TECHNISCHE UNIVERSITÄT DRESDEN

Real-time Business Intelligence through Compact and Efficient Query Processing Under Updates

Dissertation

zur Erlangung des akademischen Grades
Ph.D.

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Muhammad Idris
geboren am 09. April 1990 in Pakistan

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Stijn Vansummeren
Université Libre de Bruxelles
Department of Computer and Decision Engineering
Laboratory for Web and Information Technology
Av. F. Roosevelt , CP 165/15
B-1050 Brussels, Belgium

Tag der Verteidigung: 05. March 2018

CURRICULUM VITAE

Muhammad Idris

Muhammad Idris obtained his graduate degree in information technology (BSIT) in 2012 from *School of Electrical Engineering and Computer Science* (SEECS) at the *National University of Science and Technology* (NUST), Islamabad in Pakistan. During his graduation, he also completed a 2-months summer internship from KTH-AIS Lab at SEECS in information security. His graduation thesis was entitled *Next Generation Fixed Assets Management Systems* (XGAM) and was pursued under the supervision of Dr. Muhammad Bilal and Mr. Bilal Ali.

In July 2012, he joined Mindblaze Technologies (MBT), a software firm based in Pakistan, and worked as *Application Programmer* until March 2013. During this period, he was involved in both the back-end and front-end development of systems that provide online shopping and enable the end-users to create customized websites and buy their own domain names in a *one-click* manner.

In March 2013, Mr. Idris joined *Kyung Hee University* (KHU) in the Gyeonggi-do province in South Korea to pursue his master degree in computer engineering. During this time, he also worked as researcher and developer in the *Big Data and Cloud* team at the *Ubiquitous Computing Laboratory* (UCLab) at KHU. As a team member, he was involved in several health-care and life-care provisioning projects and was responsible for the development of Big Data frameworks for these projects. As a researcher at the UCLab, he pursued his master thesis entitled *"Concurrent Algorithm Execution on Large Dataset using Hadoop MapReduce"* under the supervision of Professor Sungyoung Lee. He completed his master in February 2015 and stayed as a member of the UCLab till August 2015.

Mr. Idris started his Ph.D. studies under the European Commission-funded Erasmus Mundus joint degree doctoral program, known as *Information Technology for Business Intelligence* (IT4BI-DC) at Université Libre de Bruxelles (ULB), Belgium as home university and Technische Universität Dresden (TUD), Germany as host university. He pursued his Ph.D. entitled *"Real-time Business Intelligence through Compact and Efficient Query Processing Under Updates"* under the supervision of Professor Stijn Vansummeren (ULB) and

Professor Wolfgang Lehner (TUD). As part of his joint Ph.D. studies, Idris stayed at TUD, his host university, working in the *Dresden Database Group* under the supervision of Professor Wolfgang Lehner (February 2017 - December 2017).

His main research interests fall into responsive analytics with a focus on real-time business intelligence. In particular, his research in Ph.D. is focused on the dynamic evaluation queries under updates in a main-memory model. The outcomes of his Ph.D. research work has been published in the top-tier peer-reviewed international conferences SIGMOD 2017 and VLDB 2018. An extended version of his work in VLDB 2018 was invited to the VLDB Journal, where it is currently under review. His Ph.D. research work was selected for presentation at the IFIP world computer congress (WCC) under the *Ph.D. student research competition* (SRC) that was held in Poland in September 2018.

ACKNOWLEDGMENT

First of all, I would like to thank the Almighty Allah (SWT) who entrusted me with this honor. I would like to express my great appreciation to my advisor Professor Stijn Vansummeren for his continuous support and direct supervision throughout the course of my Ph.D., and Professor Wolfgang Lehner for his support throughout this duration and making my stay at TUD wonderful. I am also grateful to Martin Ugarte (ex Post-doc at ULB) for his contributions and help during the whole Ph.D. duration and Hannes Voigt (ex Post-doc at TUD) for his direct supervision during my one-year long stay at Dresden in 2017.

I am also grateful to the members of the IT4BI-DC evaluation committee for their patience and support in overcoming the obstacles throughout this Ph.D. duration. I would also like to thank my colleagues at ULB - especially Mahmoud Sakr - for bearing me for three years, and others for their patience and making my stay at ULB wonderful. I extend my gratitude to my colleagues at Dresden database group for making my stay in Dresden enjoyable. Nevertheless, I am also grateful to my IT4BI-DC colleagues for accepting nothing less than excellence from me.

Last but not the least, I would like to thank my family - my father for his continuous support and admiration throughout my life; my mother for her unconditional love, support, prayers, and my existence; and my brothers and sisters who whole-heartedly supported me in achieving every goal of my life and did not let me bother about the obstacles that they had to overcome - especially in my undergraduate studies and Ph.D. studies. Finally, I would like to extend my gratitude to the special Zombie people.

Dedication

I dedicate this Ph.D. to my beloved father who dreamed of this Ph.D. when I was only 14 years old. His consistent commitment and support have led me to achieve this milestone which he was deprived of for himself. I also dedicate this Ph.D. to my mother whose unconditional love and support make me live my life peacefully.

ABSTRACT

Responsive analytics are rapidly taking over the traditional data analytics dominated by the post-fact approaches in traditional data warehousing. Recent advancements in analytics demand placing analytical engines at the forefront of the system to react to updates occurring at high speed and detect patterns, trends, and anomalies. These kinds of solutions find applications in Financial Systems, Industrial Control Systems, Business Intelligence and on-line Machine Learning among others. These applications are usually associated with Big Data and require the ability to react to constantly changing data in order to obtain timely insights and take proactive measures. Generally, these systems specify the analytical results or their basic elements in a query language, where the main task then is to maintain query results under frequent updates efficiently. The task of reacting to updates and analyzing changing data has been addressed in two ways in the literature: traditional business intelligence (BI) solutions focus on historical data analysis where the data is refreshed periodically and in batches, and stream processing solutions process streams of data from transient sources as flows of data items. Both kinds of systems share the niche of *reacting* to updates (known as *dynamic evaluation*), however, they differ in architecture, query languages, and processing mechanisms. In this thesis, we investigate the possibility of a reactive and unified framework to model queries that appear in both kinds of systems.

In traditional BI solutions, evaluating queries under updates has been studied under the umbrella of *incremental evaluation* of queries that are based on the relational *incremental view maintenance* model and mostly focus on queries that feature equi-joins. Streaming systems, in contrast, generally follow *automaton* based models to evaluate queries under updates, and they generally process queries that mostly feature comparisons of temporal attributes (e.g., timestamp attributes) along with comparisons of non-temporal attributes over streams of bounded sizes. Temporal comparisons constitute inequality constraints while non-temporal comparisons can either be equality or inequality constraints. Hence these systems mostly process inequality joins. As starting point for our research, we postulate the thesis that queries in streaming systems can also be evaluated *efficiently* based on the paradigm of *incremental evaluation* just like in BI systems in a main-memory model. The efficiency of such a model is measured in terms of runtime memory footprint and the update processing cost. To this end, the existing approaches of dynamic evaluation in both kinds of systems present a trade-off between memory footprint and the update processing cost. More specifically, systems that avoid materialization of query (sub)results incur high update latency and systems that materialize (sub)results incur high memory footprint. We are interested in investigating the possibility to build a model that can address this trade-off. In particular, we overcome this trade-off by investigating the possibility of practical dynamic evaluation algorithm for queries that appear in both kinds of systems and present a main-memory data representation that allows to enumerate query (sub)results without materialization and can be maintained efficiently under updates. We call this representation the *Dynamic Constant Delay Linear Representation* (DCLR).

We devise DCLRs with the following properties: 1) they allow, without materialization, enumeration of query results with bounded-delay (and with constant delay for a sub-class of queries), 2) they allow tuple lookup in query results with logarithmic delay (and with constant delay for conjunctive queries with equi-joins only), 3) they take space linear in the size of the database, 4) they can be maintained efficiently under updates. We first study the DCLRs with the above-described properties for the class of acyclic conjunctive queries featuring equi-joins with projections and present the dynamic evaluation algorithm called the *Dynamic Yannakakis* (DYN) algorithm. Then, we present the generalization of the DYN algorithm to the class of acyclic queries featuring multi-way θ-joins with projections and call it *Generalized* DYN (GDYN). We devise DCLRs with the above properties for acyclic conjunctive queries, and the working of DYN and GDYN over DCLRs are based on a particular variant of join trees, called the *Generalized Join Trees* (GJTs) that guarantee the above-described properties of DCLRs. We define GJTs and present algorithms to test a conjunctive query featuring θ-joins for acyclicity and to generate GJTs for such queries. We extend the classical GYO algorithm from testing a conjunctive query with equalities for acyclicity, to testing a conjunctive query featuring multi-way θ-joins with projections for acyclicity. We further extend the GYO algorithm to generate GJTs for queries that are acyclic.

GDYN is hence a unified framework based on DCLRs that enables processing of queries that appear in streaming systems as well as in BI systems in a unified main-memory model and addresses the space-time trade-off. We instantiate GDYN to the particular case where all θ-joins involve only equalities and inequalities and call this instantiation IEDYN. We implement DYN and IEDYN as query compilers that generate executable programs in the Scala programming language and provide all the necessary data structures and their maintenance and enumeration methods in a continuous stream processing model. We evaluate DYN and IEDYN against state-of-the-art BI and streaming systems on both industrial and synthetically generated benchmarks. We show that DYN and IEDYN outperform the existing systems by over an order of magnitude efficiency in both memory footprint and update processing time.

ZUSAMMENFASSUNG

Herkömmliche Data-Warehouse-Systeme werden zusehends durch neue Datenanalysesystem abgelöst, die in der Lage sind Muster, Trends und Anomalien in immer schneller werdenden Datenströme zu erkennen Anwendungen hierfür finden sich unter anderem in Finanzsystemen, industriellen Steuerungssystemen, im Bereiche Business Intelligence (BI) und Online Machine Learning. Diese Anwendungen erfordern die Fähigkeit, auf sich ständig ändernde Daten zu reagieren, um zeitnahe Erkenntnisse zu erhalten und proaktive Maßnahmen zu ergreifen. Im Allgemeinen definieren diese Datenanalysesysteme die entsprechende Ergebnisse bzw. Teile davon in einer Abfragesprache, wobei die Hauptaufgabe darin besteht, diese Ergebnisse unter häufigen Aktualisierungen effizient zu verwalten. Die Aufgabe, auf Aktualisierungen zu reagieren und sich ändernde Daten zu analysieren, wird in der Literatur auf zwei Arten adressiert: 1) Traditionelle Business-Intelligence-Lösungen konzentrieren sich auf die Analyse historischer Daten, bei denen diese periodisch in Batches aktualisiert werden, und 2) Datenstromsysteme welche in der Lage sind einen oder mehrere Datenströme, d.h. Folgen von Datenelementen, aus transienten Quellen zu verarbeiten. Beide Systemarten sind in der Lage dynamische Auswertungen angesichts von Aktualisierung zu realisiereb, unterscheiden sich jedoch in Architektur, Abfragesprache und Verarbeitungsmechanismen. In dieser Arbeit soll ein vereinheitlichtes Framework geschaffen werden, dass es ermöglicht Anfragen beider Systemarten einheitlich zu modellieren.

In klassischen BI-Lösungen wurde das Problem der Wartung relationaler Sichten angesichts inkrementeller Aktualisierungen zumeist für Equi-Joins untersucht. Datenstromsysteme stattdessen basieren konzeptuell auf Automaten. Diese verarbeiten zumeist Anfragen die Vergleiche von zeitlichen Attributen (z. B. Zeitstempelattributen) zusammen mit Vergleichen von nicht-zeitlichen Attributen über Datenströme begrenzten Größen realisieren. Zeitliche Attribute werden meist auf Ungleichheit geprüft, während nichtzeitliche Attribute entweder auf Gleichheit oder Ungleichheit geprüft werden. Im Allgemeinen verarbeiten die Systeme meist Theta-Verbünde.

Als Ausgangspunkt postulieren wir die These, dass Abfragen in Datenstromsystemen analog zu Abfragen hauptspeicherbasierten BI-Systemen effizient inkrementell ausgewertet werden können. Die Effizienz eines solchen Modells wird im Hinblick auf den Speicherbedarf zur Anfragelaufzeit und die Aktualisierungsverarbeitungskosten gemessen. Zu diesem Zweck bieten die bestehenden Ansätze der dynamischen Auswertung in beiden Arten von Systemen einen Kompromiss zwischen Speicherplatzbedarf und den Aktualisierungsverarbeitungskosten. Systeme, die die Materialisierung von Abfrage-(Unter-)Ergebnissen vermeiden, haben eine hohe Aktualisierungslatenz und Systeme, die (Unter-) Ergebnisse vorhalten, haben einen hohen Speicherbedarf. Das Ziel der Dissertation bestand darin, zu untersuchen ob dieser Kompromiss vermeidbar ist. Dazu haben wir einen dynamischen Auswertealgorithmus für Abfragen untersucht, der in beiden Systemarten vorkommt. Weiterhin haben wir eine Hauptspeicherdatenstruktur entwickelt, die es ermöglicht, Abfrage-(Unter-)Ergebnisse ohne Materialisierung zu enumerieren

und unter Aktualisierung effizient verwaltet. Wir bezeichnen diese Darstellung als Dynamic Constant Delay Linear Representation (DCLR).

Wir entwickeln DCLRs mit den folgenden Eigenschaften: 1) Sie ermöglichen die Enumeration von Abfrageergebnissen mit begrenzter Verzögerung ohne Materialisierung (und mit konstanter Verzögerung für eine Unterklasse von Abfragen). 2) sie ermöglichen Tupel-Lookups in Abfrageergebnissen mit logarithmischer Verzögerung (und mit konstanter Verzögerung für konjunktive Abfragen nur mit Equi-Joins); 3) ihr Speicherplatzbedarf ist linear Bezug zur Datenbankgröße; 4) sie können unter Aktualisierungen effizient verwaltet werden.

Wir untersuchen zunächst die DCLRs mit den oben beschriebenen Eigenschaften für die Klasse der azyklischen konjunktiven Abfragen mit Equi-Joins mit Projektionen und stellen den dynamischen Auswertungsalgorithmus vor, der als Dynamic Yannakakis (DYN)-Algorithmus bezeichnet wird. Anschließend stellen wir die Verallgemeinerung des DYN-Algorithmus auf die Klasse der azyklischen Abfragen mit mehrwegigen Theta-Joins mit Projektionen vor und bezeichnen diese als Generalized DYN (GDYN). Mit den obigen Eigenschaften für azyklische konjunktive Abfragen legen wir fest DCLRs, und das Arbeiten von DYN und GDYN über DCLRs basiert auf einer bestimmten Variante von Verbindungsbäumen, den so genannten Generalized Join Trees (GJTs), die die oben beschriebenen Eigenschaften von DCLRs gewährleisten. Wir definieren GJTs und präsentieren Algorithmen, um eine konjunktive Abfrage mit Theta-Joins auf Azyklizität zu testen und GJTs für solche Abfragen zu generieren. Wir erweitern den klassischen GYO-Algorithmus um eine konjunktive Abfrage mit mehrwegigen Theta-Joins mit Projektionen auf Azyklizität zu testen. Wir erweitern den GYO-Algorithmus weiter, um GJTs für azyklische Abfragen zu generieren

RESUMÉ

L'analyse en ligne (responsive analytics) est en train de prendre le dessus sur les analyses de données hors ligne supportés par les entrepôts de données traditionnels. Les avancements récents en analyse exigent le placement du moteur d'analyse en avant du système afin de prendre en considération les mises à jour effectuées à grande vitesse et détecter les régularités, les tendances et les anomalies. Ces solutions sont appliquées entre autres dans les systèmes de finance, les systèmes de control industriel, l'informatique décisionnelle (business intelligence) et l'apprentissage automatique en ligne (on-line Machine Learning).

Ces applications sont associées aux mégadonnées (big data) et demandent la capacité de réagir aux modifications constantes des données dans l'objectif d'obtenir une compréhension de la situation actuelle pour prendre des mesures proactives. Généralement, ces systèmes spécifient les résultats analytiques ou leurs éléments de base avec un langage d'interrogation qui consiste à maintenir l'efficacité de ces résultats dans un contexte de mises à jour fréquentes. Les tâches de réagir aux mises à jour et d'analyser les données modifiées ont été étudiées selon deux approches dans la littérature. D'un côté, les solutions de l'informatique décisionnelle traditionnelle se focalisent sur l'historique de l'analyse des données où les données sont mises à jour périodiquement et en batch. D'un autre, les solutions de traitement par flux (stream processing) traitent les données reçus comme des flux de données élémentaires. Ces deux types de systèmes partagent la problématique de réagir aux mises à jour (appelée évaluation dynamique). Cependant, ils diffèrent dans leurs architectures, leurs langages d'interrogation et les mécanismes de traitement. Dans cette thèse, nous investiguons la possibilité d'une infrastructure logicielle (framework) réactif et unifié afin de modéliser les requêtes relatives aux deux types de système.

Dans l'informatique décisionnelle traditionnelle, l'évaluation dynamique des requêtes a été étudié dans le contexte de l'évaluation incrémentale des mises à jour mais en se basant uniquement sur des requêtes avec des equi-jointures. En revanche, les systèmes de flux de données suivent généralement les modèles basés sur des automates et traitent généralement des requêtes qui présentent principalement des comparaisons d'attributs temporels et non temporels avec des flux de tailles limitées. Les comparaisons temporelles constituent des contraintes d'inégalité, alors que les comparaisons non temporelles peuvent être des contraintes d'égalité ou d'inégalité. Par conséquent, ces systèmes traitent principalement les jointures d'inégalité. Comme point de départ, nous postulons l'hypothèse suivante : les requêtes dans les systèmes de traitement de flux peuvent également être évaluées de manière efficace sur la base du paradigme de l'évaluation incrémentale, tout comme dans les systèmes d'informatique décisionnelle dans un modèle de mémoire principale. L'efficacité d'un tel modèle se mesure en termes de la consommation de mémoire au moment de l'exécution et du coût de traitement de la mise à jour. Pour atteindre cet objectif, les approches existantes d'évaluation dynamique dans les deux types de systèmes présentent un compromis entre consommation de mémoire

et le coût de traitement de la mise à jour. Plus spécifiquement, les systèmes qui évitent la matérialisation de résultats de (sous) requêtes entraînent une latence de mise à jour élevée et les systèmes qui matérialisent des résultats de (sous) requièrent une consommation de mémoire importante. Nous nous intéressons à la possibilité de construire un modèle qui aborde ce compromis. En particulier, nous étudions la possibilité d'utiliser un algorithme d'évaluation dynamique pour les requêtes apparaissant dans les deux types de systèmes. Pour cela nous utilisons une représentation des données succincte permettant d'énumérer les résultats de requêtes sans matérialisation et pouvant être gérée efficacement avec des mises à jour. Nous désignons cette représentation Dynamic Constant Delay Linear Representation (DCLR).

Nous concevons les DCLRs avec les propriétés suivantes: 1) ils permettent, sans matérialisation, l'énumération des résultats de requête avec un délai borné (et avec un délai constant pour une sous-classe de requêtes), 2) ils permettent la recherche de tuples dans les résultats de requête avec un délai logarithmique (et avec un retard constant pour les requêtes conjonctives avec des équi-jointures uniquement), 3) ils consomment un taille linéaire dans la base de données, 4) ils peuvent être maintenus efficacement avec des mises à jour. Nous étudions d'abord les DCLR avec les propriétés décrites ci-dessus pour la classe de requêtes conjonctives acycliques comportant des équi-jointures avec des projections et nous introduisons un algorithme d'évaluation dynamique appelé Dynamic Yannakakis (DYN). Ensuite, nous présentons la généralisation de l'algorithme DYN à la classe de requêtes acycliques comportant des thêta-jointures et des projections et l'appelons le Generalized DYN (GDYN). Nous concevons les DCLR avec les propriétés ci-dessus pour les requêtes conjonctives acycliques. Le fonctionnement de DYN et GDYN sur DCLRs se base sur une variante particulière des arbres de jointure, appelée Generalized Join Trees (GJT), qui garantit les propriétés décrites ci-dessus des DCLR. Nous définissons les GJT et présentons des algorithmes pour tester l'acyclicité d'une requête conjonctive comportant des thêta-jointures et pour générer des GJT pour de telles requêtes. Pour cela, nous étendons l'algorithme classique GYO en testant l'acyclicité d'une requête conjonctive avec seulement des équi-jointures.

GDYN est une infrastructure logicielle basée sur les DCLR qui permet le traitement des requêtes dans les systèmes de flux de données ainsi que dans les systèmes d'informatique décisionnelle de manière unifiée et prend en considération le compromis espace-temps. En particulier, nous instancions GDYN au cas particulier où toutes les jointures sont composées uniquement des égalités et des inégalités, et appelons cette instanciation IEDYN. Nous implémentons DYN et IEDYN en tant que compilateurs de requêtes qui génèrent des programmes exécutables dans le langage de programmation Scala et fournit toutes les structures de données nécessaires ainsi que leurs méthodes de maintenance et d'énumération dans un modèle de traitement de flux continu. Nous évaluons DYN et IEDYN par rapport aux systèmes d'informatique décisionnelle et de flux de données avec des jeux de données standards industriels et des jeux de données synthétiques. Nous montrons que DYN et IEDYN surperforment les systèmes existants d'une efficacité d'un ordre de grandeur en termes de consommation de mémoire et de temps de traitement de la mise à jour.

CONTENTS

CURRICULUM VITAE i

ABSTRACT iv

ZUSAMMENFASSUNG vi

RESUMÉ viii

PAGE

1 INTRODUCTION 4

1.1 Background . . . 4
1.1.1 Dyanmic Evaluation in BI Systems . . . 5
1.1.2 Dynamic Evaluation in IFP Systems . . . 6

1.2 Problem Definition and Research Questions . . . 7

1.3 Proposed Solution: The Dynamic Constant-Delay Linear Representations (DCLRs) . . . 8

1.4 Summary of Contributions . . . 10
1.4.1 Contribution for NCQ . . . 10
1.4.2 Contributions for GCQs . . . 12
1.4.3 Contributions for Computing GJTs . . . 13

1.5 Statement . . . 14

1.6 Thesis Structure . . . 14

2 RELATED WORK 15

2.1 Dynamic Query Evaluation . . . 15
2.1.1 BI systems . . . 15
2.1.2 IFP systems . . . 17

2.2 Constant Delay Enumeration . . . 19
2.2.1 Practical and Static CDE . . . 20
2.2.2 Theoretical and Dynamic CDE . . . 22

2.3 Join Algorithms . . . 23
2.3.1 Equi-Join Algorithms . . . 23
2.3.2 Inequality-Join Algorithms . . . 24

2.4 Join Tree Computation . . . 25

3 THE DYNAMIC YANNAKAKIS ALGORITHM (DYN) 27

3.1 Preliminaries . . . 27
3.1.1 Computational Model . . . 29
3.1.2 Acyclicity . . . 30
3.2 Dynamic Yannakakis . . . 32
3.2.1 Constant delay enumeration . . . 32
3.2.2 Constant Delay Yannakakis . . . 33
3.2.3 Dynamic Yannakakis . . . 35
3.2.4 Complexity of Dynamic Yannakakis . . . 40
3.2.5 Enumeration . . . 46
3.2.6 Optimalitiy . . . 51
3.3 Experimental Evaluation . . . 55
3.3.1 Practical Implementation . . . 55
3.3.2 Experimental Setup . . . 56
3.3.3 Experimental results . . . 57
3.4 Conclusion . . . 60

4 GENERALIZED DYN (GDYN) 62

4.1 Preliminaries . . . 62
4.2 Generalized Acyclicity . . . 63
4.3 Dynamic Joins with Equalities and Inequalities: An Example . . . 70
4.4 Dynamic Yannakakis over GCQs . . . 73
4.5 Experimental Evaluation . . . 81
4.5.1 Experimental Setup . . . 81
4.5.2 Experimental Evaluation . . . 85
4.6 Conclusion . . . 90

5 JOIN TREE COMPUTATION 92

5.1 Classical GYO . . . 92
5.2 GYO-reduction for GCQs . . . 93
5.3 Correctness of GYO for GCQs . . . 98
5.3.1 Soundness and Completeness . . . 98
5.4 Proof of Confluence of GYO for GCQs . . . 105
5.5 Conclusion . . . 110

6 DISCUSSION AND FUTURE DIRECTIONS 111

6.1 Summary . . . 111
6.2 Strength, Limitations and Future Directions . . . 112
6.2.1 Strengths . . . 112

6.2.2 Limitations and Future Directions 112

BIBLIOGRAPHY **114**

LIST OF FIGURES **120**

LIST OF TABLES **121**

LIST OF ACRONYMS **122**

A BENCHMARK QUERIES **124**

A.1 Queries for evaluation of DYN 124
A.1.1 Full Join Queries 124
A.1.2 TPC-H Queries 125
A.1.3 TPC-DS Queries 127

A.2 Queries for Evaluation of IEDYN 128
A.2.1 Queries in SQL Syntax for IEDYN and DBToaster 128
A.2.2 Queries in Esper EPL 130
A.2.3 Queries in Tesla/TRex Rule Language Syntax 131
A.2.4 SASE rule language 133

1 INTRODUCTION

In this thesis, we discuss issues related to data analysis under high update rates in a main-memory model. In this chapter, we will first introduce the reader to the research background in the area and the identified problems. From the identified problems, we will postulate the thesis (research) questions that will be the base of the study. Then, we will briefly present an overview of the proposed solution and research contributions to address the research questions. Lastly, we will show an overview of the thesis structure.

1.1 Background

Realtime analytics find applications in Financial Systems, Industrial Control Systems, Business Intelligence and On-line Machine Learning, among many others (see [CM12b] for a survey). These applications are usually associated with Big Data, and require the ability to analyze dynamically changing data in order to obtain timely insights and implement reactive and proactive measures. Generally, the analytical results that need to be kept up-to-date, or at least their basic elements, are specified in a query language. The main task is then to efficiently update the query results under frequent data updates. In particular, the requirement of analyzing changing data has been addressed in two ways.

- Traditional business intelligence (BI) solutions focus on the setting where one has to analyze historical data in order to evaluate potential strategic business options. As a consequence, a traditional data warehouse is mostly static: its data is refreshed only periodically and in batches, typically under a long time interval (e.g., a week). The desire to push operational data as soon as possible to the data warehouse for timely decision support has led to research on *near realtime data warehouses* [FMF13] and *active data warehouses [DKRC+14].* Abstractly speaking, these approaches try to integrate Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) functionalities as much as possible.
- For a rising number of applications, data arriving in a streaming fashion from transient sources must be taken into account. Examples here include financial stock analysis, twitter sentiment stream analysis, traffic and mobility management, click-stream inspection, etc. To query such transient data sources, systems have been built that are specially designed to process information as a flow (or a set of flows) of data items. These systems, to which we will refer collectively as Information Flow Processing (IFP) systems, support so-called continuous queries, i.e., queries that inspect the incoming data streams and produce streams of (updated) answers.

At a high level, it is clear that in both of these examples, *reacting* to updates, also called *dynamic evaluation* under updates, is the central feature: realtime and active data warehouses must update information in their OLAP cube data structures; while IFP systems have to update their continuous answers. In the following sections 1.1.1 and 1.1.2, we present brief overviews of *dynamic evaluation* in traditional BI and IFP systems respectively.

1.1.1 Dyanmic Evaluation in BI Systems

In this section, we focus on the problem of *dynamic query evaluation* in traditional BI solutions, where a given query Q has to be evaluated against a database that is constantly updated. In this setting, when database *db* is updated to database $db + u$ under update u, the objective is to efficiently compute the result $Q(db + u)$, taking into consideration that $Q(db)$ was already evaluated and re-computations could be avoided. Dynamic query evaluation has traditionally been approached from Incremental View Maintenance (IVM) [CY12a]. IVM techniques materialize $Q(db)$ and evaluate *delta queries*. These delta queries take as input *db*, u and the materialized $Q(db)$, and return the set of tuples to add/delete from $Q(db)$ to obtain $Q(db + u)$. If u is small w.r.t. *db*, this is expected to be faster than recomputing $Q(db + u)$ from scratch. Research in this area has received a big boost with the introduction of Higher-Order IVM (HIVM) [NDK16, KAK+14, Koc10]. HIVM essentially obtains low processing times by trading time for space: it aggressively stores query subresults in order to guarantee fast response times upon updates. This is most effective when the query to be maintained results in a single aggregate (such as, e.g., in the case of aggregate sum

```
SELECT SUM(A * C) from R, S where R.B = S.B        (⋆)
```

over relation $R(A, B)$ and $S(B, C)$). Given a query Q, HIVM not only defines the delta query ΔQ, but also materializes it. Moreover, it defines higher-order delta queries (i.e., delta queries for delta queries, denoted $\Delta^2 Q, \Delta^3 Q, \dots$), where every $\Delta^j Q$ describes how the materialization of $\Delta^{j-1} Q$ should change under updates. This method is highly efficient in practice, and is formally in a lower complexity class than IVM [Koc10].

(H)IVM present important drawbacks, however. First, materialization of $Q(db)$ requires $\Omega(\|Q(db)\|)$ space, where $\|db\|$ denotes the size of *db*. Therefore, when $Q(db)$ is large compared to *db*, materializing $Q(db)$ quickly becomes impractical, especially for main-memory based systems. HIVM is even more affected by this problem than IVM since it not only materializes the result of Q but also the results of the higher-order delta queries. Second, IVM and HIVM only exploit the information provided by the materialized views to process updates, while additional forms of information could result in better update rates. Consider for example the query $Q = R(A, B) \bowtie S(B, C)$ and a database with N tuples in R and N tuples in S, all with the same B value. The materialization of $Q(db)$ in this case uses $\Theta(N^2)$ space and is useless for re-computing Q under updates. In contrast, a simple index on B for R and S would allow for efficient enumeration of the set of tuples that need to be added or removed from $Q(db)$ to obtain $Q(db + u)$. It is important to note that even for queries whose result is smaller than the database, aggressive materialization of higher-order delta queries in HIVM can still cause these problems to appear. Indeed, some higher-order delta queries are partial join results, which can be larger than both *db* and $Q(db)$.

Dynamic evaluation in traditional BI systems is a relatively old topic and has been extensively studied by the database community over the last four decades. Many approaches and systems have been developed to support incremental computation. Notable early examples of these systems include RETE, TREAT, and RETE* algorithms [Pro12,

Mir14, WM03] for production rule systems, realtime data warehousing systems [FMF13] among others (see [ZGMHW95] for details). However, in all these systems, IVM and HIVM remain at the core of the problem i.e. all of these systems investigate the different possibilities to make decisions about the materialization of views and the evaluation of queries. These systems hence inherit the problems faced by (H)IVM. We present a detailed overview of the key BI systems in chapter 2.

1.1.2 Dynamic Evaluation in IFP Systems

In this section, we focus on the problem of *dynamic query evaluation* for queries in IFP systems. To do so, we first present an overview of the working model and then present dynamic query evaluation in IFP systems.

IFP systems have been categorized in two groups in the literature: classical stream processing systems and composite event recognition (CER) systems (see [Bui09] for a survey). Classical stream processing systems [ABW06, AAB+05, ACÇ+03, CBB+03] employ networks of operators (e.g. join operators) to process incoming event streams of bounded windows, and these systems leave it to the endpoints or clients to attach meaningful information to the resultset. On the other hand, CER systems [ASAP15] are fed with streams of events where an event can be a sensor reading or a credit card transaction, etc. These systems then apply computations to correlate different incoming event streams and generate complex (derived) events. For example, a derived event *Fire* may be generated from the sequence of events *Smoke* and *Heat* in a time window of x seconds where *Area a* and *TimeStamp ts* are the common attributes among Heat and Smoke events, and event *Heat* has an additional attribute *Value v*. The derived event imposes constraints (temporal and non-temporal) on the input events. These constraints, if temporal, generally involve comparisons of temporal values (e.g. comparison of *TimeStamps ts* between Smoke and Heat) of events, hence they constitute inequality joins between event streams. Non-temporal constraints could be simple equi-joins between the common attributes (e.g. Smoke and Heat joined on the attribute *Area a*). Among the inequality joins resulting from temporal constraints, there can be standard inequality joins between event streams on non-temporal attributes e.g. Smoke may have an additional attribute *Intensity s* that can be compared to the attribute *Value v* of the Heat event using an inequality (e.g. $<, \leq, >, \geq$). Systems that use CER are known as Complex Event Processing (CEP) systems.

In general, IFP systems feature temporal and non-temporal constraints. In other words, they feature multi-way θ-joins in addition to standard equi-joins. These systems feature a high degree of inequality joins because of the windowing semantics that require processing of temporal constraints. To illustrate this, consider that we wish to detect potential credit card frauds. Credit card transactions specify their timestamp (*ts*), the account number (*acc*), amount (*amnt*), among others. A typical fraud pattern is that the criminal tests the credit card with a few small purchases to then make larger purchases (cf. [SMP09]). In this respect, we would like to dynamically evaluate the following query, assuming new transactions arrive in a streaming fashion and the pattern must be detected in less than 1 hour.

```
SELECT * FROM Trans as S1, Trans as S2, Trans as L
WHERE S1.ts < S2.ts AND S2.ts < L.ts AND L.ts < S1.ts + 1h
AND S1.acc = S2.acc AND S2.acc = L.acc
AND S1.amnt < 100 AND S2.amnt < 100 AND L.amnt > 400
```

Queries like this with inequality joins appear in both CER and BI scenarios. The traditional BI techniques to process these queries dynamically based on IVM and HIVM has been discussed in section 1.1.1. In IFP systems, *automaton* based approaches are used to evaluate these queries dynamically. We next discuss the automaton based models and their drawbacks.

Automaton-based Models. Automaton-based approaches (e.g., [Esp, ZDI14, BDG+07b, CM12a, WDR06, CM10, ADGI08, MM09]) assume that the arrival order of event tuples corresponds to the timestamp order (i.e., there are no out-of-order events) and build an automaton to recognize the desired temporal patterns in the input stream. There are two automaton-based recognition approaches. In the first approach (also called *lazy evaluation*), followed by [WDR06, ADGI08], events are cached per state and once a final state is reached a search through the cached events is done to recognize the complex events. While the temporal constraints need no longer be checked during the search, the additional constraints (in our example, $L.ts < S1.ts + 1h$ and $S_1.\text{acc} = S_2.\text{acc} = L.\text{acc}$) must still be verified. If the additional constraints are highly selective this approach creates an unnecessarily large update latency, given that each event triggering a transition to a final state may cause re-evaluation of a sub-join on the cached data, only to find that few tuples contribute to the output. In the second approach (also called *materialization-based*), followed by [BDG+07b, ZDI14, CM12a, CM10], partial runs are materialized according to the automaton's topology. For our example credit card fraud query, this means that, just like HIVM, the join

$$\sigma_{amnt<100}(S_1) \bowtie_{S_1.ts<S_2.ts \wedge S_1.acc=S_2.acc} \sigma_{amnt<100}(S_2)$$

is materialized and maintained so it is available when a large amount of transaction L arrives. This approach hence shares with HIVM its high memory overhead and maintenance cost.

It is important to note here that, some CEP systems such as [Esp, MM09] use the automaton-based models for evaluation of queries, however, they employ the relational model to store the data. These systems hence inherit the drawbacks of HIVM when the materialization based approach is used, and they suffer from high update latency when the lazy evaluation approach is used.

1.2 Problem Definition and Research Questions

In general, both BI and IFP systems are similar in the required functionality of reacting to updates. However, despite this basic similarity, the developed systems—having been developed by different research (sub)communities, each one approaching the problem from a different viewpoint—differ significantly in important aspects, including: architecture, query languages, and processing mechanisms, as discussed in sections 1.1.1 and 1.1.2. In a sense, each system focuses on its own niche in the data management landscape. For example, active data warehouse systems (BI system) focus on OLAP-style queries over frequently changing datasets (where aggregation is the point of focus) while other systems, such as Complex Event Processing Systems (CEP systems), focus on temporal join processing between different streams under a time-window semantics.

Given the disagreement between the similarity in required functionality and the myriad of techniques used for providing this functionality, this thesis is motivated by the following question:

> *RQ_1: Is it possible to build a reactive data management framework, that unifies functionality of both realtime data warehouses and continuous queries as known from IFP systems?*

As a concrete starting point for answering this question, we postulate the thesis that just like in BI systems, queries in IFP systems can also be evaluated based on the paradigm of *incremental evaluation*. In particular, we are interested in investigating an *efficient* unified incremental evaluation model that can be adapted for both kinds of systems. We discuss the efficiency of the model in terms of the update processing delay and the runtime memory footprint. To this end, the traditional BI and IFP approaches either materialize the full query results and partial results to speedup update processing, or use lazy evaluation approach to reduce memory footprint. We, however, are interested in a setting where we can avoid materialization of query results and partial results, and still be able to efficiently evaluate queries under updates and generate their results. This relates to the tradeoff between runtime memory footprint and update processing cost posed by the traditional (H)IVM approaches. Given this tradeoff, this thesis is further motivated by the following question:

> *RQ_2: Is it possible to build a model that can address the tradeoff between runtime memory footprint and update processing delay in BI and IFP systems?*

In this thesis, we investigate the research questions RQ_1 and RQ_2, and present our solution as a unified main-memory representation called the *Dynamic Constant Delay Linear Representations* (DCLRs). In the following section, we present an overview of DCLRs.

