σ, we pick the d-tree $\mathcal{D}_s$ with the lowest height for which a preorder traversal outputs σ. These properties make $\mathcal{D}_s$ the d-tree describing the most opportunities of caching and reuse in

P_s. There is only one such d-tree. Finally, using the d-tree, we generate a dataflow with which we replace the subpipeline in the original plan. Examples of dataflows obtained from mapping WCOJ subpipelines are our two examples in Figures 5.4 and 5.6.

We take the query Q, the chosen d-tree $\mathcal{D}_s$, and subpipeline P_s as input to generate an output dataflow. If the subpipeline does not start with a `Scan` then it might depend on attributes evaluated by subpipeline prior to P_s so we add all attributes evaluated in the subpipeline before P_s as a root node to $\mathcal{D}_s$. The generation of the dataflow happens in a recursive fashion. A preorder traversal on $\mathcal{D}_s$ starts from the root and at each node labelled o we do as follows (except for leaf nodes in $\mathcal{D}_s$ which only apply step 1.):

1. Add a `Scan` or `INLJ`/`IMJ` operator, and `Filter`(s) as necessary to find bindings for o.
2. If o has children and is not obtained with a `Scan`, o might have repeated bindings during evaluation. As such, we identify the consecutive child branches that can be cached and reused starting from the left child.
3. Traverse to the cacheable child nodes which themselves start in step 1. Once done traversing the cacheable child nodes, append an appropriate `DGroup-by` operator.
4. Next, traverse to the rest of the child nodes.

Once this initial dataflow is generated, we do one more pass to add `Iterator` operators as necessary and to set the appropriate operators that joins should jump ahead to.

5.4 D-Representation Implementation

Our d-representations are implemented as nested hash tables. We explain its implementation details with an example that captures various considerations and optimizations. Consider the dataflow DF_{c1} and the d-tree $\mathcal{D}_{c1}$. This dataflow materializes the results for the subquery $M_{a_1}(a_3, a_6, a_4, a_5) = R(a_1, a_2), R(a_2, a_3), R(a_1, a_4), R(a_4, a_5), R(a_5, a_1)$. We show the internal structure of the materialized d-representation in Figure 5.8.

Each d-representation node is made up of multiple in-memory blocks of size 256 KBs in which attribute values and pointers to child nodes are stored. Each node labelled by attribute a_i for which we can reuse subtrees has a header that maps an a_i value to a slot, which is a pair of (block ID, offset). The header $\mathcal{F}_{a_1}$ is what join operators reference in our prior dataflows. We show two headers in the figure for $\mathcal{F}_{a_1}$ and $\mathcal{F}_{a_2}$, which `INLJ` operators in DF_{c1} access. Each node has value of attributes and children pointers as follows: a consecutive set of attribute values followed by the pointers. Each child pointer(s) use the highest most bit as follows. If set to 0, all child values are in a single chunk *i.e.*, belong to consecutive bytes in a block, otherwise, if set

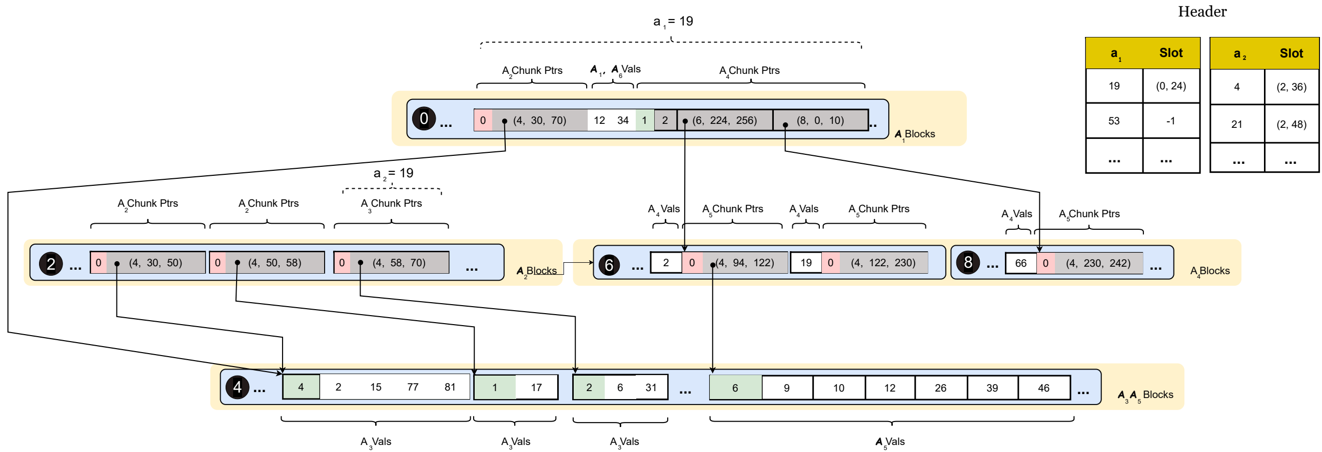


Figure 5.8: Example of internals of d-representation following $\mathcal{D}_{c1}$ in Figure 5.5b.

to 1, the child values are split across multiple chunks. When set to 0, and the bit is part of the single child pointer. Otherwise, it is set to 1 and the bit is part of the the number of chunks. Each pointer is a tuple (block ID, offset, length). The block 0 for the root a_1 node in Figure 5.8 shows an example of the layout described.

Note that a_1 points directly to a_4's and this is done as part of the `DGroup-by` projection which merges the chunks of a_3 that all a_2 children of a given v_{a_1} point to. This is shown in the figure where the various child chunks of 3 chunks of a_2 are merged into a single reference. The leaf d-representation nodes simply store the number of values followed by the values themselves. During the dataflow execution, all `DGroup-by`(s) grouping leaf nodes in the d-tree, share the same memory block to avoid fragmentation and writing to many separate blocks. Finally, notice that the a_6 which is obtained from the $a_1 {\rightarrow} a_6$ relationship in DF_{c1}'s $\texttt{Scan}_{a_6}$ which is an m-1 relationship and so the a_6 is stored in the same chunks as a_1 values. Some alternative designs are possible and are worth considering, *e.g.*, grouping the a_4s into separate lists and using multiplicity to make vector sizes smaller in subsequent accesses.

5.5 Experimental Evaluation

We present experiments to evaluate our adoption of d-representation and compare it with our factorized vector executor from Chapter 4. Our experiments aim to answer the following questions: (1) How much benefit do we obtain with d-representation adoption in an end-to-end

Name	#Vertices	#Edges
Epinions (Ep)	75.9K	508.8K
Amazon (Am)	403.4K	3.4M
Google (Go)	875.7K	5.1M
LDBC10	29.9M	176.6M
LDBC30	88.7M	540.9M
LDBC100	282.6M	1.7B

Table 5.1: Datasets used.

benchmark on top of factorized vector execution? [1] (2) What are the benefits that adoption of d-representation introduce over the plan space of WCOJs? (3) When does the adoption of d-representation bring the most benefit and for which queries and datasets?

5.5.1 Setup

Hardware: We use a single machine with two Intel E5-2670 @2.6GHz CPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical ones. All code runs on openjdk-17. We set the maximum size of the JVM heap to 500 GB and keep JVM's default minimum size.

Datasets: Table 5.1 summarizes the datasets used. We use 3 datasets from SNAP [112] and 3 LDBC social network datasets [26] generated with scale factors 10, 30, and 100, which we denote by LDBC10, LDBC30, and LDBC100, respectively. The datasets include social, web, and product co-purchasing networks, which have a variety of graph topology and sizes ranging from several million edges to over a billion edges.

Running: Each measurement is repeated five times. We treat the first two as warm-up, which we discard, and we report the median of the three subsequent runs.

We split our evaluation into end-to-end system comparisons to answer questions (1) and (2) and benchmarks to answer (3). We refer to the older version of our query processor from Chapter 4, *i.e.*, the one using factorized vector execution, as `GF-F`, and the one built on top of it adopting d-representations as `GF-D`.

[1] We tried but couldn't obtain the code of FDB [26] from its authors for a comparison.

5.6 Baseline System Comparison

We use the LDBC benchmark to assess the benefits of adopting d-representations on top of our factorized and vectorized query processor from Chapter 4. We only consider LDBC's interactive complex (IC) queries because LDBC's Short Read (IS) queries are at most 2-hop queries and do not lead to any subquery caching and reuse opportunities.

Similar to what we have done before, we modified IC queries. This is because GraphflowDB is a prototype system that lacks several features that LDBC queries exercise. The system has support for select-project-join queries, where joins are expressed as fixed-length. The system has also limited aggregation support.

We modified the IC queries similarly to the evaluation in Section 4.4 as follows. Variable length join queries that search for joins between a minimum and maximum length are set to the maximum length to make them fixed-length, and shortest path queries are removed. We also removed predicates that check the existence or non-existence of edges between nodes and the `ORDER BY` clauses.

Next, we run two types of IC queries, IC_c which have a COUNT(*) in the SELECT / RETURN clause and IC_p which project attribute. In the IC_p queries we only return numerical types as we do not support string types in our d-representation implementation. Note that in our experiments, we find the benefits to be about the same between IC_c and IC_p and in some cases IC_p queries obtain more benefits from subquery caching and reuse.

Finally, unlike Section 4.4, we turn these queries from point queries into range queries to introduce more redundant computation. We ran the IC queries on scale factors 10, 30, and 100 (SF10, SF30, and SF100) on queries IC_c and IC_p with selectivity that made the queries previously starting from a single source start from multiple. The queries would start from 0.1% or 1% of the vertices. We denote this by selectivity 0.1% and 1% respectively. Our exact queries are made available as part of the open-source code.

Similar to Chapter 4, the heavy selectivity on these queries made hand picking WCOJ plans suitable. Note that we only hand pick a WCOJ plan that uses factorized vector execution and rely on the rule-based approach of our query optimizer to map it into a dataflow that caches and reuses subqueries.

5.6.1 Discussion

We report all of our results in Tables 5.2, 5.3, and 5.4. Tables 5.2 and 5.3 report run time numbers in msecs for all IC_c queries, while 5.4 reports IC_p01 and IC_p02 as a comparison point but similar results to IC_C hold for the rest of the IC_p queries as we will explain.

By adopting d-representations, GraphflowDB improves over all queries except for IC07 and IC08, for both of which our rule-based optimizer does not add subquery caching and reuse as we will explain momentarily. The benefits in terms of query speedups are upwards of 68x. The larger the sizes of intermediate results, the larger the speedups with caching and reuse. Specifically, for the same selectivity, a larger scale factor leads to more speedups, *e.g.*, for selectivity 0.1%, IC_c03 has speedups of 9.05x, 13.6x, and 22.2x for SF10, SF30, and SF100, respectively. Similarly, for the same scale factor, the smaller the selectivity the larger the intermediate results, and therefore the larger the benefits. For SF100, IC_c03 has speedups of 22.2x and 60.1x for selectivity 0.1% and 1%, respectively.