1.3 Proposed Solution: The Dynamic Constant-Delay Linear Representations (DCLRs)

In this thesis, we overcome the problems of BI systems and IFP systems by investigating the possibility of a practical algorithm for dynamic evaluation of queries that appear in both kind of systems. We present a data structure (DCLR) that can allow bounded delay enumeration of query results without materialization and can be maintained efficiently under updates. In particular, the crucial idea behind this data structure is to make careful decisions about what should be materialized, how the results of queries can be enumerated from this data structure, and how this data structure can be maintained under updates. Since we avoid full materialization of query results and subresults, the *enumeration* of query results can be seen as a *streaming decompression* algorithm that generates tuples one by one from the data structure such that each tuple is unique, and the data structure itself can be seen as a compressed (succinct) representation of the query result. In the following, we first present an overview of the notion of *enumeration with constant and bounded delay* (and recall them in chapters 3 and 4), and then present the key properties of DCLRs.

Enumeration with constant and bounded delay. A data structure D supports enumeration of a set E if there exists a routine ENUM such that ENUM(D) outputs each element in E exactly once. Such enumeration occurs with delay d if the time to output first tuple; the time between any two consecutive tuples; and the time between the last tuple and the end of the ENUM(D) routine, are all bounded by d. These times can neither depend on the size of D nor on the size of E. Moreover, the enumeration occurs with constant delay if d is constant. We call such enumeration *Constant Delay Enumeration* (CDE).

CDE has gained particular interest form the database research community [Seg15, BDG07a, Seg13], in the last decade. These approaches employ streaming decompression

algorithms to construct results from their compressed representation. However, these approaches either work for static settings without updates or, if they apply to the dynamic setting, they study only purely theoretical algorithms. We present a detailed overview of existing CDE approaches in chapter 2. In general, the query execution process is divided into two phases: a preprocessing phase during which the compressed representation is built, and an enumeration phase during which compressed representation is used to enumerate the query output. We, however, study the enumeration of query results with bounded delay in both static and dynamic settings, and present practical algorithms. In particular, to support enumeration with bounded delay (and with constant delay for a subclass of conjunctive queries), we desire a succinct data structure $\mathcal{D}_{db}$ that posses the following properties.

We will first present the properties of DCLRs for conjunctive queries that feature equi-joins only, and then present its properties for conjunctive queries that feature arbitrary θ-joins. Queries that feature equi-joins only are hereafter referred to as *Normal Conjunctive Queries* (NCQs for short) and queries that feature arbitrary θ-joins as *Generalized Conjunctive Queries* (GCQs for short).

Properties of $\mathcal{D}_{db}$ for NCQs. For every database db and an NCQ Q, we can compute a data structure $\mathcal{D}_{db}$ with the following properties:

- (P_1) $\mathcal{D}_{db}$ allows to enumerate $Q(db)$ with constant delay.
- (P_2) For any tuple $\vec{t}$, we can use $\mathcal{D}_{db}$ to check whether $\vec{t} \in Q(db)$ in constant time.
- (P_3) $\mathcal{D}_{db}$ requires only $O(\|db\|)$ space. As such $\mathcal{D}_{db}$ depends only on db and is independent of the size of $Q(db)$.
- (P_4) $\mathcal{D}_{db}$ features efficient maintenance under updates: given $\mathcal{D}_{db}$ and update u to database db, we can compute $\mathcal{D}_{db+u}$ in time $O(\|db\| + \|u\|)$. In contrast, both IVM and HIVM may require $\Omega(\|u\| + \|Q(db+u)\|)$ time in the worst case.

We present DCLRs for NCQs in chapter 3.

Properties of $\mathcal{D}_{db}$ for GCQs. For every database db and a GCQ Q, we can compute a data structure $\mathcal{D}_{db}$ with the following properties:

- (P_1^*) $\mathcal{D}_{db}$ allows to enumerate $Q(db)$ with bounded (logarithmic) delay. If the GCQ Q is a query without predicates or with at most a single inequality predicate between a pair of relations, then $\mathcal{D}_{db}$ allows to enumerate $Q(db)$ with constant delay.
- (P_2^*) For any tuple $\vec{t}$, we can use $\mathcal{D}_{db}$ to check whether $\vec{t} \in Q(db)$ in logarithmic time. For GCQs that feature only equi-joins, we can perform $\vec{t} \in Q(db)$ in constant time since the query that features equi-joins only is an NCQ.
- (P_3^*) $\mathcal{D}_{db}$ requires only $O(\|db\|)$ space. As such $\mathcal{D}_{db}$ depends only on db and is independent of the size of $Q(db)$.
- (P_4^*) $\mathcal{D}_{db}$ features efficient maintenance under updates for a higher class of conjunctive queries: given $\mathcal{D}_{db}$ and update u to database db, we can compute $\mathcal{D}_{db+u}$ in time $O(M^2 \cdot \log(M))$, where $M = |db|+|u|$. Moreover, for GCQs that feature single inequality predicate between pairs of relations, we can compute $\mathcal{D}_{db+u}$ in $O(M \cdot \log(M))$ time.

We present DCLRs for GCQs in chapter 4. Just like the existing approaches for CDE, we also separate the preprocessing (maintenance of $\mathcal{D}_{db}$ under updates) phase from the enumeration phase. Our DCLRs apply to both static and dynamic settings and enumeration properties of DCLRs discussed above hold for enumeration of delta query results. It is important to note that we consider query evaluation in main memory and measure time and space under data complexity [Var82]. That is, the query is considered to be fixed and not part of the input. This makes sense under dynamic query evaluation, where the

query is known in advance and the data is constantly changing. In particular, the number of relations to be queried, their arity, and the length of the query are all constant.

The DCLRs exhibit the above-described properties only for the class of acyclic NCQs and GCQs, and we focus on investigating the class of *acyclic* conjunctive queries in this thesis. Testing a conjunctive query for acyclicity has been extensively studied in the literature [GLS01, CR97]. The well-known GYO-reduction algorithm [AHV95] tests a conjunctive query Q for acyclicity if Q features equi-joins only i.e. if Q is a NCQ. Moreover, using this algorithm one can also construct a traditional join tree for Q. However, our DCLRs are based on a different notion of join trees, which we call *Generalized Join Trees* (GJTs for short) that ensure the above-described properties for DCLRs. In this thesis, we present the algorithms to test a conjunctive query with θ-joins (GCQs) for acyclicity and generate GJTs for GCQs that are acyclic. In essense, we build our algorithms on the traditional GYO-reduction algorithm and modify it to test the acyclicity of a GCQ in the presence of θ-joins, as well as generate actual GJTs.

1.4 Summary of Contributions

In this section we present the contributions of this thesis in the following order. First, we present our contributions with respect to dynamic evaluation of NCQs over DCLRs under multiset semantics mainly addressing the limitations of traditional BI systems. We call this specialized algorithm for NCQs the *Dyanmic Yannakakis* (DYN) algorithm. Next, we present our contributions by extending DYN to GCQs over DCLRs, mainly addressing the limitations of IFP systems. We call this generalized solution the *Generalized Dynamic* (GDYN) algorithm. Then, we present contributions for testing NCQs and GCQs for acyclicity and computing *Generalized Join Trees* (GJTs). Finally, we present an overview of different classes of GCQs and their theoretical results.

1.4.1 Contribution for NCQ

As a first contribution, we discuss how to modify the classical Yannakakis algorithm [Y81] into a dynamic query evaluation algorithm that we call DYN. DYN's operation is driven by the specification of a *Generalized Join Tree* (GJT), which essentially acts as a query plan for dynamic query evaluation. Syntactically, a GJT is a variant of the classical join trees used by the Yannakakis Algorithm.

DYN is a good algorithmic core to build practical algorithms on, for the following reasons.

(1) Like standard Yannakakis, DYN is a conceptually simple algorithm, and therefore easy to implement.

(2) Certain topological properties of the GJT on which DYN operates directly imply one or more of the properties (P_1 – P_4) of the DCLRs. For example, we formally introduce the topological notion of a GJT being *compatible* with the projections done by an NCQ Q, and show that if DYN operates on a GJT T that is compatible with Q, then the data structure $\mathcal{D}_{db}$ that is maintained by DYN satisfies properties (P_1) and (P_2). Furthermore, we introduce the topological notion of a GJT being *simple*. If DYN operates on a GJT T that is simple, then the property (P_4) is guaranteed. Property (P_3) is always guaranteed, regardless of the shape of the GJT on which DYN operates.

(3) The topological constraints that are required on GJTs in order for DYN to satisfy all required properties are not arbitrary, but, in a sense optimal. Indeed, as we will show, results by Bagan et al. [BDG07a] and Brault-Baron [BB13] for the static setting imply that, under certain complexity-theoretic assumptions, a DCLR can exist for an NCQ Q only if Q belongs to the class of so-called *free-connex acyclic* NCQs. An NCQ Q is free-connex acylic if, and only if, it has a GJT that is compatible with Q. This hence shows that GJT compatibility is a necessary topological constraint if we want all properties (P_1 – P_4).

(4) Furthermore, in a recent work, Berkholz et al. [BKS17] have characterized the class of self-join free NCQs that feature constant delay enumeration (CDE) of query results and that can be maintained in constant time under single-tuple updates. They show that this class corresponds to the class of so-called *q-hierarchical* queries, a strict subclass of the free-connex acyclic queries. We show that an NCQ Q is q-hierarchical if, and only if, Q has a GJT that is both simple and compatible with Q. As a consequence, DYN matches the lower bound of Berkholz et al.: for (not necessarily self-join free) q-hierarchical NCQs, it processes single-tuple updates in constant time and supports enumeration of query results with constant delay. For non q-hierarchical queries, Berkholz et al.'s result imply that *any algorithm* must either (a) process some single-tuple updates with more than constant time, or (b) do query result enumeration with more than constant delay. We show that DYN continues to process all single-tuple updates in constant time, as long as the query has a simple GJT. This comes at the expense of not allowing constant delay enumeration in case the GJT is not compatible with Q, but Berkholz' et al's result show that this is unavoidable. In this sense, DYN is hence again optimal.

(5) For single-tuple updates, DYN also allows us to enumerate with constant delay the delta results $\Delta Q(db, u) = Q(db + u) - Q(db)$, provided that Q has a simple GJT. This result is relevant for *push*-based query processing systems such as IFP systems, where users do not ping the system for the complete current query answer, but instead ask to be notified of the changes to the query results when the database changes.

Building on DYN, we implement a practical algorithm that allows for dynamic evaluation of the more general class of acylic *conjunctive aggregate* queries. Acyclic conjunctive aggregate queries are queries that compute aggregates (e.g., SUM, AVG, ...) over the result of an acyclic NCQ. This practical implementation works by first computing a free-connex approximation of the NCQ that underlies aggregate query Q. Call this approximation Q'. We use DYN to dynamically process Q' and use the delta enumeration of Q' provided by DYN (since Q' is free-connex, hence has a simple GJT) to materialize and maintain $Q(db)$. It is immediate that $Q(db)$ can be enumerated with constant delay from its materialized version, at the expense of now requiring $O(\|db\| + \|Q(db)\|)$ memory and incurring an extra $O(\|\Delta Q(db, u)\|)$ penalty upon updates. Nevertheless, our experiments show that this approach remains highly effective.

In Figure 1.1, we illustrate the universe of GCQs and its subclasses, where each subclass of GCQs is shown as a circle and labeled. Since NCQs are a subclass of GCQs, Table 1.1 complements Figure 1.1 and summarizes the theoretical results discussed-above for *acyclic* NCQs and its subclasses.

Finally, we experimentally compare our approach against HIVM on the industry-standard benchmarks TPC-H and TPC-DS. Our experiments show that, for the class of acyclic NCQs, our method is up to one order of magnitude more efficient than HIVM, both in terms of update time and memory consumption. At the same time, our experiments show that the enumeration of $Q(db)$ from $\mathcal{D}_{db}$ is as fast (and sometimes, even faster) as when $Q(db)$ was materialized as an array. These contributions of the thesis (fully presented in chapter 3) has been published in SIGMOD 2017 [IUV17].

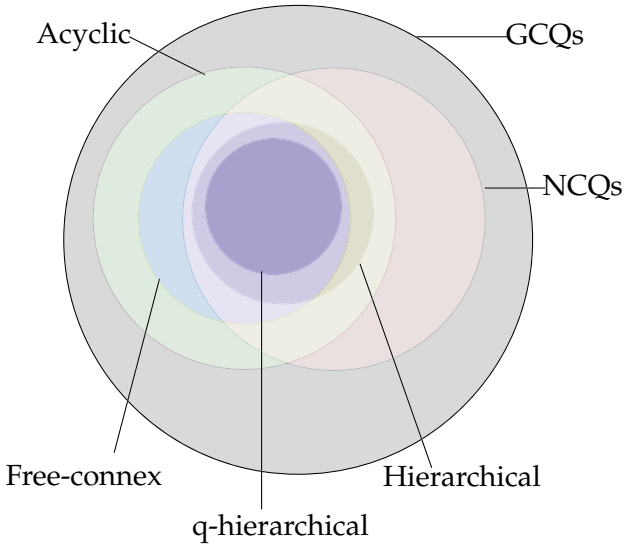


Figure 1.1: Classification of GCQs and its sub-classes of queries.

Query class	GJT	Update time	Enumeration delay	Space
Free-connex	Compatible	$O(\|db\| + \|u\|)$	$O(1)$	$O(\|db\|)$
Hierarchical	Simple	$O(\|u\|)$	$\omega(1)$	$O(\|db\|)$
q-hierarchical	Simple and compatible	$O(\|u\|)$	$O(1)$	$O(\|db\|)$
Acyclic	Arbitrary	$O(\|db\| + \|u\|)$	$\omega(1)$	$O(\|db\|)$

Table 1.1: Theoretical results for the different classes of acyclic NCQs in the DYN framework.

1.4.2 Contributions for GCQs

Our specialized approach DYN and several other approaches (e.g., [NO18]) to dynamically process aggregate-join queries are only applicable to queries with equality joins, and as such they do not support analytical queries with other types of joins like the ones with inequalities ($\leq, <, \geq, >$) or disequalities ($\neq$). Therefore, the current state of the art techniques and DYN for dynamically processing queries with joins beyond equality suffer either from a high update latency (if subresults are not materialized) or a high memory footprint (if subresults are materialized).

We overcome these problems by generalizing the Dynamic Yannakakis Algorithm (DYN) to conjunctive queries with arbitrary θ-joins. We show that, in the specific case of inequality joins, this generalization improves the state of the art for dynamically processing inequality joins by performing consistently better, with up to two orders of magnitude improvements in processing time and memory consumption. In GDYN, we focus on the dynamic evaluation of acylic queries under multiset semantics and present the following contributions.

(6) We extend our succinct data structures DCLRs to dynamically process GCQs. To this end, we first extend and generalize the notions of acyclicity and free-connexity to queries with arbitrary θ-joins (chapter 4 section 4.2). Our data structure degrades gracefully: if a GCQ only contains equalities our approach inherits the worst-case optimality provided by DYN.

(7) We present GDYN, a general framework for extending DYN to free-connex acyclic GCQs. Our treatment is general in the sense that the θ-join predicates are treated abstractly. GDYN hence applies to all predicates, not only inequality joins. We analyze the complexity of GDYN, and identify properties of indexing structures that are required in order for GDYN to support constant delay enumeration of results for a higher class of GCQs and bounded delay enumeration for the rest of queries as well as efficient update processing.

(8) We instantiate GDYN to the particular case of inequality and equality joins. We show that updates can be processed in log-linear time and results can be enumerated with logarithmic delay. Moreover, if there is at most one inequality between any pair of relations, then results can be enumerated with constant delay. We call the resulting algorithm IEDYN.GDYN satisfies the properties $P_1^* - P_4^*$ of DCLRs.

Table 1.2 compliments Figure 1.1 and summarizes the theoretical results for *acyclic* GCQs discussed-above in the GDYN framework. In this table, GCQ $^{\leq 1}$ denotes GCQs that feature only single inequality between a pair of relations, and GCQ $^{>1}$ denotes GCQs that feature more than one inequalities between pairs of relations.

Finally, we experimentally compare IEDYN against state of the art HIVM and CEP frameworks. Our extensive experiments show that IEDYN performs consistently better, with up to two order of magnitude improvements in both speed and memory consumption. These contributions have been published in the international conference on Very Large Databases (VLDB'18) [IUV+18].

Query class	Update time	Enumeration delay	Space
GCQs $^{\leq 1}$	$O((\|db\| + \|u\|)log(\|db\| + \|u\|))$	$O(1)$	$O(\|db\|)$
GCQs $^{>1}$	$O((\|db\| + \|u\|)^2log(\|db\| + \|u\|))$	$O(log\ \|db\|)$	$O(\|db\|)$

Table 1.2: Theoretical results for different classes of GCQs in the GDYN framework

1.4.3 Contributions for Computing GJTs

Traditionally, join trees define the order in which relations in a traditional conjunctive query Q are joined. Usually, each node in a join tree represents a relation in the query Q, and the important question than is, how to link/join those nodes to compute the query results efficiently. Extensive literature exists in this respect that we will discuss in chapter 2 in detail. Most of the existing works only address this issue for static queries i.e. join trees (also called evaluation plans) are defined and computed for static settings. We, however, present GJTs that not only support GCQs in static settings, but also support evaluation of GCQs in dynamic settings e.g. in case of DYN and GDYN. Our contributions in this regard are as follows:

(9) We present the notion of *generalized join trees* (GJTs) for NCQs and GCQs. These GJTs are the building blocks for the DCLRs and ensure the theoretical guarantees $P_1 - P_4$ and $P_1^* - P_4^*$ of DCLRs.

(10) GYO-reduction is an algorithm that tests a traditional conjunctive query (with equi-joins only) for acyclicity [AHV95]. We present an extension of the traditional *GYO-reduction* to test a GCQ Q for acyclicity.

(11) We extend the GYO-reduction such that, if a GCQ Q is acyclic then it must return a GJT T for Q. Moreover, our algorithm provides necessary theroetical guarantees and works for queries with projections as well.

These contributions have been submitted to the VLDB Journal for review.

1.5 Statement

The work in this dissertation is sponsored under the Erasmus Mundus Joint Doctorate in Information Technology for Business Intelligence – Doctoral College (IT4BI[1]) cotutellate program between Universite Libre de Bruxelles[2], Belgium and Technische Universität Dresden[3], Germany. This dissertation is drawn from several joint works with my colleagues. The dissertation contains theoretical and empirical results from the papers of Martin Ugarte, Stijn Vansummeren, Hannes Voigt, Wolfgang Lehner and myself [IUV17, IUV$^+$18]. In the following I enumerate the specific contributions of each author.

Chapter 1 contains main ideas which can be attributed to Stijn Vansummeren, Martin Ugarte, Wolfgang Lehner, Hannes Voigt and myself [IUV17, IUV$^+$18]. Chapter 2 has roots in [IUV17, IUV$^+$18] by Stijn Vansummeren and myself. Sections 3.1 - 3.2.6 in chapter 3 contains theoretical results with respect to evaluation of NCQs under updates which are obtained and formulated by Martin Ugarte, Stijn Vansummeren and myself; and section 3.3 contains practical results that are obtained by myself under the guidance of Stijn Vansummeren.

Chapter 4 presents the idea of the GDYN framework from [IUV$^+$18] which can be attributed to Martin Ugarte, Stijn Vansummeren, Hannes Voigt, Wolfgang Lehner and myself. The main idea and theme of this chapter is from Stijn Vansummeren, Wolfgang Lehner, and myself. The results in sections 4.1 - 4.2 are obtained by Stijn, Vansummeren, Martin Ugarte, Hannes Voigt and myself. The empirical results in Section 4.5 are obtained by myself under the supervision of Stijn Vansummeren and Hannes Voigt. Chapter 5 presents theoretical results from our work that is currently submitted for a review to the VLDB journal, and contains results that can be mainly attributed to Stijn Vansummeren and myself.

1.6 Thesis Structure

This thesis is further organized as follows. We discuss related work with respect to evaluation of NCQs, GCQs in static as well as dynamic settings, computation of join trees, and constant delay enumeration in chapter 2. In chapter 3 we present our approach DYN for evaluating NCQs over DCLRs and in chapter 4 we present GDYN for evaluating GCQs over DCLRs. We will define and use GJTs throughout chapters 3 and 4 for building DYN and GDYN. We devote chapter 5 to present algorithms for computing GJTs for NCQs and GCQs. Finally, we discuss the future directions and conclude the discussion in chapter 6.

[1] https://it4bi-dc.ulb.ac.be/
[2] ULB
[3] TUD

2

RELATED WORK

In this chapter we present the state of the art related to query evaluation under updates in BI and IFP systems; constant delay enumeration; and join tree computation. In Figure 2, we present the taxonomy of the related work and this chapter is organized according to this taxonomy. Moreover, at the end of each section in this chapter, we present a comparison of supported features and functionalities of the most relevant systems as compared to our proposed unified model (GDYN) in this thesis.

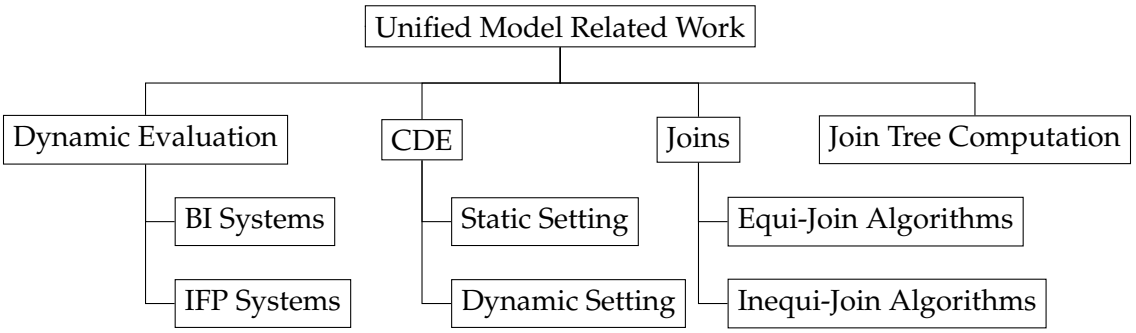


Figure 2.1: Taxonomy of related work

2.1 Dynamic Query Evaluation

The problem of dynamic query evaluation has been extensively studied in the literature. In BI systems, it has been mainly addressed under the umbrella of incremental view maintenance (IVM), and in IFP systems it has been studied most often as continuous query processing and composite event recognition. In the following, we discuss dynamic query evaluation in both types of systems in detail.

2.1.1 BI systems

One of the early methods to support incremental computation is the RETE algorithm developed for production rule systems [For82]. A production system is defined by a set

of rules called productions. Each production has an IF (condition) and a THEN (else) part. A production is closely related to a conjunctive query with the exception of supporting negated atoms. The RETE algorithm evaluates rules by means of RETE trees and these trees can be seen as evaluation plans where internal nodes (also called Beta memories) materialize the partial results so that re-evaluation of these partial results could be avoided on updates. RETE is superseded by TREAT [Mir14] and RETE* [WM03]. Both TREAT and RETE* propose improvements to the size of materialized subresults (Beta memories). In particular, TREAT completely drops materialization of partial results at beta nodes and hence re-computes them under updates whereas RETE* maintains a middle ground by providing fine-grained control over the space-time trade-off by providing controlled storage of productions at beta memories through a mechanism called dynamic beta cut i.e. it treats insertions and deletions asymetrically. Similarly, in another work by Francois et al. in [FRS93], an incremental algorithm has been proposed that determines what needs to be materialized (not necessarily all subresults) to speedup production rules processing. In particular, they maintain/materialize a set of relational expressions in response to input rules for program evaluation.

In data warehousing systems, many approaches have been proposed to shorten the data loading (ETL) intervals (data latency) while keeping the indexes and other structures upto date such as in real-time and active data warehousing. This is typically done by recording fresh tuples in *staging* areas, which are tables that feature low insertion overhead (to allow rapid insertions) and yet support high-performance querying (to allow analysis). Queries are answered transparantly on data from both the staging area and the static data. Periodically, data in the staging area is indexed and moved to the static area. One of the solutions in this regard defines real-time partitions for fact tables where real-time data is stored separately from static data. Realtime partitions maintain three types of partitions having same schemas as fact tables including transaction granularity, periodic snapshot, and accumulating snapshot. The transaction granularity contains only the transactions occuring since last fact table refresh, periodic snapshot contains the records of current snapshot period and summarized at the granularity of fact table on period completion, and accumulating snapshot are used to process short updates such as order handling [FMF13].

There are several other proposed techniques to achieve the goal of real-time data warehouse. Trickle and Flip is an approach to continuously feed data in data warehouse as staging tables (copy of fact tables). It periodically swaps staging tables with fact tables to reflect changes [Zut11]. Realtime data caching is another approach to real-time data warehouse where incoming data is stored in a cache and processed to fulfill the fast processing requirement [ZAL08]. In-memory data processing systems can be exploited to handle large volume of data. In some cases, part of the data may need to be processed directly after arrival while in others, the data can be loaded and processed at regular intervals. This approach is known as Right-time data warehousing, where the right time could be current time instance or many seconds and hours, it needs to load data once needed. The advancements in real-time data acquisition and capturing has led to new loading paradigms in data warehouses such as Extraction-Load-Transform (ELT). In ELT, data from sources are staged and after necessary corrections it is stored in the warehouse where transformations can be performed. This technique adds the ability to integrate new data sources into data warehouses [VZ14].

Traditionally, in the databases and data warehousing community, base relations are used to compute views. Views are derived data and can be stored in the database management system (materialized) to query against. When some of the relations change (i.e. when an update occurs to the database) then the materialized views also need to be updated accordingly to ensure that the database is at the correct state. In essence, recomputing the materialized views on each update (change) to the database is an expensive operation, hence it makes sense to define and recompute the changes that needs to be

made to the view. This has been studied under the umbrella of *Incremental view maintenance*, and it has been addressed from both set and bag semantics [GL95]. In a recent work, HIVM was introduced by Y. Ahmed et al. [AKKN12] to not only materialize views defined by queries but also materialize the partial query results (also called sub-results) and maintain them under updates. HIVM is fast in updating views, however, it also has to maintain the partial results at the same time.

Unfortunately, materializing views (as query results and partial results) in a main-memory model is expensive and not well-suited for large databases that change frequently, in particular when the update frequency is very high. This is because on the one hand, the memory footprint required to materialize the results is high and on the other hand, if only the query results are materialized (without materializing partial results) then the recomputation cost is very high, as discussed in chapter 1 Section 1.1.1. For example, for a join query $Q = R(a, b) \bowtie S(b, c) \bowtie T(c, d)$, HIVM not only materializes the results of Q, it also materializes the partial joins $R \bowtie S$, $S \bowtie T$ and $R \bowtie T$ so that when an update arrives to one of the base relations, then these partial results can be used to compute changes to the result of Q. Here, the join $R \bowtie T$ is a cross product and the result can be quadratic in the size of the input database. In this thesis, we address this memory and update delay tradeoff in the following chapters.

2.1.2 IFP systems

A survey by Cugola and Margara [CM12b] classify IFP systems into *active databases, data stream management systems (DSMS) and complex event processing systems (CEP)*. Moreover, processing streaming data has also been addressed in the big data world. In the following we will present an overview of each of these systems.

Active Databses and DSMS

Historically speaking, the first systems that provided support for reactive behavior in the database community on the detection of pre-defined situations on updates were the so-called active databases with HiPac [DBB+88], Ode [LGA96], Samos [GGDD91], and Snoop [CM94] being notable examples. Active databases are extensions of classical DBMS' that allow triggers to be installed that automatically allow certain actions to be taken if certain modifications are made to the database. As such they are meant to provide a reactive behavior and this behavior is decoupled (partially) from the application layer to the DBMS itself. However, like traditional database, active database persist data on disk where modifications are relatively infrequent and hence this inherently limits the number of registered reactive rules. Active databases use RETE trees (see section 2.1.1) or classical IVM techniques.

To allow processing of high-rate streams, the so-called *data stream management systems* (DSMS) were developed. These systems process *continuous queries* over transient data sources. DSMS' mostly work by first setting up a data flow network consisting of variants of well-known relational algebra operators (i.e. an evaluation plan), and optimize this plan dynamically so that the streams are processed based on the observed stream behavior and statistics. Notable examples of DSMS include academic prototypes such as Aurora [ACÇ+03], Borealis [AAB+05], and STREAM [ABW06] as well as commercial systems such as Coral[1], StreamBase[2] and Oracle CEP [Pro09]. Aurora is one of the earliest

[1] www.coral8.com
[2] www.streambase.com

centralized streams processing system specified by drawing network of boxes and arrows working as operators and data streams respectively. Operators execution can be prioritized based on quality of service specifications. Similarly, Aurora and Medusa [CBB+03] is an extension of the original Aurora where distributed processing on federation of computing nodes runnining instances of Aurora are enabled while implementing resource multiplexing, fault tolerance and other distributed processing functionalities. STREAM on the other hand provides support for a continuous query language (known as CQL) that provides the operators such as mapping of input streams to relations, relations to relations and mapping of relations to streams. In essence, the main workload is done by the relation to relation operators while the other two operators mainly work as transformation between streams and relations. The CQL in STREAM maintains the intermediate data while evaluating the stream and hence suffers from the runtime memory footprint. In particular, most of the data streaming systems assume that, the state that needs to be maintained is small and hence they materialize the partial as well as full results. This, however, is not always the case especially when the input streams are timestamped and the joins are inequalities or the state needs to be big.

With the advent of big data, some of the techniques for DSMS have been further refined in the big data streaming processing systems. Among others two notable systems are Storm [TTS+14] and Spark streaming [ZDL+13]. Many other approaches have developed in mapreduce in the Hadoop framework. In the following we, however, discuss Storm and Spark streaming. Storm is a distributed stream processing system tracing its lineage back to the earlier systems like Borealis, packaged in a single open-source framework. It employs topological directed acyclic graph(DAG) of operators (also called spouts and bolts). A spout is an external communicating node while bolts perform the role of actual executers at each individual node. The whole framework works in a master-slave architecture and provides intra-topological parallelism. Spark streaming on the other hand models a stream as a sequence of resilient distributed datasets (RDDs) called DStreams. RDDs are immutable and support a wide set of operation (actions) including map, group by, filter etc. in a micro-batching fashion. DStreams can recover in case of failures through its lineage structure that it maintains as RDDs. Both of these systems are designed for distributed settings and generally work on micro-batches (not fine-grained updates).

Composite Event Recognition

There is a multitude of research on Composite Event Recognition (CER). For an overall background and depth of the field, we refer the reader to [AMU+17]. Unlike traditional BI systems, CER systems mostly do not maintain the legacy (historical) data for long-term analytical queries and are mainly based on windowing semantics. Hence, the goal is to react to updates in the current window. An important direction in CER research is the design of declarative languages for specifying CER patterns. In this thesis, however, we focus on the dynamic execution of events joins. In particular, we focus on the multi-way θ-joins, which naturally occur in CER **SEQ** patterns as exemplified by the fraud query in chapter 1. As discussed in chapter 1, CER systems can be categorized in general into two categories: those that are based on relational model and those that are based on automaton model.

In either of the models i.e. relational and automaton, the following two approaches have been adopted for the evaluation of sequence of events. Suppose the streams of events A,B,C in a sequence **SEQ(A,B,C)** where A must occur before B, and B must occur before C. In the first approach, whenever there is an update in B, all the matching As are searched (using particular indexes) and the partial join result (AB) is materialized so that this partial result can be used to generate the full sequence of events when an

		BI systems		CER systems		GDYN	
		IVM	HIVM	Automaton	Relational	DYN	IEDYN
Equalities	Updates	-	-	-	-	✓	✓
	Enumeration	✓*	✓*	✓*	✓*	✓	✓
	Memory	×	×	×	×	✓	✓
Inequalities	Updates	×	×	-	-	×	✓
	Enumeration	✓*	✓*	✓*	✓*	×	✓
	Memory	×	×	×	×	×	✓

Table 2.1: Comparison of GDYN against BI and IFP systems: (×) means the corresponding functionality (reducing update processing time, CDE, or reducing runtime memory footprint) is not addressed, (-) means supports functionality with good performance but degrades because of the memory-time trade-off, (✓) means supports with good performance, and (✓*) for enumeration means CDE when query results are materialized while without "*" means without materialization.

update occurs in C. In the second approach, instead of materializing the partial results, they are recomputing whenever there is an update in C. For example, when an update occurs in A or B, nothing happens except A and B being stored separately. However, when an update arrives in C then first B is searched and for each B, A is searched and the final output is produced. Note that the first approach suffers from memory footprint since a sequence is normally interpreted as inequality join and the latter approach suffers from update latency. The first approach resembles the HIVM approach where the query results and partial results are both materialized. The latter case is similar to the lazy evaluation. Note here that materialization speeds up the processing time, however, it increases the memory footprint exponentially as is the case in HIVM. On the other hand, recomputation approach is memory efficient, yet, suffers from high update processing cost.

In this thesis, we address this trade-off between memory footprint and update processing time for CER systems as well as BI systems. Note that, our solution (as we will show in chapters 3 and 4) maintains partial results without full materialization in DCLRs (i.e. the views are maintained but not fully materialized), and the full query results can be enumerated from DCLRs without materialization and with the same complexity (or even better) just as it is done from a materialized in-memory array. In the following section, we present work related to the enumeration of query results with constant delay. Table 2.1 shows comparison of the BI and IFP systems against GDYN for processing of updates, enumeration of query results and the runtime memory footprint for both equality and inequality conjunctive queries. In this table, we only show the comparison for dynamic processing systems, and show that both IFP and BI systems support update processing, enumeration of results, however, they do not address the runtime memory footprint, and hence their update processing time degrades because of the memory and update trade-off.

2.2 Constant Delay Enumeration

Constant delay enumeration (CDE for short) has received increasing attention in various research directions. In database terms, CDE amounts to enumerate each tuple in the result $Q(db)$ from a data structure D where Q is a conjunctive query over the database db. This enumeration is such that the delay to enumerate the first tuple, the delay between a

pair of tuples in the result and the delay to enumerate the last tuple is in constant time, and is independent of the input database and the query output. Most of the works in CDE decouple the work required to build the data structure D and the effort to enumerate $Q(db)$. In [BDG07a], Bagan et al. studied CDE for the class of acyclic conjunctive queries and in particular a subclass of acyclic queries called *free-connex acyclic* queries. They show that for such queries, $Q(db)$ can be enumerated with constant delay after linear time precomputation. In their further theoretical results, they show that if a query is not in the class free-connex acyclic, then CDE does not hold after linear time precomputation. Moreover, they also show that if a query has a tree-width bounded by a constant k, then we require $O(|D|^{k+1})$ steps of precomputation to support CDE.

There is an increasing work on CDE, we refer the reader to [Seg14] for a survey. In general, all CDE techniques separate query processing life cycle into a preprocessing stage (precomputation stage) and an enumeration stage, and guarantee that, during enumeration, only constant amount of work is required to produce each new result tuple. However, most of the existing approaches either work for static settings, or present theoretical results for a restricted class of queries with equi-joins only. In the following, we first present practical CDE for static setting and then theoretical results of [BKS17] for the dynamic setting.

2.2.1 Practical and Static CDE

In a relatively new method of representing relational data in the database systems instead of using the traditional NSM (N-ary storage model) or DSM (decomposed storage model, also called column store), Olteanu et al. [OZ12] have introduced factorized databases. Since the NSM and DSM models incur high redundancy in the data particularly in the query results, they incur high memory footprint. Factorized representations, also called *f-representations* for short, are succinct representations where tuples are represented by expressions and each expression is either: a unary relation of a single data item, a union of two expressions, or a product of two expressions. The succinctness of the f-representations results from the distributivity of products over union as shown in Figures 2.3 and 2.4, and exemplified next. Consider a query Q over the relations Customer(custkey, custname), Orders(okey, custkey, odate), Item(okey, disc) in Figure 2.2 as follows:

$$Q = Customer \bowtie Order \bowtie Item.$$

The f-representations of the result of Q are shown in Figure 2.4 as an expression. and Figure 2.3 shows the f-representations in its flat (expanded) form. These f-representations are encoded into factorized trees (f-trees) which shows the dependency of attributes in a relation. Since a tree will give a dependency of attributes, it hence intuitively gives a way to perform some operations such as group-by and aggregates (i.e. SUM, AVG etc.). Moreover, as it can be seen in the Figure 2.4 that f-representations can be orders of magnitude memory efficient.

In a further attempt, the f-representations have been extended to D-representations that encode the sub-expression in f-representations [OZ15]. It has been shown that for a conjunctive query Q on database D, there exist parameters $s(Q)$, $s^{\uparrow}(Q)$, $\rho^{*}(Q)$ such that for $Q(D)$:

- size of f-representation is $O(|D|^{s(Q)})$
- size of d-representation is $O(|D|^{s^{\uparrow}(Q)})$

Cust	
ckey	name
1	Joe
2	Dan
3	Li
4	Mo

Ord		
ckey	okey	date
1	1	1995
1	2	1996
2	3	1994
2	4	1993
3	5	1995
3	6	1996

Item	
okey	disc
1	0.1
1	0.2
3	0.4
3	0.1
4	0.4
5	0.1

Figure 2.2: Relations *Customer, Order, and Item*

Ckey		Name		Okey		Date		Disc	
1	x	Joe	x	1	x	1995	x	0.1	U
1	x	Joe	x	1	x	1995	x	0.2	U
2	x	Dan	x	3	x	1994	x	0.4	U
2	x	Dan	x	3	x	1994	x	0.1	U
2	x	Dan	x	4	x	1993	x	0.4	U
3	x	Li	x	5	x	1995	x	0.1	

Figure 2.3: Relational representation of $Q = Customer \bowtie Order \bowtie Item$

1	x	Joe	x	1 x 1995 x (0.1 U 0.2)	U
2	x	Dan	x	(3 x 1994 x (0.4 U 0.1) U 4 x 1993 x 0.4)	U
3	x	Li	x	5 x 1995 x 0.1	

Figure 2.4: Factorized representation of $Q = Customer \bowtie Order \bowtie Item$

- size of flat relational result is $O(|D|^{\rho^*(Q)})$, where $\rho^*(Q)$ is the factional edge cover of the query Q.

Moreover, the upper bounds are tight and there exists an arbitrary database D for which $Q(D)$ admits f-representations of size $\Omega(|D|^{s(Q)})$, d-representation of size $\Omega(|D|^{s\uparrow(Q)})$ and flat relational result of size $\Omega(|D|^{\rho^*(Q)})$. It is shown that $1 \leq O(|D|^{s\uparrow(Q)}) \leq O(|D|^{s(Q)}) \leq O(|D|^{\rho^*(Q)}) \leq O(|D|^{(Q)})$. Since f-representations are logically compressed structures, they can be used to enumerate the query result with constant delay simply by visiting each path from root to leaf giving a tuple in the result. It is interesting to see that not only query results can be stored as f-representations, partial results and input database can also be stored as f-representations In general, f-representations are more efficient when there is high dependency among attributes in a relation/result. CDE has also been studied with regards to enumeration of *regular document spanners* from text documents [FRU+18]. In general, data of interest in a document can be very large in size, in this work the authors propose an approach to maintain a data structure that allows enumeration of spanners with CDE after a linear time precomputation. However, in this approach, the spanners are based on variables set automata and not on relational model.

			f-representations	GDYN		other
				DYN	IEDYN	
CDE	Equalities	Static	✓	✓	✓	✓
		Dynamic	✓	✓	✓	✓*
	Inequalities	Static	×	×	✓	×
		Dynamic	×	×	✓	×

Table 2.2: GDYN against f-representations and other approaches: (✓) - CDE is supported, (✓*) - CDE is shown to be supported theoretically, (×) - CDE not supported or not discussed

Closely Related Work. In a recent work [OZ15], Olteanu et al. present *covers* for query results i.e. cover represents minimal subset of query result from which one can losslessly reconstruct the query result itself. These results are only applicable for static queries. For dynamic queries, an approach named *factorized-IVM* (F-IVM), based on *f-representations*, is developed to dynamically process aggregate-join queries that are not necessarily acyclic [NO18]. It is important, however, to note that F-IVM addresses NCQs that are not necessarily acyclic, hence addresses a higher class of queries for dynamic evaluation without θ-joins, and F-IVM also support complex aggregates. Moreover, the F-IVM contributions for acyclic NCQs exactly match to that of our DCLRs for NCQs. However, F-IVM was published after our work on conjunctive queries with equalities under updates [IUV17].

2.2.2 Theoretical and Dynamic CDE

Berkhloz et al. in [BKS17] show that a subclass of conjuctive queries can be efficiently maintained under updates. They characterize the class of acyclic self-join free NCQs that feature CDE and that can be maintained in $O(1)$ time under single tuple updates after linear time preprocessing. This particular subclass of queries is called the class of *q-hierarchical* queries, a strict subclass of acyclic free-connex queries. The notion of *hierarchicality* is closely related to the hierarchical property discussed earlier in [DS13]. For queries that are self-join free but are not hierarchical (not in the restricted subclass), they show that it is unlikely to process updates with sublinear time and to enumerate query results with sublinear delay. Moreover, for answering queries with counting results or bolean queries, they show that if the query is q-hierarchical then the count of query result can be computed in linear time and maintained in constant time under updates. Otherwise, the size cannot be maintained under sublinear update time. It is important to note here that the results presented in [DS13] are theoretical results for NCQs only and our DCLRs match their theoretical bounds for NCQs. Moreover, the DCLRs not only achieve these results for NCQs, it also works for GCQs that feature θ-joins and presents a general optimal framework.

In Table 2.2 we show the comparison of CDE for GDYN against state of the art practical and theoretical CDE supported systems discussed above. In this table, we only show summary that f-representations supports CDE practicaly for NCQs (equalities only) in both dynamic and static settings, other approaches either only provide practical static CDE or theoretical dynamic CDE, and GDYN provides CDE for NCQs and GCQs in both static and dyanmic settings. Note here that for GCQs, GDYN supports CDE for a higher sub-class of queries that have at most one inequality constraint between pair of relaitons, and for the rest it supports enumeration with bounded delay. Since, here the comparison is for CDE, we show only for those queries where GDYN supports CDE.

2.3 Join Algorithms

Efficiently processing joins is one of the most fundamental and widely studied problems in the database research community. In this section, we briefly present an overview of the related join algorithms for both equi and inequality joins.

2.3.1 Equi-Join Algorithms

Relational joins evaluation is one of the classic and well studied problem in the database community. Numerous join processing algorithms have been presented over time including *Nested loop join, Hash join, Sort-merge join* among others (see [NRR14] for a survey). Commercial database systems go one step further by incorporating semantics such as cardinalities of relations, IO, memory etc. In particular, there are two main approaches to evaluate joins: examining the structural properties of the join, and cardinality estimations. In the former approach, most theoretical algorithms consider the structural properties of join queries such as testing if a query is acyclic or not, or defining the "width" of a query. As an example, the classical Yannakakis algorithm [Y81] performs well when the query is acyclic (of width one) i.e. the join of an acyclic query can be computed in time linear in the size of input database *db* and output $Q(db)$ of a query Q over the database *db* (i.e. $O(||db||+||Q(db)||)$). In chapter 3 we present a detailed working of the Yannakakis algorithm since this algorithm is the baseline for our GDYN framework. The latter approaches of cardinality estimations mainly focus on the actual input database cardinalities. Commercial database systems in general use the heuristics and adopt pair-wise join processing of a multi-way join, as initially presented in [SAC+79]. As rightly said by the authors in [NRR14], "by throwing away the structural properties, any join project plan is destined to be slower than the best optimal runtime by a polynomial time in the data complexity". This holds true for systems that only consider one of the approaches for defining the join plan.

Works on structural properties of queries have further been extended to study and investigate whether a query is tractable or on in the sense that there there exists a polynomial solution to process it. Then other solutions such as [GLS03a] discuss "hypertree width", the "treewidth" related to the hypergraph of a query. In [AGM08], the authors presented much finer method of bringing the approaches based on structural properties of the query and heuristics together by presenting tighter bounds on the size of query results, also known as the AGM bound. They presented bounds in terms of the query width as well as the input cardinalities. In their proofs, they have introduced the notion of *fractional hypertree width* (fhw) and show that

$$fhw \leq ghw \leq qw \leq tw + 1$$

where ghw, qw, and tw are *generalized hypertree width, query width, and tree width* respectively. We refer the reader to [GM14] for more details.

In the first optimal join algorithm presented by Ngo et al. [NPRR12], the authors show that given the bounds on input relation sizes, the runtime of their join algorithm is bounded by the maximum output size possible. Another algorithm called "Leap-frog Trie Join" that is implemented in the commercial database system *LogicBlox*[3] and is also optimal algorithm, was shortly presented after the first optimal algorithm by Ngo et al. The Leap-from trie join algorithm takes into account many traditional database heuristics and optimally handles skew in the data [Vel12]. Apart from these join algorithms, works

[3] https://developer.logicblox.com/

on processing the traditional triangle (cyclic) query $Q_\Delta = R(a,b) \bowtie S(b,c) \bowtie T(c,a)$ have been presented that include the "the power of two choices" [DNS+92] and the "delaying computation" [NRR13] algorithms. Since cyclic queries are out of the scope of this thesis, we omit details in this respect.

It is important to note here that a wide range of works exists on query optimization and join processing. Most of the existing join algorithms consider the traditional join trees as the join evaluation plan, and then it remains an important question to determine which join ordering is the best to reduce the space and time complexity of the algorithm [SMK97]. However, most of the existing works related to join algorithms focus on static settings without updates. In this thesis, we build upon the classical Yannakakis algorithm (optimal join algorithm for queries of width one i.e. acylic) and present our variants of it for multi-way θ-joins and show that this works under updates efficiently.

2.3.2 Inequality-Join Algorithms

In the static setting, common ways to compute an inequality join between a pair of relations include variants of sort-merge join and partition join [DNS91], as well as index joins using interval-based indexing [HNP95, EHS04]. Unlike equi-joins, inequality joins have less attracted the attention of the research community and the existing works mainly focus on joins between pair of relations. Existing works such as [BG81, YK84] present evaluating inequality joins based on semi-joins. In [BG81], the authors present an approach to process queries that feature natural inequalities such as $(\leq, <, >, \geq)$ in the static setting. However, this work focused on single inequality joins between relations. In [YK84], the authors extend the work presented in [BG81] by incorporating queries that do not necessarily feature single inequalities and present a multi-variable semi-join reduction.

In a recent work [KLS+17], Khayyam et al. have proposed fast multi-way inequality join algorithms based on sorted arrays and space efficient bit arrays. In this work, they focus on pairwise joins with exactly two inequality conditions between variables/attributes. They also present an incremental algorithm to compute such inequality join under updates. However, their algorithm does not take into account the constant delay enumeration of query results and makes no effort to minimize the update processing cost. Rather, after each update processing step, they require at-least linear processing of the data structure to enumerate query results (delta result). As a result, they either need to materialize the partial or full results (incurring space-overhead that is potentially more than linear) or recompute subresults on each update (incurring update time-overhead).

Other common systems and approaches such as CER continuous query processing systems, mostly make use of the sort-merge joins and their variants to process inequality joins. These systems typically compute inequality joins based on timestamp variables. The whole focus is how to enrich the model (relational or automaton) with a maximum amount of functionality in the sense of supporting operators and enriching the system. Therefore, the effort is less made to make inequality joins in the static as well as in the dynamic setting efficient.

In this thesis, we are concerned with maintenance of queries that feature multi-way θ-joins with at-least one inequality join condition between variables (attributes) of a pair of relations under updates. Just like for equi-joins, we separate the (pre)computation phase from the enumeration phase for inequality joins and provide algorithms that can support enumeration of query results with bounded delay. Our algorithms for inequality joins work for both static and dynamic settings, and support enumeration with bounded delay for query and delta results.

		Reducers		
		GYO	B&G	GDYN
Cyclicity check	NCQs	✓	✓*	✓
	GCQs	×	✓*	✓
Join trees	NCQs	✓	✓	✓
	GCQs	×	✓*	✓

Table 2.3: Acyclicity test and GJT construction for NCQs and GCQs in GDYN against GYO-reduction and Bagan & Goodman (B&G) semi-join inequality reduction: (×) means cyclicity test and join tree construction not supported or discussed, (✓) means supported, and (✓*) means supported partially or with limited support.

2.4 Join Tree Computation

The problem of evaluating joins efficiently has been extensively addressed in database systems [SMK97]. The main theme in these works can be divided into two different ways: how the joins of relations in a query itself should be evaluated (the join strategy e.g. Nested loop join etc.) and the algorithms to determine the order in which the joins should be evaluated. In this section, we focus on the latter i.e. the order in which the joins of relations should be evaluated. Moreover, the order in which the joins should be evaluated has been studied in terms of defining join trees for queries (especially for queries that are acyclic) of different widths [AGM08].

One of the worst-case optimal algorithms for evaluating joins of queries that are acyclic is the Yannakakis algorithm [Y81] where the notion of join trees for acyclic queries was introduced. To test a query for acyclicity, the notion of *generalized hypertree decomposition* (GHD) was introduced. In particular, a conjunctive query that admits a GHD of width one is an acyclic query. Moreover, another algorithm called *GYO-reduction* determines the acyclicity of a query by means of defining a reduction procedure of the hypergraph of the query [AHV95]. In particular, the GYO-reduction determines a query to be acyclic if the hypergraph associated to it can be fully reduced (i.e. after finite number of *reduction* operations defined in the GYO-reduction, the hypergraph must result into an empty hypergraph). We explain the working of GYO algorithm in detail in chapter 5. However, these algorithms, the GYO-recution algorithm to test a conjunctive query for acyclicity and the Yannakakis algorithm to compute the join of relation in a conjunctive query, only work for queries that feature equi-joins (NCQs). In [BG81], Bernstein and Goodman consider conjunctive queries with inequalities and classify the class of such queries that admit full reducers. A *full reducer* presented by Bernstein and Goodman in [BG81] is defined as: a query Q admits a full reducer, given a database *db*, if there exists a program in the semi-join algebra (a variant of relational algebra where the joins are replaced by semi-joins) that selects a *minimal* subset of *db* needed to answer Q. However, the notion of acyclicity by Bernstein and Goodman only considers queries where a pair of relations can have at most one comparison of attributes. In other words, there can be at most either one equality or inequality constraint between a pair of relations. We, in contrast, investigate reducers for arbitrary number of joins (comparisons) between pair of relations in multi-way joins. To ensure the properties defined for DCLRs in the introduction of this thesis, we present the notion of generalized join tree GJTs for acyclic queries. Unlike traditional join trees, GJTs are special trees where the relations always remain at the leaves of the tree and internal nodes are sets of variables. The important question then is, to find particular subsets (set of variables at internal nodes) to join the relations at the leaves. To this end, we first extend the GYO-reduction to test a query with θ-joins for acyclicity and compute a GJT if it is acyclic. Then, we exploit the basic idea of Yannakakis algorithm and extend it to work on GJTs i.e. queries with θ-joins and

support update processing and enumeration with constant delay. In Table 2.3, we show the summary of checking conjunctive queries for acyclicity and generating GJTs or traditional join trees for queries that are acyclic in GDYN against the GYO-reduction and the semi-join reduction algorithm presented by Bagan and Goodman.

3

THE DYNAMIC YANNAKAKIS ALGORITHM (DYN)

In this chapter, we start our study of dynamic query processing. In order to illustrate the main ideas underlying our solution, we will focus in this chapter on the processing of acyclic conjunctive queries with equi-joins (NCQs) only. We show how to generalize this solution to accommodate θ-joins in chapter 4.

Specifically, we obtain DCLRs for processing acyclic conjunctive queries dynamically by building on the well-known algorithm by Yannakakis [Y81] for processing acyclic conjunctive queries in the static setting. Concretely, we show that a modified version of this algorithm, which we call *Constant Delay Yannakakis* (CDY for short) allows to enumerate query results with constant delay in the static setting. We then further modify this algorithm into *Dynamic Yannakakis* (DYN for short) to accomodate the dynamic processing of updates. The data structures processed by DYN are DCLRs. We next analyze the complexity of DYN, showing in particular that its complexity is optimal for two important subclasses of the acyclic conjunctive queries. Finally, we experimentally compare DYN against the DBToaster[1], a well-known state-of-the-art HIVM based dynamic query processing engine. Our experiments based on the industry-standard TPC-H[2] and TPC-DS[3] benchmarks show that DYN is up to one order of magnitude more efficient than DBToaster, both in terms of update time and memory consumption. We structure the rest of this chapter as follows: we first present preliminary and necessary notations and definitions in Section 3.1, then we present the DYN algorithm in Section 3.2 followed by its experimental evaluation in Section 3.3.

3.1 Preliminaries

We adopt the data model of Generalized Multiset Relations (GMRs for short) [KAK$^+$14, Koc10]. A GMR is a relation in which each tuple is associated to an integer in $\mathbb{Z}$. Figure 3.1 shows several examples. Note that in a GMR, in contrast to classical multisets, the multiplicity of a tuple can be negative. This allows to treat insertions and deletions symmetrically, as we will later see. To avoid ambiguity, we give a formal definition of GMRs that will be used throughout this thesis.

[1] https://dbtoaster.github.io/
[2] http://www.tpc.org/tpch/
[3] http://www.tpc.org/tpcds/

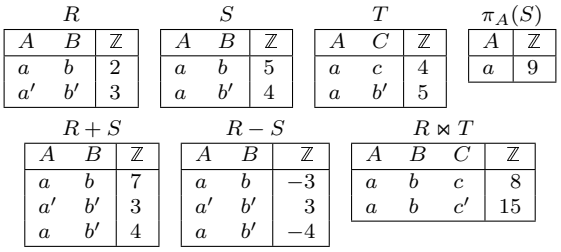


Figure 3.1: Operations on GMRs

Tuples. We first introduce some notation for tuples. Let $\overline{x}$ be a set of *variables* (also commonly known as *column names* or *attributes*). We write $\mathbb{T}[\overline{x}]$ for the universe of all possible tuples over $\overline{x}$. If $\vec{t} \in \mathbb{T}[\overline{x}]$ and y is a variable in $\overline{x}$ then we write $\vec{t}(y)$ for the value assigned to y by $\vec{t}$. If $\overline{y} \subseteq \overline{x}$ then we write $\vec{t}[\overline{y}]$ for the tuple over $\overline{y}$ obtained from $\vec{t}$ by removing all variables in $\overline{x} \setminus \overline{y}$. For example, if $\vec{t} = \langle A\colon 5, B\colon 4, C\colon 3\rangle$ then $\vec{t}(A) = 5$ and $\vec{t}[B, C] = \langle B\colon 4, C\colon 3\rangle$.

GMRs. A *generalized multiset relation* (GMR) over $\overline{x}$ is a function $R\colon \mathbb{T}[\overline{x}] \to \mathbb{Z}$ from relation tuples over $\overline{x}$ to integers. Every GMR R is a *total* function from the (possibly infinite) set $\mathbb{T}[\overline{x}]$ to $\mathbb{Z}$ and hence, conceptually, is an infinite object. However, every GMR is required to have finite *support* $\mathrm{supp}(R) := \{\vec{t} \in \mathbb{T}[\overline{x}] \mid R(\vec{t}) \neq 0\}$. Intuitively, $R(\vec{t}) = 0$ indicates that $\vec{t}$ is absent from R. The fact that R must have finite support indicates that R is a finite relation. To illustrate, in Figure 3.1, $R(\langle a, b\rangle) = 2$, hence present, while $R(\langle a, b'\rangle) = 0$, hence absent (and not shown). In what follows, we abuse notation and write $(\vec{t}, \mu) \in R$ to indicate that $\vec{t} \in \mathrm{supp}(R)$ and $R(\vec{t}) = \mu$; $\vec{t} \in R$ to indicate $\vec{t} \in \mathrm{supp}(R)$; and $|R|$ for $|\mathrm{supp}(R)|$. We say that R is empty if $\mathrm{supp}(R) = \emptyset$. The set of all GMRs over $\overline{x}$ is denoted by $\mathbb{GMR}[\overline{x}]$. A GMR is *positive* if $R(\vec{t}) > 0$ for all $\vec{t} \in \mathrm{supp}(R)$.

Operations on GMRs. Let R and S be GMRs over $\overline{x}$, T a GMR over $\overline{y}$, and $\overline{z} \subseteq \overline{x}$. The operations union ($R + S$), minus ($R - S$), join ($R \bowtie T$) and projection ($\pi_{\overline{z}} R$) over GMRs are defined as follows.

$$\begin{array}{rll} R + S & \in \mathbb{GMR}[\overline{x}] & : \vec{t} \mapsto R(\vec{t}) + S(\vec{t}) \\ R - S & \in \mathbb{GMR}[\overline{x}] & : \vec{t} \mapsto R(\vec{t}) - S(\vec{t}) \\ R \bowtie T & \in \mathbb{GMR}[\overline{x} \cup \overline{y}] & : \vec{t} \mapsto R(\vec{t}[\overline{x}]) \times S(\vec{t}[\overline{y}]) \\ \pi_{\overline{z}} R & \in \mathbb{GMR}[\overline{z}] & : \vec{t} \mapsto \sum_{\vec{s} \in \mathbb{T}[\overline{x}], \vec{s}[\overline{z}] = \vec{t}} R(\vec{s}) \end{array}$$

Figure 3.1 illustrates these operations. Note that GMRs there are positive, modeling standard multisets. Hence union, join, and projection correspond to the classical operations from relational algebra under multiset (i.e., bag) semantics. Minus is not relational difference, since it simply subtracts multiplicities (notice this could yield negative multiplicities).

Query Language. Conjunctive Queries (NCQs) are expressions of the form

$$Q = \pi_{\overline{y}}(r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n})).$$

Here, $r_1, \ldots, r_n$ are *relation symbols*; $\overline{x_1}, \ldots, \overline{x_n}$ are sets of variables, and $\overline{y} \subseteq \overline{x_1} \cup \ldots \cup \overline{x_n}$ is the set of *output variables*, also denoted by $out(Q)$. If $\overline{y} = \overline{x_1} \cup \ldots \cup \overline{x_n}$ then Q is a *join query* and simply denoted as $r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n})$. The pairs $r_i(\overline{x_i})$ are called *atomic queries* (or simply *atoms*).

A *database* over a set $\mathcal{A}$ of atoms is a function *db* that maps every atom $r(\overline{x}) \in A$ to a positive GMR $db_{r(\overline{x})}$ over $\overline{x}$. Given a database *db* over the atoms occurring in query Q, the evaluation of Q over *db*, denoted $Q(db)$, is the GMR over $\overline{y}$ constructed in the expected way: substitute each atom $r(\overline{x})$ in Q by $db_{r(\overline{x})}$, and subsequently apply the operations according to the structure of Q.

Discussion. For ease of notation in the rest of the chapter, we have not included relational selection $\sigma_\theta(r(\overline{x}))$ in queries. This is without loss of generality, as to dynamically process a Select-Project-Join query we can always filter out irrelevant tuples. For example, for $Q = \pi_{\overline{z}}(\sigma_{\theta_1}(r(\overline{x})) \bowtie \sigma_{\theta_2}(s(\overline{y})))$ we can consider new relation symbols r' and s' and dynamically process $Q' = \pi_{\overline{z}}(r'(\overline{x}) \bowtie s'(\overline{y}))$ instead. Then, whenever r and/or s are updated, it suffices to *discard* the tuples that do not satisfy the corresponding filter, and propagate the rest of the updates to relations r' and s' to update Q'.

Updates and deltas. An *update to a GMR* R is simply a GMR ΔR over the same variables as R. Applying update ΔR to R yields the GMR $R + \Delta R$. An *update to a database db* is a collection u of (not necessarily positive) GMRs, one GMR $u_{r(\overline{x})}$ for every atom $r(\overline{x})$ of *db*, such that $db_{r(\overline{x})} + u_{r(\overline{x})}$ is positive. We write $db + u$ for the database obtained by applying u to each atom of *db*, i.e., $(db + u)_{r(\overline{x})} = db_{r(\overline{x})} + u_{r(\overline{x})}$, for every atom $r(\overline{x})$ of *db*. For every query Q, every database *db* and every update u to *db*, we define the delta query $\Delta Q(db, u)$ of Q w.r.t. *db* and u by

$$\Delta Q(db, u) := Q(db + u) - Q(db).$$

As such, $\Delta Q(db, u)$ is the update that we need to apply to $Q(db)$ in order to obtain $Q(db + u)$.

3.1.1 Computational Model

We focus on dynamic query evaluation in main-memory and analyze performance under data complexity [Var82]. We assume a model of computation where tuple values and integers take $O(1)$ space and arithmetic operations on integers as well as memory lookups are $O(1)$ operations. We further assume that every GMR R can be represented by a data structure that allows (1) enumeration of R with constant delay (as defined in Section 3.2.1); (2) multiplicity lookups $R(\vec{t})$ in $O(1)$ time given $\vec{t}$; (3) single-tuple insertions and deletions in $O(1)$ time; while (4) having size that is proportional to the number of tuples in the support of R. We write $||R||$ for the size of GMR R over $\overline{x}$, i.e. $||R|| = |R| \times |\overline{x}|$. We further assume the existence of dynamic data structures that can be used to index GMRs on a subset of their variables. Concretely if R is a GMR over $\overline{x}$ and I is an index of R on $\overline{y} \subseteq \overline{x}$ then we assume that for every $\overline{y}$-tuple $\vec{s}$ we can retrieve in $O(1)$ time a pointer to the GMR $I(\vec{s}) \in \mathbb{GMR}[\overline{x}]$ consisting of all tuples that project to $\vec{s}$, as formally defined by

$$I(\vec{s}) \in \mathbb{GMR}[\overline{x}] \colon \vec{t} \mapsto \begin{cases} R(\vec{t}) & \text{if } \vec{t}[\overline{y}] = \vec{s} \\ 0 & \text{otherwise} \end{cases}$$

Moreover, we assume that single-tuple insertions and deletions to R can be reflected in the index in $O(1)$ time and that an index takes space linear in the support of R. Essentially, our assumptions amount to perfect hashing of linear size [Cor09]. Although this is not realistic for practical computers [Pap03], it is well known that complexity results for this model can be translated, through amortized analysis, to average complexity in real-life implementations [Cor09].

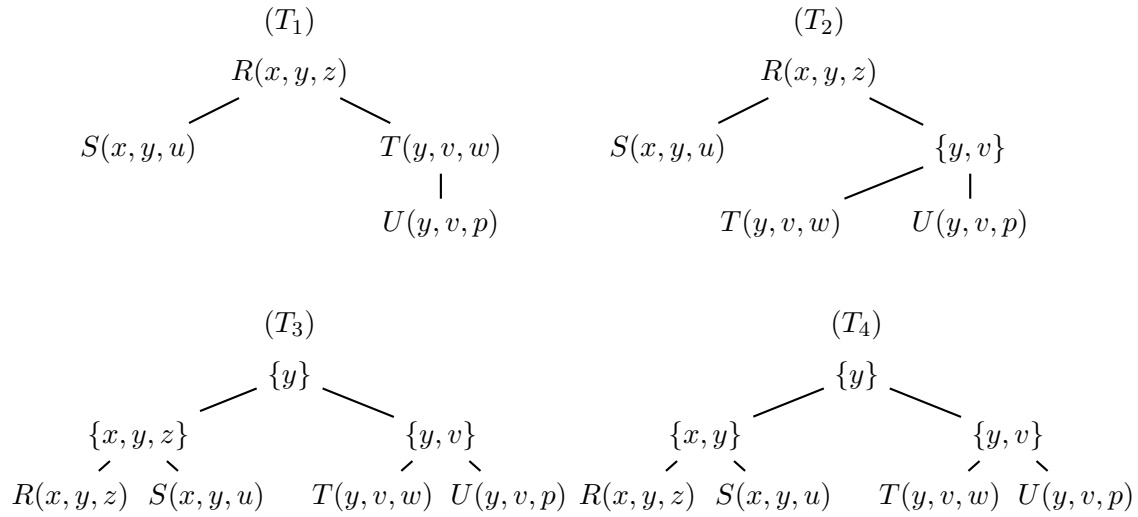


Figure 3.2: Width-one GHDs for $\{R(x,y,z), S(x,y,u), T(y,v,w), U(y,v,p)\}$. T_1 is a traditional join tree, T_3 and T_4 are generalized join trees. In addition, T_4 is simple.

3.1.2 Acyclicity

Throughout the thesis we focus on the class of *acyclic* queries. While there are many equivalent ways of defining acyclic queries [AHV95] we will use here a characterization of the acyclic queries in terms of those queries that have a Generalized Hypertree Decomposition (GHD for short) of width one [GLS03a]. Width-one GHDs generalize traditional join trees [AHV95] by also allowing partial hyperedges to occur as nodes in the tree. Intuitively, these partial hyperedges represent projections of single atoms. The importance of this feature will become clear at the end of Section 3.2, where we show the existence of acyclic queries for which traditional join trees (where only full hyperedges can occur) do not induce optimal complexity algorithms under the setting of dynamic query evaluation.

To simplify notation, we denote the set of all variables (resp. the set of all atoms) that occur in a mathematical object X (such as a query) by $var(X)$ (resp. $at(X)$). In particular, if X is itself a set of variables, then $var(X) = X$.

Definition 1 (Width-1 GHD)**.** Let $\mathcal{A}$ be a finite set of atoms. A *hyperedge in* $\mathcal{A}$ is a set $\overline{x}$ of variables such that $\overline{x} \subseteq var(\boldsymbol{a})$ for some atom $\boldsymbol{a} \in A$. We call $\overline{x}$ *full* in $\mathcal{A}$ if $\overline{x} = var(\boldsymbol{a})$ for some $\boldsymbol{a} \in A$, and partial otherwise. A *Generalized Hypertree Decomposition (GHD) of width one* for $\mathcal{A}$ is a directed tree $T = (V, E)$ such that:

- All nodes of T are either atoms or hyperedges in $\mathcal{A}$. Moreover, every atom in $\mathcal{A}$ occurs in T.
- Whenever the same variable x occurs in two nodes m and n of T, then x occurs in each node on the unique undirected path linking m and n.

If all nodes in T are atoms, then T is a *traditional join tree*.

To illustrate this, Figure 3.2 shows four width-one GHDs for the join of relations $R(x,y,z), S(x,y,u), T(y,v,w), U(y,v,p)$ where T_1 is traditional while the others are not.

Definition 2 (Acyclicity)**.** A NCQ Q is *acyclic* if there exists a width-one GHD T for *at*(Q), and is cyclic otherwise.

For example the width-one GHDs of Figure 3.2 show that $R(x, y, z) \bowtie S(x, y, u) \bowtie T(y, v, w) \bowtie U(y, v, p)$ is acyclic. In contrast, $R(x, y) \bowtie S(y, z) \bowtie T(x, z)$, the triangle query, is the prototypical cyclic join query.

For the rest of the thesis, it will be convenient to focus on width-one GHDs of a particular form. We call such restricted GHDs *generalized join trees*.

Definition 3 (Generalized Join Tree)**.** A *generalized join tree* for set of atoms $\mathcal{A}$ is a width-one GHD T for $\mathcal{A}$ in which all atoms occur as leafs. Moreover, every interior node n must have at least one child c such that *var*$(n) \subseteq$ *var*(c).

In Figure 3.2, trees T_3 and T_4 are generalized join trees; trees T_1 and T_2 are not. The following proposition shows that we may restrict our attention to generalized join trees without loss of generality.

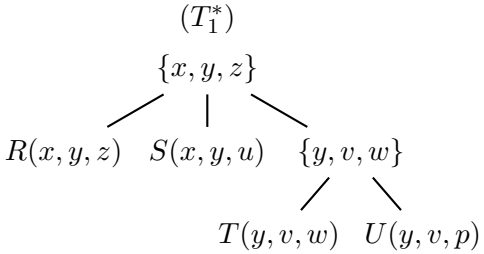
Readers familiar with GHDs of arbitrary width may observe that GHDs are normally defined as triples (T, χ, λ) with T a tree; χ a function that assigns a set of variables to each node, and λ a function that assigns a set of atoms to each node (see [GLS03a]). Since we focus on GHDs of width one, and hence do not need the full richness of GHDs, we omit χ and λ from our definition. These can be recovered by fixing $\chi\colon n \to$ *var*(n) and λ to be the function that maps atoms $\boldsymbol{a} \mapsto \{\boldsymbol{a}\}$ and hyperedges $\overline{x} \mapsto \{\boldsymbol{b}\}$ where $\boldsymbol{b}$ is some atom with $\overline{x} \subseteq$ *var*$(\boldsymbol{b})$. Note that, since under this definition $|\lambda(n)|= 1$ for all nodes, this indeed yields a GHD of width 1.

Proposition 1. *(Generalized Join Tree) If there exists a width-one GHD for the set of atoms $\mathcal{A}$, then there also exists a generalized join tree for $\mathcal{A}$. Consequently, a NCQ Q is acyclic iff at(Q) has a generalized join tree.*

Crux. A *traditional join tree* for $\mathcal{A}$ is a width-one GHD T for $\mathcal{A}$ in which all the nodes are atoms in $\mathcal{A}$ (no hyperedges allowed). It is well-known that a width-one GHD exists for $\mathcal{A}$ if, and only if, a traditional join tree T exists for $\mathcal{A}$ [GLS03a]. It hence suffices to show that every traditional join tree T for $\mathcal{A}$ can be transformed into a generalized join tree T' for $\mathcal{A}$. We do this by recursively applying the following transformation rule to nodes in T, starting at the root:

Let n be the current node being transformed. If hyperedge var(n) is not yet in T', then add var(n) to T' and if n is not the root in T then add an edge from var(n) to var(p) with p the parent of n in T. Finally (and even if var(n) were already in T'), add n (which is an atom) to T' and add an edge from n to var(n). Then, recursively apply this procedure to each child of n in T.

To illustrate, if we apply this procedure to the traditional join tree T_1 of Figure 3.2 then we obtain the GJT T_1^* generalized join tree.



It is a standard exercise to show that this transformation indeed always yields a generalized join tree. □

In what follows, we will refer to generalized join trees simply as join trees. When n is a node of a join tree T, c is a child of n, and $var(n) \subseteq var(c)$, we call c a *guard* of n. By definition, there is a guard for every hyperedge. We denote by $\mathrm{grd}(n)$ the set of guards of n, by $\mathrm{ch}(n)$ the set of children of n, and by $\mathrm{ng}(n)$ the set $\mathrm{ch}(n) \setminus \mathrm{grd}(n)$ of *non-guards* of n. Finally, we define $\mathrm{pvar}(c)$ to be the set of variables that c has in common with its parent ($\mathrm{pvar}(c) = \emptyset$ for the root). For example, in T_3 of Figure 3.2, $\mathrm{pvar}(S(x, y, u)) = \{x, y\}$.

3.2 Dynamic Yannakakis

In this section we develop DYN, a dynamic version of the Yannakakis algorithm. In Section 3.2.1, we introduce the notion of constant-delay enumeration. Then, in Section 3.2.2 we show that, for acyclic conjunctive queries, a representation satisfying properties P_1 and P_3 of the proposed solution in chapter 1 can be obtained by slightly modifying the Yannakakis algorithm. We introduce DYN in Section 3.2.3, and show in Sections 3.2.4-3.2.5 that the representation gives the properties $P_1 - P_4$ of DCLRs for NCQs that are free-connex acyclic. We show that this is optimal in two distinct ways in Section 3.2.6.

3.2.1 Constant delay enumeration

Definition 4. A data structure D supports enumeration of a set E if there exist a routine ENUM such that ENUM(D) outputs each element in E exactly once. Such enumeration occurs with delay d if the time to output the first tuple; the time between any two consecutive tuples; and the time between the last tuple and the end of the ENUM(D) routine, are all bounded by d. These times can neither depend on the size of D nor on the size of E. We say that D supports bounded-delay enumeration of a GMR R if D supports bounded-delay enumeration of the set $E_R = \{(\vec{t}, R(\vec{t})) \mid \vec{t} \in \mathrm{supp}(R)\}$. When evaluating a query Q, we will be interested in representing the possible outputs of Q by means of a family $\mathcal{D}$ of data structures, one data structure $D_{db} \in \mathcal{D}$ for each possible input database *db*. We say that *Q can be enumerated from $\mathcal{D}$ with delay f*, if for every input *db* we can enumerate $Q(db)$ from D_{db} with delay $O(f(D_{db}))$, where f assigns a natural number to each D_{db}. Intuitively f measures D_{db} in some way. In particular, if f is constant we say the results are generated from the data structure with *constant-delay enumeration* (CDE) as exemplified below.