We find that the benefits between IC_c and IC_p are very comparable and in some cases those of IC_p are even larger as shown in Table 5.4. For instance, GraphflowDB obtains more speedups due to caching on IC_p01 and IC_p02 for SF10 than on IC_c01 and IC_c02. This is due to the type of plans that GraphflowDB supports which relies on `INLJ`/`IMJ` operators using adjacency list indexes and then scanning column attributes that are needed for predicates or are part of the final output. These scans perform random memory access on column attributes. These accesses lead to slowdowns that can be avoided due to subquery caching. More traditional `HJ` based analytical plans would reduce these speedups but we would still see benefits from caching subqueries that evaluate joins and push-down projections.

SF10 - Selectivity 0.1%

	IC$_c$01	**IC$_c$02**	**IC$_c$03**	**IC$_c$04**	**IC$_c$05**	**IC$_c$06**
`GF-F`	172.4	173.3	9860.9	295.0	40.3	6653.9
`GF-D`	51.4	148.9	1089.0	228.2	11.6	898.0
	3.35x	**1.16x**	**9.05x**	**1.29x**	**3.47x**	**7.40x**
	IC$_c$07	**IC$_c$08**	**IC$_c$09**	**IC$_c$11**	**IC$_c$12**	
`GF-F`	4.5	4.2	5633.7	112.7	993.8	
`GF-D`	4.5	4.2	892.2	31.7	639.4	
	1.0x	**1.0x**	**6.3x**	**3.5x**	**1.5x**	

SF10 - Selectivity 1%

	IC$_c$01	**IC$_c$02**	**IC$_c$03**	**IC$_c$04**	**IC$_c$05**	**IC$_c$06**
`GF-F`	573.2	691.7	29233.1	13811.7	140.2	20743.4
`GF-D`	65.0	329.8	1119.6	458.5	20.7	974.3
	8.8x	**2.0x**	**26.1x**	**30.1x**	**6.7x**	**21.2x**
	IC$_c$07	**IC$_c$08**	**IC$_c$09**	**IC$_c$11**	**IC$_c$12**	
`GF-F`	14.7	3.5	18206.0	437.6	2901.3	
`GF-D`	14.7	3.5	1312.2	53.4	2691.2	
	1.0x	**1.0x**	**13.8x**	**8.1x**	**1.0x**	

SF30 - Selectivity 0.1%

	IC$_c$01	**IC$_c$02**	**IC$_c$03**	**IC$_c$04**	**IC$_c$05**	**IC$_c$06**
`GF-F`	1118.8	721.7	56069.2	2260.9	149.8	34585.5
`GF-D`	134.6	530.9	4119.9	866.2	41.4	2784.9
	8.3x	**1.3x**	**13.6x**	**2.6x**	**3.6x**	**12.4x**
	IC$_c$07	**IC$_c$08**	**IC$_c$09**	**IC$_c$11**	**IC$_c$12**	
`GF-F`	7.1	3.6	29998.5	403.0	3587.5	
`GF-D`	7.1	3.6	2884.1	96.1	2598.5	
	1.0x	**1.0x**	**10.4x**	**4.1x**	**1.3x**	

SF30 - Selectivity 1%

	IC$_c$01	**IC$_c$02**	**IC$_c$03**	**IC$_c$04**	**IC$_c$05**	**IC$_c$06**
`GF-F`	573.2	691.7	29233.1	13811.7	140.2	20743.4
`GF-D`	214.7	1413.8	4301.6	1450.8	86.3	3004.2
	21.6x	**2.2x**	**44.5x**	**48.5x**	**9.9x**	**38.1x**
	IC$_c$07	**IC$_c$08**	**IC$_c$09**	**IC$_c$11**	**IC$_c$12**	
`GF-F`	131.9	18.1	3120.6	205.1	11859.6	
`GF-D`	131.9	18.1	1312.2	53.4	11780.3	
	1.0x	**1.0x**	**35.4x**	**9.2x**	**1.0x**	

Table 5.2: Run time (msec) for IC LDBC Queries (SFs 10 and 30) for `GF-F` and `GF-D`.

Selectivity 0.1%

	IC$_c$01	**IC$_c$02**	**IC$_c$03**	**IC$_c$04**	**IC$_c$05**	**IC$_c$06**
GF-F	6513.4	3019.2	380768.1	14904.5	801.4	167647.3
GF-D	603.4 **10.7x**	2344.1 **1.2x**	17104.0 **22.2x**	3267.0 **4.5x**	162.7 **4.9x**	11535.2 **14.5x**
	IC$_c$07	**IC$_c$08**	**IC$_c$09**	**IC$_c$11**	**IC$_c$12**	
GF-F	49.6	11.5	189069.7	1642.4	17144.9	
GF-D	49.6 **1.0x**	11.5 **1.0x**	10834.9 **17.4x**	256.5 **6.4x**	9738.7 **1.7x**	

Selectivity 1%

	IC$_c$01	**IC$_c$02**	**IC$_c$03**	**IC$_c$04**	**IC$_c$05**	**IC$_c$06**
GF-F	20580.7	11521.0	1015240.9	319178.6	3712.5	548625.8
GF-D	774.1 **26.5x**	5348.9 **2.1x**	16880.0 **60.1x**	4690.6 **68.0x**	329.3 **11.2x**	14884.5 **36.8x**
	IC$_c$07	**IC$_c$08**	**IC$_c$09**	**IC$_c$11**	**IC$_c$12**	
GF-F	201.8	34.4	576191.8	7074.7	49427.3	
GF-D	201.8 **1.0x**	34.4 **1.0x**	10779.8 **29.1x**	395.7 **17.8x**	48870.8 **1.0x**	

Table 5.3: Run time (msec) for IC LDBC Queries (SF 100) for GF-F and GF-D.

Selectivity 0.1%

	SF10		**SF30**		**SF100**	
	IC$_p$01	**IC$_p$02**	**IC$_p$01**	**IC$_p$02**	**IC$_p$01**	**IC$_p$02**
GF-F	473.8	337.5	2460.5	1403.9	15509.6	6062.0
GF-D	139.0 **3.6x**	146.4 **2.3x**	495.9 **4.9x**	537.4 **2.6x**	1901.4 **8.15x**	1981.8 **3.0x**

Selectivity 1%

	SF10		**SF30**		**SF100**	
	IC$_p$01	**IC$_p$02**	**IC$_p$01**	**IC$_p$02**	**IC$_p$01**	**IC$_p$02**
GF-F	1611.5	1200.4	9263.4	4448.6	49172.3	20002.0
GF-D	162.8 **9.8x**	367.6 **3.2x**	596.7 **15.5x**	1208.8 **3.6x**	2453.6 **20.0x**	4906.0 **4.0x**

Table 5.4: Run time (msec) for IC LDBC Queries (SF 100) for GF-F and GF-D.

On IC LDBC queries, we are seeing large benefits, our sweep of IC$_c$ queries across three scale factors and two selectivity percentages lead to 66 queries. Out of the 66 queries, `GF-D` leads to: i) 23 queries with speedups of more than 10x; ii) 10 queries with speedups between 5x to 10x; iii) 11 between 2x to 5x; iv) 10 between 1x-2x; and v) 12 queries use the same plan as `GF-F`, for IC$_c$07 and IC$_c$08. These large benefits are due to the fact that LDBC queries contain several paths over many-to-many relationships. The queries also contain minimal simple predicates with no theta joins, *i.e.*, the queries are highly 'factorizable'.

Note that adding caching to IC$_c$07 and IC$_c$08 leads to slowdowns of 0.4-0.5x. Consider IC$_c$07 with a graph pattern of a 2-hop query $person \xleftarrow{hasCreator} comment \xrightarrow{likes} friend$. The evaluation starts at person and as such we can only cache friend vertices per comment. This is however exactly the type of data access provided by the adjacency list indexes. Our rule-based optimizer avoids this cache and generates fully pipelined plans.

5.6.2 Impact on WCOJ Plan Space

The adoption of d-representation happens within WCOJ subpipelines of hybrid plans. In this section, we examine the impact of subquery caching and reuse on WCOJ plans for LDBC IC queries. To do so, we compare the set of actual run times of WCOJ plans for queries with that of the dataflows. Broadly, we find that our dataflows tend to make the plan space more robust, *i.e.*, in some cases eliminating many bad plans as well as improving good plans.

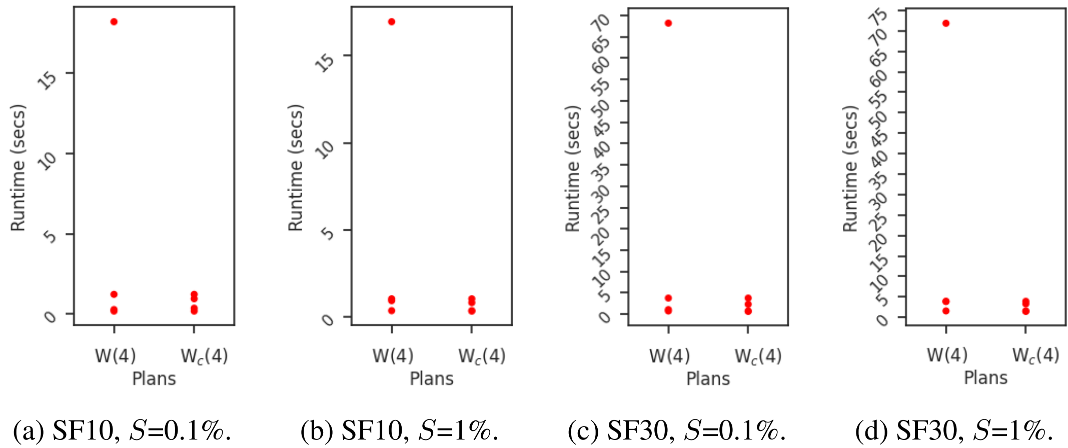
Table 5.5 shows the speedups on the WCOJ plan spectrum on IC$_c$01. The spectrum contains a total of 16 plans and 4-5 out of these plans has more than an order of magnitude speedups. For SF10, selectivity 0.1%, 13 of the plans run in less than 2 secs and 5 plans run for more than 70 secs. Our dataflow plans improve the performance of 3 of these very bad plans making them run in less than 1 sec for two of them and less than 3 for a third. Being able to completely avoid these plans makes the plan space more robust.

IC$_c$01's plan space contains a lot of plans, which is why we only show the speedups bucketed. On the other hand, IC$_c$02's plan space contains only four plans for which we show the full spectrum plan with and without subquery caching. Figure 5.9 shows the spectrums and helps make the case for adding robustness to a plan space. For example, for SF10, selectivity 0.1%, three plans running in 1.3 secs or less keep the same run-time while a fourth plan improves by 19x from 18.2 secs to 0.9 secs. The same applies to other configurations where one plan improves by $>$ 20x, another by 2x and two plans stay with the same run time.