As a trivial example of CDE of a GMR R, assume that the pairs $(\vec{t}, R(\vec{t}))$ of E_R are stored in an array A (without duplicates). Then A supports CDE of R: ENUM(A) simply iterates over each element in A, one by one, always outputting the current element. To see that this is correct, first observe that all pairs of E_R will be output exactly once. Moreover, the time required to output the first pair is the time required to fetch the first array element, hence constant. Similarly, the time required to produce each subsequent output tuple is the time required to fetch the next array element, again constant. Finally, checking whether we have reached the end of E_R amounts to checking whether we have reached the end of the array, again taking constant time.

This example shows that in order to do CDE of the result $Q(db)$ of a query Q on input database *db*, we can always (naively) materialize $Q(db)$ in an in-memory array A.

Unfortunately, A then requires memory proportional to $\|Q(db)\|$ which, depending on the query, can be of size polynomial in $\|db\|$. We hence search for other data structures that can represent $Q(db)$ using less space, while still allowing enumeration with the same (worst-case) complexity as enumeration from a materialized array A: namely, with constant delay. The key idea to obtain this is *delayed evaluation*. To illustrate this, consider that we are asked to compute the Cartesian product of R and S. Then it suffices to simply store R and S, requiring $O(\|R\| + \|S\|) = O(\|db\|)$ memory. To enumerate $R \times S$, ENUM simply executes a nested-loop based Cartesian product over R and S. This satisfies the properties of CDE. Indeed, every element of $R \times S$ will be output exactly once. Moreover, the time required to output the first element of $R \times S$ is the time required to initialize a pointer to the first elements of R and S (hence constant). The time required to produce each subsequent element is bounded by the time required to either advance the pointer in S, or advance the pointer in R and reset the pointer in S to the beginning. In both cases, this is constant. Finally, checking whether we have reached the end of $R \times S$ again takes constant time.

The situation becomes more complex for queries that involve joins instead of Cartesian products. Consider for example the query $Q = R(A, B) \bowtie S(B, C)$. Simply delaying evaluation does not yield constant-delay enumeration. Indeed, suppose that we evaluate Q using a simple in-memory hash join with R as build relation and S as probe relation. Assume that the corresponding index of R on B (i.e. the hash table) has already been computed. When iterating over S to probe the hash table, we may have to visit an unbounded number of S-tuples that do not join with any of the R-tuples. Consequently, there is no constant that bounds the delay between consecutive outputs. A similar analysis shows that other join algorithms, such as the sort-merge join, do not yield enumeration with constant delay.

In essence, therefore, a data structure that allows CDE of $Q(db)$ must be able to produce all output tuples and their multiplicities without spending any extra time in building auxiliary data structures to help in enumeration (such as hash tables or sorted versions of the input relations), nor can it afford to waste time in processing input tuples that in the end do not appear in $Q(db)$.

How do we obtain CDE for $R(A, B) \bowtie S(B, C)$? Intuitively speaking, if in our hash join algorithm we can ensure to only iterate over those S-tuples that have matching R-records, we trivially obtain a CDE algorithm. In a broader sense, we need to maintain under updates, for the relations that are used as *probe* relations, the set of tuples that will match the corresponding build relation(s). We call these tuples the *live tuples*. In the following sections we gradually devise a more general algorithm that follows this idea. Intuitively, this algorithm dynamically maintains the hash tables and the live values for a query in a DCLR.

3.2.2 Constant Delay Yannakakis

Acyclic full join queries are evaluated in $O(\|db\| + \|Q(db)\|)$ time by the well-known Yannakakis algorithm. For future reference, we recall the operation of the Yannakakis algorithm [Y81], formulated in our setting. We first need to introduce the semi-join operation for GMRs.

Definition 5. The semijoin $R \ltimes S$ of a GMR $R[\overline{x}]$ by a GMR S is the GMR over $\overline{x}$ defined by

$$R \ltimes S \in \mathbb{GMR}[\overline{x}] \colon \vec{s} \mapsto \begin{cases} R(\vec{s}) & \text{if } \vec{s} \in \pi_{\overline{x}}(R \bowtie S) \\ 0 & \text{otherwise.} \end{cases}$$

Classical Yannakakis. In its standard formulation, Yannakakis takes as input a *traditional* join tree T for a join query Q and a database db on Q. The algorithm starts by assigning a GMR R_n over $var(n)$ to each node n in T. Initially, $R_n := db_n$. The algorithm then works in three stages.

(1) The nodes of T are visited in some bottom-up traversal order of T. When node n is visited in this order, its parent p is considered and R_p is updated to $R_p := R_p \ltimes R_n$.
(2) The nodes of T are visited in a top-down traversal order. When node n is visited in this order, each child c of n is considered, and R_c is updated to $R_c := R_c \ltimes R_n$.
(3) The interior nodes of T are again visited in a bottom-up order. In this stage, however, the actual join results are computed: when node n with children $c_1, \ldots, c_k$ is visited, its GMR is updated to $R_n := R_{c_1} \bowtie \cdots \bowtie R_{c_k}$.

After the final stage, the GMR materialized at the root is precisely $Q(db)$. The initialization together with stages (1) and (2) run in time $O(\|db\|)$ while stage 3 can be shown to run in time $O(\|Q(db)\|)$. It is worth noting that the Yannakakis algorithm fully materializes the query result $Q(db)$ at the root, requiring $O(\|Q(db)\|)$ space. Notice also that this algorithm works over the static setting, and does not consider updates.

To extend Yannakakis to work on generalized join trees in addition to traditional join trees, one only needs to modify the initialization step as follows. If n is a hyperedge, simply set $R_n := \pi_{var(n)} R_c$ for some arbitrary but fixed $c \in \mathrm{grd}(n)$ (which we may assume to have been initialized before if we initialize in a bottom-up fashion). It is not difficult to see that, with this initalization, every hyperedge n has $R_n = \pi_{var(n)}\, db_{\boldsymbol{a}}$ for some descendant atom $\boldsymbol{a}$ of n. In other words, R_n is the projection of some input atom. This ensures that, even on generalized join trees, Yannakakis exhibits the same complexity guarantees.

Yannakakis with constant delay enumeration (CDY). Our dynamic query processing algorithm is based on the simple observation that, after the first bottom-up traversal stage, the join query result $Q(db)$ can be enumerated with constant delay. As such, there is no need to materialize the query result in stage 3. To illustrate this claim, consider the following variant of the Classical Yannakakis algorithm, called CDY for Constant Delay Yannakakis.

(1) Do the first stage of Classical Yannakakis.
(2) For each node n construct an index L_n of R_n on $\mathrm{pvar}(n)$.

Given this pre-processing, the constant-delay enumeration method ENUM is essentially a multi-way hash join, where the GMR materialized at the root is used as probe relation, and the other R_n as build relations, with the hash tables given by L_n. Because of the way in which R_n is computed, we are ensured that for every probe we will have matching join tuples, ensuring constant-delay enumeration. Note, moreover, that the GMRs materialized after the first step of the Yannakakis algorithm, as well as the constructed indexes, require $O(\|db\|)$ space. We delay a formal definition of the enumeration algorithm until Section 3.2.5, but illustrate its working by means of the following example.

Example 1. *Consider generalized join tree T_3 of Figure 3.2.* ENUM *works as follows. Let $\vec{s}$ be the empty tuple. Then* ENUM *is defined by:*

for each $\vec{t}_{\{y\}} \in L_{\{y\}}(\vec{s})$ ***do***
  for each $\vec{t}_{\{x,y,z\}} \in L_{\{x,y,z\}}(\vec{t}_{\{y\}})$ ***do***
    for each $(\vec{t}_R, \mu_R) \in L_{R(x,y,z)}(\vec{t}_{\{x,y,z\}})$ ***do***
      for each $(\vec{t}_S, \mu_S) \in L_{S(y,v,w)}(\vec{t}_{\{y,v,w\}}[y,v])$ ***do***
        for each $\vec{t}_{\{y,v\}} \in L_{\{y,v\}}(\vec{t}_{\{y\}})$ ***do***

for each $(\vec{t_T}, \mu_T) \in L_{T\{y,v,w\}}(\vec{t}_{\{y,v\}})$ ***do***
  for each $(\vec{t_U}, \mu_T) \in L_{U\{y,v,p\}}(\vec{t}_{\{y,v\}})$ ***do***
    output $(\vec{t_R}\vec{t_S}\vec{t_T}\vec{t_U}, \mu_R * \mu_S * \mu_T * \mu_U)$ □

From our discussion so far, we obain:

Proposition 2. *Given an acyclic full join query Q, a join tree T of Q and a database db, $CDY(T, db)$ runs in time $O(\|db\|)$ using space $O(\|db\|)$. Once CDY has completed,* ENUM *effectively enumerates $Q(db)$ with constant delay.*

We delay the proof of this proposition and present it together with the proposition 10 in Section 3.2.5.

3.2.3 Dynamic Yannakakis

Definition 6. Let T be a join tree and db a database. Let R_n, for $n \in T$, be the GMR associated to n after executing the first stage of the Yannakakis algorithm. A tuple $\vec{t}$ is called *live* in (db, n) w.r.t. T if $\vec{t} \in R_n$.[4]

CDY shows that we can suitably index the live tuples to enumerate $Q(db)$ with constant delay. To turn CDY into a dynamic algorithm, it hence suffices to maintain the live tuples and indices under updates. A naive approach for doing this would be to re-run CDY from scratch whenever the database is updated. This would spend time linear in the size of the updated database. Of course, this naive approach introduces unnecessary overhead. Indeed, consider an update that inserts a single tuple to an atom $\boldsymbol{a}$. In that case, only the set of live tuples associated to $\boldsymbol{a}$ and its ancestors in join tree T can change, while the rest of the nodes would remain unchanged. Moreover, the new set of live tuples of $\boldsymbol{a}$ and its ancestors can be computed incrementally. At the end of Section 3.3, we will see in particular that avoiding naive recomputation is highly effective in practice.

In order to be able to explain how we maintain the live tuples incrementally, we require the following definitions.

Definition 7. Let T be a join tree. To every node n of T we associate two queries, Λ_n^T and Ψ_n^T, over $var(n)$ and $\mathrm{pvar}(n)$, respectively. To every hyperedge n of T we also associate an additional query Γ_n^T over $var(n)$. The definition of these queries is recursive: for each atom $\boldsymbol{a}$ we define $\Lambda_{\boldsymbol{a}}^T$ simply as $\boldsymbol{a}$, and $\Psi_{\boldsymbol{a}}^T := \pi_{\mathrm{pvar}(\boldsymbol{a})}\boldsymbol{a}$. Then, in a bottom-up traversal order, for every hyperedge n we define

$$\Lambda_n^T := \Gamma_n^T \ltimes \bowtie_{c \in \mathrm{ng}(n)} \Psi_c^T$$
$$\Psi_n^T := \pi_{\mathrm{pvar}(n)} \Lambda_n^T$$
$$\Gamma_n^T := \bowtie_{c \in \mathrm{grd}(n)} \Psi_c^T$$

We often omit the superscript if it is clear from the context. Intuitively, Λ_n contains the set of live tuples of n, while Ψ_n and Γ_n are auxiliary queries that will help maintain Λ_n under updates. The following proposition shows that, indeed, the queries Λ_n characterize the live tuples. The proof is by induction on the height of node n in T.

[4] Recall that $\vec{t} \in \mathrm{supp}(R_n)$ indicates that $R_n(\vec{t}) \neq 0$.

Proposition 3. *A tuple $\vec{t}$ is live in (db, n) w.r.t. join tree T if, and only if, $\vec{t} \in \Lambda_n^T(db)$.*

Proof. Two queries Q and Q' are said to be *support-equivalent*, denoted by $Q \simeq Q'$, if $\mathrm{supp}(Q(db)) = \mathrm{supp}(Q'(db))$, for every database db. Clearly, $\simeq$ is an equivalence relation.

Next, define, for every node n in T, the query Q_n inductively as follows. If $n = \boldsymbol{a}$ for some atom $\boldsymbol{a}$, then $Q_n := \boldsymbol{a}$. Otherwise, n is an internal node. Let $c \in \mathrm{grd}(n)$ be the child of n that is arbitrarily chosen during the initalization phase of Yannakakis to initialize $R_n := \pi_{var(n)} R_c$. Then we define

$$Q_n := (\pi_{var(n)} Q_c) \ltimes_{d \in \mathrm{ch}(n)} Q_d.$$

With this definition, it is readily verified that after the bottom-up phase of Yannakakis, $R_n = Q_n(db)$, for every node n in T. We show the proposition by proving $Q_n \simeq \Lambda_n$. We do so by induction on the height h of node n.[5]

(Base case.) If this height is 0 (i.e., n is a leaf), then n is an atom $\boldsymbol{a}$. As such $Q_n = \boldsymbol{a} = \Lambda_n$ and therefore $Q_n \simeq \Lambda_n$.

(Induction step.) Otherwise, let $h > 0$ be the height of n and assume that the induction hypothesis holds for all nodes with height $< h$. First verify the following claim.

Claim. Let R and S be positive GMRs over $\overline{x}$ and $\overline{y}$, respectively. Then $R \ltimes S \equiv R \ltimes \pi_{\overline{x}} S$.

Since databases contain only positive GMRs, this result is easily extended to queries: For all databases db and all queries P, Q with $out(Q) = \overline{x}$ and $out(P) = \overline{y}$ it holds that $(Q \ltimes P) \simeq (Q \ltimes \pi_{\overline{x}} P)$. Indeed, it suffices to observe that the multiplicities computed by a query are the result of sums and multiplications of multiplicities in base relations. Since databases contain only positive multiplicities, the multiplicities output by queries must also be positive, and hence, the above claim can be used to show the result for queries.

Now, we reason as follows.

$$\begin{aligned}
Q_n &= (\pi_{var(n)} Q_c) \ltimes_{d \in \mathrm{ch}(n)} Q_d \\
&= ((\pi_{var(n)} Q_c) \ltimes_{g \in \mathrm{grd}(n)} Q_g) \ltimes_{d \in \mathrm{ng}(n)} Q_d \\
&\simeq ((\pi_{var(n)} Q_c) \ltimes_{g \in \mathrm{grd}(n)} \pi_{var(n)} Q_g) \ltimes_{d \in \mathrm{ng}(n)} \pi_{var(n)} Q_d \\
&\simeq ((\pi_{var(n)} \Lambda_c) \ltimes_{g \in \mathrm{grd}(n)} \pi_{var(n)} \Lambda_g) \ltimes_{d \in \mathrm{ng}(n)} \pi_{var(n)} \Lambda_d \\
&= (\Psi_c \ltimes_{g \in \mathrm{grd}(n)} \Psi_g) \ltimes_{d \in \mathrm{ng}(n)} \Psi_d \\
&\simeq (\bowtie_{g \in \mathrm{grd}(n)} \Psi_g) \ltimes_{d \in \mathrm{ng}(n)} \Psi_d \\
&= \Gamma_n \ltimes_{d \in \mathrm{ng}(n)} \Psi_d \\
&\simeq \Gamma_n \bowtie \bowtie_{d \in \mathrm{ng}(n)} \Psi_d \\
&= \Lambda_n
\end{aligned}$$

The first $\simeq$-equivalence follows from our observation; the second $\simeq$-equivalence from the induction hypothesis; and the third and fourth $\simeq$-equivalences follow from the fact that if R and S are GMRs over $\overline{x}$ and $\overline{y}$ respectively, and $\overline{y} \subseteq \overline{x}$, then $R \ltimes S \simeq R \bowtie S$. □

[5] Recall that the height of a node in a tree is the number of edges on the longest downward path starting at that node.

Next, we show that the sizes of Λ_n, Ψ_n and Γ_n for a node n in a GJT T are bounded by the size of a descendant atom $\boldsymbol{a}$ of n by the following lemma.

Lemma 1. *Let T be a join tree and let db be a database.*

1. *For every atom $\boldsymbol{a} \in T$ we have $|\Psi^T_{\boldsymbol{a}}(db)| \leq |\Lambda^T_{\boldsymbol{a}}(db)| = |db_{\boldsymbol{a}}|$.*
2. *For every hyperedge $n \in T$ we have*

$$|\Psi^T_n(db)| \leq |\Lambda^T_n(db)| \leq |\Gamma^T_n(db)| \leq \min_{\substack{\boldsymbol{a} \in T|_n \\ var(n) \subseteq var(\boldsymbol{a})}} |db_{\boldsymbol{a}}|.$$

Here, $T|_n$ denotes the subtree of T rooted at n.

Proof. Claim (1) follows directly from the observation that tuples in $\Psi^T_{\boldsymbol{a}}(db)$ are projections of tuples in $\Lambda^T_{\boldsymbol{a}}(db)$; therefore $\Psi^T_{\boldsymbol{a}}(db)$ can have at most as many tuples as $\Lambda^T_{\boldsymbol{a}}(db)$.

To prove Claim (2), observe that for every interior node n we have

$$|\Psi^T_n(db)| \leq |\Lambda^T_n(db)| \leq |\Gamma^T_n(db)| \leq \min_{c \in \mathrm{grd}(n)} |\Psi^T_c(db)|.$$

The first inequality follows as before; the second from the fact that Γ^T_n has the same schema as Λ^T_n and $\Lambda^T_n = \Gamma^T_n \ltimes \ltimes_{c \in \mathrm{ng}(n)} \Psi^T_c$; the last inequality follows from the observation that $\Gamma^T_n = \ltimes_{c \in \mathrm{grd}(n)} \Psi^T_c$ where the Ψ^T_c all have the same schema, for every $c \in \mathrm{grd}\, n$—therefore the join of $\Gamma_n(db)$ is actually an intersection.

By iterating this equality again on c, recursively until the leaves, we obtain that the number of tuples output by Ψ^T_n, Λ^T_n and Γ^T_n is bounded by $|db_{\boldsymbol{a}}|$ for each descendant atom $\boldsymbol{a}$ of n with $var(n) \subseteq var(\boldsymbol{a})$. In particular, it is bounded by the minimum such cardinality. □

We can now define the data structure maintained by DYN.

Definition 8 (T-representation)**.** Let T be a join tree and let *db* be a database. A *T-representation* (T-rep for short) of *db* is a data structure $\mathcal{D}$ that for each node n of T contains:

- an index L_n of $\Lambda_n(db)$ on $\mathrm{pvar}(n)$;
- a GMR P_n that materializes $\Psi_n(db)$, i.e., $P_n = \Psi_n(db)$;
- a GMR G_n that materializes $\Gamma_n(db)$, i.e., $G_n = \Gamma_n(db)$;
- for every non-guard child $c \in \mathrm{ng}(n)$, an index $G_{n,c}$ of G_n on $\mathrm{pvar}(c)$.

Example 2. *Consider join tree T_3 from Figure 3.2. In Figure 3.3, we show the T_3-rep $\mathcal{D}$ for the database db consisting of the GMRs $R(x, y, z)$, $S(x, y, u)$, $T(y, v, w)$, $U(y, v, p)$ presented at the leaves of Figure 3.3. For each node n, the live tuples $L_n = \Lambda_n(db)$ are given by the white-colored tables (shown below n) while $P_n = \Psi_n(db)$ is given by the gray-colored tables (shown above n on the edge from n to its parent). For reasons of parsimony, we do not show the G_n: for $\{y\}$ and $\{y, v\}$ this equals L_n; for $\{x, y, z\}$ this equals $L_{R(x,y,z)}$. The indexes are likewise not shown.* □

A first important feature of T-representations is that they use only linear space.

Proposition 4. *Let T be a join tree and $\mathcal{D}$ a T-rep of db. Then $\|\mathcal{D}\| = O(\|db\|)$.*

Proof. The crux to prove this proposition lies in observing that, as illustrated in Figure 3.3, for all nodes n, if $\vec{t}$ is in $\Lambda_n(db)$, $\Psi_n(db)$, or $\Gamma_n(db)$, then there is some descendant atom $\boldsymbol{a} \in T$ such that $\vec{t} \in \pi_{\overline{x}}(db_{\boldsymbol{a}})$ with $\overline{x} = var(n)$ or $\overline{x} = \mathrm{pvar}(n)$. Therefore, $\Lambda_n(db)$, $\Psi_n(db)$, and $\Gamma_n(db)$ as well as indexes thereon, all take space $O(\|db\|)$. □

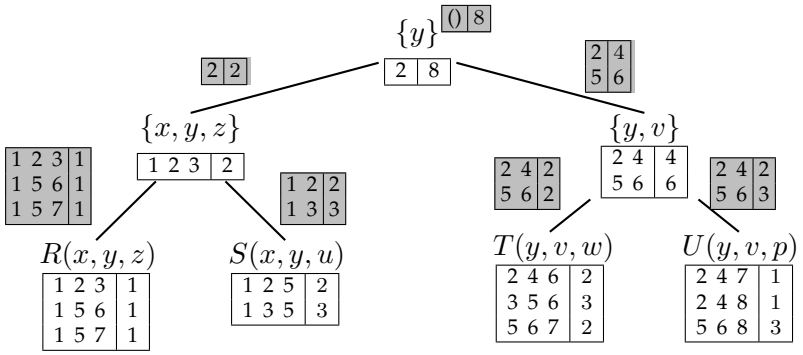


Figure 3.3: Illustration of T-representations (Example 2).

Algorithm 1 DYN$_T$: Update trigger maintaining T-rep $\mathcal{D}$ under update u

1: **Assume:** T is a join tree
2: **Input:** T-rep $\mathcal{D}$ for (db); and update u
3: **Output:** T-rep for $db + u$.
4: **for each** node $n \in T$, visited in bottom-up order **do**
5: compute ΔL_n, ΔP_n, and ΔG_n (if applicable)
6: **for each** node $n \in T$ **do**
7: L_n+=ΔL_n; P_n+=ΔP_n
8: **if** n is a hyperedge **then**
9: G_n+=ΔG_n
10: **for** each $c \in \mathrm{ng}(n)$ **do** $G_{n,c}$+=ΔG_n

Dynamic Yannakakis. We now describe the Dynamic Yannakakis algorithm (DYN) presented in Algorithm 1. DYN maintains T-representations under updates. To explicitly assert the join-tree over which DYN operates we write DYN$_T$.

Like classical Yannakakis, DYN$_T$ traverses the nodes of T in a bottom-up fashion upon update u. During this traversal, the goal is to materialize, for each node n, the deltas $\Delta\Lambda_n(db, u)$, $\Delta\Psi_n(db, u)$, and $\Delta\Gamma_n(db, u)$ into GMRs ΔL_n, ΔP_n, and ΔG_n, respectively. These represent the updates that we need to apply to $\mathcal{D}$'s components in order to obtain a T-rep for $db + u$. This application happens in lines 6–10.

The delta GMRs are computed as follows. When n is an atom $\boldsymbol{a}$, DYN$_T$ uses the update u to compute $\Delta L_{\boldsymbol{a}} = u_{\boldsymbol{a}}$ and $\Delta P_{\boldsymbol{a}} = \pi_{\mathrm{pvar}(\boldsymbol{a})}\, u_{\boldsymbol{a}}$. The latter projection can be done using a simple hash-based aggregation algorithm. When n is a hyperedge, DYN$_T$ uses Algorithm 2 to compute ΔL_n, ΔP_n and ΔG_n. This algorithm uses the materialized index of $\Lambda_n(db)$ on $\mathrm{pvar}(n)$, and the materialized GMRs $P_n = \Psi_n(db)$ and $G_n = \Gamma_n(db)$, which are already available in $\mathcal{D}$. In addition, it uses the delta GMRs ΔP_c for each child c of n, which was previously computed when visiting c. In order to compute ΔL_n, ΔP_n, and ΔG_n efficiently, we use the following insight.

Lemma 2. *If $\vec{t} \in \Delta\Gamma_n(db, u)$ then $\vec{t} \in \Delta\Psi_c(db, u)$ for some guard $c \in \mathrm{grd}(n)$. Moreover, if $\vec{t} \in \Delta\Lambda_n(db, u)$ then either (1) $\vec{t} \in \Delta\Psi_c(db, u)$ for some guard $c \in \mathrm{grd}(n)$ or (2) $\vec{t} \in (\Gamma_n(db) \ltimes \Delta\Psi_c(db, u))$ for some child $c \in \mathrm{ng}(n)$.*

Proof. We only show the reasoning when $\vec{t} \in \Delta\Lambda_n(db, u)$. The reasoning when $\vec{t} \in \Delta\Gamma_n(db, u)$ is similar.

Algorithm 2 Delta computation for hyperedge n

1: Initialize $\Delta L_n, \Delta P_n$ and ΔG_n to the empty GMRs
2: Initialize $U := \bigcup_{c \in \mathrm{grd}(n)} \mathrm{supp}(\Delta P_c)$
3: **for each** $c \in \mathrm{ng}(n)$ **and each** $\vec{t_c} \in \Delta P_c$ **do**
4: $\quad U := U \cup \mathrm{supp}(G_{n,c}[\vec{t_c}])$
5: **for each** $\vec{t} \in U$ **do**
6: $\quad \Delta G_n[\vec{t}] := \prod_{c \in \mathrm{grd}(n)} (P_c + \Delta P_c)[\vec{t}] - G_n[\vec{t}]$
7: $\quad \Delta L_n[\vec{t}] := \prod_{c \in \mathrm{ch}(n)} (P_c + \Delta P_c)[\vec{t}[\mathrm{pvar}(c)]] - L_n[\vec{t}]$
8: $\quad \Delta P_n[\vec{t}[\mathrm{pvar}(n)]]$ += $\Delta L_n[\vec{t}]$

Suppose that $\vec{t} \in \Delta\Lambda_n(db, u)$. By definition, $\Lambda_n := \Gamma_n \bowtie \bowtie_{c \in \mathrm{ng}(n)} \Psi_c$. By definition of join tree, $\mathrm{grd}(n)$ is non-empty. If $\mathrm{ng}(n)$ is empty, then in particular, $\vec{t} \in \Delta\Lambda_n(db, u) = \Delta\Gamma_n(db, u)$. In that case, by the first part of the lemma, we hence obtain $\vec{t} \in \Delta\Psi_c(db, u)$ for some $c \in \mathrm{grd}(n)$. It remains to confirm the result when $\mathrm{ng}(n)$ is non-empty. Hereto, first observe that taking deltas distributes over joins as follows.

$$\begin{aligned}\Delta(r(\overline{x}) \bowtie s(\overline{y}))(db, u) = {} & \Delta r(\overline{x})(db, u) \bowtie s(\overline{y})(db) \\ & + (\Delta r(\overline{x})(db, u) \bowtie \Delta s(\overline{y})(db, u) \\ & + (r(\overline{x})(db, u) \bowtie \Delta s(\overline{y})(db, u)\end{aligned}$$

By application of this equality to $\Delta\Lambda_n(db, u)$, we obtain that there are three cases possible.

- Case $\vec{t} \in \Delta\Gamma_n(db, u) \bowtie \left(\bowtie_{c \in \mathrm{ng}(n)} \Psi_c\right)(db)$. Then, $\vec{t} \in \Delta\Gamma_n(db, u)$ since Γ_n has the same schema as Λ_n. By the first part of the lemma, we hence obtain $\vec{t} \in \Delta\Psi_c(db, u)$ for some $c \in \mathrm{grd}(n)$.
- The case $\vec{t} \in \Delta\Gamma_n(db, u) \bowtie \Delta\left(\bowtie_{c \in \mathrm{ng}(n)} \Psi_c\right)(db, u)$ is similar.
- Case $\vec{t} \in \Gamma_n(db) \bowtie \Delta\left(\bowtie_{c \in \mathrm{ng}(n)} \Psi_c\right)(db, u)$. Then in particular, $\vec{t} \in \Gamma_n(db)$. Moreover, $\vec{t}[\bigcup_{c \in \mathrm{ng}(n)} \mathrm{pvar}(c)]$ is in $\Delta\left(\bowtie_{c \in \mathrm{ng}(n)} \Psi_c\right)(db, u)$. Then by application of the above distribution of delta's over joins on expression $\Delta\left(\bowtie_{c \in \mathrm{ng}(n)} \Psi_c\right)(db, u)$ we obtain that there is at least one $c \in \mathrm{ng}(n)$ such that $\vec{t}[\mathrm{pvar}(c)] \Delta\Psi_c(db, u)$. Therefore, $\vec{t} \in \mathrm{supp}(\Gamma_n(db) \ltimes \Delta\Psi_c(db, u))$, as desired. □

Algorithm 2 uses Lemma 2 to compute a bound on $\mathrm{supp}\,(\Delta\Lambda_n(db, u))$. In particular, in lines 2–4 it computes

$$U = \bigcup_{c \in \mathrm{grd}(n)} \mathrm{supp}(\Delta P_c) \cup \bigcup_{c \in \mathrm{ng}(n)} \mathrm{supp}(G_n \ltimes \Delta P_c).$$

As such, U contains all tuples that can appear in $\Delta\Gamma_n(db, u)$ or $\Delta\Lambda_n(db, u)$. Lines 5–8 compute $\Delta G_n = \Delta\Gamma_n(db, u)$, $\Delta L_n = \Delta\Lambda_n(db, u)$ and $\Delta P_n = \Delta\Psi_n(db, u)$ by iterating over the tuples in U and using the fact that $\Psi_c(db + u)(\vec{s}) = P_c(\vec{s}) + \Delta P_c(\vec{s})$, for every tuple $\vec{s}$. From Lemma 2 and our explanation so far, we hence obtain:

Theorem 1. *Let $\mathcal{D}$ be a T-rep for db and let u be an update to db. $\mathrm{DYN}_T(\mathcal{D}, u)$ produces a T-rep for $db + u$.*

Proof. Correctness follows from lemma 2 and the explanation above. □

3.2.4 Complexity of Dynamic Yannakakis

Next, we study the efficiency with which DYN_T maintains T-reps under updates. Towards this end, we first illustrate DYN_T's operation by example.

Example 3. *Consider join tree T_3 from Figure 3.2 and the T_3-rep $\mathcal{D}$ shown in Figure 3.3 that we discussed in Example 2. Consider the update $\{(1,5,9) \mapsto 2\}$ on S (i.e., tuple $(1,5,9)$ is inserted into $S(x,y,u)$ with multiplicity 2). When DYN_{T_3} executes, it will compute empty delta GMRs for all nodes, except for $S(x,y,u)$ and its ancestors. In particular, when Algorithm 2 is run on hyperedge $\{x,y,z\}$, it operates as follows. First observe that $\{x,y,z\}$ has only one guard child, namely $R(x,y,z)$. Hence $G_{\{x,y,z\}} = P_{R(x,y,z)} = L_{R(x,y,z)} = R$. Since R is not updated, U is initialized to $\emptyset$ in line 2. Then, in lines 3 and 4, Algorithm 2 uses the index $G_{\{x,y,z\},S[x,y,u]}$ of $G_{\{x,y,z\}}$ on $\{x,y\}$ to directly retrieve all tuples in $R(x,y,z)$ that satisfy $x=1$ and $y=5$. This is the main purpose of the indexes $G_{n,c}$: we do not have to iterate over the entire GMR G_n nor check for equalities. During the rest of its computation, Algorithm 2 then calculates $\Delta G_{\{x,y,z\}} = \emptyset$, $\Delta L_{\{x,y,z\}} = \{(1,5,6) \mapsto 2, (1,5,7) \mapsto 2\}$, and $\Delta P_{\{x,y,z\}} = \{(5) \mapsto 4\}$. When processing $\{y\}$, we have to* propagate *$\Delta P_{\{x,y,z\}}$ from $\{x,y,z\}$ to $\{y\}$. This is done by initializing $U = \{(5)\}$ in line 2 of Algorithm 2, after which $\Delta G_{\{y\}} = \Delta L_{\{y\}} = \{(5) \mapsto 24\}$ and $\Delta P_{\{y\}} = \{() \mapsto 24\}$.* □

It is important to observe that in this example the single-tuple update $\{(1,5,9) \mapsto 2\}$ on S triggers *multiple* tuples to become live in $\{x,y,z\}$. This occurs simply because the variable z of $\{x,y,z\}$ is not in $S(x,y,u)$; therefore a single-tuple update to S can match many tuples in $G_{\{x,y,z\}}$ with different z values. In fact, in the worst case, it may cause as many tuples to become live in $\{x,y,z\}$ as there are tuples in $G_{\{x,y,z\}} = R$. In contrast, single-tuple updates into $R(x,y,z), T(y,v,w)$, or $U(y,v,p)$, can cause at most 1 tuple to become live in any of their parents. This is because R's (resp. T's and U's) parent contains only variables that are also mentioned in R (resp. T, resp. U). Likewise, updates to $\{x,y,z\}$ (resp. $\{y,v\}$) that we need to propagate to $\{y\}$ can only cause as many $\{y\}$ tuples to become live as have become live in $\{x,y,z\}$ (resp. $\{y,v\}$).

So, the fact that a node contains all variables mentioned in its parent makes it efficient to propagate updates from that node to its parent. Trees for which all nodes (except for the root) contain all variables mentioned in their parent are called simple trees.

Definition 9 (Simplicity)**.** A width-one GHD T is *simple* if every child node in T is a guard of its parent. A query Q is simple if it has a simple join tree.

For example, T_4 of Figure 3.2 is simple, but T_3 is not since $S(x,y,u)$ is a child but not a guard of $\{x,y,z\}$.

Because in simple trees the number of tuples that can propagate from a child update to its parent is bounded by the size of the child update, we obtain that, for a simple tree T, DYN_T maintains a T-rep under update u in time linear in u, independent of the databse *db*.

Theorem 2. *$\text{DYN}_T(\mathcal{D}, u)$ produces a T-rep for $db + u$ in time $O(\|u\|)$ for every database db and every update u if, and only if, T is simple.*

To prove this theorem, we first require the following definitions and propositions.

Definition 10. For a join tree T and node $n \in T$, we write $T|_n$ to denote the subtree of T rooted at n.

Proposition 5. *Let n be a hyperedge in join tree T and let $\mathcal{D}$ be a T-rep for db. Let u be an update. Assume that, for each child c of n, we have already computed $\Delta L_c = \Delta\Lambda_c(db, u)$, $\Delta P_c = \Delta\Psi_c(db, u)$ and $\Delta G_c = \Delta\Gamma_c(db, u)$. In that case, Algorithm 2 correctly computes $\Delta L_n = \Delta\Lambda_n(db, u)$, $\Delta P_n = \Delta\Psi_n(db, u)$ and $\Delta G_n = \Delta\Gamma_n(db, u)$ and runs in time*

$$\Theta\left(\sum_{c\in\mathrm{ch}(n)}|\Delta P_c|+\sum_{c\in\mathrm{ng}(n)}|G_n\ltimes\Delta P_c|\right).$$

Proof. Correctness follows from the explanation in the paragraph following Lemma 2

To see that Algorithm 2 runs in time

$$\Omega\left(\sum_{c\in\mathrm{ch}(n)}|\Delta P_c|+\sum_{c\in\mathrm{ng}(n)}|G_n\ltimes\Delta P_c|\right).$$

we consider the running time of lines 2–4. In particular, in line 2, algorithm 2 computes $\bigcup_{c\in\mathrm{grd}(n)}\mathrm{supp}(P_c)$, which requires $\Omega(\sum_{c\in\mathrm{grd}(n)}|\Delta P_c|)$ time. In line 3, algorithm 2 iterates over all tuples in P_c with $c\in\mathrm{ng}(n)$. Clearly, this takes $\Omega(\sum_{c\in\mathrm{ng}(n)}|\Delta P_c|)$ time. Finally, in line 4, algorithm 2 adds $G_n\ltimes\vec{t_c}$ to U, for each $\vec{t_c}\in\Delta P_c$. Per child $c\in\mathrm{ng}(n)$, this hence takes total time $\Omega(\sum_{c\in\mathrm{ng}(n)}|G_n\ltimes\Delta P_c|$. Therefore, lines 2–4 run in time

$$\Omega\left(\sum_{c\in\mathrm{grd}(n)}|\Delta P_c|+\sum_{c\in\mathrm{ng}(n)}|\Delta P_c|+\sum_{c\in\mathrm{ng}(n)}|G_n\ltimes\Delta P_c|\right)=\Omega\left(\sum_{c\in\mathrm{ch}(n)}|\Delta P_c|+\sum_{c\in\mathrm{ng}(n)}|G_n\ltimes\Delta P_c|\right).$$

To see that Algorithm 2 runs in time

$$O\left(\sum_{c\in\mathrm{ch}(n)}|\Delta P_c|+\sum_{c\in\mathrm{ng}(n)}|G_n\ltimes\Delta P_c|\right). \tag{3.1}$$

we first observe that, assuming that set U is represented by a hash table (which has $O(1)$ inserts in our model), lines 2–4 compute the set U in this time. The crucial insight here is that the index $G_{n,c}$ of G_n on $\mathrm{pvar}(c)$ with $c\in\mathrm{ng}(n)$ allows us to compute $\mathrm{supp}(G_n\ltimes\Delta P_c)$ in Line 4 in time $O(|\Delta P_c|+|G_n\ltimes P_c|)$.

In particular, $|U|$ is bounded by (3.1). The time required to do lines 5–8 is hence also bounded by (3.1) since there we iterate over the elements of U and do only $O(1)$ work per tuple. (The tree T is fixed and hence the number of GMR and index lookups as well as the number of multiplications, additions, and substractions are constant.) □

Lemma 3. *If join tree T is simple, then we have for every database db, every update u, and every node $n\in T$ that $|\Delta P_n|\leq|\Delta L_n|\leq\sum_{\boldsymbol{a}\in T|_n}|u_{\boldsymbol{a}}|$.*

Proof. Every tuple in ΔP_n is the projection of a tuple in ΔL_n. Hence, it remains to show the second inequality. We do so by induction on n.

If n is an atom $\boldsymbol{a}$, then $\Delta L_n = u_{\boldsymbol{a}}$ and the result trivially follows.

If n is a hyperedge, then by Lemma 2,

$$|\Delta L_n|\leq\sum_{c\in\mathrm{grd}(n)}|\Delta P_c|+\sum_{c\in\mathrm{ng}(n)}|G_n\ltimes P_c| \tag{3.2}$$

However, since T is simple, all nodes have only guard children. Therefore,

$$|\Delta L_n| \leq \sum_{c \in \mathrm{grd}(n)} |\Delta P_c| = \sum_{c \in ch(n)} |\Delta P_c| \tag{3.3}$$

Then, by induction hypothesis

$$(3.3) \leq \sum_{c \in \mathrm{ch}(n)} \sum_{\boldsymbol{a} \in T|_c} |u_{\boldsymbol{a}}| \leq \sum_{\boldsymbol{a} \in T|_n} |u_{\boldsymbol{a}}| \qquad \square$$

The following proposition shows that, if T is simple, then DYN processes an update u in time $O(\|u\|)$.

Proposition 6. *If T is simple, then DYN_T processes updates in time $O(\|u\|)$.*

Proof. If T is simple, then all nodes have only guard children. Therefore, for every node n in T, Algorithm 2 runs in time $O(\sum_{c \in \mathrm{ch}(n)} |P_c|)$ by Proposition 5. Moreover, by Lemma 3, $|\Delta G_n|$, $|\Delta L_n|$, and $|\Delta P_n|$ are all bounded by $O(\sum_{a \in T|_n} |u_{\boldsymbol{a}}|)$. (Note that $G_n = L_n$.)

Now reason as follows. The computation of $\Delta L_{\boldsymbol{a}}$ and $\Delta P_{\boldsymbol{a}}$ for atoms $\boldsymbol{a}$ in lines 4–5 are easily seen to take time $O(\| u \|)$. Furthermore, for each hyperedge n, DYN_T calls Algorithm 2 once for to compute ΔL_n, ΔP_n, and ΔG_n. It follows from our discussion above regarding Proposition 5 and Lemma 3 that each call takes time $O(\|u\|)$.

Since T is fixed, lines 4–5 hence take time $O(\|u\|)$.

Once the delta GMRs are computed, it remains to apply the deltas to the materialized views in $\mathcal{D}$'s components (lines 6–10). By the size bound in Lemma 3, this also takes time $O(\|u\|)$. $\square$

Proposition 7. *Let Q be a NCQ, T be a generalized join tree for Q, and n a hyperedge in T. Then, for every GMR R defined over $var(n)$, there is a database db such that (1) $\|db\| = \Omega(\|R\|)$, (2) $\Lambda_n^T(db) = R$, and (3) for every atom $\boldsymbol{a}$ of Q not occurring in $T|_n$, $db_{\boldsymbol{a}} = \emptyset$.*

Proof. We proceed by induction on the depth of n. If n is an atom, define $db_n = R$ and $db_{\boldsymbol{a}} = \emptyset$ for every $\boldsymbol{a} \in at(Q) \setminus \{n\}$. Conditions (2) and (3) hold immediately in this case. As n is an atom, we know that $\Lambda_n^T(db) = db_n$, and (1) is also satisfied.

If n is not an atom, for each $c \in \mathrm{ch}(n)$ and each tuple $\vec{t} \in R$ define $\vec{t}_c$ to be a tuple over $var(c)$ that assigns $\vec{t}(x)$ to each variable $x \in var(n) \cap var(c)$ and an arbitrary value v_0 to each variable in $var(c) \setminus var(n)$. Let g be an arbitrary node in $\mathrm{grd}(n)$, and define the GMR $R_g = \{\vec{t}_c \to \mu \mid (\vec{t}, \mu) \in R\}$. Intuitively, when joining the GMRs associated to the children of n, R_g will provide the correct multiplicities. Following this, for each node $c \in \mathrm{ch}(n) \setminus \{g\}$, define the GMR $R_c = \{\vec{t}_c \to 1 \mid \vec{t} \in \mathrm{supp}(R)\}$. Notice that for two tuples $\vec{t}$ and $\vec{t'}$ it could be the case that $\vec{t}_c = \vec{t'}_c$, but the *assignment* $\vec{t}_c \to 1$ would appear only once in R_c. Now, by induction hypothesis, for each $c \in \mathrm{ch}(n)$ we can make $\Lambda_c^T(db) = R_c$ by defining a database db_c that only instantiates the atoms under $T|_c$. Let $db = \bigcup_{c \in \mathrm{ch}(n)} db_c$. Since the result of $\Lambda_c^T(db)$ only depends on the subtree $T|_c$, and for each node c we have that the database db_c and db are exactly the same for the atoms in $T|_c$, we obtain that $\Lambda_c^T(db_c) = \Lambda_c^T(db) = R_c$.

To show that $\Lambda_n^T(db) = R$ a key observation is that given a tuple $\vec{t} \in \text{supp}(R)$ and a node $c \in \text{ch}(n)$, $\vec{t}_c$ is the only tuple in $\text{supp}(\Lambda_c^T(db))$ such that $\vec{t}_c \sim \vec{t}$. Indeed, by the construction of the $\vec{t}_c$, two different tuples in $\text{supp}(\Lambda_c^T(db))$ cannot have the same values in $var(n) \cap var(c)$ (notice that in the border case $var(n) \cap var(c) = \emptyset$, there is only one tuple in $\text{supp}(\Lambda_c^T(db))$ that assigns the value v_0 to each variable). Now we show that $\Lambda_n^T(db) = R$

- $\Lambda_n^T(db) \subseteq R$. Let $(\vec{t}, \mu) \in \Lambda_n^T(db)$. By definition, there must be a set containing, for each $c \in \text{ch}(n)$, a tuple $\vec{s^c} \in \text{supp}(\Lambda_c^T(db))$ such that $\vec{t} = \bowtie_{c \in \text{ch}(n)} \vec{s^c}[var(n)]$. Since each tuple $\vec{s^c}$ belongs to $\text{supp}(\Lambda_c^T(db))$, by construction it must be equal to $\vec{t'}_c$ for some $\vec{t'} \in \text{supp}(R)$. As we know that $\vec{t'}_c$ and $\vec{t}$ are equivalent on $var(n) \cap var(c)$, by the discussion above we conclude that $\vec{t'}_c = \vec{t}_c$. Then, we have that $\vec{t} = \bowtie_{c \in \text{ch}(n)} \vec{t}_c[var(n)]$. Moreover, for each child c of n the tuple $\vec{t}_c$ is the only tuple in $\in \text{supp}(\Lambda_c^T(db))$ compatible with $\vec{t}$, and thus the multiplicity $\Lambda_n^T(db)(\vec{t})$ is equal to $\Pi_{c \in \text{ch}(n)} \Lambda_c^T(db)(\vec{t}_c)$. But by construction this is a multiplication of only 1's, except for the multiplicity of $\vec{t}_g$ which is equal to the multiplicity of $\vec{t}$ in R. We then conclude that $(\vec{t}, \mu) \in R$.
- $R \subseteq \Lambda_n^T(db)$. Let $(\vec{t}, \mu) \in R$. As discussed before, for each $c \in \text{ch}(n)$ the tuple $\vec{t}_c$ is compatible with $\vec{t}$. Therefore we obtain that $\vec{t} = \bowtie_{c \in \text{ch}(n)} \vec{t}_c[var(n)]$. Moreover, $\vec{t}_c$ is the only tuple compatible with $\vec{t}$ in $\text{supp}(\Lambda_c^T(db))$, and therefore $\Lambda_n^T(db)(\vec{t}) = \Pi_{c \in \text{ch}(n)} \Lambda_c^T(db)(\vec{t}_c)$. By the same discussion above, we have that this is the multiplicity of $\vec{t}$ in R, namely μ, and therefore $(\vec{t}, \mu) \in \Lambda_n^T(db)$.

Finally, notice that property (3) holds trivially, and property (1) holds because of the amount of nodes in T is a constant. □

In the following proposition, we show that for a non-simple GJT T, DYN can process updates in time $\Omega(\|db\| + \|u\|)$.

Proposition 8. *If T is not simple, then there exists a family of database db and update u such that $\text{DYN}_T(\mathcal{D}, u)$ with $\mathcal{D}$ the T-rep for db runs in time $\Omega(\|db\| + \|u\|)$.*

Proof. Let T be a non-simple generalized join tree for a query Q. We show how to construct a database *db* of arbitrary size and a constant-size update u such that $\text{DYN}(\mathcal{D}, u)$ runs in time proportional to $\|db\|$, where $\mathcal{D}$ is the T-rep for *db*. To this end, consider two nodes n, m of T such that n is the parent of m and $var(n) \not\subseteq var(m)$. Let x be a variable in $var(n) \setminus var(m)$ and let v_0 be an arbitrary value. Intuitively, if a sibling of m does not mention x, we will populate it with a single tuple assigning the value v_0 to all variables. Otherwise, we will populate it with an arbitrary amount of tuples, all assigning a different value to x and v_0 to all other variables.

Let us now proceed formally. Let $N \in \mathbb{N}$. For each $c \in \text{ch}(n) \setminus \{m\}$ such that $x \notin var(c)$ define the GMR R_c as $\{\vec{t} \to 1\}$, where $\vec{t}$ is the tuple that assigns v_0 to each variable in $var(c)$. For each $c \in \text{ch}(n)$ such that $x \in var(c)$, define the GMR R_c as $\{\vec{t_k} \to 1 | k \in [1..N]\}$, where $\vec{t_k}$ is the tuple that assigns v_0 to each variable in $var(c) \setminus \{x\}$ and k to x. For each $c \in \text{ch}(n) \setminus \{m\}$ let db_c be the database provided by Proposition 7 that makes $\Lambda_c^T(db_c) = R_c$, and define $db = \bigcup_{c \in \text{ch}(n) \setminus \{m\}} db_c$. By the same argument as in the proof of Proposition 7, $\Lambda_c^T(db_c) = \Lambda_c^T(db)$ for each $c \in \text{ch}(n)$. It is important to notice that $db = O(N)$, since there is at least one node c mentioning the variable x (a guard of n) and the amount of nodes is constant.

Now, since *db* assigns the empty GMR to all atoms in the subtree of m, it is clear that $\Lambda_m^T(db) = \emptyset$. It immediately follows that $\Lambda_n^T(db) = \emptyset$. Now define the GMR $R_m = \{\vec{t} \to 1\}$, where $\vec{t}$ assigns v_0 to each variable in $var(m)$. By Proposition 7 there is a database db_m that only populates atoms in the subtree of m such that $\Lambda_m^T(db_m) = R_m$. Since $\mathrm{supp}(R_m)$ contains only one tuple, db_m is of constant size. Let $u = db_m$.

To prove that $\textsc{Dyn}_T(\mathcal{D}, u)$ runs in time $\Omega(\|db\| + \|u\|)$, where $\mathcal{D}$ is the T-rep for *db*, we show that $\|\Lambda_n^T(db + u)\| = \Omega(N)$. This proves the statement since $\|\Lambda_n^T(db)\| = 0$ and the T-rep must be updated to materialize the result of Λ_n^T, which will take time $\Omega(N)$.

As u only populates atoms under m, for each $c \in \mathrm{ch}(n) \setminus \{m\}$ it is the case that $\Lambda_c^T(db) = \Lambda_c^T(db + u)$. Since *db* only populates atoms not under m, we have $\Lambda_c^T(db + u) = R_m$. For each $k \in [1..N]$ define $\vec{t}_k$ to be the tuple that assigns k to x and v_0 to each variable in $var(n) \setminus x$. From the construction of the GMRs R_c it immediately follows that for each $k \in [1..N]$ the tuple $\vec{t}_k$ is compatible with at least one tuple on R_c. Since $R_c = \Lambda_c^T(db_c) = \Lambda_c^T(db) = \Lambda_c^T(db + u)$, we obtain that $\vec{t}_k$ is compatible with at least one tuple in $\Lambda_c^T(db + u)$, for each $c \in \mathrm{ch}(n)$. As every multiplicity is 1, it follows that $\vec{t}$ occurs with multiplicity one in $\Lambda_n^T(db + u)$.

We have shown that $\Lambda_n^T(db) = \emptyset$ and that $\| \Lambda_n^T(db + u) \| = \Omega(\| db \|)$. Since the size of u is constant, it is straightforward that $\textsc{Dyn}_T(\mathcal{D}, u)$ runs in time $\Omega(\| db \| + \| u \|)$. Since the number N, the value v_0 and the multiplicities were all chosen arbitrarily, it is straightforward to generate a family of database-update pairs that satisfy the statement of this proposition. □

Now we prove theorem 2.

Theorem 2. $\textsc{Dyn}_T(\mathcal{D}, u)$ *produces a T-rep for* $db + u$ *in time* $O(\|u\|)$ *for every database db and every update* u *if, and only if,* T *is simple.*

Proof. The if direction is given by Proposition 6; the only-if direction by Proposition 8. □

Theorem 2 indicates that, before using DYN to dynamically process Q, it is important to check for the existence of a generalized simple join tree for Q.

On non-simple trees T, such as the tree T_3 from Example 3, $\textsc{Dyn}_T$ is less efficient in the worst case. Indeed, as already illustrated above, a single-tuple update can trigger multiple-tuple updates to its ancestors and in the worst case the parent update may be as big as $\|db\|$. In principle, the multiple-tuple update to the parent may cause an even bigger update to the grand-parent (assuming that the latter is not a guard of the grand-parent). The number of tuples in an update to a node can be shown to be always bounded by $\|db\| + \|u\|$, however. Using this observation, we can show:

Proposition 9. *Let* T *be a join tree,* $\mathcal{D}$ *a* T*-rep for db and* u *an update to db.* $\textsc{Dyn}_T(\mathcal{D}, u)$ *produces a T-rep for* $db + u$ *in time* $O(\|db\| + \|u\|)$.

To the prove this proposition, we first prove the following result.

Lemma 4. *For every db, every update* u*, every join tree* T *and every node* $n \in T$ *we have* $|\Delta P_n| \leq |\Delta L_n| \leq \sum_{\boldsymbol{a} \in T|_n} |db_{\boldsymbol{a}}| + |u_{\boldsymbol{a}}|$.

Proof. Every tuple in ΔP_n is the projection of a tuple in ΔL_n. Hence, it remains to show the second inequality. We do so by induction on n.

If n is an atom $\boldsymbol{a}$, then $\Delta L_n = \Delta \boldsymbol{a}(u) = u_{\boldsymbol{a}}$. Hence, $|\Delta L_n| \leq |u_{\boldsymbol{a}}| \leq |db_{\boldsymbol{a}}| + |u_{\boldsymbol{a}}|$, as desired.

If n is a hyperedge, then by Lemma 2,

$$|\Delta L_n| \leq \sum_{c \in \mathrm{grd}(n)} |\Delta P_c| + \sum_{c \in \mathrm{ng}(n)} |G_n \ltimes P_c| \tag{3.4}$$

Note in particular that $|G_n \ltimes P_c| \leq |G_n|$. Then, by induction hypothesis and Lemma 1

$$\begin{aligned}
(3.4) &\leq \sum_{c \in \mathrm{grd}(n)} \sum_{\boldsymbol{a} \in T|_c} |db_{\boldsymbol{a}}| + |u_{\boldsymbol{a}}| + \sum_{c \in \mathrm{ng}(n)} \sum_{\boldsymbol{a} \in T|_c} |db_{\boldsymbol{a}}| \\
&\leq \sum_{c \in \mathrm{ch}(n)} \sum_{\boldsymbol{a} \in T|_c} |db_{\boldsymbol{a}}| + |u_{\boldsymbol{a}}| \\
&\leq \sum_{\boldsymbol{a} \in T|_n} |db_{\boldsymbol{a}}| + |u_{\boldsymbol{a}}|
\end{aligned}$$

□

Having the above results, we prove proposition 9 as follows.

Proposition 9. *Let T be a join tree, $\mathcal{D}$ a T-rep for db and u an update to db.* $\textsc{Dyn}_T(\mathcal{D}, u)$ *produces a T-rep for $db + u$ in time $O(\|db\| + \|u\|)$.*

Proof. For each atom $\boldsymbol{a}$, the computation of $\Delta L_{\boldsymbol{a}}$ and $\Delta P_{\boldsymbol{a}}$ $\boldsymbol{a}$ in lines 4–5 in algorithm 1 are easily seen to take time $O(\|db\| + \|u\|)$. Furthermore, for each hyperedge n, $\textsc{Dyn}_T$ calls Algorithm 2 once for to compute ΔL_n, ΔP_n, and ΔG_n. From Proposition 5, each call takes time

$$O\left(\sum_{c \in \mathrm{ch}(n)} |\Delta P_c| + \sum_{c \in \mathrm{ng}(n)} |G_n \ltimes \Delta P_c|\right) = O\left(\sum_{c \in \mathrm{ch}(n)} |\Delta P_c| + \sum_{c \in \mathrm{ng}(n)} |G_n|\right)$$

Then, by the size bounds on ΔP_c and G_n of Lemma 1 and Lemma 4, respectively, it follows that each call takes time $O(\|db\| + \|u\|)$.

Since T is fixed, lines 4–5 hence take time $O(\|db\| + \|u\|)$.

Once the delta GMRs are computed, it remains to apply the deltas to the materialized views to $\mathcal{D}$'s components (lines 6–10). By the size bounds in Lemmas 4, this also takes time $O(\|db\| + \|u\|)$. □

In other words, in the worst case, $\textsc{Dyn}_T$ runs in time $O(\|db\| + \|u\|)$, which is unfortunately similar to recomputing everything from scratch using CDY after every update. Fortunately, while recomputing everything from scratch will always cost $\Omega(\| db + u \|)$ time, in practice DYN performs much better than its $O(\|db\| + \|u\|)$ upper bound. This is discussed at the end of Section 4.5.1.

Algorithm 3 $\text{ENUM}_{(T,N)}(\mathcal{D})$

for each $(\vec{t}_{n_1}, \mu_{n_1}) \in L_{n_1}([\,])$ **do**
  for each $(\vec{t}_{n_2}, \mu_{n_2}) \in L_{n_2}(\vec{t}_{p(n_2)}[\overline{x_{n_2}}])$ **do**
    for each $(\vec{t}_{n_3}, \mu_{n_3}) \in L_{n_3}(\vec{t}_{p(n_3)}[\overline{x_{n_3}}])$ **do**
      ...
      for each $(\vec{t}_{n_k}, \mu_{n_k}) \in L_{n_k}(\vec{t}_{p(n_k)}[\overline{x_{n_k}}])$ **do**
        let $\mu = \mu_{c_1} * \cdots * \mu_{c_l} * \Pi_{d \in D} P_d(\vec{t}_{p(d)}[\overline{x_d}])$
        output $(\vec{E}_{n_1}\vec{E}_{n_2}\ldots\vec{E}_{n_l}, \mu)$

3.2.5 Enumeration

In this section, we show that a T-rep $\mathcal{D}$ for *db*, with T a join tree for join query Q can be used not only to enumerate $Q(db)$ with constant delay, but also some of its projections $\pi_{\overline{x}}Q(db)$. In particular, CDE of projections is possible if there exists a subtree of T that includes the root and contains precisely the set $\overline{x}$ of projected variables. Intuitively, if such subtree exists, the enumeration algorithm will be able to *traverse* $\mathcal{D}$ to find the required values of $\overline{x}$ without traversing tuples containing variables in $var(Q) \setminus \overline{x}$, which may cause unbounded delays in the enumeration. Essentially the same idea has been used before in the context of static CDE [BDG07a] (as discussed in Section 3.2.6), factorized databases [BKOZ13], and worst-case optimal algorithms [JPR16]. We proceed formally.

Definition 11. Let $T = (V, E)$ be a join tree. A subset $N \subseteq V$ is *connex* if it includes the root and the subgraph of T induced by N is a tree.

To illustrate, $\{\{y\}, \{x, y, z\}, \{y, v\}\}$ is a connex subset of the join tree T_3 of Figure 3.2, but $\{\{y\}, S(x, y, u)\}$ is not.

For each join tree T and each connex subset N of the nodes in T, we define the enumeration algorithm $\text{ENUM}_{(T,N)}$ as follows. Let T' be the subtree of T induced by N and let $(n_k, n_{k-1}, \ldots, n_1)$ be a topological sort of T'. In particular, n_1 is the root of T. Let $C = \{c_1, \ldots, c_l\}$ be the set of leaf nodes of T'. Let $p(n)$ denote the parent of node n and let $\overline{x_n}$ denote $\text{pvar}(n)$. Finally, let D be the subset of all nodes in T that are not in N, but for which some sibling is in N. With this notation, $\text{ENUM}_{(T,N)}$ is shown in Algorithm 3. Example 1 shows $\text{ENUM}_{(T_3,N)}(\mathcal{D})$ for the join tree T_3 of Figure 3.2 with N consisting of all nodes in T_3.

Proposition 10. *Let T be a join tree for join query Q. Assume that N is a connex subset of T. Then $\text{ENUM}_{(T,N)}(\mathcal{D})$ enumerates $Q'(db) := \pi_{var(N)}Q(db)$ with constant delay, for every database db and every T-rep $\mathcal{D}$ of db.*

We begin the proof of this proposition by observing that the classical rewrite rule from relational algebra which pushes projections over joins continues to hold for GMRs. This result straighforward follows from the distributivity of product over sums.

Lemma 5. *Let $\overline{x}$ and $\overline{y}$ be sets of variables, not necessarily disjoint. Assume that $\overline{u}$ and $\overline{v}$ are additional variables, disjoint with $\overline{x}$ resp. $\overline{y}$, such that all variables shared between $\overline{xu}$ and $\overline{yv}$ are in $\overline{x} \cap \overline{y}$. Then, for all queries $Q(\overline{xu})$ and $P(\overline{xv})$ we have*

$$\pi_{\overline{xy}}\left(Q(\overline{xu}) \bowtie P(\overline{yv})\right) \equiv \left(\pi_{\overline{x}}Q(\overline{xu})\right) \bowtie \left(\pi_{\overline{y}}P(\overline{yv})\right)$$

Next, we prove Proposition 10 in three steps as follows. Let O be the GMR output by $\text{ENUM}_{(T,N)}(\mathcal{D})$. In step (1), given in Lemma 6, we show that $\text{ENUM}_{(T,N)}$ enumerates this set with constant delay. In step (2), given in Lemmas 7 and 8, we show that O can be written as the join of certain queries associated to certain nodes in T. Finally, in step (3) we show that this join is equivalent to the projection query Q' of Proposition 10.

Lemma 6. *Let T be a join tree, N a connex subset of T and $\mathcal{D}$ a T-rep. Let O be the GMR output by $\text{ENUM}_{(T,N)}(\mathcal{D})$. Then $\text{ENUM}_{(T,N)}(\mathcal{D})$ enumerates O with constant delay.*

Proof. We need to establish that (1) every pair in O is output exactly once; (2) the time required to output the first element; between two consecutive elements; and between the output of the last element and the termination of enumeration, are all constant.

(Part 1) It suffices to recognize that if tuple $\vec{t}_{n_1} \dots \vec{t}_{n_k}$ were to be output more than once (possibly with different multiplicities), then this implies that some node n_i would have to enumerate $\vec{t}_{n_i}$ more than once, for exactly the same values of $\vec{t}_{n_1}, \dots, \vec{t}_{n_{i-1}}$ in the above-lying for-loops. This implies by itself that the GMR returned by index lookup $L_{n_i}(\vec{t}_{p(n_i)}[\overline{x_{n_i}}]$ would enumerate $\vec{t}_{n_i}$ twice, which is impossible.

(Part 2) To see that we spend a constant amount of work to produce the first output pair, observe that $\text{ENUM}_{(T,N)}(\mathcal{D})$ starts with the index lookup $L_{n_1}([\,])$ with n_1 the root of T and $[\,]$ the empty tuple. The GMR resulting from this lookup is hence equivalent to $\Lambda_{n_1}(db)$. Recall that in our model index lookups take constant time and GMRs allow enumeration of their support with constant delay. As such, in constant time we find that either $\Lambda_{n_1}(db)$ is empty—in which case $\text{ENUM}_{(T,N)}(\mathcal{D})$ immediately terminates—or we identify a first pair in $(\vec{t}_{n_1}, \mu_{n_1}) \in \Lambda_{n_1}(db)$. Then, in the following line, $\text{ENUM}_{(T,N)}(\mathcal{D})$ does an index lookup $L_{n_2}(\vec{t}_{p(n_2)}[\overline{x_{n_2}}])$. Again, this lookup takes constant time in our model, and returns the GMR consisting of all tuples in Λ_{n_2} that are compatible with $\vec{t}_{p(n_2)}[\overline{x_{n_2}}]$. Since $\vec{t}_{p(n_2)} \in \Lambda_{p(n_2)} = \Lambda_{n_1}$ we obtain, per definition of Λ_{n_1} that the latter GMR cannot be empty (otherwise, this would invalidate $\vec{t}_{p(n_2)} \in \Lambda_{n_1}$ with n_1 parent of n_2). Since GMRs allow themselves enumeration with constant delay, we can hence identify in constant time a pair $(\vec{t}_{n_2}, \mu_{n_2}) \in \Lambda_{n_2}(db)$. Continuing this reasoning on the other nested for-loops of $\text{ENUM}_{(T,N)}(\mathcal{D})$ we obtain that in constant time we can identify suitable pairs $(\vec{t}_{n_i}, \mu_{n_i}) \in P_{n_i}(\vec{t}_{p(n_i)}[\overline{x_{n_i}}]$, for every $2 \leq i \leq k$. This makes the overall delay to output the first tuple linear in k, but since k depends only on $|T|$ which itself depends only on the number of atoms in Q and since Q is fixed, this is constant. To actually output the first pair, it then remains to do a number of multiplications that is again linear in T (hence constant). Since multiplications take constant time in our model, the overall delay is hence constant.

A similar analysis shows that we spend a constant amount of work between any two consecutive output pairs; and between the last tuple and termination of the algorithm. □

Lemma 7. *If N is a connex subset of join tree T and C and D are as defined before, then $var(N) = \bigcup_{c \in C} var(c) \cup \bigcup_{d \in D} \text{pvar}(d)$.*

Proof. $\supseteq$. Since $C \subseteq N$, obviously, $\bigcup_{c \in C} var(c) \subseteq N$. To see that also $\bigcup_{d \in D} \text{pvar}(d) \subseteq var(N)$, consider $d \in D$. By definition, there is a sibling of d in T that belongs to N. Let n be this sibling. Then, since N is connex, $p(n) \in N$. Since d is a child of $p(n)$, necessarily $\text{pvar}(d) \subseteq var(p(n)) \subseteq var(N)$.

$\subseteq$. We show that $\mathit{var}(n) \subseteq \bigcup_{c\in C} \mathit{var}(c) \cup \bigcup_{d\in D} \mathrm{pvar}(d)$ for every $n \in N$. Hereto, let $n \in N$. We discern two cases. If $n \in C$ then the desired inclusion obviously holds. Otherwise, n is a non-leaf node in T'. We observe:

$$\begin{aligned}
\mathit{var}(n) = & \bigcup_{m\in \mathrm{grd}(n)} \mathrm{pvar}(m) \subseteq \bigcup_{m\in \mathrm{ch}(n)} \mathrm{pvar}(m) \\
\subseteq & \bigcup_{m\in \mathrm{ch}(n)\cap N} \mathrm{pvar}(m) \cup \bigcup_{m\in \mathrm{ch}(n)\setminus N} \mathrm{pvar}(m) \\
\subseteq & \bigcup_{m\in \mathrm{ch}(n)\cap N} \mathit{var}(m) \cup \bigcup_{m\in \mathrm{ch}(n)\cap D} \mathrm{pvar}(m) \\
\subseteq & \bigcup_{m\in \mathrm{ch}(n)\cap N} \mathit{var}(m) \cup \bigcup_{d\in D} \mathrm{pvar}(d)
\end{aligned}$$

By iterating this reasoning on the children $m \in \mathrm{ch}(n) \cap N$ until the leaves in T', we obtain that $\mathit{var}(n) \subseteq \bigcup_{c\in C} \mathit{var}(c) \cup \bigcup_{d\in D} \mathrm{pvar}(d)$. □

Lemma 8. *Let T be a join tree for join query Q. Assume that N is a connex subset of T and let C and D be defined as before. Then* $\textsc{enum}_{(T,N)}(\mathcal{D})$ *enumerates*

$$(\bowtie_{c\in C} \Lambda_c \bowtie \bowtie_{d\in d} \Psi_d)(db) \tag{3.5}$$

for every database db and every T-rep $\mathcal{D}$ for db.

Proof. Fix database *db* and T-rep $\mathcal{D}$ for *db*. Let $R := (\bowtie_{c\in C} \Lambda_c \bowtie \bowtie_{d\in d} \Psi_d)(db)$. The proof proceeds in two steps. Step 1 verifies that for all $\vec{t} \in \mathrm{supp}(R)$ there is a pair $(\vec{t}, \mu)$ enumerated by $\textsc{enum}_{(T,N)}(\mathcal{D})$ with $\mu = R(\vec{t})$. Step 2 verifies that for all pairs $(\vec{t}, \mu)$ output by $\textsc{enum}_{(T,N)}(\mathcal{D})$ we have $\vec{t} \in \mathrm{supp}(R)$ and $\mu = R(\vec{t})$.

(Step 1: $\supseteq$). Let $\vec{t} \in \mathrm{supp}(R)$. By definition, $\vec{t}[\mathit{var}(c)] \in \Lambda_c(db)$ and $\vec{t}[\mathrm{pvar}(d)] \in \Psi_d(db)$, for every $c \in C$ and $d \in D$.

Claim: For every interior node $n \in N$ in T', there exists a tuple $\vec{t_n} \in \Lambda_n(db)$ which is compatible: (i) with $\vec{t}$, (ii) with all $\vec{t_{n'}}$ for n' a descendant of n in T'; and (iii) with $\vec{t}[\mathrm{pvar}(d)]$, for every $d \in D$.

Proof of Claim: straightforward, by induction on the height of n in T'. If n is a leaf of T', then it suffices to take $\vec{t_n} := \vec{t}[\mathit{var}(n)] \in \Lambda_n(db)$. If n is a non-leaf node, the result follows straightforward from the induction hypothesis, using the fact that T is a GJT.

It immediately follows from this claim and the fact that L_n is an index of $\Lambda_n(db)$ on $\mathrm{pvar}(n)$ that during execution of the nested for-loop of $\textsc{enum}_{(T,N)}(\mathcal{D})$ we will reach a point where the current visited tuples for node $n \in N$ are exactly the tuples $\vec{t_n}$ of the claim. At that point, it follows from Lemma 7 that $\vec{t} = (\vec{t_{n_1}} \ldots \vec{t_{n_k}})$. Moreover, the μ-value calculated at that point is $\mu = \Pi_{c\in C}\mu_c * \Pi_{d\in D} P_d(\vec{t_{p(d)}}[\overline{x_d}])$ with $\mu_c = \Lambda_c(db)(\vec{t_c})$. Since $p(d) \in T'$ by definition of D; and since $\mathrm{pvar}(d) \subseteq var(d)$ we also have that $\vec{t_{p(d)}}[\overline{x_d}] = \vec{t}[\mathrm{pvar}(d)]$, and therefore

$$\begin{aligned}
\mu =& \Pi_{c\in C}\mu_c * \Pi_{d\in D} P_d(\vec{t_{p(d)}}[\overline{x_d}]) \\
=& \Pi_{c\in C}\Lambda_c(db)(\vec{t}[\mathit{var}(c)]) \times \Pi_{d\in D}\Psi_d(\vec{t}[\mathrm{pvar}(d)]) = R(\vec{t})
\end{aligned}$$

as desired.

(Step 2: $\subseteq$). Consider a point during the execution of the nested for-loop of algorithm $\text{ENUM}_{(T,N)}(\mathcal{D})$ where a new output pair is produced. Then in particular, we have, for every $1 \leq i \leq k$, a pair $(\vec{t}_{n_i}, \mu_{n_i})$ such that $(\vec{t}_{n_i}, \mu_{n_i}) \in L_{n_i}(\vec{t}_{p(n_i)}[\overline{x_{n_i}}])$. Let $\vec{t} = \vec{t}_{n_1} \dots \vec{t}_{n_k}$ be the constructed output tuple, and $\mu = \Pi_{c \in C} \mu_c * \Pi_{d \in D} P_d(\vec{t}_{p(d)}[\overline{x_d}])$ its multiplicity.

To show that $\vec{t} \in R$, we need to show that $\vec{t}[\mathit{var}(c)] \in \Lambda_c(db)$ and $\vec{t}[\text{pvar}(d)] \in \Psi_d(db)$, for every $c \in C$ and $d \in D$. In this respect, observe that $\vec{t}[\mathit{var}(c)] = \vec{t}_c$, for every $c \in C$. Then, since $\vec{t}_c \in L_c(\vec{t}_{p(c)}[\overline{x_c}])$ and since L_c is an index of $\Lambda_c(db)$ on $\text{pvar}(c)$, we know that $\vec{t}[\mathit{var}(c)] = \vec{t}_c \in \Lambda_c(db)$. Moreover, for every $d \in D$ we know that $p(d) \in N$ and $\text{pvar}(d) \subseteq \mathit{var}(p(d))$. Therefore, $\vec{t}[\text{pvar}(d)] = \vec{t}_{p(d)}[\text{pvar}(d)]$. Since $\vec{t}_{p(d)} \in L_{p(d)}(\vec{t}_{p(p(d))}[\overline{x_{p(d)}}])$, we know in particular that $\vec{t}_{p(d)} \in \Lambda_{p(d)}(db)$. By definition of Λ, this implies that $\vec{t}_{p(d)}[\text{pvar}(m)] \in \Psi_m(db)$, for every child m of $p(d)$. In particular, this holds for $m = d$. Therefore, $\vec{t}[\text{pvar}(d)] = \vec{t}_{p(d)}[\text{pvar}(d)] \in \Psi_d(db)$. We hence conclude that $\vec{t} \in R$.