(a)	∼1x	1-10x	10-100x	>100x
# plans	9	3	2	2

(b)	∼1x	1-10x	10-100x	>100x
# plans	9	2	3	2

Table 5.5: Number of WCOJ plans on IC_c01 per speedup when using subquery caching and reuse buckets based on order of magnitude on: (a) SF10, selectivity 0.1% and 1% and SF30, selectivity 0.1%; and (b) SF30, selectivity 1%.

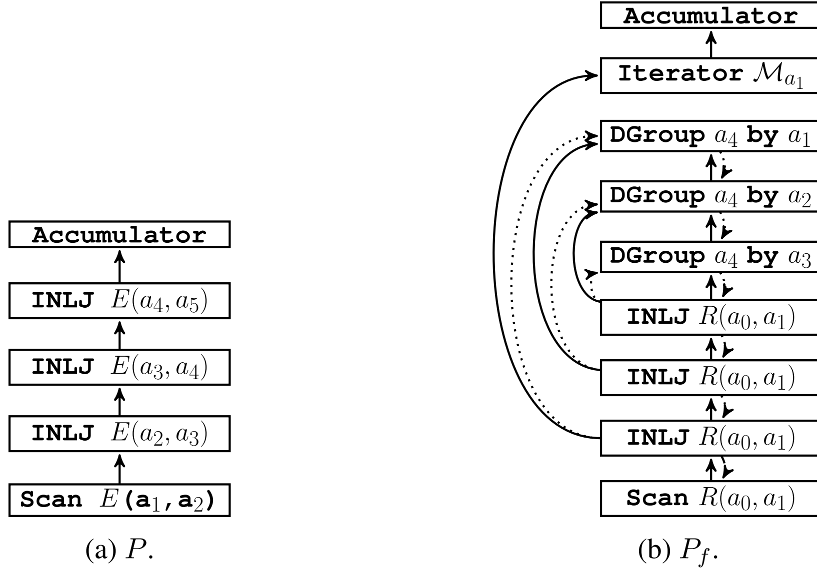


(a) SF10, S=0.1%. (b) SF10, S=1%. (c) SF30, S=0.1%. (d) SF30, S=1%.

Figure 5.9: Plan spectrum for QC_c02 showing the run time of WCOJ plans (W) and dataflows with subquery caching and reuse (W_c). This is shown on different SF and selectivity (S) configurations.

5.6.3 Microbenchmarks

The goal of our microbenchmarks is to answer our third question regarding the benefits of adopting d-representations. To do so, we aim at looking at two different aspects, the first is the amount of computation avoided and the second is the change of benefits dependent on the density/sparsity of the input graph. We also use path queries as they are simple and a common query template part of larger queries.



(a) P.

(b) P_f.

Figure 5.10: A WCOJ plan P and a dataflow P_f which adds d-representation usage to P evaluating $Q_{4H}(a_1, a_4) = E(a_0, a_1), E(a_1, a_2), E(a_2, a_3), E(a_3, a_4)$.

5.6.3.1 Computation Avoided

In order to empirically evaluate the amount of computations avoided, we evaluate the 4-hop join path query $\text{Q}_{4H} = E(a_1, a_2), E(a_2, a_3), E(a_3, a_4), E(a_4, a_5)$ on SNAP datasets.

The queries are evaluated using plan P, which is a WCOJ plan with JAO $(a_1, a_2, a_3, a_4, a_5)$ and dataflow P_f in Figures 5.10a and 5.10b. For $1 \leq i \leq 3$, each a_i attribute has materialized results of subqueries in $\mathcal{M}_{a_i}$ in P_f such that:
1) $\mathcal{M}_{a_2}(a_2 = v_{a_2}, a_5) = E(v_{a_2}, a_3), E(a_3, a_4), E(a_4, a_5)$
2) $\mathcal{M}_{a_3}(a_3 = v_{a_3}, a_5) = E(v_{a_3}, a_4), E(a_4, a_5)$
3) $\mathcal{M}_{a_4}(a_4 = v_{a_4}, a_5) = E(v_{a_4}, a_5)$
Note that for Q_{4H_c}, the reusable subquery contains as final results the number of output tuples of $\mathcal{M}(a_4)$ and not attribute values a_4.

Table 5.6 reports the run time in seconds for `GF-F` and `GF-D`. `GF-D` leads to significant speedups of upto 236x on epinions. To understand these benefits further, we next look at the computation avoided at different depths of the Q_{4H}.

The 4-hop query Q_{4H} is doing a graph traversal starting from v_{a_1}(s), *i.e.*, possible bindings for a_1, and finding a_5 ID bindings per v_{a_1}. We identify depths within the traversal where the depth D_i refers to the output of a join operator matching i-hop queries. For instance, $D1$ maps

Dataset	`GF-F`	`GF-D`
Ep	3068.0	13.0 **(236.0x)**
Am	36.2	4.2 **(8.6x)**
Go	177.2	17.8 **(9.9x)**

Table 5.6: Run time (secs) comparing `GF-F` and `GF-D` when evaluating $Q_{4H}(a_1, a_4) = E(a_0, a_1)$,$E(a_1, a_2)$, $E(a_2, a_3)$, $E(a_3, a_4)$ on:
1) Epinions (Ep); 2) Amazon (Am); and 3) Google (Go).

	Depth	`#TJ`	`#SJ`	`#M`	`#H`
Ep	**D1**	508.8K	0	51.9K	456.8K
	D2	39.9M	39.4M **(98.7%)**	49.8K	424.3K
	D3	3.721B	3.720B **(99.9%)**	49.6K	422.3K
Am	**D1**	3.3M	0	403.3K	2.9M
	D2	32.3M	28.9M **(89.4%)**	403.3K	2.9M
	D3	314.2M	310.8M **(98.9%)**	403.3K	2.9M
Go	**D1**	5.1M	0	714.5K	4.3M
	D2	60.6M	56.3M **(92.9%)**	669.0K	3.6M
	D3	878.8M	874.6M **(99.5%)**	658.7K	3.5M

Table 5.7: Evaluating $Q_{4H}(a_1, a_4) = E(a_0, a_1)$,$E(a_1, a_2)$, $E(a_2, a_3)$, $E(a_3, a_4)$, with a WCOJ with JAO $(a_1,)$, we report the total number of joins (`#TJ`) with no caching at each depth. We also report the % and number of saved joins (`#SJ`) by caches in prior depths. Finally, we report the number of cache misses (`#M`) & hits (`#H`) at each depth. Evaluation on:
1) Epinions (Ep); 2) Amazon (Am); and 3) Google (Go).

to the output matches (a_1, a_2) by operator `INLJ` R(a$_1$, a$_2$). Table 5.7 reports the computation avoided when adopting d-representations. The table reports the total number of joins done (`#TJ`), *i.e.*, number of input tuples processed by a join operator in the WCOJ plan P. The table also reports the number of saved joins (`#SJ`) by having cached subqueries at prior depths, that is the number of input tuples not processed due to the subpipeline being skipped. For D_1, `#SJ` is 0. The dataflow P_f has to process each input tuple including checking the set of visited v_{a_1} values. Joins that are completely saved start at D_2.

Both plans at D_1 join a_1 values as many times as the number of edges in the network with no work saved. Next both plans, join a_2 values. In this case, P joins a_2 as many times as there are $E(a_1, a_2), E(a_2, a_3)$ tuples, *i.e.*, 2-hop instances in the network while P_f would only do so as many times as the number of edges in the network since the a$_2$s traversed are the neighbours

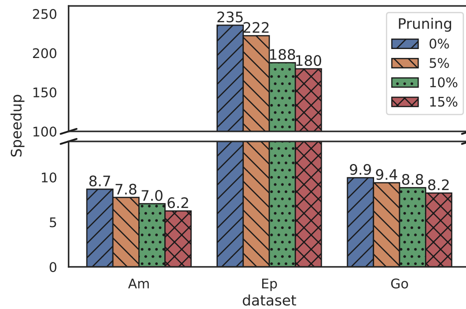


Figure 5.11: Speedup over factorized vector execution when we reuse expressions as as we prune a percentage of edges from three datasets: 1) Amazon(Am); 2) Epinions(Ep); and 3) Google(Go). Query evaluated is: $R(a_0, a_1)$, $R(a_1, a_2)$, $R(a_2, a_3)$, $R(a_3, a_4)$.

of unique a_1s which would be at most the set of edges. The same applies to joining a_3, where P joins a_3 as many times as there are $E(a_1, a_2)$, $E(a_2, a_3)$, $E(a_3, a_4)$ tuples, *i.e.*, 3-hop instances in the graph and P_f does so at most as many times as number of edges in the network. This is consistent with our expectation where the run time would be linear in the number of edges when adopting d-representation on path queries. Table 5.7 shows further benefits from the caching at each depth level. The number of cache misses for a_i joins is $|V|$ while the number of cache hits is $|E| - |V|$.

5.6.3.2 Dataset Impact

First, we make a note about the impact of skew on the benefits based on Table 5.7. In general, the more skew, *i.e.*, some vertices having very large neighbourhoods compared to the average, the larger the joins grow over many-to-many joins, this leads to more benefits from d-representation due to more computation saved at later depths. From 5.7, Amazon is a uniform dataset, while Google and Epinions have skew with Epinions having more of it. We can see from the table that Epinions has the most `#SJ`s followed by Google, and then Amazon. This also translates directly to run time improvements.

Second, we look at the impact of density/sparsity of the input graph on the benefits of factorization. For our three SNAP datasets: Amazon, Epinions, and Google, we prune 5%, 10%, and 15% of the edges and compare the speedup obtained by `GF-D` over `GF-F`. Figure 5.11 shows the amount of speedups. As the network is more sparse, factorization leads to less benefits, this is because the joins grow less and is closely connected to the amount of skew in a dataset. In general, a network with a large expansion rate over path queries gets more benefits from factorization.

Chapter 6

Related Work

The central thesis of this work is that modern query processors of DBMSs should adopt WCOJ algorithms and factorization to efficiently evaluate graph workloads, which contain join queries that are over m-n relationships. The core problem with these queries is that they can generate very large intermediate results, which worst-case optimal join algorithms and factorization address. The research conducted as part of this thesis contributes to an extensive literature on these topics. The technical contributions of this thesis are on novel approaches to adopt these two techniques into systems. There is also extensive implementation-based work on subgraph matching on graphs, which is equivalent to what we referred to as graph conjunctive queries, without any predicates aside from the join conditions. This thesis relates further to other systems approaches that make DBMSs more efficient on graph workloads by integrating other techniques that are orthogonal to WCOJs and factorization. In this chapter, we review related work in these topics. Within Chapters 2-5, we covered the most directly related work to our contributions, *e.g.*, any prior work we used as a baseline was covered in depth and as such will not be covered in depth here again.