It remains to show that $\mu = R(\vec{t})$, but this trivially follows from the observations already made. □

Proposition 10. *Let T be a join tree for join query Q. Assume that N is a connex subset of T. Then $\text{ENUM}_{(T,N)}(\mathcal{D})$ enumerates $Q'(db) := \pi_{\mathit{var}(N)} Q(db)$ with constant delay, for every database db and every T-rep $\mathcal{D}$ of db.*

Proof. Let db be a database and let $\mathcal{D}$ be T-rep of db. By combining of Lemma 6 and Lemma 8 we obtain that $\text{ENUM}_{(T,N)}(\mathcal{D})$ enumerates $(\bowtie_{c \in C} \Lambda_c \bowtie \bowtie_{d \in d} \Psi_d)(db)$. It remains to show that

$$\bowtie_{c \in C} \Lambda_c \bowtie \bowtie_{d \in d} \Psi_d \equiv \pi_{\mathit{var}(N)} Q$$

Towards this equivalence, observe that, for every node $n \in T$:

$$\Lambda_n(db) \equiv \pi_{\mathit{var}(n)} \bowtie_{\boldsymbol{a} \in \mathit{at}(T|_n)} \boldsymbol{a}$$
$$\Psi_n(db) \equiv \pi_{\text{pvar}(n)} \bowtie_{\boldsymbol{a} \in \mathit{at}(T|_n)} \boldsymbol{a}.$$

where $\mathit{at}(T|_n)$ denotes the set of atoms in the subtree of T rooted at N. Therefore, $\bowtie_{c \in C} \Lambda_c \bowtie \bowtie_{d \in d} \Psi_d$ is equivalent to

$$\bowtie_{c \in C} \left(\pi_{\mathit{var}(c)} \bowtie_{\boldsymbol{a} \in \mathit{at}(T|_c)} \boldsymbol{a} \right) \bowtie \bowtie_{d \in D} \left(\pi_{\text{pvar}(d)} \bowtie_{\boldsymbol{a} \in \mathit{at}(T|_d)} \boldsymbol{a} \right) \tag{3.6}$$

Further observe that, since T is a join tree, we know that, for every $c, c' \in C$ with $c \neq c'$, the only variables that atoms in $T|_c$ can share with atoms in $T|_{c'}$ must be in $\mathit{var}(c_1) \cap \mathit{var}(c_2)$. Therefore, we can repeatedly apply the equivalence of Lemma 5 to lift projection over joins, and hence obtain that (3.6) is equivalent to

$$\pi_{\bigcup_{c \in C} \mathit{var}(c)} \left(\bowtie_{c \in C} \bowtie_{\boldsymbol{a} \in \mathit{at}(T|_c)} \boldsymbol{a} \right) \bowtie \bowtie_{d \in D} \left(\pi_{\text{pvar}(d)} \bowtie_{\boldsymbol{a} \in \mathit{at}(T|_d)} \boldsymbol{a} \right) \tag{3.7}$$

Further note that, again because T is a join tree and the nodes $d \in D$ are nodes in T that themselves are not in N but one of their siblings is, the only variables that the atoms in $T|_d$ can share with the atoms in one of the $T|_c$ (with $c \in C$) or $T|_{m_g}$ (with $d' \in D$, $d' \neq d$) must be in $\text{pvar}(d)$. Hence, we can again repeatedly apply the of equivalence Lemma 5 to lift projection over joins, and hence obtain that (3.7) is equivalent to

$$\pi_{\bigcup_{c \in C} \mathit{var}(c) \cup \bigcup_{d \in D} \text{pvar}(d)} \left(\bowtie_{c \in C} \bowtie_{\boldsymbol{a} \in \mathit{at}(T|_c)} \boldsymbol{a} \bowtie \bowtie_{d \in D} \bowtie_{\boldsymbol{a} \in \mathit{at}(T|_d)} \boldsymbol{a} \right) \tag{3.8}$$

Algorithm 4 $\text{LookUp}_{(T,N)}(\mathcal{D}, \vec{t}[\overline{y}], n)$

```
result= true
x̄ = var(n) ∩ ȳ
if t⃗[x̄] ∈ L_n(t⃗[pvar(n)]) then
    for each child c ∈ ch(n) do
        result := (LookUp(D, t⃗[ȳ], c) && result)
    return result
return false
```

Now, note that (3.8) mentions all atoms in Q (every atom in Q must appear in $T|_c$, for some $c \in C$ or in $T|_d$ for some $d \in D$). Therefore, this is equivalent to:

$$\pi_{\bigcup_{n \in C \cup D} var(n)} \Big(\bowtie_{\boldsymbol{a} \in at(T)} \boldsymbol{a} \Big)$$

This is equivalent to Q' since $var(N) = \bigcup_{c \in C} var(c) \cup \bigcup_{d \in D} \text{pvar}(d)$ by Lemma 7. □

Next we present the proposition to show that DCLRs allow to check if a tuple $\vec{t} \in Q(db)$ in constant time for a NCQ Q over the database *db*.

Proposition 11. *Let T be a join tree for join query Q. Assume that N is a connex subset of T. Then the algorithm $\text{LookUp}_{(T,N)}(\mathcal{D}, \vec{t}[\overline{y}], n)$ checks if an arbitrary tuple $\vec{t} \in Q(db)$ is in $\mathcal{D}_{db}$ in constant time.*

Proof. Using the T-rep for a query Q, we can test if a tuple $\vec{t}$ is in the result $Q(db)$ of Q over the database *db* in constant time. This result follows from the proposition 3 since all the tuples in the index L_n for each node $n \in T$ are live and represent the join of the subtree rooted by n. With this observation, it is straight forward to check if $\vec{t} \in Q(db)$ using L_n for each node $n \in T$ starting from the root recursively. Let T be a GJT for a query Q, N a connex set in T and $\mathcal{D}_T$ be the T-rep for Q. Using the live indexes L_n for each node n in N, the algorithm 4 tests if a tuple $\vec{t} \in Q(db)$ in constant time as follows: let $\vec{t}$ be a tuple over the sequence of variables $\overline{y} = out(Q)$ and let $\overline{x}$ be the sequence of variables a node $n \in T$ shares with $\vec{t}$ i.e. $\overline{x} = \overline{y} \cap var(n)$. Starting from the root n of T, if $\vec{t}[\overline{x}] \in L_n(\vec{t}[\text{pvar}(n)])$ then we iterate over the children of n to check if the tuple exists in its children, otherwise the algorithm terminates and the tuple is not found. Since by the proposition 10 we know that each L_n contains tuples that are live for the node n and tuples in $Q(db)$ can be enumerated with constant delay, the algorithm 4 hence performs this check with constant delay. □

Note that $\text{ENUM}_{T,V}(\mathcal{D})$ with V the set of all nodes in T enumerates $Q(db)$. Combining all of our results so far we obtain that join-tree representations are DCLRs for the class of all *free-connex acyclic NCQs*, which is defined as follows.

Definition 12. (Compatible, Free-Connex Acyclic) Let T be a join tree. A NCQ Q is *compatible with* T if T is a join tree for Q and T has a connex subset N with $var(N) = out(Q)$. A NCQ is *free-connex acyclic* if it has a compatible join tree.

In particular, every acyclic join query is free-connex acyclic. Let T be a join tree. It now follows that the class of all T-reps form a DCLR of every NCQ Q compatible with T.

Algorithm 5 $DeltaEnum_{(T,N)}(\Delta\,\mathcal{D})$

1: **for each** $(\vec{t}_{n_1}, \mu_{n_1}) \in \Delta L_{n_1}([\,])$ **do**
2: **for each** $(\vec{t}_{n_2}, \mu_{n_2}) \in \Delta L_{n_2}(\vec{t}_{p(n_2)}[\overline{x_{n_2}}])$ **do**
3: ...
4: **for each** $(\vec{t}_{n_j}, \mu_{n_j}) \in \Delta L_{n_j}(\vec{t}_{p(n_j)}[\overline{x_{n_j}}])$ **do**
5: **for each** $(\vec{t}_{n_{j+1}}, \mu_{n_{j+1}}) \in L_{n_{j+1}}(\vec{t}_{p(n_{j+1})}[\overline{x_{n_{j+1}}}])$ **do**
6: ...
7: **for each** $(\vec{t}_{n_k}, \mu_{n_k}) \in L_{n_k}(\vec{t}_{p(n_k)}[\overline{x_{n_k}}])$ **do**
8: let $\mu = \mu_{c_1} * \cdots * \mu_{c_l} * \Pi_{d\in D} P_d(\vec{t}_{p(d)}[\overline{x_d}])$
9: **output** $(\vec{E}_{n_1}\vec{E}_{n_2}\ldots\vec{E}_{n_l}, \mu)$

Delta-enumeration. Using DYN_T we can actually also enumerate deltas $\Delta Q(db, u)$ of $Q(db)$ under single-tuple update u. This result is relevant for *push-based* query processing systems, where users do not ping the system for the complete current query answer, but instead ask to be notified of the changes to the query results when the database changes. In addition, as we will discuss in Section 3.3, it also provides a key method for dynamic processing of Conjunctive Aggregate Queries (CAQs) as we discuss in the subsection 3.3.1.

Definition 13 (ΔT-rep)**.** Let T be a join tree, *db* be a database and let u be an update to *db*. A *ΔT-representation* of (db, u) is a data structure $\Delta\mathcal{D}$ that contains (1) a T-rep for *db*; (2) an index ΔL_n of $\Delta\Lambda_n(db, u)$ on $\text{pvar}(n)$, for every $n \in T$; and (3) a GMR ΔP_n that materializes $\Delta\Psi_n(db, u)$, for every $n \in T$.

Note that DYN_T needs to compute $\Delta L_n = \Delta\Lambda_n(db, u)$ and $\Delta P_n = \Delta\Psi_n(db, u)$ anyway when processing T-rep $\mathcal{D}$ under update u. Hence, after the bottom-up pass in lines 4–5 of DYN_T we obtain a ΔT-rep $\Delta\mathcal{D}$, provided that we represent ΔL_n as an index on $\text{pvar}(n)$.

Theorem 3. *Let u be a single-tuple update to database db. Let $\Delta\mathcal{D}$ be a ΔT-rep of (db, u) and let Q be compatible with T. Then $\Delta Q(db, u)$ can be enumerated with constant delay from $\Delta\mathcal{D}$.*

Proof. First of all, observe that when an update occurs to an atom $\boldsymbol{a}$ in the database *db*, then only the nodes at the path from $\boldsymbol{a}$ until the root node of T have non-empty deltas. Then, it makes sense to modify the algorithm for enumeration such that, for the nodes in the path between $\boldsymbol{a}$ and the root node we iterate on the new tuples only i.e. on ΔL_x for every node x in this path. To show this, let T be a GJT where $\boldsymbol{a}$ is an atom and a leaf node in T and n_1 is the root in T, N be a connext subset in T, and let X denote the set of nodes in the path between $\boldsymbol{a}$ and n_1. Let $n_1, \ldots, n_j$ be the nodes in $N \cap X$, and let $n_{j+1}, \ldots, n_k$ be the nodes in $N \setminus X$. Moreover, let D be the set of all nodes in T that are not in N but has a sibling in N. Then, with this notation $DeltaEnum(T, N)$ is shown in algorithm 5. This algorithm works similar to the algorithm for full enumeration except that for the node in X, we iterate over the deltas instead of their full GMRs. Since each ΔL_x for $x \in X$ is also a GMR, the proof of full enumeration proposition 10 holds true for delta enumeration. □

3.2.6 Optimalitiy

In this section we show that Dynamic Yannakakis is optimal in two aspects. (1) It is able to dynamically process the largest subclass of NCQs for which DCLRs can reasonably be expected to exist. (2) The class of queries for which DYN processes updates in $O(\|u\|)$ time (Theorem 2) is the largest class of queries for which we can reasonably expect to have such update processing time as well as CDE of results.

DCLR-optimality. In the static setting without updates, a query Q is said to be in class CD $\circ$ LIN if there exists an algorithm that, for each database *db* does an $O(\|db\|)$-time precomputation and then proceeds in CDE of $Q(db)$, evaluated under set semantics. Bagan et al. showed that, under the so-called *binary matrix multiplication conjecture*, an acyclic NCQ is in CD $\circ$ LIN if and only if it is free-connex [BDG07a]. Recently, Brault-Baron extended this result under two assumptions: the triangle hypothesis (checking the presence of a triangle in a hypergraph with n nodes cannot be done in time $O(n^2)$) and the tetrahedron hypothesis (for each $k > 2$, checking whether a hypergraph contains a k-simplex cannot be done in time $O(n)$). Under these assumptions, he shows that a NCQ is in CD $\circ$ LIN if and only if it is free-connex acyclic [BB13].

Proposition 12. *Under the above-mentioned hypotheses, a DCLR exists for NCQ Q if, and only if, Q is free-connex acyclic.*

Crux. The if direction follows from all of our results so far. For the only if direction, assume that query Q is not free-connex acyclic and suppose that a DCLR exists for query Q. In particular, we can compute, for every database *db* a data structure $\mathbb{D}$ that represents $Q(db)$, for every database *db*. Let U be the algorithm that maintains these datastructures under updates and let ϵ be the $DCLR$ that represents the empty query result (which is obtained when Q is evaluated on the empty database). Then, starting from ϵ, $U(\epsilon, db)$ must construct a DCLR in time $O(\|\epsilon\| + \|db\|) = O(\|db\|)$ since ϵ is constant. Now enumerate $Q(db)$ from $U(\epsilon, db)$ with constant delay but do not output tuple multiplicities. This enumerates $Q(db)$ evaluated under set semantics. Then $Q \in$ CD$\circ$LIN, contradicting Brault-Baron [BB13]. □

Processing-time optimality. Berkholz et al. have recently characterized the class of self-join free NCQs that feature a representation that allows both CDE of results and $O(1)$ maintenance under single-tuple updates [BKS17]. In particular, they show that, under the assumption of hardness of the Online Matrix-Vector Multiplication problem [HKNS15], the following dichotomy holds. When evaluated under set semantics, a NCQ Q without self joins features a representation that supports CDE and maintenance in $O(1)$ time under single-tuple updates if, and only if, Q is *q-hierarchical*. Notice that Q can be maintained in $O(1)$ time under single-tuple updates if, and only if, it can also be maintained in $O(\|u\|)$ time under arbitrary updates. The definition of *q-hierarchical* queries is the following.

Definition 14 (q-hierarchicality)**.** Given a NCQ Q and a variable $x \in var(Q)$, let $at(x)$ denote the set of all atoms in which x occurs in Q. Q is called *hierarchical* if for every pair of variables $x, y \in var(Q)$, either $at(x) \subseteq at(y)$ or $at(y) \subseteq at(x)$ or $at(x) \cap at(y) = \emptyset$. A NCQ Q is q-hierarchical if it is hierarchical and for every two variables $x, y \in var(Q)$, if $x \in out(Q)$ and $at(x) \subsetneq at(y)$, then $y \in out(Q)$.

Example 4. *Consider the join query $Q = R(x, y, z) \bowtie S(x, y, u) \bowtie T(y, v, w) \bowtie U(y, v, p)$. Q is hierarchical. Moreover, $\pi_{x,y}Q$ is q-hierarchical. In contrast, $\pi_u Q$ is not q-hierarchical since $u \in out(Q)$, $at(u) \subsetneq at(y)$, yet $y \notin out(Q)$.*

Observe that a join query is q-hierarchical iff it is hierarchical. The hierarchical property has actually already played a central role for efficient query evaluation in various contexts [DS13, FO16, KS11], see [BKS17] for a discussion.

The following two propositions establish the relationship between DYN and the dichotomy of Berkholz et al.

Proposition 13. *If a NCQ Q is q-hierarchical, then it has a join tree which is both simple and compatible.*

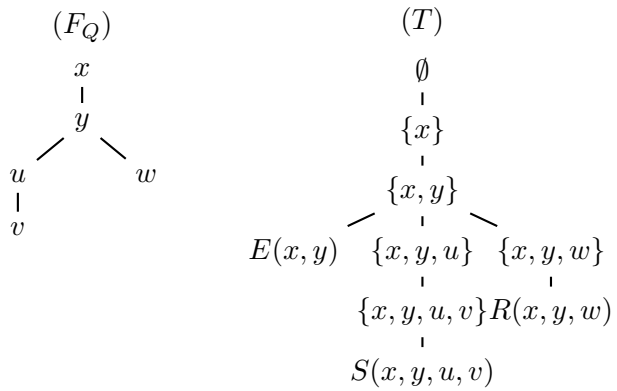


Figure 3.4: Illustration of the proof of Proposition 13

Proof. A NCQ Q is *connected* if for any two $x, y \in \mathit{var}(Q)$ there is a path $x = z_0, \ldots, z_l = y$ such that for each $j < l$ there is an atom $\boldsymbol{a}$ in Q such that $\{z_j, z_{j+1}\} \subseteq \mathit{var}(\boldsymbol{a})$. It is a standard observation that every NCQ can be written as a join $Q_1 \bowtie \cdots \bowtie Q_k$ of connected NCQs with pairwise disjoint sets of output variables. Call these Q_i the connected components of Q. Berkholz et al. [BKS17] show that NCQ Q is hierarchical if and only if every connected component Q_i of Q has a q-tree, which is defined as follows.

Definition 15. Let Q_i be a connected NCQ. A q-tree for Q_i is a rooted directed tree $F_{Q_i} = (V, E)$ with $V = \mathit{var}(Q)$ s.t. (1) for all atoms $\boldsymbol{a}$ in Q_i the set $\mathit{var}(\boldsymbol{a})$ forms a directed path in F_Q starting at the root, and (2) if $\mathit{out}(Q_i) \neq \emptyset$, then $\mathit{out}(Q_i)$ is a connected subset in F_{Q_i} containing the root.

To show the proposition, assume that NCQ Q is hierarchical. From the q-trees for the connected components of Q we can construct a simple join tree T for Q that is compatible with Q, as follows. For ease of exposition, let us assume that Q has a single connected component; the general case is similar. Let F_Q be the q-tree for Q. Then, for every node x in F_Q, define $p(x)$ to be set of variables that occur in the unique path from x to the root in F_Q. In particular, $p(x)$ contains x. By definition of q-trees, $p(x)$ must be a partial hyperedge of A, the set of atoms in Q. Construct T as follows. Initially, T contains only the empty hyperedge $\emptyset$. For all variables $x \in \mathit{var}(Q)$, add hyperedge $p(x)$ to T. For every edge $x \to y$ in F_Q, add an edge $p(x) \to p(y)$ to T. If x is the root in F_Q, then also add the edge $p(x) \to \emptyset$ to the root $\emptyset$ in T. Next, add all atoms of Q to T, and for each atom $\boldsymbol{a}$, add an edge from $\boldsymbol{a}$ to the hyperedge h in T with $h = \mathit{var}(\boldsymbol{a})$. (This hyperedge has been generated by $p(x)$ with x the variable in $\mathit{var}(\boldsymbol{a})$ that is the lowest among all variables of $\mathit{var}(\boldsymbol{a})$ in F_Q). Figure 3.4 illustrates this construction for $Q = \pi_{x,y,u}(E(x, y) \bowtie R(x, y, w) \bowtie S(x, y, u, v))$ and the q-tree F_Q shown in Figure 3.4. Note that in this example, T is indeed a simple generalized join tree. It can be shown that this is always the case. It remains to show that T is compatible with Q. To do so, observe that, by definition of q-tree, $\mathit{out}(Q)$ is a connected subset of F_Q that contains the root. Then $N = \{\emptyset\} \cup \{p(x) \mid x \in \mathit{out}(Q)\}$ must be a connex subset of T with $\mathit{var}(N) = \mathit{out}(Q)$, as desired. □

It then follows from Theorem 2 and Proposition 10 that, for q-hierarchical queries, DYN also processes single-tuple updates in $O(1)$ time while allowing the query result to be enumerated with constant delay (given the join tree). This hence matches the algorithm provided by Berkholz et al. for processing q-hierarchical queries under updates. Note that, by Proposition 13, all q-hierarchical queries must be free-connex acyclic.

Proposition 14. *If a NCQ Q has a join tree T which is both simple and compatible with Q, then Q is q-hierarchical.*

Proof. Assume that T is a simple join tree for Q that is also compatible with Q. We first show that Q is hierarchical. Let x and y be two variables in Q. If $at(x) \cap at(y) = \emptyset$ we are done. Hence, assume $at(x) \cap at(y) \neq \emptyset$. Let $\boldsymbol{c} \in at(x) \cap at(y)$. We need to show that either $at(x) \subseteq at(y)$ or $at(y) \subseteq at(x)$. Assume for the purpose of contradiction that neither holds. Then there exists $\boldsymbol{a} \in at(x) \setminus at(y)$ and similarly an atom $\boldsymbol{b} \in at(y) \setminus at(x)$. Since T is a join tree, and since x occurs both in $\boldsymbol{a}$ and $\boldsymbol{c}$, we know that x must occur in every node on the unique undirected path between $\boldsymbol{a}$ and $\boldsymbol{c}$ in T. In particular, let n be the least common ancestor of $\boldsymbol{a}$ and $\boldsymbol{c}$. Then $x \in var(n)$. Similarly, y must occur in every node on the unique undirected path between $\boldsymbol{b}$ and $\boldsymbol{c}$ in T. In particular, let m be the least common ancestor of $\boldsymbol{b}$ and $\boldsymbol{c}$. Then $y \in var(m)$. Now there are two possibilities. Either (1) n is an ancestor of m. But then, since T is simple, $x \in var(n) \subseteq var(m)$. Since $\boldsymbol{b}$ is a descendant of m then by simplicity of T hence $x \in var(n) \subseteq var(m) \subseteq var(\boldsymbol{b})$. This contradicts our assumption that $\boldsymbol{b} \in at(y) \setminus at(x)$. Otherwise, (2) m is an ancestor of m and we similarly obtain a contradiction to our assumption that $\boldsymbol{a} \in at(x) \setminus at(y)$.

It remains to show q-hierachicality. Hereto, assume that $at(x) \subsetneq at(y)$ and $x \in out(Q)$. We need to show that $y \in out(Q)$. Let $\boldsymbol{a} \in at(x)$ and let $\boldsymbol{b} \in at(y) \setminus at(x)$. In particular, $\boldsymbol{a}$ contains both x and y, while $\boldsymbol{b}$ contains only y. From compatibility of T with Q, it follows that there is a connex subset N of T such that $var(N) = out(Q)$. Let n be the lowest ancestor node of $\boldsymbol{a}$ in N that contains x. Because T is simple, all descendants of n hence also have x. As a consequence, $\boldsymbol{b}$ cannot be a descendant of n. Since $\boldsymbol{a}$ and $\boldsymbol{b}$ share y, this implies that the unique undirected path between $\boldsymbol{a}$ and $\boldsymbol{b}$ must pass through n. Because all nodes on this path must share all variables in common between $\boldsymbol{a}$ and $\boldsymbol{b}$, it follows that $y \in var(n) \subseteq var(N) = out(Q)$. □

This result is to be expected, since from Theorem 2 and Proposition 10 we know that for such T we can do CDE of Q and do maintenance under updates in $O(\| u \|)$ time. If other than q-hierarhical queries had simple compatible join trees, Berkholz et al.'s dichotomy would fail. Also observe that, as seen in Section 3.2.4, DYN may process updates in $\omega(\|u\|)$ time on non-simple join trees. Berkholz et al.'s dichotomy implies that this is unavoidable in the worst case.

At this point, we can explain why it is important to work with join trees based on width-one GHDs rather than classical join trees (which do not allow partial hyperedges to occur). Indeed, the following proposition shows that there are hierarchical queries for which no classical join tree is simple. Therefore, if we restrict ourselves to classical join trees we will fail to obtain an $O(\|u\|)$ update time for some q-hierarchical queries.

Proposition 15. *Let Q be the hierarchical join query $R(x,y,z) \bowtie S(x,y,u) \bowtie T(y,v,w) \bowtie U(y,v,p)$. Every simple width-one GHD for Q has at least one partial hyperedge.*

Proof. Let T be a simple width-one GHD for Q and assume, for the purpose of contradiction that T contains only atoms and full hyperedges. T's nodes are hence elements of $\{R(x,y,z), S(x,y,u), T(y,v,w), U(y,v,p), [x,y,z], [x,y,u], [y,v,w], [y,v,p]\}$. Partition this set into

$$XY = \{R(x,y,z), S(x,y,u), [x,y,z], [x,y,u]\}$$
$$YV = \{T(y,v,w), U(y,v,p), [y,v,w], [y,v,p]\}.$$

Now consider the unique undirected path $m, n_1, n_2, \ldots, n_k, p$ between $m = R(x,y,z)$ and $p = T(y,v,w)$. There are two possibilities: either this undirected path shows that some node in XY is a parent of a node in YV, or it shows that some node in YV is a parent of some node in XY. In either case, hierarchicality is violated since nodes in XY all have variable x while nodes in YV don't and, conversely, nodes in YV all have variable v while nodes in XY don't. □

3.3 Experimental Evaluation

In this section, we experimentally measure the performance of DYN, focusing on both throughput and memory consumption. We start by describing how our implementation addresses some practical issues, then we describe in detail the operational setup, and finally present the experimental results.

3.3.1 Practical Implementation

We have described how DYN processes free-connex acyclic NCQs under updates. In this subsection, we first explain how to use DYN as an algorithmic core for practical dynamic query evaluation of the more general class of acyclic conjunctive aggregate queries (not necessarily free-connex).

Definition 16. A conjunctive aggregate query (CAQ) is a query of the form

$$Q' = (\overline{x}, f_1, \ldots, f_n)Q,$$

where Q is a NCQ; $\overline{x} \subseteq \mathit{out}(Q)$ and f_i is an aggregate function over $\mathit{out}(Q)$ for $1 \leq i \leq n$. Q' is *acyclic* if its NCQ Q is so.

Example aggregate functions are $(\text{SUM}(u) \times 3)$ or $\text{AVG}(x)$, assuming u, x in $\mathit{out}(Q)$. The semantics of CAQs is best illustrated by example. Consider $Q' = (x, y, \text{AVG}(v))\pi_{x,y,v}$ $(R(x, y, z) \bowtie S(y, z, v))$. This query groups the result of $\pi_{x,y,v}(R(x, y, z) \bowtie S(y, z, v))$ by x, y, and computes $\text{AVG}(v)$ (under multiset semantics) for each group. It should be noted that we assume that the aggregate functions need to be *streamable*. This means that one should be able to update the aggregate function results by only inspecting the updates to $Q(db)$ and the previous aggregate value plus, possibly, a constant amount of extra information per tuple

We can dynamically process an acyclic CAQ Q' using DYN by means of a simple strategy: use DYN to maintain a DCLR for the acyclic NCQ Q of Q', but materialize the output of the CAQ in an array. Use delta-enumeration on Q to maintain this array under single-tuple updates. Note that, in order to support delta-enumeration with constant delay, we require that Q is free-connex (Theorem 3). If this is not the case, (which frequently occurs in practice), we let DYN maintain a DCLR for a free-connex acyclic approximation Q_F of Q. Q_F can always be obtained from Q by extending the set of output variables of Q (in the worst case by adding all variables to the output). Of course, under this strategy, we require $\Omega(\|Q'(db)\|)$ space, just like (H)IVM, but we avoid the (partial) materialization of Q and its deltas. As shown in Section 3.3.3, this property actually makes DYN outperform HIVM in both processing time and space.

An important optimization that our implementation applies in this context, is that of early computation of aggregate functions that are restricted to variables of a single atom. For example, consider $Q' = (x, y, \text{SUM}(t))\pi_{x,y,t}(R(x, y, t) \bowtie S(y, z, v))$. Our implementation will actually run DYN on $\pi_{x,y}(R'(x, y) \bowtie S(y, z, v))$ where R' is the GMR that maps tuple $(x, y) \mapsto \sum_t t \times R(x, y, t)$. Note that R' can be maintained under updates to R.

Benchmark		Query	# of tuples
TPC-H	Full joins	**FQ1**	2,833,827
		FQ2	2,617,163
		FQ3	2,820,494
		FQ4	2,270,494
	Aggregate queries	**Q1**	7,999,406
		Q3	10,199,406
		Q4	9,999,406
		Q6	7,999,406
		Q9	11,346,069
		Q12	9,999,406
		Q13	2,200,000
		Q16'	1,333,330
		Q18	10,199,406
TPC-DS	Full joins	**FQ5**	10,669,570
	Aggregate queries	**Q3**	11,638,073
		Q7	13,559,239
		Q19	11,987,115
		Q22	36,138,621

Table 3.1: Number of tuples in the stream file of each query

Sub-queries. Before proceeding to the experimental evaluation of DYN, we briefly discuss how to evaluate queries with sub-queries. Recall from Proposition 10 that T-reps have a particularly interesting property: If $\mathcal{D}$ is a T-rep and Q is compatible with T, then the multiplicity of an arbitrary tuple $\vec{t}$ in $Q(db)$ can be calculated in constant time from $\mathcal{D}$. This is highly relevant in practice, since when evaluating queries with IN or EXIST subqueries, it suffices to maintain two DCLRs, one for the subquery and one for the outer query. From the viewpoint of the outer query, the subquery DCLR then behaves as an input GMR.

3.3.2 Experimental Setup

We have implemented DYN as a query compiler that generates executable code in the Scala programming language for full NCQs, NCQs with projections and CAQs. The generated Scala code instantiates a T-rep and defines *trigger functions* that are used for maintaining the T-rep under updates. Our implementation is basic in the sense that we use off-the-shelf Scala collection libraries (notably `HashMap`) to implement the required indices. It is important to note here that faster implementations with specialized code for the index structures are certainly possible.

Queries and update streams. We evaluate the subset of queries available in the industry-standard benchmarks TPC-H and TPC-DS that can be evaluated by the methods described throughout this paper. In particular, we evaluate those queries involving only equijoins, whose FROM-WHERE clauses are acyclic. Queries are divided into acyclic full-join queries (called FQs) and acyclic aggregate queries. Acyclic full join queries are generated by taking the FROM clause of the corresponding queries on the benchmarks. It is important to mention that we omit the ORDER BY and LIMIT clauses, we replaced the left-outer join in query Q13 by an equijoin, and modified Q16 to remove an inequality. We discard those queries using the MIN and MAX aggregate functions as this is not supported by our current implementation. We report all the evaluated queries in Appendix A.1.

Our update streams consist of single-tuple insertions only and are generated as follows. We use the data-generating utilities of the benchmarks, namely *dbgen* for TPC-H and *dsdgen* for TPC-DS[6]. We used scale factor 0.5 and 2 for the FQs from TPC-H and TPC-DS, respectively, and scale factor 2 and 4 for the aggregate queries from TPC-H and TPC-DS, respectively. Notice that the data-generating tools create datasets for a fixed schema, while most queries do not use the complete set of relations. The update streams are generated by randomly selecting the tuples to be inserted from the relations that occur in each query. To use the same update streams for evaluating both DYN and HIVM, each stream is stored in a file. The number of tuples on each file is depicted in Table 3.1.

Comparison to HIVM. As discussed in the introduction, HIVM is an efficient method for dynamic query evaluation that highly improves processing time over IVM [KAK+14]. We compare our implementation against DBToaster [KAK+14], a state-of-the-art HIVM engine. DBToaster is particularly meticulous in that it materializes only useful views, and therefore it is an interesting implementation for comparison in both throughput and memory usage. Moreover, DBToaster has been extensively tested and proven to be more efficient than a commercial database management system, a commercial stream processing system and an IVM implementation [KAK+14]. DBToaster compiles SQL queries into trigger programs for different programming languages. We compare against those in Scala, the same programming language used in our implementation. It is important to mention that programs compiled by DBToaster use the so-called *akka actors*[7] to generate update tuples. During our experiments, we have found that this creates an unnecessary memory overhead by creating many temporary objects. For a fair comparison we have therefore removed these actors from DBToaster.

Operational setup. The experiments are performed on a machine running GNU/Linux with an Intel Core i7 processor running at 3.07 GHz. We use version 2.11.8 of the Scala programming language, version 1.8.0_101 of the Java Virtual Machine, and version 2.2 of the DBToaster compiler. Each query is evaluated 10 times against each of the two engines for measuring time, and two times for measuring memory; the presented results are the average measurements over those runs. Every time a query is evaluated, 16 GB of main memory are freshly allocated to the corresponding program. To measure memory usage we use the Java Virtual Machine (JVM) System calls. We measure the memory consumption every 1000 updates, and consider the maximum value. For a fair comparison, we call the garbage collector before each memory measurement. The time used by the garbage collector is not considered in the measurements of throughput.

3.3.3 Experimental results

Figure 3.5 depicts the resources used by DYN as a percentage of the resources used by DBToaster. For each query, we plot the percentage of memory used by DYN considering that 100% is to the memory used by DBToaster, and the same is done for processing time. This improves readability and normalizes the chart. To present the absolute values, on top of the bars corresponding to each query we write the memory and time used by DBToaster. Some executions of DBToaster failed because they required more than 16GB of main memory. In those cases, we report 16GB of memory and the time it took the execution to raise an exception. We mark such queries with an asterisk (*) in Figure 3.5. Note that DYN never runs out of memory, and times reported for DYN are the times required to process the entire update stream.

[6] *dbgen* and *dsgen* are available at http://www.tpc.org/

[7] http://doc.akka.io/docs/akka/snapshot/scala/actors.html

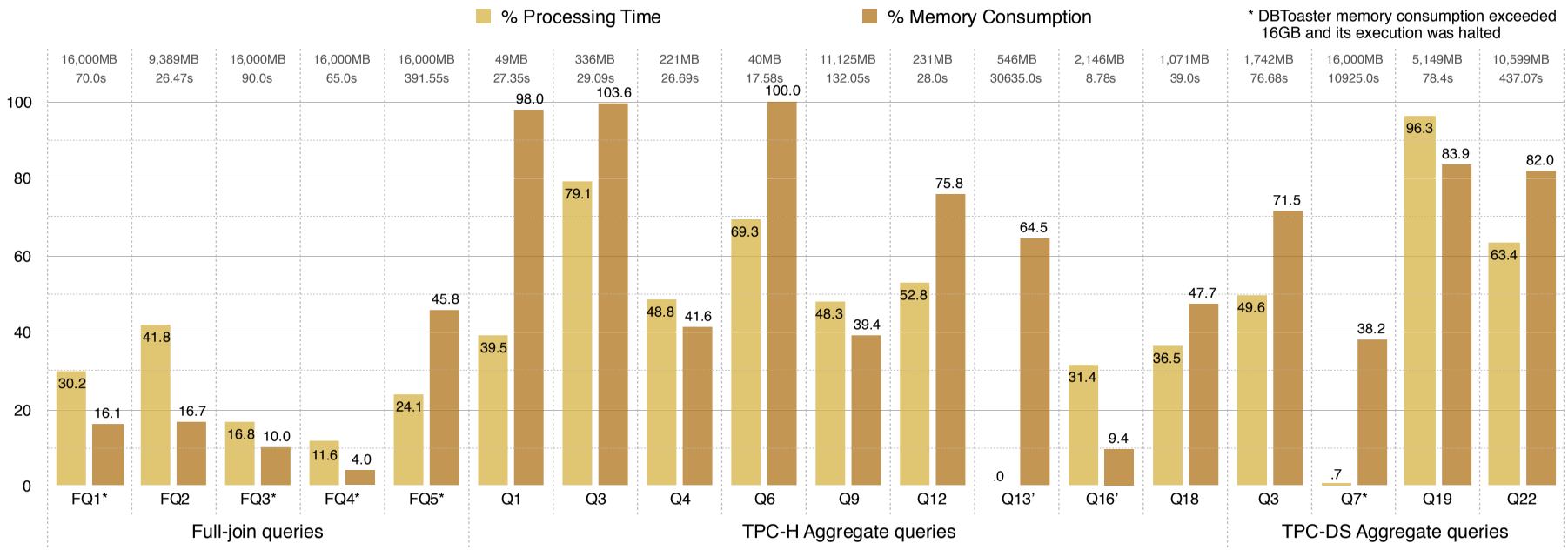


Figure 3.5: DYN usage of resources as a percentage of the resources consumed by DBToaster (lower is better).

Full-join queries. For full join queries (FQ1-FQ5), Figure 3.5 shows that DYN outperforms DBToaster by close to one order of magnitude, both in memory consumption and processing time. The difference in memory consumption is expected, since the result of full-join queries can be polynomially larger than the input dataset, and DBToaster materializes these results. The difference in processing time, then, is a consequence of DYN's maintenance of T-reps rather than the results themselves. The average processing time for DBToaster over FQ1-FQ5 is 128.49 seconds, while for DYN it is 29.85 seconds. This includes FQ1, FQ3, FQ4 and FQ5, for which DBToaster reached the memory limit. Then, 128.49 seconds is only a lower bound for the average processing time of DBToaster over FQ1-FQ5. Regarding memory consumption, DBToaster requires in average 14.68 GB for FQ1-FQ5 (considering a limit of 16 GB), compared to the 2.74 GB required by DYN. Note that the query presenting the biggest difference, FQ4, is a q-hierarchical query (see Section 3.2.4).

Aggregate queries. For aggregate queries, Figure 3.5 shows that DYN can significantly improve the memory consumption of HIVM while improving processing time up to one order of magnitude for TPC-H Q13' and TPC-DS Q7.

For TPC-H queries Q1, Q3, and Q6, DYN equals DBToasters memory consumption. For these queries, the algorithms used by DYN and DBToaster are nearly identical which is why DYN and DBToaster require the same amount of memory. The difference in execution time for these queries is due to implementation specifics. For example we have detected that DBToaster parses tuple attributes before filtering particular attributes in the WHERE clause. Our implementation, in contrast, does lazy parsing, meaning that each attribute is parsed only when it is used. In particular, if a certain attribute fails its local condition, then subsequent attributes are not parsed.

The biggest difference in processing time is observed for TPC-H query Q13' and TPC-DS query Q7. Q13' has a sub-query that computes the amount of orders processed for each customer. It then counts the number of customers for which k orders were processed, for each k. To process this, DBToaster almost fully recomputes the sub-query each time a new update arrives, which basically yields a quadratic algorithm. In contrast, our implementation uses DYN to maintain the sub-query as a T-rep, supporting, for this particular case, constant update time. For Q7, the aggressive materialization of delta queries causes DBToaster to maintain 88 different GMRs. In contrast, to maintain its T-rep, DYN only needs to store 5 GMRs and 5 indexes.

Scalability. To show that DYN performs in a consistent way against streams of different sizes, we report the processing time and memory consumption each time a 10% of the stream is processed (Figure 4.14). The results show that for all queries the memory and time increase linearly with the amount of tuples processed. We can see that DYN is constantly faster and scales more consistently. The same phenomena occur for memory consumption. We report scalability results for representative queries FQ1, Q3 and Q18.

Enumeration of query results. We know from Section 3.2.1 that T-reps feature constant delay enumeration, but this theoretical notion hides a constant factor that could decrease performance in practice. To show that this is not the case, we have measured the time needed for enumerating and writing to secondary memory the results of FQ1 to FQ4 from their corresponding DCLRs. We use update streams of different sizes, and for comparison we measure the time needed to iterate over the materialized set of results (from an in-memory array) and write them to secondary memory. The results are depicted in Figure 3.6. Interestingly, for larger result sizes, enumerating from a T-rep was slightly more efficient than enumerating from an in-memory array. A possible explanation is illustrated by the following example. Consider the full-join query $R(A, B) \bowtie S(B, C)$, and assume there are several tuples in the join result. It is not hard to see that given a fixed B value, from a T-rep we can iterate over the C values corresponding to each A value. This way, the A and B values are not re-assigned while generating several tuples. In contrast, every time a tuple is read from an array each value needs to be read again.

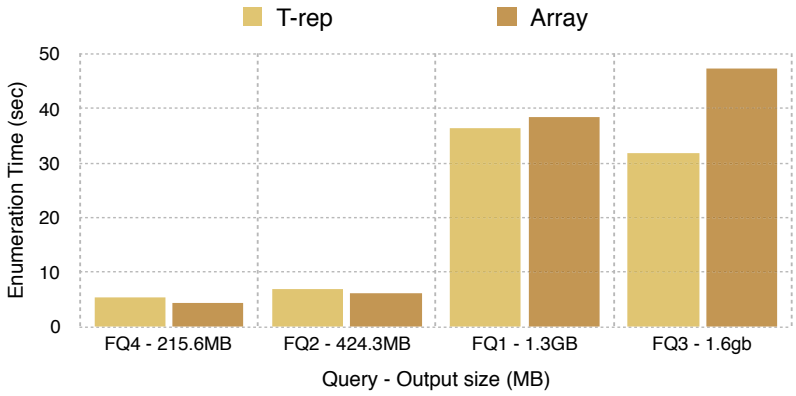


Figure 3.6: Time for enumerating output (lower is better)

Full query recomputation. In Section 3.2.3 we mentioned that, in theory, the worst-case complexity for updating a T-rep when T is not simple is the same as that of recomputing the Yannakakis algorithm from scratch. However, we can expect DYN to be much faster than the naive full-recomputation algorithm as it only updates those portions of the T-rep that are affected. This is indeed the case in practice. We tested both strategies over different datasets for FQ1 and FQ4. In average, the naive recomputation turned out to process updates 190 times slower than DYN.

3.4 Conclusion

In this chapter, we studied the dynamic evaluation of NCQs and presented the DYN algorithm. We started with defining *Generalized Join Trees* (GJTs) for NCQs, presented *Constant Delay Yannakakis* (CDY) algorithm to efficiently process joins in a static setting and support enumeration of query results with constant delay. Then we extended CDY to support update processing and presented the DYN algorithm with its complexity analysis and optimality. We showed that we can maintain a main-memory compact representation for NCQs as *T-representation*, that features the properties P_1 through P_4 of DCLRs for NCQs. Moreover, we also showed that, for the specific class of *hierarchical* queries, DYN can process updates in constant time. Finally, we compared DYN with state of the art benchmark system DBToaser - an HIVM based incremental evaluation system - on the industry benchmark datasets TPC-H and TPC-DS, and showed that DYN outperforms it by over an order of magnitude efficiency in both memory footprint and update processing. In the coming chapters, we will extend the DYN framework to the class of GCQs.

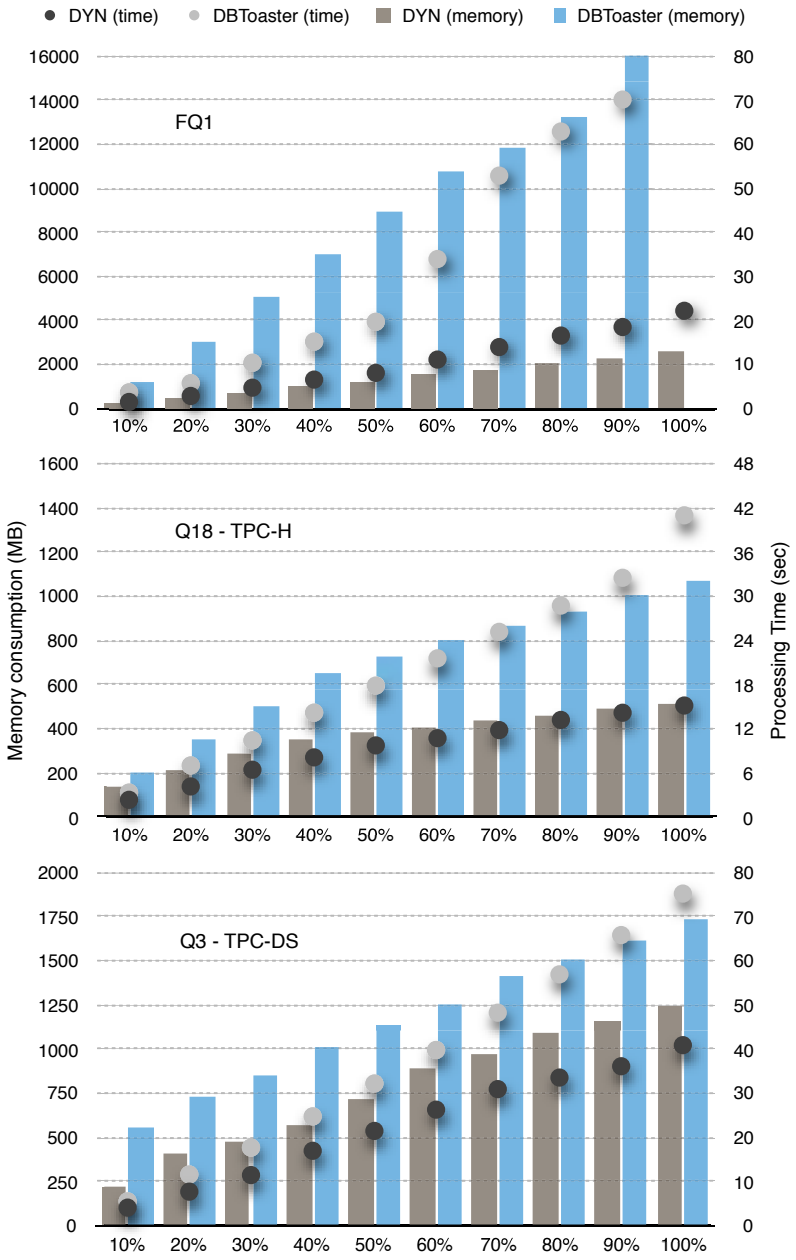


Figure 3.7: Resource utilization v/s % of tuples processed

4

GENERALIZED DYN (GDYN)

In this chapter, we extend the DYN algorithm that we developed in chapter 3 from NCQs to conjunctive queries with θ-joins. We call our solution the *Generalized* DYN (GDYN). In doing so, we extend the basic definitions, and generalize the notion of acyclicity (respectively generalized join trees (GJTs)) to the class of acyclic queries with θ-joins. For the sake of simplicity and understanding, we first present, with the help of an extensive example, how to dynamically process free-connex acyclic conjunctive queries with predicates when all the predicates are inequalities ($\leq, <, \geq, >$).Then, we present the GDYN framework which works for arbitrary predicates but is formulated at an abstract level. The remainder of this chapter is organized as follows: we first present the extended preliminaries section followed by the notion of generalized acyclicity. Then we present the extensive example to illustrate processing of queries with inequalities followed by the GDYN framework. Finally, we discuss the results of experimental evaluation at the end of the chapter.

4.1 Preliminaries

In this section, we provide a recap and extend the preliminaries for DYN to support acyclic queries with inequalities.

Traditional conjunctive queries are cross products between relations, restricted by equalities. Similarly, generalized conjunctive queries (GCQs) are cross products between relations, but restricted by arbitrary predicates. We use the following notation for queries.

GCQs. Let $x, y, z, \ldots$ be variables. A *hyperedge* is a finite set of variables denoted by $\overline{x}, \overline{y}, \ldots$. A *Generalized Conjunctive Query* (GCQ) is an expression of the form

$$Q = \pi_{\overline{y}}(r_1(\overline{x_1}) \bowtie \cdots \bowtie r_n(\overline{x_n}) \mid \bigwedge_{i=1}^{m} \theta_i(\overline{z_i})) \tag{4.1}$$

Here $r_1, \ldots, r_n$ are *relation symbols*; $\overline{x_1}, \ldots, \overline{x_n}$ are hyperedges (of the same arity as relation symbols $r_1, \ldots, r_n$); $\theta_1, \ldots, \theta_m$ are predicates over $\overline{z_1}, \ldots, \overline{z_m}$, respectively; and both $\overline{y}$ and $\bigcup_{i=1}^{m} \overline{z_i}$ are subsets of $\bigcup_{i=1}^{n} \overline{x_i}$. We treat predicates abstractly: for our purpose, a predicate over $\overline{x}$ is a (not necessarily finite) decidable set θ of tuples over $\overline{x}$. For example, $\theta(x, y) = x\theta y$ is the set of all tuples (a, b) satisfying $a\theta b$ where $\theta \in \{<, \leq, >, \geq\}$. We indicate that θ is a predicate over $\overline{x}$ by writing $\theta(\overline{x})$. Throughout the thesis, we consider only non-nullary predicates, i.e., predicates with $\overline{x} \neq \emptyset$.

R

x	y	z	$\mathbb{Z}$
1	2	2	2
2	4	6	3
1	2	3	3

S

u	v	$\mathbb{Z}$
4	5	5
2	3	4
1	4	2

T

u	v	$\mathbb{Z}$
4	5	-4
2	1	6
1	4	3

$S \bowtie T$

u	v	$\mathbb{Z}$
4	5	-20
1	4	6

$\pi_y(R)$

y	$\mathbb{Z}$
2	5
4	3

$S + T$

u	v	$\mathbb{Z}$
4	5	1
2	3	4
1	4	5
2	1	6

$S - T$

u	v	$\mathbb{Z}$
4	5	9
2	3	4
1	4	-1
2	1	-6

$R \bowtie_{y<u} S$

x	y	z	u	v	$\mathbb{Z}$
1	2	2	4	5	10
1	2	3	4	5	15

Figure 4.1: Operations on GMRs

Example 5. *The following query is a GCQ.*

$$\pi_{y,z,w,u}(r(x,y) \bowtie s(y,z,w) \bowtie t(u,v) \mid x < z \wedge w < u)$$

Intuitively, the query asks for the natural join between $r(x,y)$, $s(y,z,w)$, and $t(u,v)$, and from this result select only those tuples that satisfy both $x < z$ and $w < u$.

We call $\overline{y}$ the *output variables* of Q and denote it by $out(Q)$. If $\overline{y} = \overline{x_1} \cup \ldots \cup \overline{x_n}$ then Q is called a *full query* and we may omit the symbol $\pi_{\overline{y}}$ altogether for brevity. The elements $r_i(\overline{x_i})$ are called *atomic queries* (or *atoms*). We write $at(Q)$ for the set of all atoms in Q, and $pred(Q)$ for the set of all predicates in Q. A GCQ Q where $pred(Q) = \emptyset$ naturally evalutes to traditional join query (i.e. an NCQ) as presented in chapter 3.

Semantics. We evaluate GCQs over GMRs [KAK$^+$14, Koc10, IUV17]. The operations of GMR union ($R + S$), minus ($R - S$), projection ($\pi_{\overline{z}} R$), natural join ($R \bowtie T$) and selection ($\sigma_P(R)$) are defined similarly as in relational algebra (resp. DYN in chapter 3) with multiset semantics where P in $\sigma_P(R)$ is a non-empty set of predicates. Figure 4.1 illustrates these operations.We refer to chapter 3 Section 3.1 for a formal semantics. We abbreviate $\sigma_P(R \bowtie T)$ by $R \bowtie_P T$ and, if $\overline{x} = var(R)$, we abbreviate $\pi_{\overline{x}}(R \bowtie_P T)$ by $R \ltimes_P T$.

We adapt the approach similar to DYN for: defining updates and deltas to GMRs, defining enumeration with bounded and constant delays from DCLRs, and using the computational model for GCQs.

4.2 Generalized Acyclicity

Join queries are GCQs without projections that feature equality joins only. The well-known subclass of acyclic join queries [AHV95, Y81], in contrast to the entire class of join queries, can be evaluated in time $O(\|db\| + \|Q(db)\|)$, i.e., linear in both input and output. This result relies on the fact that acyclic join queries admit a tree structure that can be exploited during evaluation. In chapter 3, we showed that this tree structure can also be exploited for efficient processing of NCQs under updates. In this section, we therefore extend the tree structure and the notion of acyclicity from join queries to GCQs with both projections and arbitrary θ-joins. We begin by defining this tree structure and the related notion of acyclicity for full GCQs. Then, we proceed with the notion corresponding to GCQs that feature projections, known as free-connex acyclicity.

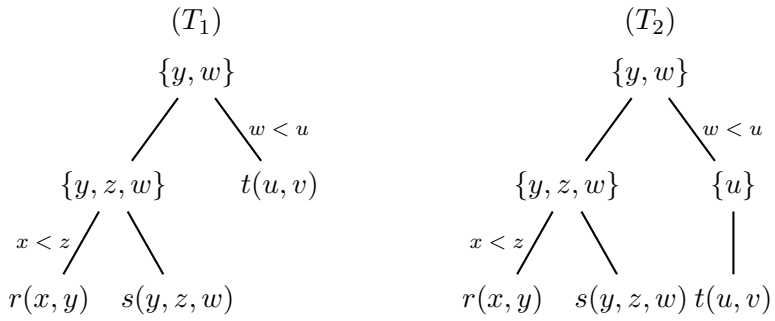


Figure 4.2: Two example GJTs.

Generalized Join Trees. To simplify notation, we denote the set of all variables (resp. atoms, resp. predicates) that occur in an object X (such as a query) by *var*(X) (resp. *at*(X), resp. *pred*(X)). In particular, if X is itself a set of variables, then *var*$(X) = X$. We extend this notion uniformly to labeled trees. E.g., if n is a node in tree T, then *var*$_T(n)$ denotes the set of variables occurring in the label of n, and similarly for edges and trees themselves. Finally, we write $\mathrm{ch}_T(n)$ for the set of children of n in tree T. If T is clear from the context, we omit subscripts from our notation. In the following, we present an extended definition of GJTs to incorporate predicates.

Definition 17 (GJT)**.** A *Generalized Join Tree* (GJT) is a node-labeled and edge-labeled directed tree $T = (V, E)$ such that:

- Every leaf is labeled by an atom.
- Every interior node n is labeled by a hyperedge and has at least one child c such that *var*$(n) \subseteq$ *var*(c).
- Whenever the same variable x occurs in the label of two nodes m and n of T, then x occurs in the label of each node on the unique path linking m and n. This condition is called the *connectedness condition*.
- Every edge $p \to c$ from parent p to child c in T is labeled by a set *pred*$(p \to c)$ of predicates. It is required that for every predicate $\theta(\overline{z}) \in$ *pred*$(p \to c)$ we have *var*$(\theta) = \overline{z} \subseteq$ *var*$(p) \cup$ *var*(c).

Let n be a node in GJT T. Every node m with *var*$(n) \subseteq$ *var*(m) is called a *guard* of n. Observe that every interior node must have a guard child by the second requirement above. Since this child must itself have a guard child, which must itself have a guard child, and so on, it holds that every interior node has at least one guard descendant that is a leaf.

Definition 18. A GJT T is a *GJT for GCQ* Q if *at*$(T) =$ *at*(Q) and the number of times that an atom occurs in Q equals the number of times that it occurs as a label in T, and *pred*$(T) =$ *pred*(Q). A GCQ Q is *acyclic* if there is a GJT for Q. It is *cyclic* otherwise.

Example 6. *The two trees depicted in Fig. 4.2 are GJTs for the following full GCQ* Q*, which is hence acyclic.*

$$Q_1 = (r(x, y) \bowtie s(y, z, w) \bowtie t(u, v) \mid x < z \wedge w < u)$$

In contrast, the query $r(x, y) \bowtie s(y, z) \bowtie t(x, z)$ *(also known as the triangle query) is the prototypical cyclic join query.*

If Q does not contain any predicates, that is, if Q is a NCQ, then the last condition of Definition 17 vacuously holds. In that case, the definition corresponds to the definition of a generalized join tree given in chapter 3, where it was also shown that a NCQ is acyclic under any of the traditional definitions of acyclicity (e.g., [AHV95]) if and only if the query has a GJT T for Q with $\mathit{pred}(T) = \emptyset$. In this sense, Definition 18 indeed generalizes acyclicity from NCQs to GCQs.

Discussion. The notion of acyclicity for normal NCQs is well-studied in database theory [AHV95] and has many equivalent definitions, including a definition based on the existence of a *full reducer*. Here, a full reducer for a NCQ Q is a program $\mathcal{S}$ in the semijoin algebra (the variant of relational algebra where joins are replaced by semijoins) that, given a database *db* computes a new database $\mathcal{S}(db)$ with the following properties. (1) $Q(\mathcal{S}(db)) = Q(db)$; (2) $\mathcal{S}(db)_{r(\overline{x})} \subseteq db_{r(\overline{x})}$ for every atom $r(\overline{x})$; and (3) no strict subset of $\mathcal{S}(db)$ has $Q(\mathcal{S}(db)) = Q(db)$. In other words, $\mathcal{S}$ selects a *minimal* subset of *db* needed to answer Q.

Bernstein and Goodman [BG81] consider conjunctive queries with inequalities and classify the class of such queries that admit full reducers. As such, one can view this as a definition of acyclicity for conjunctive queries with inequalities. Bernstein and Goodman's notion of acyclicity is incomparable to ours. On the on hand, our definition is more general: Bernstein and Goodman consider only queries where for each pair of atoms there is exactly one variable being compared by means of equality or inequality. We, in contrast, allow an arbitrary number of variables to be compared per pair of atoms. In particular, Bernstein and Goodman's disallow queries like $(r(x, y), s(x, z) \mid y < z)$ since it compares $r.x$ with $s.x$ by means of equality and $r.y < s.z$ by means of inequality, while this is trivially acyclic in our setting.

On the other hand, for this more restricted class of queries, Bernstein and Goodman show that certain queries that we consider to be cyclic have full reducers (and would be hence acyclic under their notion). An example here is

$$r(x_r) \bowtie s(x_s, y_s) \bowtie t(x_t, y_t) \bowtie u(y_u) \mid x_s \leq x_r \wedge x_t \leq x_r \wedge y_s \leq y_u \wedge y_t \leq y_u$$

The crucial reason that this query admits a full reducer is due to the transitivity of $\leq$. Since our notion of acyclicity interprets predicates abstractly and does hence not assume properties such as transitivity on them, we must declare this query cyclic (as can be checked by running the join tree algorithm of chapter 5 on it). It is an interesting direction for future work to incorporate Bernstein and Goodman's notion of acyclicity in our framework.

Free-connex acyclicity. Acyclicity is actually a notion for full GCQs. Indeed, note that whether or not Q is acyclic does not depend on the projections of Q (if any). To also process queries with projections efficiently, we will need, similar to chapter 3, a related structural property known as free-connex acyclicity. In the following, we present the corresponding definitions of connexity for GCQs.

Definition 19 (Connex, Frontier)**.** Let $T = (V, E)$ be a GJT. A *connex subset* of T is a set $N \subseteq V$ that includes the root of T such that the subgraph of T induced by N is a tree. The *frontier* of a connex set N is the subset $F \subseteq N$ consisting of those nodes in N that are leaves in the subtree of T induced by N.

To illustrate, the set $\{\{y, w\}, \{u\}, \{y, z, w\}\}$ is a connex subset of the tree T_2 shown in Fig. 4.2. Its frontier is $\{\{y, z, w\}, \{u\}\}$. In contrast, $\{\{y, w\}, \{y, z, w\}, t(u, v)\}$ is not a connex subset of T_2.

Definition 20 (Compatible, Free-Connex Acyclic)**.** A *GJT pair* is a pair (T, N) with T a GJT and N a connex subset of T. A GCQ Q is *compatible with* (T, N) if T is a GJT for Q and $\mathit{var}(N) = \mathit{out}(Q)$. A GCQ is *free-connex acyclic* if it has a compatible GJT pair.

In particular, every full acyclic GCQ is free-connex acyclic since the entire set of nodes V of a GJT T for Q is a connex set with $\mathit{var}(V) = \mathit{out}(Q)$. Therefore, (T, V) is a compatible GJT pair for Q.

Example 7. *Let $Q_2 = \pi_{y,z,w,u}(Q_1)$ with Q_1 the GCQ from Example 6. Q_2 is free-connex acyclic since it is compatible with the pair $(T_2, \{\{y, w\}, \{y, z, w\}, \{u\}\})$ with T_2 the GJT from Fig. 4.2. By contrast, Q_2 is not compatible with any GJT pair containing T_1, since any connex set of T_1 that includes a node with variable u will also include variable v, which is not in $\mathit{out}(Q_2)$. Finally, it can be verified that no GJT pair is compatible with $\pi_{x,u}(Q_1)$; this query is hence not free-connex acyclic.*

In chapter 5 we show how to efficiently check free-connex acyclicity and compute compatible GJT pairs.

Binary GJTs and sibling-closed connex sets. As we will see in Sections 4.3 and 4.4, a GJT pair (T, N) essentially acts as query plan by which GDYN and IEDYN (a specialized variant of GDYN presented in the following section 4.3) process queries dynamically. In particular, the GJT T specifies the data structure to be maintained and drives the processing of updates, while the connex set N drives the enumeration of query results.

In order to simplify the presentation of what follows, we will focus exclusively on the class of GJT pairs (T, N) with T a binary GJT and N sibling-closed.

Definition 21 (Binary, Sibling-closed)**.** A GJT T is binary if every node in it has at most two children. A connex subset N of T is *sibling-closed* if for every node $n \in N$ with a sibling m in T, m is also in N.

Our interest in limiting to sibling-closed connex sets is due to the following property, which will prove useful for enumerating query results, as explained in Section 4.3.

Lemma 9. *If N is a sibling-closed connex subset, then $\mathit{var}(N) = \mathit{var}(F)$ where F is the frontier of N.*

Proof. Since $F \subseteq N$ the inclusion $\mathit{var}(F) \subseteq \mathit{var}(N)$ is immediate. It remains to prove $\mathit{var}(N) \subseteq \mathit{var}(F)$. To this end, let n be an arbitrary but fixed node in N. We prove that $\mathit{var}(n) \subseteq \mathit{var}(F)$ by induction on the *height* of n in N, which is defined as the length of the shortest path from n to a frontier node in F. The base case is where the height is zero, i.e., $n \in F$, in which case $\mathit{var}(n) \subseteq \mathit{var}(F)$ trivially holds. For the induction step, assume that the height of n is $k > 0$. In particular, n is not a frontier node, and has at least one child in N. Because N is sibling-closed, all children of n are in N. In particular, the guard child m of n is in N and has height at most $k - 1$. By induction hypothesis, $\mathit{var}(m) \subseteq \mathit{var}(F)$. Then, because m is a guard of n, $\mathit{var}(n) \subseteq \mathit{var}(m) \subseteq \mathit{var}(F)$, as desired. □

Let us call a GJT pair (T, N) *binary* if T is binary, and *sibling-closed* if N is sibling-closed. We say that two GJT pairs (T, N) and (T', N') are *equivalent* if T and T' are equivalent and $\mathit{var}(N) = \mathit{var}(N')$. Two GJTs T and T' are equivalent if $\mathit{at}(T) = \mathit{at}(T')$, the number of times that an atom appears as a label in T equals the number of times that it appears in T', and $\mathit{pred}(T) = \mathit{pred}(T')$.

The following proposition shows that we can always convert an arbitrary GJT pair into an equivalent one that is binary and sibling-closed. As such, we are assured that our focus on binary and sibling-closed GJT pairs is without loss of generality.

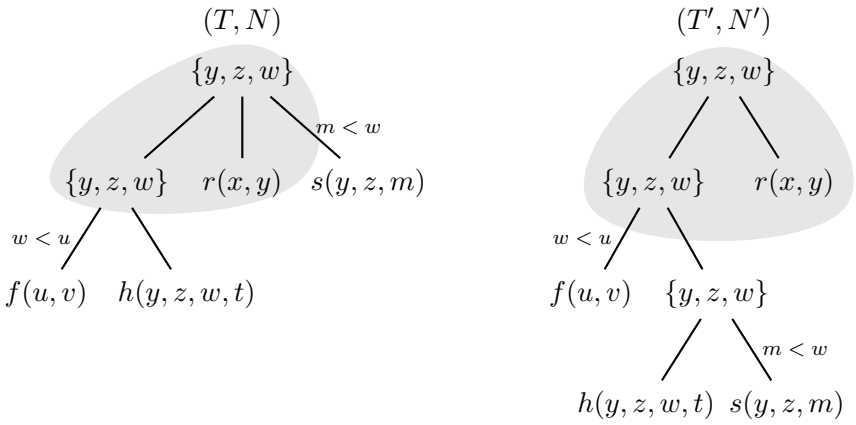


Figure 4.3: Illustration of the sibling-closed transform: removal of type-1 violator. The connex sets N and N' are indicated by the shaded areas.

Proposition 16. *Every GJT pair can be transformed in polynomial time into an equivalent pair that is binary and sibling closed.*

The rest of this section is devoted to proving Proposition 16. We do so in two steps. First, we show that any pair (T, N) can be transformed in polynomial time into an equivalent sibling-closed pair. Next, we show that any sibling-closed GJT pair (T, N) can be converted in polynomial time into an equivalent binary and sibling-closed pair. Proposition 16 hence follows by composing these two transformations.

Sibling-closed transformation. We say that $n \in T$ is a *violator* node in a GJT pair (T, N) if $n \in N$ and some, but not all children of n are in N. A violator is of *type 1* if some node in $\mathrm{ch}(n) \cap N$ is a guard of n. It is of *type 2* otherwise. We now define two operations on (T, N) that remove violators of type 1 and type 2, respectively. The sibling-closed transformation is then obtained by repeatedly applying these operators until all violators are removed.

The first operator is applicable when n is a type 1 violator. It returns the pair (T', N') obtained as follows:

- Since n is a type 1 violator, some $g \in \mathrm{ch}_T(n) \cap N$ is a child guard of n (i.e., $\mathit{var}(n) \subseteq \mathit{var}(g)$).
- Because every node has a guard, there is some leaf node l that is a descendant guard of g (i.e. $\mathit{var}(g) \subseteq \mathit{var}(l)$). Possibly, l is g itself.
- Now create a new node p between node l and its parent with label $\mathit{var}(p) = \mathit{var}(l)$. Since l is a descendant guard of n and g, p becomes a descendant guard of n and g as well. Detach all nodes in $\mathrm{ch}(n) \setminus N$ from n and attach them as children to p, preserving their edge labels. This effectively moves all subtrees rooted at nodes in $\mathrm{ch}(n) \setminus N$ from n to p. Denote by T' the final result.
- If l was not in N, then $N' = N$. Otherwise, $N' = N \setminus \{l\} \cup \{p\}$.

We write $(T, N) \xrightarrow{1,n} (T', N')$ to indicate that (T', N') can be obtained by applying the above-described operation on node n.

Example 8. *Consider the GJT pair (T, N) from Fig. 4.3 where N is indicated by the nodes in the shaded area. Let us denote the root node by n and its guard child with label $\{y, z, w\}$ by g. The node $l = h(y, z, w, t)$ is a descendant guard of g. Since $s(y, z, m)$ is not in N, n is violator of type 1. After applying the operation 1 for the choice of guard node g and descendant guard node l, (T', N') shows the resulting valid sibling-closed GJT.*

Lemma 10. *Let n be a violator of type 1 in (T, N) and assume $(T, N) \xrightarrow{1,n} (T', N')$. Then (T', N') is a GJT pair and it is equivalent to (T, N). Moreover, the number of violators in (T', N') is strictly smaller than the number of violators in (T, N).*

Proof. The lemma follows from the following observations. (1) It is straightforward to observe that T' is a valid GJT: the construction has left the set of leaf nodes untouched; took care to ensure that all nodes (including the newly added node p) continue to have a guard child; ensures that the connectedness condition continues to hold also for the relocated children of n because every variable in n is present on the entire path between n and p; and have ensured that also edge labels remain valid (for the relocated nodes this is because $\mathit{var}(p) = \mathit{var}(g) \subseteq \mathit{var}(n)$).

(2) N' is a connex subset of T' because the subtree of T induced by N equals to subtree of T' induced by N', modulo the replacement of l by p in case that l was in N and p is hence in N'.

(3) (T, N) is equivalent to (T', N') because the construction leaves leaf atoms untouched, preserves edge labels, and $\mathit{var}(N) = \mathit{var}(N')$. The latter is clear if $l \notin N$ because then $N = N'$. It follows from the fact that $\mathit{var}(l) = \mathit{var}(p)$ if $l \in N$, in which case $N' = N \setminus \{l\} \cup \{p\}$.

(4) All nodes in $\mathrm{ch}_T(n) \setminus N$ (and their descendants) are relocated to p in T'. Therefore, n is no longer a violator in (T', N'). Because we do not introduce new violators, the number of violators of (T', N') is strictly smaller than the number of violators of (T, N). □

This concludes the proof of lemma 10.

The second operator is applicable when n is a type 2 violator. When applied to n in (T, N) it returns the pair (T', N') obtained as follows:

- Since n is a type 2 violator, no node in $\mathrm{ch}_T(n) \cap N$ is a guard of n. Since every node has a guard, there is some $g \in \mathrm{ch}(n) \setminus N$ which is a guard of n.
- Create a new child p of n with label $\mathit{var}(p) = \mathit{var}(n)$; detach all nodes in $\mathrm{ch}(n) \setminus N$ (including g) from N, and add them as children of p, preserving their edge labels. This moves all subtrees rooted at nodes in $\mathrm{ch}(n) \setminus N$ from n to p. Denote by T' the final result.
- Set $N' = N \cup \{p\}$.

We write $(T, N) \xrightarrow{2,n} (T', N')$ to indicate that (T', N') was obtained by applying this operation on n.

Example 9. *Consider the GJT pair (T, N) in Fig. 4.4. Let us denote the root node by n. Since its guard child $h(y, z, w, t)$ is not in N, n is violator of type 2. After applying operation 2 on n, (T', N') shows the resulting valid sibling-closed GJT.*

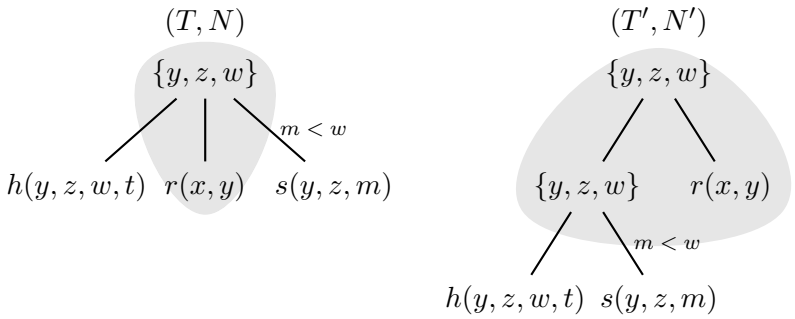


Figure 4.4: Illustration of the sibling-closed transform: removal of type-2 violator. The connex sets N and N' are indicated by the shaded areas.

Lemma 11. *Let n be a violator of type 2 in (T, N) and assume $(T, N) \xrightarrow{2,n} (T', N')$. Then (T', N') is a GJT pair and it is equivalent to (T, N). Moreover, the number of violators in (T', N') is strictly smaller than the number of violators in (T, N).*

Proof. The lemma follows from the following observations. (1) It is straightforward to observe that T' is a valid GJT: the construction has left the set of leaf nodes untouched; took care to ensure that all nodes (including the newly added node p) continue to have a guard child; ensures that the connectedness condition continues to hold also for the relocated children of n because every variable in n is also present in p, their new parent; and have ensured that also edge labels remain valid (for the relocated nodes this is because $\mathit{var}(p) = \mathit{var}(n)$).

(2) N' is a connex subset of T' because (i) the subtree of T induced by N equals to subtree of T' induced by N' $\{p\}$, (ii) $n \in N$, and (iii) p is a child of n in T'. Therefore, N' must be connex.

(3) (T, N) is equivalent to (T', N') because the construction leaves leaf atoms untouched, preserves edge labels, and $\mathit{var}(N) = \mathit{var}(N')$. The latter follows because $\mathit{var}(N') = \mathit{var}(N \cup \{p\})$ and because $\mathit{var}(p) = \mathit{var}(n) \subseteq \mathit{var}(N)$ since $n \in N$.

(4) All nodes in $\mathrm{ch}_T(n) \setminus N$ (and their descendants) are relocated to p in T'. Therefore, n is no longer a violator in (T', N'). Because we do not introduce new violators, the number of violators of (T', N') is strictly smaller than the number of violators of (T, N). □

This concludes the proof of lemma 11. Next we present the sibling-closed transformation proposition.

Proposition 17. *Every GJT pair can be transformed in polynomial time into an equivalent sibling-closed pair.*

Proof. The two operations introduced above remove violators, one at a time. By repeatedly applying these operations until no violator remains we obtain an equivalent pair without violators, which must hence be sibling-closed. Since each operator can clearly be executed in polynomial time and the number of times that we must apply an operator is bounded by the number of nodes in the GJT pair, the removal takes polynomial time. □

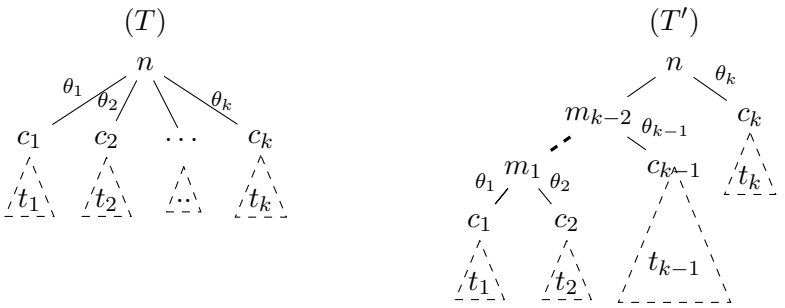


Figure 4.5: Binarizing a k-ary node n.

Binary transformation. Now we show how to transform a sibling-closed pair (T, N) into an equivalent binary and sibling-closed pair (T', N'). The idea here is to "binarize" each node n with $k > 2$ children as shown in Fig. 4.5. There, we assume without loss of generality that c_1 is a guard child of n. The binarization introduces $k-2$ new intermediate nodes $m_1, \ldots, m_{k-2}$, all with $\mathit{var}(m_i) = \mathit{var}(n)$. Note that, since c_1 is a guard of n and $\mathit{var}(m_i) = \mathit{var}(n)$, it is straightforward to see that c_1 will be a guard of m_1, which will be a guard of m_2, which will be a guard of m_3, and so on. Finally, m_{k-2} will be a guard of n. The connex set N is updated as follows. If none of n's children are in N i.e. n is a frontier node, set $N' = N$. Otherwise, since N is sibling-closed, all children of n are in N, and we set $N' = N \cup \{m_1, \ldots, m_{k-2}\}$. Clearly, N' remains a sibling-closed connex subset of T' and $\mathit{var}(N') = \mathit{var}(N)$. We may hence conclude:

Lemma 12. *By binarizing a single node in a sibling-closed GJT pair (T, N) as shown in Fig. 4.5, we obtain an equivalent GJT pair (T', N') that has strictly fewer non-binary nodes than (T, N).*

Binarizing a single node is a polynomial-time operation. Then, by iteratively binarizing non-binary nodes until all nodes have become binary we hence obtain:

Proposition 16. *Every GJT pair can be transformed in polynomial time into an equivalent pair that is binary and sibling closed.*

4.3 Dynamic Joins with Equalities and Inequalities: An Example

In this section we illustrate how to dynamically process free-connex acyclic GCQs when all predicates are inequalities ($\leq, <, \geq, >$). We do so by means of an extensive example that shows the indexing structures and GMRs. The definitions and algorithms (that apply to arbitrary θ-joins) will be formally presented in Section 4.4.