6.1 WCOJ Algorithms

We begin in Section 6.1.1 by covering algorithmic work on WCOJ algorithms. In Section 6.1.2, we discuss work that have implemented WCOJ algorithms in actual systems but that was not covered in Chapter 3. In addition to covering the implementations in DBMSs, this section covers implementations of WCOJ algorithms in distributed data processing systems and the work to maintain join queries using incremental versions of these algorithms. Our implementation in

Chapter 3 included a detailed description of how we optimize these queries. Part of this description was about how we estimate the cardinality of subqueries. In Section 6.1.3, we briefly discuss work on cardinality estimation that relates to the catalogue-based approach we implemented. Interestingly, ideas behind WCOJ algorithms have also been used for cardinality estimation, which we cover here. We also described how to evaluate WCOJ plans adaptively by picking between multiple JAOs during runtime. This is the only adaptive evaluation of WCOJ algorithms in the literature. There is a rich body of work on adaptive query processing in relational systems [51], which we do not cover here.

6.1.1 Algorithmic Work on WCOJ Algorithms

The theory of worst-case optimal joins started with Atserias *et al.* 's work that introduced the AGM bound as the worst-case tight size for the outputs of join queries [24]. This work introduced the fractional edge covering number, ρ^* that we have used in prior chapters. In addition, this work introduced a new join algorithm that would run in time $O(IN^{\rho^*+1})$, if IN represents the size of the database on a query Q whose fractional edge covering number is ρ^*[1]. This algorithm already contained some of the key ingredients of the eventual worst-case optimal join algorithms NPRR and Generic Join that were introduced by Ngo *et al.* [135, 137]. Specifically, the Atserias algorithm would also perform the joins one attribute at a time, which is the key algorithmic step that differentiates these algorithms from traditional binary join algorithms. Interestingly, Veldhuizen has revealed that prior to the publication of the NPRR algorithm, LogicBlox, which is a Datalog system that Veldhuizen has worked on, had implemented another join algorithm called Leapfrog Triejoin algorithm, that also performed joins an attribute-at-a-time [179]. Veldhuizen has analyzed the runtime of this algorithm and shown that it is also worst-case optimal [179].

Leapfrog Triejoin, NPRR, and Generic Join are all worst-case optimal in the sense that their worst-case runtimes are asymptotically bounded by the AGM bounds of queries. They all evaluate joins one attribute at a time. There are however several differences between the algorithms. For example, NPRR separates values in attributes as "heavy" and "light", depending on the "degree" of the values on the join attributes. Degree of a value, *e.g.*, $a_1 = 5$, refers to the number of tuples in the input relations that have this value. Heavy attribute values can generate large intermediate results and are handled separately from light values. However, this separation is later understood to not be necessary in both Leapfrog TrieJoin algorithm and Generic Join, which is perhaps the simplest of these algorithms. This is the core algorithm we covered in Chapter 3.

[1]The algorithm was described as a constraint satisfaction algorithm but can also be easily translated into a join algorithm.

We mentioned in Chapter 1 when stating our thesis statement that WCOJ algorithms are particularly suitable for cyclic queries to reduce the intermediate results generated with binary join plans. An important clarification and connection to make here is with the seminal Yannakakis algorithm [185]. Yannakakis's semi-join based algorithm is instance optimal for acyclic join queries, *i.e.*, it runs in time $O(IN + OUT)$, where IN and OUT, are respectively the input size of the database and output size of the query. This algorithm takes an acyclic join query, generates a binary join/semi-join plan P, performs two rounds of semi-joins to remove any "dangling" tuples that cannot contribute to the final output. Finally, the algorithm performs a bottom-up join following P using traditional binary join algorithms. The algorithm, however, only has guarantees for acyclic queries. For cyclic queries, such as our running example triangle query, there are no known semijoin plans that can remove dangling tuples. In a sense, the inability to remove dangling tuples is part of the problem with cyclic queries. It is important to also note that Yannakakis's algorithm has a much stronger guarantee than worst-case optimal join algorithms, which are optimal only in the sense that their worst-case runtimes are no larger than any other algorithms' worst-case run time. On any input, they can be sub-optimal. There is no known instance optimal algorithm for cyclic queries.

Yannakakis's algorithm is often a subroutine in more advanced join algorithms to achieve provable bounds on algorithms [12, 16, 89]. Several prior work proposed using WCOJ algorithms in conjunction with GHDs [16, 89]. Recall from Chapter 3 that GHDs are effectively join plans whose nodes are subqueries that are intended to be evaluated with a WCOJ algorithm. Once these subqueries are evaluated, their results can be fed into the rest of the binary join plan. At this stage, the resulting join query (with results of base subqueries evaluated) is acyclic, so can be evaluated in an instance optimal manner with Yannakakis's algorithm. This approach of using a GHD can be understood as finding an "acyclic structure" of the query by treating cyclic subqueries as base relations, so at least Yannakakis's algorithm can be used on this structure (even if not on the entire query). Different work has shown that this approach can generate more nuanced worst-case runtimes for queries. For example, Afrati *et al.* [16] has shown that this approach can yield $O(IN^{ghw} + OUT)$ runtime for a query Q whose generalized hyper tree width (which the EH system used) is ghw. Let GHD D^* be the GHD of Q such that the maximum ρ^* exponent of any of the subqueries in D^* is the minimum across any GHD of Q. Ghw of Q is the maximum ρ^* of the subqueries in D^*. Importantly, ghw of Q is guaranteed to be at most the ρ^* of Q, which is the worst-case runtime one could achieve by running a WCOJ algorithm blindly on Q (*i.e.*, without any decompositions). The key takeaway is that combinations of Yannakakis's algorithm and WCOJ algorithms have been used in several work to derive more nuanced worst-case run times than the AGM bound of the query. However, we are unaware of any system that implements Yannakakis's algorithm when evaluating join queries. This is likely due to the fact that the 2 round of semi-joins is a large overhead despite the algorithmic guarantees that it

gives. In addition, query plans come with many other operators that intermix with join operators and executing a semijoin plan on general plans may be more challenging than executing them on join-only plans. Systems instead optimistically evaluate the joins bottom up assuming tuples will not be dangling. An empirical analyses of whether there are queries where semi-join reductions would be beneficial is an interesting research question.

Finally, there have been other join algorithms that aim to achieve stronger notions of optimality than worst-case optimality. Ngo *et al.* have introduced a join algorithm called Minesweeper and its generalization Tetris that achieve "beyond worst-case optimality" [95, 136]. The runtimes of these algorithms are characterized by a notion called certificate complexity, which captures the size of the "implicit proof" a join algorithm generates to guarantee that its output is correct. A detailed coverage of these algorithms are beyond the scope of this thesis. One interesting aspect of these algorithms is that instead of working on actual values in tuples, these algorithms perform computations on the gaps between the values (assume the domain of each attribute is integers). As such, it is unclear how to implement these algorithms inside actual DBMSs and the only work to date that has implemented these algorithms [140] is a standalone implementation. Whether these algorithms can be integrated into actual DBMSs is an interesting future work direction.

6.1.2 Other Systems Implementations of WCOJ Algorithms

Implementations in DBMSs: There has been several work other than EH and our implementation on GraphflowDB that have implemented WCOJ algorithms and evaluated their performance. Leapfrog TrieJoin algorithm, which we discussed above has been implemented in the LogicBlox system. Leapfrog TrieJoin algorithm is designed for evaluating joins on general relations and assumes that the inputs are indexed according to a global attribute ordering. When evaluating a query Q, the algorithm picks a JAO that's consistent with this global ordering. The rest of the algorithm is similar to GenericJoin and based on multiway intersections. Unlike EH and GraphflowDB implementation, Leapfrog TrieJoin algorithm does not use any binary join operators [179]. That implementation only generates WCOJ plans.

The Umbra RDBMS [130] implemented a WCOJ-style algorithms [64]. The premise of this work is that all prior implementations, including the work in this thesis, assume that the inputs are indexed in some sort order, which can be expensive in practice. Recall that because GraphflowDB assumes binary relations, we only needed to index the relations twice. However, for arbitrary n-ary relations, a system would need to order in possibly $n!$ different ways to be able to support all possible JAOs, which would be prohibitively expensive to maintain. Instead, Umbra describes a new multiway join operator that computes the trie indexes required by its WCOJ algorithms on the fly when a query is issued. These indices are built as nested hash tables,

where each level corresponds to exactly one join key attribute and the leaf nodes are sorted using a linked list structure. The system has a new HashTrieJoin operator that takes in these hash-trie indexes of $k \geq 2$ relations $R_1, ..., R_k$ that are being joined in the query. This is a very practical approach for integrating WCOJ algorithms into a general RDBMS. Umbra's approach can also mix this multiway join operator with other binary join operators in hybrid plans. Umbra uses a rule-based optimizer to improve the default binary join plan of the system with a modified join plan that replaces sequences of binary join algorithms with the system's multiway join operator.

However, Umbra's approach has the clear disadvantage of having to generate indices on the fly, which GDBMSs can avoid as they, primarily, process joins over binary relations. So, the approach proposed in this thesis can be more desirable for GDBMSs. Kùzu [62, 63], which is the successor system to GraphflowDB and integrates many of the query processing techniques described in this thesis, drops the assumption of GraphflowDB that the input adjacency lists are sorted according to neighbor IDs. Instead, Kùzu's join operator that performs multiway intersections reads unsorted adjacency lists from storage, sorts them, and indexes them once in hash tables. Then for each prefix tuple, these sorted adjacency lists are read from hash tables and intersected. EH, GraphflowDB, Umbra, and Kúzu are currently in the full scope of design points that have been explored to implement WCOJ algorithms and mix them with binary join algorithms.

Implementations in Distributed Systems: All of the implementations described so far have been implemented in single node shared memory systems. We briefly cover work in the distributed setting. These work assume a different setting, are not implemented in actual DBMSs, and aim to optimize distributed resources, such as the amount of communication and number of rounds of communication an algorithm takes. Ammar *et al.* [20] have implemented a version of GenericJoin they call *BiGJoin* on top of the Timely Dataflow system [125, 175]. One of the primary concerns in distributed query evaluation is to reduce the memory/load overhead of any single machine, while minimizing communication. Ammar *et al.* describe a batching technique to control the memory use. To minimize communication, the authors describe a technique to pick *intersection plans* to plan in which order the multiway intersections should be performed to extend each prefix tuple. For a prefix tuple t, if multiple adjacency lists that are in different machines need intersecting, these plans aim to shuffle the smaller lists around to ensure minimal communication.

Chu *et al.* [43] have described another distributed join algorithm that uses a WCOJ algorithm as a primitive. Unlike the join algorithm of Ammar *et al.* , Chu *et al.* 's algorithm aims to evaluate the entire join algorithm in a single round of communication[2]. The algorithm mixes

[2] Ammar *et al.* 's algorithm's round requirements is commensurate with the AGM bound of the query divided by a batching factor.

the Hypercube algorithm [17, 27], which is a communication optimal generic one-round shuffling algorithm for join queries, with an optimized Leapfrog TrieJoin algorithm called Tributary join. Tributary join indexes the shuffled tuples on the fly, similar to Umbra, and runs Leapfrog TrieJoin algorithm with a specific attribute ordering. The attribute ordering is tailored for the case when the tuples come from different relations with different data distributions. Tributary join's approach to optimizing attribute ordering is not suitable for graph pattern queries, which are generally self join queries where tuples come from the same relation.

Incremental Versions of WCOJ Algorithms: Several papers use incremental versions of WCOJ algorithms to maintain the results of join queries upon updates to the base tables. The earliest work here was by Veldhuizen who designed an incremental Leapfrog TrieJoin algorithm [180] to maintain join queries. Broadly, this algorithm keeps the computational trace of the Leapfrog TrieJoin algorithm in indices which are then used and fixed upon updates. This index can be as large as the AGM bound of the query that is being maintained.

Ammar *et al.* as part of implementing BiGJoin [20], proposed an incremental version of GenericJoin. They adopt the incremental view maintenance (IVM) approach of Blakeley *et al.* [30] that decomposes a join query with n relations into n *delta queries*. For example, for a join query $R(a, b), S(b, c)$, one of the delta queries would be $\Delta R(a, b), S(b, c)$, where $\Delta R(a, b)$ is a small relation that contains only the updates to R. Each of the n delta queries in the decomposition contains a small delta relation, so each delta query can be evaluated efficiently. Upon updates to a system, Blakeley *et al.* have shown that the union of the set of delta queries of a query contains the new tuples that should be added to and the tuples that should be deleted from the latest output of the query. Ammar *et al.* have proposed a new IVM algorithm called Delta-GenericJoin that decomposes join queries into delta queries and evaluates each one using GenericJoin. Unlike Veldhuizen's IVM algorithm, Delta-GenericJoin does not require any additional memory beyond the temporary memory required during computing the delta queries. Ammar *et al.* have also shown that upon insertion-only workloads, this approach is worst-case optimal in the sense that at any point in time t, Delta-GenericJoin's worst-case runtime is bounded by the AGM bound of the query at time t. They have also implemented a distributed version of Delta-Generic Join that they call Delta-BiGJoin.

Kankanamge, Salihoglu, and the author of this thesis have integrated Delta-GenericJoin of Ammar *et al.* to a continuous query processor that they developed for GraphflowDB [92, 118]. The goal of the continuous query processor is to facilitate graph pattern triggers to support applications that need to detect the emergence and deletion of subgraph patterns they register in GraphflowDB and perform an action for each new or deleted pattern. An example application is a fraud detection application that needs to detect the emergence of cycles in financial transaction network, which are assumed to be indicative of fraudulent money transfers, and alert a fraud agent. They tackle the problem of sharing common computations across multiple regis-

tered queries in the system (specifically their delta queries) and to optimize JAOs of delta queries to maximize computation sharing.

6.1.3 Cardinality Estimation

Recall from Chapter 3 that GraphflowDB uses a subgraph catalogue-based cardinality estimator. Our catalogue is closely related to Markov tables [13], and MD- and Pattern-tree summaries techniques [115]. Similar to our catalogue, both of these techniques store information about small-size subgraphs to make cardinality estimates for larger subgraphs. Markov tables were introduced to estimate cardinalities of paths in XML trees and store exact cardinalities of small size paths to estimate longer paths. MD- and Pattern-tree techniques store exact cardinalities of small-size acyclic patterns, and are used to estimate the cardinalities of larger subgraphs (acyclic and cyclic) in general graphs. These techniques are limited to cardinality estimation and store only acyclic patterns. In contrast, our catalogue stores information about acyclic and cyclic patterns and is used for both cardinality and i-cost estimation. In addition to selectivity (μ) estimates that are used for cardinality estimation, we store information about the sizes of the adjacency lists (the $|A|$ values), which allows our optimizer to differentiate between WCOJ plans that generate the same number of intermediate results, so have same cardinality estimates, but incur different i-costs. Storing cyclic patterns in the catalogue allow us to make accurate estimates for cyclic queries.

A very interesting application of the theory of WCOJ algorithms has been to use the worst-case optimal bounds of queries for cardinality estimation. These estimators, introduced by Cai *et al.* [36, 50], are called *pessimistic estimators* as they are guaranteed to never make underestimates. As such, they address the well known problem of existing cardinality estimators, which suffer from underestimations [110]. These estimators are also summary-based and use a subset of the inequalities in the linear programs that describe the worst-case optimal bounds, *e.g.*, the AGM bound.

At first sight, our cardinality estimation technique can seem very different from the linear program-based pessimistic estimators. Interestingly, Chen *et al.* [38] have recently shown that both our catalogue-based estimator (and other catalogue-based estimators, such as Markov tables and MD- and Pattern-tree summary estimators) and the pessimistic estimators can be seen as instances of a more general estimator based on finding paths in a *cardinality estimation graph* (CEG). The nodes of a CEG are subqueries and an edge between subquery S and S' has a weight describing the factor by which to grow the estimate of the number of S's to estimate the number of S''s. Chen *et al.* have shown that all of these estimators can be seen as using similar (though not the same) CEGs, except that our catalogue-based estimator (and other catalogue-

based ones) use average degrees of values as edge weights in CEGs, while pessimistic estimators use maximum degrees.

6.2 Factorized Representations

The core idea of factorization is to detect multi-valued dependencies in the intermediate results that are generated when processing join queries. The conditional dependencies between attributes describe these multi-valued dependencies and ultimately lead to redundancy if one represents relationships using flat tuples (specifically Cartesian products of sets of tuples). Factorized representations exploit these dependencies to represent these intermediate relations in compressed formats. Multi-valued dependencies have been studied in other context prior to the work on factorized relationships, such as database design [57]. Factorized representations introduced by Olteanu *et al.* [144, 145], describe the principles of how to exploit multi-valued dependencies when processing queries that contain m-n joins. In the rest, we cover work related to factorized query processing in DBMSs. Recent work by Nikolic and Olteanu [143] have shown how to use factorization to speed up various related tasks, including incremental maintenance of queries with group by and aggregations and joins, matrix chain multiplication and gradient descent computations. They show that these sets of tasks can commonly be seen as maintaining view trees whose leaves are views that perform partial computations, *e.g.*, partial group by and aggregates, and the internal nodes are joins of views with further partial computations. They show how to achieve similar benefits from factoring out common computations in different branches of these views and how to maintain these views upon updates. These more advanced applications of factorization relate less to this thesis and are not covered in detail.

The research described in Chapters 4 and 5 of this thesis tackled how to modify modern vectorized query processor architectures to benefit from factorized representations. The closest to our work are the FDB system [26] and WireFrame [14, 67] which also described a query processor design that uses factorized representations. We covered FDB in detail in Chapter 4 and do not cover it here again. There has been further work on the FDB system beyond what we covered in Chapter 4. This further work studied query processing techniques for aggregations and ordering [25]. Similar to the original work on FDB, this work extends FDB with operators that take as input entire f-representations and output f-representations. Specifically this work discusses how to push group by and aggregation operations through the branches of f-representations or break the aggregations into partial aggregations to speed up aggregations over f-representations. In addition, this work characterizes which sort orders can be quickly enumerated from a given f-representation and how to transform one f-representation to another one (complying to another f-tree) to enable fast enumeration of a specific sort order. As we discussed earlier, because this

work also assumes that the entire intermediate relations are stored in tries, this type of processing seems to require major changes to traditional query processors of existing DBMSs. We note however that it is an interesting research direction to better understand how one can speed up aggregations and ordering in factorized vector execution approach of this thesis.

Answer Graph [14, 67] is a recent system that extends PostgreSQL's query processor for a join-only subset of SPARQL (*i.e.*, without projections) that performs a two-stage query evaluation for acyclic queries. The first stage is a full semi-join reduction, similar to Yannakakis's algorithm that identify only and all of the edges that participate in the final output. This is done by performing a sequence of "forward extensions" according to a join order that is picked by a traditional cost-based optimizer which are followed by cascading deletes in case a particular data node, say $a_i = 5$ does not extend to the next attribute a_{i+1}. After this step, a second stage called the embedding generation stage generates a set of flat tuples by performing a left-deep join plan. The result of the first phase of Answer Graph is similar to our d-representations. However, the following enumeration phase flattens all results. The authors also describe an envisioned but not implemented version of semi-join reduction for cyclic queries, which is based on a more complex cascading logic. In contrast to this approach, we do not need to handle cascading deletes as tuples that are not in the output never arrive at our DGroup operator. We also evaluate a larger class of queries and produce d-representations as outputs.

Finally we cover the work by Xirogiannopoulos and Deshpande on the GraphGen system [183]. GraphGen is a system to extract large graphs from RDBMSs. These extracted graphs are results of join queries with duplicate elimination. GraphGen is a system that is built as a layer over an RDBMS and has its own expressive graph extraction domain specific language. For example users can extract a co-authors graph from a database of papers and their authors stored in an RDBMS. The premise of the work is that since edges (*e.g.*, co-authorship) between entities can be defined as possible (growing) joins between multiple records (authors), graphs that users might want to analyze can be orders of magnitude larger than the raw records (*e.g.*, papers and their authors). Therefore extracting these large graphs from RDBMSs and then ingesting them into a graph analytics system, say to run a PageRank computation, can be very expensive. Instead, GraphGen stores these large graphs in compressed formats in its own specialized storage in memory. Then users can program GraphGen's vertex-centric analytics API to implement algorithms such as PageRank, which will run over these compressed graphs. The GraphGen paper offers multiple compressed formats, one of which, called C-Dup, similar to d-representations or the Answer Graph approach [14]. For example, the C-Dup representation for the co-authorship example stores the subgraph of authors and papers that contains the set of co-authors and the papers that make these co-authors (instead of expanding each co-authorship edge between authors). This can be seen as the d-representation of the join result that produces the co-authorship

graph. When users access the co-authorship edges of any node in GraphGen's graph analytics API, GraphGen implicitly recomputes the part of the join from these compressed representations. One specific challenge this paper tackles is that often in these graph extraction queries, users would remove duplicate edges. If one were to blindly traverse the C-Dup representation, same edges could be produced multiple times (*e.g.*, if two authors are co-authors through multiple papers). The authors describe alternative compression techniques that trade off how much memory is stored vs how much work is done to remove duplicates. Although these representations relate to factorized representations we covered in this thesis, the problem GranGen addresses, which is to compress large extracted graphs, is different than our work and the other work we covered on factorization, which is efficient m-n join processing in DBMSs.

6.3 Subgraph Matching Algorithms

The queries we focused on in this thesis are equivalent to the problem of subgraph matching, or subgraph isomorphism, that has been extensively studied by many different communities in computer science. This is the problem of finding instances of a given query graph pattern in a larger data graph. We briefly cover this literature here.

Many of the earlier subgraph matching algorithms are based on Ullmann's branch and bound or backtracking method [177]. In the terminology used in this thesis, the algorithm conceptually performs an attribute-at-a-time matching using a JAO. This algorithm has been improved with different techniques to pick good JAOs and filter partial matches, often focusing on queries with labels [47, 48, 165]. Several recent algorithms perform preprocessing to find *candidate vertex sets* (the set of possible data vertices for each query vertex), build an auxiliary data structure for these sets and finally pick a JAO for the evaluation. Such algorithms include Turbo$_{ISO}$ [78], CFL [29], CECI [28], and DP-iso [76]. Each of these algorithms include optimizations on the auxiliary data structure as well as query processing. Turbo$_{ISO}$, for example, proposes to merge similar query vertices (same label and neighbours) to minimize the number of partial matches and once the merged and smaller query is evaluated, perform a Cartesian product to enumerate the final outputs. CFL, which we covered extensively in Chapter 3 decomposes the query into a dense subgraph and a forest, and processes the dense subgraph first to reduce the number of partial matches. CFL also uses an index called *compact path index (CPI)* which estimates the number of matches for each root-to-leaf query path in the query and is used to enumerate the matches as well. CECI and DP-iso rely on an auxiliary data structure which maintains edges between candidates and also rely on multiway intersections when finding candidate sets. Each of the algorithms has its own optimization, *e.g.*, CECI divides the data graph into multiple embedding clusters for parallel processing while DP-iso relies on an adaptive QVO selection and a pruning

technique called *pruning by failing sets* which are partial matches with no possible extensions in the data graph.

Interestingly some of the optimizations that are described in this literature are akin to the core algorithmic ideas of WCOJ algorithms and factorization. As we mentioned, many, but not all, of these works match subgraphs query vertex (or attribute)-at-a-time similar to WCOJ algorithms. CFL and Turbo$_{ISO}$ have optimizations to delay Cartesian products that appear when matching different parts of queries, which is akin to factorization. Yet, the work on the theory of WCOJ and factorization explain the advantages of these optimizations more formally. In addition because these theories are developed in the context of query processing in relational DBMSs, they give insights about how they can be integrated into DBMSs. This is not a concern for the many of the work on subgraph matching where the algorithms are not implemented in actual DBMSs. Instead they are separate standalone codebases. That is why it is not clear how to decompose some of the algorithmic steps of CFL [29] or CECI [28] into operator-based query plans.

At the same time, some of the techniques from this literature seems complementary and can be integrated to further improve existing query processors. For example, some works index different structures in input graphs, such as frequent paths, trees, or triangles, to speed up query evaluation [184, 190], akin to maintaining a view of small-size joins. Similarly, some of these algorithms [78] find symmetries in the graph pattern and compute these parts once. Symmetry detection can also be directly applied in the query processors of DBMSs by evaluating a subquery once and reusing the results of this subquery in different branches of a join tree. These are practical optimizations that are complementary to the techniques we integrated into the GraphflowDB query processor.

6.4 Other Systems Approaches for Efficient Graph Query Processing in DBMSs

We next cover several other work that has integrated techniques other than WCOJ algorithms and factorization into DBMSs to make query processing more efficient on graph workloads. There are several systems work, mainly from commercial companies, that build separate layers on top of an existing RDBMSs to support a graph model, such as the property graph model or RDF, without modifying the core RDBMS's query processor [157, 170, 173]. There has also been some work that extends RDBMS to do batch graph analytics, such as a recursive PageRank computation [58, 87, 88]. We do not cover this literature here in detail. The primary focus of this line of work is to tackle the problem of how to map their target computations to SQL and use the RDBMSs as a core engine. They do not modify the query processors of the underlying RDBMS.

Instead our work proposes different approaches to modify the query processors of RDBMSs to make them more efficient on m-n joins.

GR-Fusion [79, 80] is a multi-model system that allows users to define graphs over relational systems. GR-Fusion is developed on top of the VoltDB RDBMS [168] and extends the SQL supported by VoltDB to contain graph specific constructs to: (i) define graphs, such as "Create Graph"; and (ii) query these graphs, such as "Path" or "ShortestPath" to generate paths from these graphs. GR-Fusion's approach to evaluating these path queries has two key ingredients. First, GR-Fusion indexes the edges in the constructed graph in adjacency list indices, similar to the adjacency list index of GraphflowDB. The adjacency list indices store record identifiers (RIDs) of the source and destination "node" records they index. Second, GR-Fusion adopts a dual query processor approach where the graph constructs of the queries are evaluated in a separate plan that consists of graph-optimized operators, such as "ScanPath", and the results of these are given to VoltDB's default processor. The authors implement graph algorithms, such as Dijkstra's algorithm to find shortest paths or the BFS graph traversal algorithm to enumerate paths using the native adjacency list indices to find paths. These traversals are akin to left-deep join plans that use `INLJ`-like operators. GR-Fusion does not implement WCOJ algorithm or factorization. In addition, its dual processor approach is different from our approach where we aim to develop designs that can be directly integrated into the query processors of a DBMS. Although we have implemented our techniques in the context of a prototype GDBMS, the core design principles of our processor is very similar to traditional vector-based query processors of analytical RDBMSs. GQ-Fast [113] is another system that follows the design of GR-Fusion. Although GQ-Fast does not extend SQL with graph-specific constructs, it contains means to create join indices that index RIDs that are stored in adjacency lists. For several query templates the authors identify, GQ-Fast uses a dual query processor that uses these indices to evaluate joins using BFS traversals.

GrainDB [86] is a recent work that aims to extend the DuckDB RDBMS [152] with *predefined joins*, which refers to joins that use system-level RIDs. Virtually all GDBMSs evaluate their joins using system level node IDs, instead of equality conditions on, say primary keys of records. We took this style of joins as given in this thesis[3]. However, RDBMSs evaluate joins by explicit comparisons of values and without using a join index akin to the adjacency list indices of GDBMSs. The premise of GRainDB is that the lack of predefined joins is a disadvantage for RDBMS. At the same time, many GDBMSs use `INLJ`-like algorithms as their core join operator, which the authors show is not robust compared to plans that use bushy hash joins, which are common in RDBMSs. The goal of the GRainDB work is to benefit from both of these advantages.

[3]We note that GR-Fusion and GQ-Fast also effectively implement predefined joins when they run their BFS traversal algorithms to enumerate paths

The authors tackle the problem of extending DuckDB with a join index they call RID index for primary foreign key joins (so users do not model parts of their databases as graphs), which is similar to the indices of GR-Fusion and GQ-Fast. Then they describe a modified hash join operators called `SJoinIdx` that uses these join indices. One additional optimization GRainDB introduces is the following. During the build phase of `SJoinIdx`, as the operator constructs a hash table, it also records the RIDs that have been hashed as keys. These RIDs identify the records that need to be scanned from the probe side. `SJoinIdx` passes this information sideways from its build side to the probe side scans to ensure that only the "node" records that will successfully join are scanned. They show the combination of a RID index and sideways information passing can yield more efficient and robust join plans compared to vanilla DuckDB and some of the existing native GDBMSs. Yet they report that these plans are still generally slower than GraphflowDB, which further benefits from WCOJ algorithms and factorization. Since GraphflowDB is an in-memory system, it is likely that our queries would not benefit as much from sideways information passing compared to disk-based systems like DuckDB. GraphflowDB's successor Kúzu, which is disk-based has also adopted GRainDB's sideways information passing optimization. Finally, in a follow up demo paper [85], the authors of GRainDB extend DuckDB to an actual multimodal system, where users can indicate that parts of their tables represents nodes and edges in a property graph. This extension directly indexes the edge tables in RID indices. They further extend SQL with constructs to seamlessly query the modeled graphs and tables together. All of these queries compile to plans that DuckDB generates that are extended with the `SJoinIdx` operator that uses the RID index.

Finally, DuckPGQ [171] is an extension to DuckDB that implements SQL/PGQ [4], which is a new SQL standard to support property graphs. In SQL/PGQ, similar to the approaches described above for GR-Fusion or GRainDB's demo paper, users can model parts of their database as a property graph and query them in more graph-specific syntax. In terms of query processing, DuckPGQ focuses on recursive computations of Kleene star to find arbitrary-length paths (*e.g.*, $p{=}(a : Person){-}/know^*/{\rightarrow}(b : Person)$) and shortest paths. Specifically, DuckDB extends DuckPGQ processor with the BellmanFord and multi-source BFS algorithm by Then *et al.* [172] and compiles SQL/PGQ queries that contain these constructs into plans that use these algorithms. We have not focused on advanced recursive algorithms in this thesis. In contrast, DuckPGQ has not focused on WCOJ algorithms and factorization and refers to these two techniques as future work.

Chapter 7

Conclusion

What we referred to as graph workloads in this thesis, *i.e.*, those containing joins over m-n relationships, are integral to a wide range of application domains. These domains not only include applications that process classic network-structured datasets, *e.g.*, social networks, protein-protein interaction networks, and financial and transactional networks, but also applications that are naturally thought of as "relational" such as those containing orders, parts, and manufacturers. Such traditional datasets have been found to be modelled and queried as graphs in practice [159, 161]. Practitioners, both in industry and academic settings, use graph workloads on very large datasets and report scalability on these workloads as the paramount challenge to overcome for existing DBMSs [159, 161].

The primary challenge when evaluating graph workloads is that the m-n joins in the queries of these workloads can generate very large intermediate results. Although traditional join optimizers aim to pick plans whose intermediate results generate as small relations as possible, the plans of these relations use traditional binary join operators that process flat tuples, which can be suboptimal under m-n joins. This is the gap that has motivated the work in this thesis.

7.1 Contributions

In this thesis, we explored the adoption of worst-case optimal join algorithms and factorized representations, two novel techniques introduced by the database theory community in DBMSs. We asked the following research question: *how should the query processor and optimizer of an analytical DBMS change to adopt these techniques without deviating from standard architectural designs, such as pipelining and vector-based execution?* The thesis described design and

implementations of operators, changes to intermediate tuple representations, as well as query optimizers that we believe give a blueprint for implementing these techniques in a practical manner in current DBMSs.

Our first contribution when adopting worst-case optimal joins introduced a new multi-way join operator and proposed a query optimizer to choose a join attribute ordering when evaluating queries. Prior techniques from EmptyHeaded and CTJ relied either on the query-specific notions of cyclicity, such as widths of GHDs of queries, to pick orders or picked orders arbitrarily, *e.g.*, based on lexicographic order of the attributes in the query. In contrast, we used the traditional approach of using a dynamic programming-based optimizer with a novel cost model called intersection cost. Our plan space seamlessly mixes WCOJ plans with binary joins that introduce decomposition and therefore bushy plans with WCOJ subplans. Parts of our plan space were not part of prior approaches. As part of this contribution, we also provided extensive experiments to study which types of plans, WCOJ, binary join, or hybrid were suitable for which types of queries. We found that the suitability of plans depended on the query structure and on the dataset characteristics. We also introduced an adaptive technique to make the attribute ordering chosen by our optimizer more robust.

The multi-way intersection-based join evaluation of WCOJs can avoid generating some large intermediate results that binary join plans might generate. However, WCOJ plans are equivalent to left-deep binary join plans on acyclic queries and do not offer any intermediate-result reduction compared to binary join plans. In fact, on some acyclic queries, the results can be inherently large, so no matter what type of join algorithm is used, the number of tuples in these intermediate results cannot be reduced. However, the theory of factorization has shown that these results can be compressed using the factorized representations called f- and d-representations. While these representations can lead to much smaller sizes, their physical representation is trie-like and differs significantly from that of vector-based intermediate tuple representations. As our second major contribution, we designed and implemented an approach to extend traditional vector-based query processors to use multiple vector groups instead of a single vector group. This design trades off the set of factorized representations that can be supported with ease of integration and preserving the traditional architecture of analytical query processors. Specifically, this design supports only those factorized representations in which any Cartesian products with a set that contains more than one value cannot contain nested Cartesian products.

Factorized vector execution adopts f-representations, which can contain duplicate sub-relations that correspond to results of re-computing the same subquery. D-representations, which extend f-representations with definitions, is based on the idea of caching such sub-relations once and using them with pointers. In our third and final contribution, we introduce subquery caching and reuse by using d-representations which have not been used by any prior prototype or commercial DBMS. We show how caching and reuse can be implemented in vectorized query processors

without changing the core operators of query processors of DBMSs, *e.g.*, adopting tries as intermediate relation representation. Our approach is based on using nested hash tables, a new operator called `D-Group` that caches sub-relations, and adding new branches to join operators to skip parts of query pipelines when the sub-relations that would be computed by those sub-plans have already been computed and cached.

We also evaluated the benefits of our approaches on micro-benchmarks and end-to-end benchmarks and demonstrated that they lead to major performance benefits over existing systems and approaches on graph workloads. We believe the designs proposed in this thesis are both practical and easy to adopt and act as a stepping stone towards more scalable and efficient DBMSs on graph workloads in the future. At the time of writing, there have already been several other work that has adopted these designs in part. For example, Graphscope [59] adopts our WCOJ query processing and optimization techniques. Specifically, Graphscope uses our plan space and cardinality estimation technique with minor changes. GraphflowDB also acts as a predecessor to the Kúzu DBMS [63] developed at UWaterloo. Kúzu adopts our WCOJ query processing approach and factorized vector execution.

7.2 Future Work

The work presented in this thesis also has many limitations that could serve as directions for future research. We conclude this thesis by overviewing two of these directions.

Factorization-aware Query Optimization: Although we have studied how to generate plans that use WCOJ algorithms and binary joins in Chapter 3, we have not rigorously studied how to optimize queries considering factorization and arbitrary queries in Chapters 4 and 5. In the approaches described in both of these chapters, we either do not take factorization into account or use a rule-based approach when generating plans. Prior work describes query-specific (and database-independent) techniques to generate query plans, *e.g.*, f-trees, which are likely to not be robust in practice. An important question is: *how can one design a factorization-aware query optimizer?* One interesting approach is to consider ways to extend the traditional cost- and DP-based optimizers to take factorization into account, *e.g.*, how much caching can be expected when using a d-representation. Such extensions can be adopted more easily than designing completely novel optimizers.

Factorized aggregations with vectorized query processors: A second limitation of this thesis is that we focused solely on select-join-project queries, with the exception of simple count(*) aggregations. Perhaps the biggest performance advantages using factorization would come from queries with group-by and aggregations. We have not studied the principles of how to perform

efficient group by and aggregations in our factorized vector execution design. There are several prior work on aggregations over factorized representations [25, 143]. Specifically the work of Bakibayev *et al.* on FDB system [25] considers aggregation queries in a DBMS but this work assumes that intermediate relations are stored in fully materialized tries.

References

[1] AliPay. https://global.alipay.com/platform/site/ihome.

[2] DBToaster Release. https://dbtoaster.github.io/download.html.

[3] Graphflow. https://graphflow.github.io/.

[4] SQL/PGQ ISO Standard. https://www.iso.org/standard/79473.html/.

[5] The On-line Encyclopedia of Integer Sequences. https://oeis.org/A086345.

[6] Turboiso code. https://github.com/bookug/TurboISO.

[7] Twitter. https://twitter.com/.

[8] The WebGraph Framework I: Compression Techniques. 2004.

[9] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2006*, pages 671–682, 2006.

[10] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, Panos Kalnis, and Nikos Mamoulis. Spartex: A vertex-centric framework for RDF data analytics. *PVLDB*, 2015.

[11] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. *TODS*, 2017.

[12] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the Symposium on Principles of Database Systems*, 2017.

[13] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. *PVLDB*, 2001.

[14] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Stanley Clark, and Mark H. Chignell. Answer Graph: Factorization Matters in Large Graphs. In *Proceedings of the International Conference on Extending Database Technology*, 2021.

[15] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *TKDE*, 2011.

[16] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround distributed join algorithm. In *International Conference on Database Theory*, 2017.

[17] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9), 2011.

[18] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 2012.

[19] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *PVLDB*, 2012.

[20] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *PVLDB*, 2018.

[21] Renzo Angles. The property graph database model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, volume 2100 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[22] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. *SIGMOD*, 2015.

[23] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *TODS*, 1976.

[24] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. In *Symposium on Foundations of Computer Science*, 2008.

[25] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. Aggregation and Ordering in Factorised Databases. *Proceedings of the VLDB Endowment*, 6(14), 2013.

[26] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. Fdb: A query engine for factorised relational databases. *PVLDB*, 2012.

[27] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication Steps for Parallel Query Processing. *Journal of the ACM*, 64(6), 2017.

[28] Bibek Bhattarai, Hang Liu, and H. Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. *SIGMOD*, 2019.

[29] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. *SIGMOD*, 2016.

[30] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. *SIGMOD Record*, 1986.

[31] Arezo Bodaghi and Babak Teimourpour. *Automobile Insurance Fraud Detection Using Social Network Analysis*. 2018.

[32] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. *WWW*, 2011.

[33] Peter Boncz. *Monet: A Next-Generation Database Kernel for Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, 2002.

[34] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[35] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation- vs. Index-Based XML Multi-Query Processing. *ICDE*, 2003.

[36] Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2019.

[37] Surajit Chaudhuri. An overview of query optimization in relational systems. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43. ACM Press, 1998.

[38] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. Accurate Summary-Based Cardinality Estimation through the Lens of Cardinality Estimation Graphs. *Proceedings of the VLDB Endowment*, 15(8), 2022.

[39] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *SIGMOD*, 2000.

[40] Rada Chirkova and Jun Yang. Materialized views. *Found. Trends Databases*, 4(4):295–405, 2012.

[41] S. Choudhury. *Subgraph Search for Dynamic Graphs*. PhD thesis, Washington State University, 2014. https://research.libraries.wsu.edu/xmlui/handle/2376/5114.

[42] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. A Selectivity Based Approach to Continuous Pattern Detection in Streaming Graphs. *EDBT*, 2015.

[43] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. *SIGMOD*, 2015.

[44] Sophie Cluet and Guido Moerkotte. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. 1995.

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

[46] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty Years Of Graph Matching In Pattern Recognition. *IJPRAI*, 2004.

[47] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance Evaluation of the VF Graph Matching Algorithm. *ICIAP*, 1999.

[48] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)graph Isomorphism Algorithm for Matching Large Graphs. *TPAMI*, 2004.

[49] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. *PODS*, 2001.

[50] Kyle B. Deeds, Dan Suciu, and Magdalena Balazinska. SafeBound: A Practical System for Generating Cardinality Bounds. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2023.

[51] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 2007.

[52] https://dgraph.io/.

[53] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. YFilter: Efficient and Scalable Filtering of XML Documents. *ICDE*, 2002.

[54] EmptyHeaded Query Planner on Github. https://github.com/HazyResearch/EmptyHeaded/blob/master/query_compiler/src/main/scala/solver/QueryPlanner.scala.

[55] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 2015*, pages 619–630, 2015.

[56] Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *TODS*, 2(3):262–278, 1977.

[57] Ronald Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems*, 2, 1977.

[58] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The Case Against Specialized Graph Analytics Engines. In *Conference on Innovative Data Systems Research, CIDR*, 2015.

[59] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Kai Zeng, Kun Zhao, Jingren Zhou, Diwen Zhu, and Rong Zhu. Graphscope: A unified engine for big graph processing. *Proc. VLDB Endow.*, 14(12):2879–2892, 2021.

[60] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental Graph Pattern Matching. *SIGMOD*, 2011.

[61] Xiyang Feng. DPQP: A D-representation-based Pipelined Factorized Query Processor for Graph Database Management Systems. Master's thesis, University of Waterloo, 2021.

[62] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu Database Management System Source Code. https://github.com/kuzudb/kuzu, November 2022.

[63] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *CIDR*, 2023.

[64] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proceedings of the VLDB Endowment*, 13(12), 2020.

[65] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11), 2020.

[66] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. Continuous Pattern Detection Over Billion-edge Graph Using Distributed Framework. *ICDE*, 2014.

[67] Parke Godfrey, Nikolay Yakovets, Zahid Abul-Basher, and Mark H. Chignell. WIREFRAME: Two-phase, Cost-based Optimization for Conjunctive Regular Path Queries. In *Proceedings of the Alberto Mendelzon International Workshop on Foundations of Data Management and the Web*, 2017.

[68] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[69] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE*, 1994.

[70] Graphflow. Graphflowdb d-representation adoption Source Code. github.com/queryproc/factorized-and-vectorized-query-execution/, 2020.

[71] Graphflow. Graphflowdb factorized vector execution Source Code. github.com/queryproc/columnar-storage-and-list-based-processing-for-graph-dbms/, 2020.

[72] Graphflow. Graphflowdb wcoj adoption Source Code. github.com/queryproc/optimizing-subgraph-queries-combining-binary-and-worst-case-optimal-j 2020.

[73] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. WTF: the who to follow service at twitter. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, pages 505–514. International World Wide Web Conferences Steering Committee / ACM, 2013.

[74] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proc. VLDB Endow.*, 7(13):1379–1380, 2014.

[75] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. Columnar storage and list-based processing for graph database management systems. *Proc. VLDB Endow.*, 14(11):2491–2504, 2021.

[76] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. *SIGMOD*, 2019.

[77] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding Up Set Intersections in Graph Algorithms Using SIMD Instructions. *SIGMOD*, 2018.

[78] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. *SIGMOD*, 2013.

[79] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. Extending In-Memory Relational Database Engines with Native Graph Support. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2018.

[80] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. GRFusion: Graphs as First-Class Citizens in Main-Memory Relational Database Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2018.

[81] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. Rule-based Multi-query Optimization. *EDBT*, 2009.

[82] Hong, Mingsheng and Demers, Alan J. and Gehrke, Johannes E. and Koch, Christoph and Riedewald, Mirek and White, Walker M. Massively Multi-query Join Processing in Publish/Subscribe Systems. *SIGMOD*, 2007.

[83] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[84] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. *SIGMOD*, 2017.

[85] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. GRainDB: A Relational-core Graph-Relational DBMS. In *Conference on Innovative Data Systems Research*, 2022.

[86] Guodong Jin and Semih Salihoglu. Making RDBMSs Efficient on Graph Workloads through Predefined Joins. *Proceedings of the VLDB Endowment*, 15(5), 2022.

[87] Alekh Jindal, Samuel Madden, Malu Castellanos, and Meichun Hsu. Graph Analytics Using Vertica Relational Database. In *International Conference on Big Data*, 2015.

[88] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica: Your relational friend for graph analytics! *Proceedings of the VLDB Endowment*, 7(13), 2014.

[89] Manas Joglekar and Christopher Ré. It's All a Matter of Degree. *Theory of Computing Systems*, 62(4), 2018.

[90] Cliff Joslyn, Sutanay Choudhury, David Haglin, Bill Howe, Bill Nickless, and Bryan Olsen. Massive Scale Cyber Traffic Analysis: A Driver for Graph Database Research. *GRADES*, 2013.

[91] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. Flexible caching in trie joins. *EDBT*, 2017.

[92] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. *SIGMOD*, 2017.

[93] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. *SIGMOD*, 2017.

[94] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, 2018.

[95] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Transactions on Database Systems*, 41(4), nov 2016.

[96] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. Taming Subgraph Isomorphism for RDF Query Processing. *PVLDB*, 2015.

[97] Kim, Kyoungmin and Seo, In and Han, Wook-Shin and Lee, Jeong-Hoon and Hong, Sung-pack and Chafi, Hassan and Shin, Hyungyu and Jeong, Geonhwa. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. *SIGMOD*, 2018.

[98] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. Dbtoaster: Higher-order delta processing fordynamic, frequently fresh views. *VLDBJ*, 2014.

[99] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. *SIGMOD*, 2006.

[100] Jérôme Kunegis. KONECT: the koblenz network collection. In *22nd International World Wide Web Conference, WWW 2013*, pages 1343–1350, 2013.

[101] Jérôme Kunegis. Wikipedia Dynamic (de), (Konect). `http://konect.cc/networks/link-dynamic-dewiki/`, 2021. Last Accessed July 25, 2021.

[102] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? *WWW*, 2010.

[103] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.*, 2016.

[104] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable Multi-query Optimization for SPARQL. *ICDE*, 2012.

[105] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *PVLDB*, 2012.

[106] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. 2014.

[107] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 2015.

[108] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[109] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality Estimation Done Right: Index-Based Join Sampling. *CIDR*, 2017.

[110] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDBJ*, 2018.

[111] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. Simd compression and the intersection of sorted integers. *Softw. Pract. Exper.*, 2016.

[112] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[113] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast In-Memory SQL Analytics on Typed Graphs. *Proceedings of the VLDB Endowment*, 10(3), 2016.

[114] Longbin Lai and Lu Qin and Xuemin Lin and Ying Zhang and Lijun Chang. Scalable Distributed Subgraph Enumeration. *VLDB*, 2016.

[115] Angela Maduko, Kemafor Anyanwu, Amit Sheth, and Paul Schliekelman. Graph Summaries for Subgraph Frequency Estimation. *The Semantic Web: Research and Applications*, 2008.

[116] Maximum Common Induced Subgraph. https://en.wikipedia.org/wiki/Maximum_common_induced_subgraph.

[117] Memgraph. Memgraph. https://memgraph.com/, 2020. Last Accessed July 25, 2021.

[118] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. Optimizing one-time and continuous subgraph queries using worst-case optimal joins. *ACM Trans. Database Syst.*, 46(2):6:1–6:45, 2021.

[119] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[120] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *PVLDB*, 2019.

[121] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the flickr social network. In *Proceedings of the first Workshop on Online Social Networks, WOSN 2008*, pages 25–30, 2008.

[122] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD*, 2001.

[123] Guido Moerkotte and Thomas Neumann. Dynamic Programming Strikes Back. *SIGMOD*, 2008.

[124] MonetDB. MonetDB source code, (Jun2020-SP1). https://github.com/MonetDB/MonetDB/releases/tag/Jun2020_SP1_release, 2020. Last Accessed July 25, 2021.

[125] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. *SOSP*, 2013.

[126] Neo4j. https://neo4j.com/.

[127] Fraud Detection: Discovering Connections with Graph Databases. https://neo4j.com/use-cases/fraud-detection.

[128] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.

[129] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 2011.

[130] Thomas Neumann and Michael J. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[131] Thomas Neumann and Guido Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. *ICDE*, 2011.

[132] Thomas Neumann and Bernhard Radke. Adaptive optimization of very large join queries. *SIGMOD*, 2018.

[133] Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 2010.

[134] M. E. J. Newman. Detecting Community Structure in Networks. *The European Physical Journal B*, 2004.

[135] H. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record*, 2014.

[136] Hung Q. Ngo, Dung T. Nguyen, Christopher Re, and Atri Rudra. Beyond Worst-Case Analysis for Joins with Minesweeper. In *Proceedings of the Symposium on Principles of Database Systems*, 2014.

[137] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case Optimal Join Algorithms. *PODS*, 2012.

[138] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *CoRR*, abs/1203.1952, 2012.

[139] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record*, 2014.

[140] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join Processing for Graph Patterns: An Old Dog with New Tricks. In *International Workshop on Graph Data Management Experiences and Systems (GRADES)*, 2015.

[141] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join Processing for Graph Patterns: An Old Dog with New Tricks. *CoRR*, 2015.

[142] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. *CoRR*, 2015.

[143] Milos Nikolic and Dan Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2018.

[144] Dan Olteanu and Jakub Závodný. Factorised Representations of Query Results: Size Bounds and Readability. In *Proceedings of the International Conference on Database Theory*, 2012.

[145] Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *TODS*, 2015.

[146] openCypher. http://www.opencypher.org.

[147] Paraschos Koutris and Semih Salihoglu and Dan Suciu. Algorithmic aspects of parallel data processing. *Foundations and Trends in Databases*, 2018.

[148] Peng Peng, Lei Zou, Lei Chen, Xuemin Lin, and Dongyan Zhao. Answering subgraph queries over massive disk resident graphs. *WWW*, 2016.

[149] Andrea Pugliese, Matthias Bröcheler, V. S. Subrahmanian, and Michael Ovelgönne. Efficient multiview maintenance under insertion in huge social networks. *ACM Trans. Web*, 2014.

[150] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, 2018.

[151] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *PVLDB*, 2018.

[152] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *SIGMOD*, 2019.

[153] Rada Chirkova and Jun Yang. Materialized Views. *Foundations and Trends in Databases*, 2012.

[154] Xuguang Ren and Junhu Wang. Multi-query optimization for subgraph isomorphism search. *PVLDB*, 2016.

[155] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD*, 2000.

[156] Prasan Roy and S. Sudarshan. *Multi-Query Optimization*. 2009.

[157] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2013.

[158] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. *BTW*, 2013.

[159] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, 2017.

[160] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *VLDBJ*, 2019.

[161] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.*, 29(2-3):595–618, 2020.

[162] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. *SIGMOD*, 1979.

[163] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. *SIGMOD*, 1979.

[164] Timos K. Sellis. Multiple-query optimization. *TODS*, 1988.

[165] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *PVLDB*, 1(1), 2008.

[166] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel Subgraph Listing in a Large-scale Graph. *SIGMOD*, 2014.

[167] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. Vectorization vs. compilation in query execution. In Stavros Harizopoulos and Qiong Luo, editors, *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011*, 2011.

[168] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin*, 36, 2013.

[169] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. Rapidmatch: A holistic approach to subgraph query processing. *Proc. VLDB Endow.*, 14(2):176–188, 2020.

[170] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2015.

[171] Daniel ten Wolde, Tavneet Singh, Gabor Szarnyas, and Peter Boncz. DuckPGQ: Efficient property graph queries in an analytical RDBMS. In *Proceedings of the Conference on Innovative Data Systems Research*, 2023.

[172] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[173] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2020.

[174] TigerGraph. https://www.tigergraph.com.

[175] https://github.com/TimelyDataflow/timely-dataflow.

[176] Syunsuke Uemura, Toshitsugu Yuba, Akio Kokubu, Ryoichi Ooomote, and Yasuo Sugawara. The design and implementaion of a magnetic-bubble database machine. In *Information Processing, Proceedings of the 8th IFIP Congress 1980*, pages 433–438, 1980.

[177] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 1976.

[178] Wilco van Leeuwen, Thomas Mulder, Bram Van De Wall, George Fletcher, and Nikolay Yakovets. Avantgraph query processing engine. *Proc. VLDB Endow.*, 15(12), 2022.

[179] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, 2012.

[180] Todd L. Veldhuizen. Incremental Maintenance for Leapfrog Triejoin. *CoRR*, 2013.

[181] Vertica. Vertica 10.0.x Documentation. https://www.vertica.com/docs/10.0.x/HTML/Content/Home.html, 2020. Last Accessed July 25, 2021.

[182] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[183] Konstantinos Xirogiannopoulos and Amol Deshpande. Extracting and Analyzing Hidden Graphs from Relational Databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017.

[184] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph Indexing: A Frequent Structure-based Approach. *SIGMOD*, 2004.

[185] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *International Conference on Very Large Data Bases*, 1981.

[186] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient Maintenance of Materialized Top-k Views. *ICDE*, 2003.

[187] C. T. Yu and M. Z. Ozsoyoglu. *ALGORITHM FOR TREE-QUERY MEMBERSHIP OF A DISTRIBUTED QUERY.* 1979.

[188] C. T. Yu and M. Z. Ozsoyoglu. An Algorithm for Tree-Query Membership of a Distributed Query. *COMPSAC*, 1979.

[189] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. *PVLDB*, 2013.

[190] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph Indexing: Tree + Delta ¡= Graph. *VLDB*, 2007.

[191] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. *SIGMOD*, 2007.

[192] Lei Zou, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. Answering pattern match queries in large graph databases via graph embedding. *VLDBJ*, 2012.

[193] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A Vectorized Analytical DBMS. *ICDE*, 2012.

[194] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, 2012.