Throughout this section we consider the following query Q, which is free-connex acyclic (see Example 7):

$$\pi_{y,z,w,u}(r(x,y) \bowtie s(y,z,w) \bowtie t(u,v) \mid x < z \wedge w < u).$$

Let T_2 be the GJT from Fig. 4.2. We process Q based on a *T_2-reduct*, a data structure that succinctly represents the output of Q. For every node n, define $\mathit{pred}(n)$ as the set of all predicates on outgoing edges of n, i.e. $\mathit{pred}(n) = \bigcup_{c \text{ child of } n} \mathit{pred}(n \to c)$.

Definition 22 (T-reduct)**.** Let T be a GJT for a query Q and let *db* be a database over $\mathit{at}(Q)$. The T-reduct (or *semi-join reduction*) of *db* is a collection ρ of GMRs, one GMR ρ_n for each node $n \in T$, defined inductively as follows:

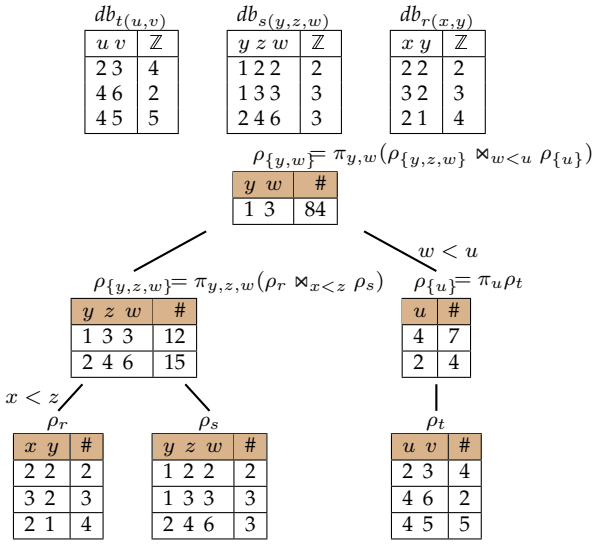


Figure 4.6: Example database and its T_2-reduct.

- if $n = r(x)$ is an atom, then $\rho_n = db_{r(x)}$
- if n has a single child c, then $\rho_n = \pi_{var(n)} \sigma_{pred(n)} \rho_c$
- otherwise, n has two children c_1 and c_2. In this case we have $\rho_n = \pi_{var(n)} \left(\rho_{c_1} \bowtie_{pred(n)} \rho_{c_2} \right)$.

Fig. 4.6 depicts an example database (top) and its T_2-reduct ρ (bottom). Note, for example, that the only tuple in the GMR at the root $\rho_{\{y,w\}}$ is the join of $\rho_{\{y,z,w\}}$ and $\rho_{\{u\}}$ restricted to $w < y$ and projected over $\{y, w\}$.

It is important to observe that the size of a T-reduct of a database *db* can be at most linear in the size of *db* (as proven in chapter 3). The reason is that, as illustrated in Fig. 4.6, for each node n there is some descendant atom α (possibly n itself) such that $\mathrm{supp}(\rho_n) \subseteq \mathrm{supp}(\pi_{var(n)}\, db_\alpha)$. Note that $Q(db)$, in contrast, can easily become polynomial in the size of *db* in the worst case.

Enumeration. From a T-reduct we can enumerate the result $Q(db)$ rather naively simply by recomputing the query results, in particular because we have access to the complete database in the leaves of T. We would like, however, to make the enumeration as efficient as possible. To this end, we equip T-reducts with a set of indices. To avoid the space cost of materialization, we do not want the indices to use more space than the T-reduct itself (i.e., linear in *db*). We illustrate these ideas in our running example by introducing a simple set of indices that allow for efficient enumeration.

Let $N = \{\{y, w\}, \{y, z, w\}, \{u\}\}$ be the connex subset of T_2 satisfying $var(N) = out(Q) = \{y, z, w, u\}$. (T_2, N) is compatible with Q, binary and sibling-closed. We rely on the sibling-closed property of N to enumerate query results, and can do so without loss of generality by Proposition 16. To enumerate the query results, we will traverse top-down the nodes in N. The traversal works as follows: for each tuple $\vec{t_1}$ in $\rho_{\{y,w\}}$, we consider all tuples $\vec{t_2}$ in $\rho_{\{y,z,w\}}$ that are compatible with $\vec{t_1}$, and all tuples $\vec{t_3} \in \rho_{\{u\}}$ that are compatible with $\vec{t_1}$. Compatibility here means that the corresponding equalities and inequalities are satisfied. Then, for each pair $(\vec{t_2}, \vec{t_3})$, we output the tuple $\vec{t_2} \cup \vec{t_3}$ with multiplicity $\rho_{\{y,z,w\}}(\vec{t_2}) \times \rho_{\{u\}}(\vec{t_3})$. A crucial difference here with naive recomputation is that, since $\rho_{\{y,w\}}$ is already a join between $\rho_{\{y,z,w\}}$ and $\rho_{\{u\}}$, we will only iterate over *relevant* tuples:

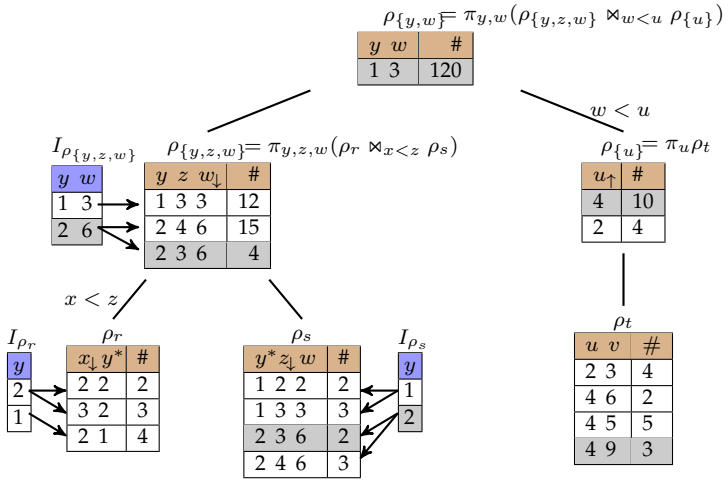


Figure 4.7: T_2-rep of $db + u$ with T_1 and db as in Fig. 4.6 and u the update containing $(\langle y\colon 2, z\colon 5, w\colon 6\rangle, 2)$ and $(\langle u\colon 4, v\colon 9\rangle, 3)$.

each tuple that we iterate over will produce a new output tuple. For example, we will never look at the tuple $\langle y : 2, z : 4, w : 6\rangle$ in $\rho_{\{y,z,w\}}$ because it does not have a compatible tuple at the root.

To implement this enumeration strategy efficiently, we desire index structures on $\rho_{\{y,z,w\}}$ and $\rho_{\{u\}}$ that allow to enumerate, for a given tuple $\vec{t_1}$ in $\rho_{\{y,w\}}$, all compatible tuples $\vec{t_2} \in \rho_{\{y,z,w\}}$ (resp. $\vec{t_3} \in \rho_{\{u\}}$) with constant delay. In the case of $\rho_{\{u\}}$ this is achieved simply by keeping $\rho_{\{u\}}$ sorted decreasingly on variable u. Given tuple $\vec{t_1}$, we can enumerate the compatible tuples from $\rho_{\{u\}}$ by iterating over its tuples one by one in a decreasing manner, starting from the largest value of u, and stopping whenever the current u value is smaller or equal than the w value in $\vec{t_1}$. For indexing $\rho_{\{y,z,w\}}$ we use a more standard index. Since we need to enumerate all tuples that have the same y and w value as $\vec{t_1}$, CDE can be achieved by using a hash-based index on y and w. This index is depicted as $I_{\rho_{\{y,z,w\}}}$ in Fig. 4.7. We can see that, since the described indices provide CDE of the compatible tuples given $\vec{t_1}$, our strategy provides enumeration of $Q(db)$ with constant delay if we assume the query to be fixed (i.e. in data complexity [Var82]).

Updates. Next we illustrate how to process updates. The objective here is to transform the T_2-reduct of db into a T_2-reduct of $db + u$, where u is the received update. To do this efficiently we use additional indexes on ρ. We present the intuitions behind these indices with an update consisting of two insertions: $\langle y\colon 2, z\colon 3, w\colon 6\rangle$ with multiplicity 2 and $\langle u\colon 4, v\colon 9\rangle$ with multiplicity 3. Fig. 4.7 depicts the update process highlighting the modifications caused by the update.

Let us first discuss how to process the tuple $\vec{t_1} = \langle y\colon 2, z\colon 3, w\colon 6\rangle$. We proceed bottom-up, starting at ρ_s which is itself affected by the insertion of $\vec{t_1}$. Subsequently, we need to *propagate* the modification of ρ_s to its ancestors $\rho_{\{y,z,w\}}$ and $\rho_{\{y,w\}}$. Concretely, from the definition of T-reduction, it follows that we need to add some modifications to ρ_s, $\rho_{\{y,z,w\}}$, and $\rho_{\{y,w\}}$ on $\vec{t_1}$:

$$\Delta\rho_s = [\vec{t_1} \mapsto 2],$$
$$\Delta\rho_{\{y,z,w\}} = \pi_{y,z,w}\,(\rho_r \ltimes_{x<z} \Delta\rho_s),$$
$$\Delta\rho_{\{y,w\}} = \pi_{y,w}\left(\Delta\rho_{\{y,z,w\}} \ltimes_{w<u} \rho_{\{u\}}\right).$$

To compute the joins on the right-hand sides efficiently, we create a number of additional indexes on ρ_r, ρ_s, and $\rho_{\{y,z,w\}}$. Concretely, in order to efficiently compute the join $\pi_{y,z,w}(\rho_r \ltimes_{x<z} \Delta\rho_s)$, we *group* tuples in the GMR ρ_r by the variables that ρ_r has in common with ρ_s (in this case y) and then, per group, *sort* tuples ascending on variable x. We mark grouping variables in Fig. 4.7 with $*$ (e.g. y^*), and sorting by $\downarrow$ (for ascending, e.g., $x_\downarrow$) and $\uparrow$ (for descending). A hash index on the grouping variables (denoted I_{ρ_r} in Fig. 4.7) then allows to find the group given a y value. The join can then be processed by means of a hybrid form of sort-merge and index nested loop join. Sort $\Delta\rho_s$ ascendingly on y and z. For each y-group in $\Delta\rho_s$ find the corresponding group in ρ_r by passing the y value to the index I_{ρ_r}. Let $\vec{t'}$ be the first tuple in the $\Delta\rho_s$ group. Then iterate over the tuples of the ρ_r group in the given order and sum up their multiplicities until x becomes larger than $\vec{t'}(z)$. Add $\vec{t'}$ to the result with its original multiplicity multiplied by the found sum (provided it is non-zero). Then consider the next tuple in the $\Delta\rho_s$ group, and continue summing from the current tuple in the ρ_r group until x becomes again larger than z, and add the result tuple with the correct multiplicity. Continue repeating this process for each tuple in the $\Delta\rho_s$ group, and for each group in $\Delta\rho_s$. In our case, there is only one group in $\Delta\rho_s$ (given by $y = 2$) and we will only iterate over the tuple $\langle x\colon 2, y\colon 2\rangle$ in ρ_r, obtaining a total multiplicity of 2, and therefore compute $\Delta\rho_{\{y,z,w\}} = [\vec{t_1} \to 4]$. In order to compute the join $\pi_{y,w}\left(\Delta\rho_{\{y,z,w\}} \ltimes_{w<u} \rho_{\{u\}}\right)$ efficiently, we proceed similarly. Here, however, there are no grouping variables on $\rho_{\{u\}}$ and it hence suffices to sort $\rho_{\{u\}}$ descendingly on u. Note that this was actually already required for efficient enumeration. Also note that $\Delta\rho_{\{y,w\}}$ is empty.

Now we discuss how to process $\vec{t_2} = \langle u\colon 4, v\colon 9\rangle$. First, we insert $\vec{t_2}$ into ρ_t. We need to propagate this change to the parent $\rho_{\{u\}}$ by calculating $\Delta\rho_{\{u\}} = \pi_u \Delta\rho_t$. This is done by a simple hash-based aggregation. Finally, we need to propagate $\Delta\rho_{\{u\}}$ to the root by computing $\Delta\rho_{\{y,w\}} = \pi_{y,w}(\rho_{\{y,z,w\}} \ltimes_{w<u} \Delta\rho_{\{u\}})$. To process this join efficiently we proceed as before. Again, there are no grouping variable on $\rho_{\{y,z,w\}}$ (since it has no variables in common with $\rho_{\{u\}}$) and it hence suffices to sort $\rho_{\{y,z,w,\}}$ ascending on w. The only tuple that we iterate over during the hybrid join is $\langle y\colon 1, z\colon 3, w\colon 3\rangle$ wich has multiplicity 12. Hence, we have $\Delta\rho_{y,w} = [\langle y\colon 1, w\colon 3\rangle \mapsto 36\ ]$, concluding the example.

4.4 Dynamic Yannakakis over GCQs

In this section we present a generalization of DYN, called GDYN, to dynamically process free-connex acyclic GCQs. Since predicates in a GCQ can be arbitrary, our approach is purely algorithmic; the efficiency by which GDYN process updates and produces results will depend entirely on the efficiency of the underlying data structures. Here we only describe the properties that those data structures should satisfy and present the general (worst-case) complexity of the algorithm. Moreover, GDYN degrades gracefully i.e. when there is no predicate in a GCQ Q then GDYN evaluates to DYN and hence the evaluation of a normal conjunctive query. The techniques and indices presented in the previous section 4.3 provide a practical instantiation of GDYN to a GCQ with equalities and inequalities, and throughout this section we make a parallel between that instantiation and the more abstract definitions of GDYN.

In this section we assume that Q is a free-connex acyclic GCQ and that (T, N) is a binary and sibling-closed GJT pair compatible with Q. Like in the case of equalities and inequalities, the dynamic processing of Q will be based on a T-reduct of the current database *db*. A set of indices will be added to optimize the enumeration of query results and maintenance of the T-reduct under updates. We formalize the notion of index as follows:

Algorithm 6 Enumerate $Q(db)$ given T-reduct ρ of *db*.

```
1: function ENUM_{T,N}(ρ)
2:   for each t⃗ ∈ ρ_{root(T)} do ENUM_{T,N}(root(T), t⃗, ρ)

3: function ENUM_{T,N}(n, t⃗, ρ)
4:   if n is in the frontier of N then yield (t⃗, ρ_n(t⃗))
5:   else if n has one child c then
6:     for each s⃗ ∈ ρ_c ⋉_{pred(n→c)} t⃗ do ENUM_{T,N}(c, s⃗, ρ)
7:   else n has two children c_1 and c_2
8:     for each t⃗_1 ∈ ρ_{c_1} ⋉_{pred(n→c_1)} t⃗ do
9:       for each t⃗_2 ∈ ρ_{c_2} ⋉_{pred(n→c_2)} t⃗ do
10:        for each (s⃗_1, μ) ∈ ENUM_{T,N}(c_1, t⃗_1, ρ) do
11:          for each (s⃗_2, ν) ∈ ENUM_{T,N}(c_2, t⃗_2, ρ) do
12:            yield (s⃗_1 ∪ s⃗_2, μ × ν)
```

Definition 23 (Index)**.** Let R be a GMR over $\overline{x}$, let $\overline{y}$ be a hyperedge, let $\overline{w}$ be a hyperedge satisfying $\overline{w} \subseteq \overline{x} \cup \overline{y}$, and let $\theta(\overline{z})$ be a predicate with $\overline{z} \subseteq \overline{x} \cup \overline{y}$. An *index* on R by $(\theta, \overline{y}, \overline{w})$ with delay f is a data structure I that provides, for any given GMR $R_{\overline{y}}$ over $\overline{y}$, enumeration of $\pi_{\overline{w}}(R \bowtie_\theta R_{\overline{y}})$ with delay $O(f(|R|+|R_{\overline{y}}|))$. The update time of index I is the time required to update I to an index on $R + \Delta R$ (by $(\theta, \overline{y}, \overline{w})$) given update ΔR to R.

For example, I_{ρ_r} in Fig. 4.7 is used as an index on ρ_r by $(x < z, \{y, z, w\}, \{y, z, w\})$. Indeed, in the previous section we precisely discussed how I_{ρ_r} allows to efficiently compute $\pi_{y,z,w}(\rho_r \bowtie_{x<z} \Delta\rho_s)$ for an update $\Delta\rho_s$ to ρ_s. Having the notion of index, we discuss how GDYN enumerates query results and processes updates.

Enumeration. Let *db* be the current database. To enumerate $Q(db)$ from a T-reduct ρ of *db* we can iterate over the reductions ρ_n with $n \in N$ in a nested fashion, starting at the root and proceeding top-down. When n is the root, we iterate over all tuples in ρ_n. For every such tuple $\vec{t}$, we iterate only over the tuples in the children c of n that are compatible with $\vec{t}$ (i.e., tuples in ρ_c that join with $\vec{t}$ and satisfy $pred(n \to c)$). This procedure continues until we reach nodes in the frontier of N at which time the output tuple can be constructed. The pseudocode is given in Algorithm 6, where the tuples that are compatible with $\vec{t}$ are computed by $\rho_c \ltimes_{pred(n\to c)} \vec{t}$.

Now we show the correctness of the enumeration algorithm, for which we need to introduce some further notation. Let Q, T and N be as above. Given a node $n \in T$ we denote the sub-tree of T rooted at n by T_n, and define the query induced by T_n as

$$Q_n = (\bowtie_{r(\overline{x}) \in at(T_n)} r(\overline{x}) \mid pred(T_n))$$

where $at(T_n)$ and $pred(T_n)$ are the sets of all atoms and predicates occurring in T_n, respectively.

Lemma 13. *Let Q, T, N, and n be defined as above, and let ρ be a T-reduct for Q. Then, $\rho_n = \pi_{var(n)} Q_n(db)$.*

Proof. We proceed by induction on the number of descendants of n. If n has no descendant then Q_n is a single atom $r(\overline{x})$, so we have $\overline{x} = out(Q_n) = var(n)$. Then $\pi_{var(n)} Q_n(db) = Q_n(db) = db_{r(\overline{x})} = \rho_n$, concluding the basic case. Now, for the inductive case we distinguish whether n has one or two children.

Assume n has a single child c and $Q_c = (\mathcal{R} \mid \Theta)$. Then, by definition we have $Q_n = (\mathcal{R} \mid \Theta \cup \mathit{pred}(n))$. Therefore $Q_n(db) = \sigma_{\mathit{pred}(n)} Q_c(db)$, which implies that $\pi_{\mathit{var}(n)} Q_n(db) = \pi_{\mathit{var}(n)} \sigma_{\mathit{pred}(n)} Q_c(db)$. Since $\mathit{pred}(n)$ only mentions variables in $\mathit{var}(c) \cup \mathit{var}(n)$ and $\mathit{var}(n) \subseteq \mathit{var}(c)$, as c is a guard of n, this is equivalent to

$$\pi_{\mathit{var}(n)} Q_n(db) = \pi_{\mathit{var}(n)} \sigma_{\mathit{pred}(n)} \pi_{\mathit{var}(c)} Q_c(db)$$

By induction, this equals $\pi_{\mathit{var}(n)} \sigma_{\mathit{pred}(n)} \rho_c = \rho_n$, showing that $\pi_{\mathit{var}(n)} Q_n(db) = \rho_n$.

Assume now that n has two children c_1 and c_2, and that $Q_{c_i} = (\mathcal{R}_i \mid \Theta_i)$ for $i \in \{1, 2\}$. We assume w.l.o.g. that c_1 is a guard for n. First, note that by definition $Q_n = (\mathcal{R}_1 \bowtie \mathcal{R}_2 \mid \Theta_1 \cup \Theta_2 \cup \mathit{pred}(n))$, and then we have $Q_n(db) = \sigma_{\mathit{pred}(n)} \sigma_{\Theta_1} \sigma_{\Theta_2} (\mathcal{R}_1 \bowtie \mathcal{R}_2)(db)$. Since Θ_i only mentions variables of atoms in $\mathcal{R}_i$ (for $i \in \{1, 2\}$), we can push the selections and obtain

$$\begin{aligned} Q_n(db) &= \sigma_{\mathit{pred}(n)} (\sigma_{\Theta_1} \mathcal{R}_1 \bowtie \sigma_{\Theta_2} \mathcal{R}_2)(db) \\ &= \sigma_{\mathit{pred}(n)} (\sigma_{\Theta_1} \mathcal{R}_1(db) \bowtie \sigma_{\Theta_2} \mathcal{R}_2(db)) \\ &= \sigma_{\mathit{pred}(n)} (Q_{c_1}(db) \bowtie Q_{c_2}(db)) \end{aligned}$$

Therefore,

$$\pi_{\mathit{var}(n)} Q_n(db) = \pi_{\mathit{var}(n)} \sigma_{\mathit{pred}(n)} (Q_{c_1}(db) \bowtie Q_{c_2}(db)) \tag{4.2}$$

Since $\mathit{var}(\mathit{pred}(n)) \subseteq \mathit{var}(c_1) \cup \mathit{var}(c_2) \cup \mathit{var}(n)$ and $\mathit{var}(n) \subseteq \mathit{var}(c_1)$ we have $\mathit{var}(\mathit{pred}(n)) \subseteq \mathit{var}(c_1) \cup \mathit{var}(c_2)$. This, combined with the fact that, due to the connectedness property of T we, have $\mathit{var}(Q_{c_1}) \cap \mathit{var}(Q_{c_2}) \subseteq \mathit{var}(c_i)$ for $i \in \{1, 2\}$, we can add the following projections

$$(4.2) = \pi_{\mathit{var}(n)} \sigma_{\mathit{pred}(n)} \left(\pi_{\mathit{var}(c_1)} Q_{c_1}(db) \bowtie \pi_{\mathit{var}(c_2)} Q_{c_2}(db) \right).$$

Then, by induction hypothesis we have

$$\pi_{\mathit{var}(n)} Q_n(db) = \pi_{\mathit{var}(n)} \sigma_{\mathit{pred}(n)} (\rho_{c_1} \bowtie \rho_{c_2}) = \rho_n,$$

concluding our proof. □

To show correctness of enumeration, we need the following additional lemma regarding the subroutine of Algorithm 6 (Line 3).

Lemma 14. *Let Q, T, and N be as above. If ρ is a T-reduct of db, then for every node $n \in N$ and every tuple $\vec{t}$ in ρ_n, $\textsc{enum}_{T,N}(n, \vec{t}, \rho)$ correctly enumerates $\pi_{\mathit{var}(N) \cap \mathit{var}(Q_n)} Q_n(db) \ltimes \vec{t}$.*

Proof. Within the proof, we abuse notation and allow for projections over supersets of variables. For example, if $\mathit{var}(Q) \subseteq \overline{x}$ then $\pi_{\overline{x}} Q = \pi_{\overline{x} \cap \mathit{var}(Q)} Q$.

Let $n \in N$ and $\vec{t} \in \rho_n$. We proceed by induction on the number of nodes in $N \cap T_n$. If $N \cap T_n = \{n\}$, we have $\mathit{var}(N) \cap \mathit{var}(Q_n) = \mathit{var}(n)$ and therefore $\pi_{\mathit{var}(N)} Q_n(db) = \pi_{\mathit{var}(n)} Q_n(db)$. Then, by Lemma 13 we have $\pi_{\mathit{var}(N)} Q_n(db) = \rho_n$. As $\vec{t} \in \rho_n$, this implies that the only tuple in $\pi_{\mathit{var}(N)} Q_n(db)$ that is compatible with $\vec{t}$ is $\vec{t}$ itself. As n is in the frontier of N, $\textsc{enum}_{T,N}(n, \vec{t}, \rho)$ will enumerate precisely $\{(\vec{t}, \rho_n(\vec{t}))\}$ (Line 4), which concludes the base case.

For the inductive step we need to consider two cases depending on the number of children of n.

Case (1). If n has a single child c then necessarily c is a guard of n, i.e., $var(n) \subseteq var(c)$. In this case, Algorithm 6 will call $\text{ENUM}_{T,N}(c, \vec{s}, \rho)$ for each tuple $\vec{s} \in \rho_c \ltimes_{pred(n)} \vec{t}$. By induction hypothesis and Lemma 13, this will correctly enumerate every tuple in $\pi_{var(N)} Q_c(db) \ltimes \vec{s}$ for every $\vec{s}$ in $\sigma_{pred(n)}(\pi_{var(c)} Q_c(db) \ltimes \vec{t})$. Therefore, this enumerates the set

$$\pi_{var(N)} Q_c(db) \ltimes \sigma_{pred(n)}(\pi_{var(c)} Q_c(db) \ltimes \vec{t}). \tag{4.3}$$

As $var(pred(n)) \subseteq var(c) \cup var(n) = var(c) \subseteq var(Q_c)$, we can pull out the projection and selection

$$(4.3) = \pi_{var(N)} \sigma_{pred(n)}(Q_c(db) \ltimes (\pi_{var(c)} Q_c(db) \ltimes \vec{t})) \tag{4.4}$$

Because the variables in $\vec{t}$ are a subset of $var(c)$, this is the same as $\pi_{var(N)} \sigma_{pred(n)}(Q_c(db) \ltimes \vec{t})$. Finally, we push the selection and projection inside and obtain

$$(4.4) = \pi_{var(N)} \sigma_{pred(n)} Q_c(db) \ltimes \vec{t} = \pi_{var(N)} Q_n(db) \ltimes \vec{t} \tag{4.5}$$

Case (2). Otherwise, n has two children c_1 and c_2. Since $|N \cap T_n| > 1$ and N is sibling closed we have $\{c_1, c_2\} \subset N$. In this case, Algorithm 6 will first enumerate $\vec{t_i} \in \rho_{c_i} \ltimes_{pred(n \to c_1)} \vec{t}$ for $i \in \{1, 2\}$. By Lemma 13 this is equivalent to enumerate every $\vec{t_i}$ in $\sigma_{pred(n \to c_i)} \pi_{var(c_i)} Q_{c_i}(db) \ltimes \vec{t}$. Then, for each such $\vec{t_i}$ the algorithm will enumerate every pair $(\vec{s_i}, \mu_i)$ generated by $\text{ENUM}_{T,N}(c_i, \vec{t_i}, \rho)$, which by induction is the same as enumerating every $(\vec{s_i}, \mu_i)$ in $\pi_{var(N)} Q_{c_i}(db) \ltimes \vec{t_i}$. Therefore the algorithm is enumerating

$$\pi_{var(N)} Q_{c_i}(db) \ltimes (\sigma_{pred(n \to c_i)} \pi_{var(c_i)} Q_{c_i}(db) \ltimes \vec{t})$$

By the same reasoning as in the previous case, this is equivalent to enumerating every $(\vec{s_i}, \mu_i)$ in

$$\sigma_{pred(n \to c_i)} \pi_{var(N)} Q_{c_i}(db) \ltimes \vec{t}.$$

From the connectedness property of T, it follows that $var(Q_{c_1}) \cap var(Q_{c_2}) \subseteq var(n)$. Thus, $var(Q_{c_1}) \cap var(Q_{c_2})$ is a subset of the variables of $\vec{t}$. Hence, every tuple $\vec{s_1}$ will be compatible with every tuple $\vec{s_2}$, and the enumeration of every pair $(\vec{s_1} \cup \vec{s_2}, \mu_1 \times \mu_2)$ is the same as the enumeration of

$$\left[\sigma_{pred(n \to c_1)} \pi_{var(N)} Q_{c_1}(db) \ltimes \vec{t}\right] \bowtie \left[\sigma_{pred(n \to c_2)} \pi_{var(N)} Q_{c_2}(db) \ltimes \vec{t}\right].$$

We can now push the projections and selections outside and obtain

$$= \pi_{var(N)} \sigma_{pred(n \to c_1)} \sigma_{pred(n \to c_2)} [(Q_{c_1}(db) \ltimes \vec{t}) \bowtie (Q_{c_2}(db) \ltimes \vec{t})]$$

Since $pred(n) = pred(n \to c_1) \cup pred(n \to c_2)$ and the variables in $var(Q_{c_1}) \cap var(Q_{c_2})$ are contained in the variables of $\vec{t}$, we have

$$\begin{aligned} &= \pi_{var(N)} \sigma_{pred(n)} [(Q_{c_1}(db) \bowtie Q_{c_2}(db)) \ltimes \vec{t}] \\ &= \pi_{var(N)} [\sigma_{pred(n)} (Q_{c_1}(db) \bowtie Q_{c_2}(db))] \ltimes \vec{t} \\ &= \pi_{var(N)} Q_n(db) \ltimes \vec{t} \end{aligned}$$

□

With these results, now we present the correctness of the enumeration algorithm.

Proposition 18. *Let Q, T, N and ρ be as above. Then $\text{ENUM}_{T,N}(\rho)$ enumerates $Q(db)$.*

Proof. Let r be the root of T. By Lemma 13 we have $\rho_n = \pi_{var(r)} Q_r(db) = \pi_{var(r)} Q(db)$, and therefore ρ_n is a projection of $Q(db)$. This implies that $Q(db) = Q(db) \ltimes \rho_r$, which is equivalent to the disjoint union $\bigcup_{\vec{t} \in \rho_r} Q(db) \ltimes \vec{t}$. By Lemma 14, it is clear that this is exactly what $\text{ENUM}_{T,N}(\rho)$ enumerates. □

We now analyze the complexity of $\text{ENUM}_{T,N}$. First, observe that by definition of T-reducts, compatible tuples will exist at every node. Hence, every tuple that we iterate over will eventually produce a new output tuple. This ensures that we do not risk wasting time in iterating over tuples that in the end yield no output. As such, the time needed for $\text{ENUM}_{T,N}(\rho)$ to produce a single new tuple is determined by the time taken to enumerate the tuples in $\rho_n \ltimes_{pred(p \to n)} \vec{t}$, where p is the parent of n. Since this is equivalent to $\pi_{var(n)}(\rho_n \bowtie_{pred(p \to n)} \vec{t})$ we can do this efficiently by creating an index on ρ_n by $(pred(p \to n), var(p), var(n))$. For example, in Section 4.3 we defined hash-maps and group-sorted GMRs so that given one tuple from a parent we could enumerate the compatible tuples in the child with constant delay. In general, the efficiency of enumeration will depend on the delay provided by the indices.

Proposition 19. *Assume that for every node $n \in N$ we have an index on ρ_n by $(pred(p \to n), var(p), var(n))$ with delay f, where p is the parent of n and f is a monotone function. Then, using these indices, $\text{ENUM}_{T,N}(\rho)$ correctly enumerates $Q(db)$ with delay $O(|N| \times f(M))$ where M is given by $\max_{n \in N}(|\rho_n|)$. Thus, the total time required to execute algorithm $\text{ENUM}_{T,N}(\rho)$ is $O(|Q(db)| \cdot f(M) \cdot |N|)$.*

Proof. We show that for every $n \in N$ and $\vec{t} \in \rho_n$, the call $\text{ENUM}_{T,N}(n, \vec{t}, \rho)$ enumerates $\pi_{var(N)} Q_n(db)$ with delay $O(|N \cap T_n| \times f(M))$. We proceed by induction in $|N|$. If $|N| = 1$ then $N = \text{root}(T)$ and the delay is clearly constant as the algorithm will only yield $\vec{t}$. Now assume that $|N| > 1$. If n has a single child c, the index on ρ_c by $(pred(n), var(n), var(c))$ allows us to iterate over $\rho_c \ltimes_{pred(n)} \vec{t}$ with delay $O(f(|\rho_c|))$ and therefore delay $O(f(M))$. For each element $\vec{s}$ of this enumeration, the algorithm calls $\text{ENUM}_{T,N}(c, \vec{s}, \rho)$, which by induction hypothesis enumerates $\pi_{var(N)} Q_c(db) \ltimes \vec{s}$ with delay $O(|N \cap T_c| \times f(M))$. Then, the maximum delay between two outputs is $O(f(|\rho_c|)) + O(|N \cap T_c| \times f(M))$, and since $|\rho_c| \leq M$ this is in

$$O\left((|N \cap T_c| + 1) \times f(M)\right) = O\left(|N \cap T_n| \times f(M)\right).$$

The final observation is that the sets $\pi_{var(N)} Q_c(db) \ltimes \vec{s}$ are disjoint for different values of $\vec{s}$, and thus the enumeration does not produce repeated values.

For the case in which n has two children c_1 and c_2, by similar reasoning it is easy to show that the maximum delay between two outputs is

$$\begin{aligned} &O(f(|\rho_{c_1}|)) + O(|N \cap T_{c_1}| \times f(M)) + O(f(|\rho_{c_2}|)) + O(|N \cap T_{c_2}| \times f(M)) \\ &= O((|N \cap T_{c_1}| + |N \cap T_{c_2}| + 2) \times f(M)) \\ &= O(|N \cap T_n| \times f(M)). \end{aligned}$$

It is also important to mention that the sets enumerated by $\text{ENUM}_{T,N}(c_i, \vec{t_i}, \rho)$ are disjoint for each $\vec{t_i} \in \rho_{c_i} \ltimes_{pred(n \to c_i)} \vec{t}\,(i \in \{1, 2\})$, and that for each $(\vec{s_1}, \mu) \in \text{ENUM}_{T,N}(c_1, \vec{t_1}, \rho)$ and $(\vec{s_2}, \mu) \in \text{ENUM}_{T,N}(c_2, \vec{t_2}, \rho)$, it is the case that $\vec{s_1}$ and $\vec{s_2}$ are compatible, thus producing outputs in every iteration. □

In particular, if f is constant we enumerate $|Q(db)|$ with delay $O(N)$ (i.e. constant in data complexity).

Algorithm 7 Update(ρ, u)

1: **Input:** A T-reduct ρ for *db* and an update u.
2: **Result:** Transforming ρ to a T-reduc for $db + u$.
3: **for each** $n \in \text{leafs}(T)$ labeled by $r(\overline{x})$ **do**
4: $\quad \Delta_n \leftarrow u_{r(\overline{x})}$
5: **for each** $n \in \text{nodes}(T) \setminus \text{leafs}(T)$ **do**
6: $\quad \Delta_n \leftarrow$ empty GMR over $var(n)$
7: **for each** $n \in \text{nodes}(T)$, traversed bottom-up **do**
8: $\quad \rho_n += \Delta_n$
9: $\quad$ **if** n has a parent p and a sibling m **then**
10: $\quad\quad \Delta_p += \pi_{var(p)} \left(\rho_m \ltimes_{pred(p)} \Delta_n\right)$
11: $\quad$ **else if** n has parent p **then**
12: $\quad\quad \Delta_p += \pi_{var(p)} \sigma_{pred(p)} \Delta_n$

Update processing. To allow enumeration of $Q(db)$ under updates to *db* we need to maintain the T-reduct ρ (and, if present, its indexes) up to date. As illustrated in the previous section, it suffices to traverse the nodes of T in a bottom-up fashion. At each node n we have to compute the delta of ρ_n. For leaf nodes, this delta is given by the update u itself. For interior nodes, the delta can be computed from the delta and original reduct of its children. Algorithm 7 gives the pseudocode.

The fundamental part of Algorithm 7 is to compute joins and produce delta GMRs (Line 10), propagating updates from each node to its parent. When there is an update Δ_n to a node n with sibling m and parent p, we need to compute $\pi_{var(p)} \left(\rho_m \ltimes_{pred(p)} \Delta_n\right)$. To do this efficiently, we naturally store an index on ρ_m by $(pred(p), var(n), var(p))$. For example, we discussed how the hash-map I_{ρ_r} in Fig. 4.7 plus the sorting on x of ρ_r allowed us to efficiently compute $\pi_{y,z,w}(\rho_r \ltimes_{x<z} \Delta\rho_s)$.

Delta Enumeration and Lookups. Given the above discussion on enumeration with constant delay and update processing, we are now ready to enumerate $\Delta Q(db)$ with constant delay and lookup a tuple in the result of a query with logarithmic delay. Since each update to an atom $\boldsymbol{a}$ that is a leaf in T is processed in a bottom-up fashion and the delta Δ_n of each node n on the path between $\boldsymbol{a}$ and the root of T is computed as given in algorithm 7, the delta $\Delta Q(db)$ is a special case of full enumeration and can be computed using algorithm 6 simply by replacing ρ_n by its delta for each node n that appears on the path between $\boldsymbol{a}$ and the root of T. Since each Δ_n is also a GMR, it hence suffices to state that the $\Delta Q(db)$ can be enumerated with the same complexity as enumerating $Q(db)$.

Next, to test if a tuple $\vec{t}$ over the sequence of variables $\overline{y} = out(Q)$ is in $Q(db)$ for a GCQ Q, we can use the T-reduct developed so far. In particular, for each node n in the GJT T we can do as follows: starting from the root of T and doing recursively, use the index $I_{\rho_n}(\vec{t}[\text{pvar}(n)])$ to check if $\vec{t} \in \rho_n$, and since ρ_n is sorted on the variables $\overline{v}$ that ρ_n shares with predicates on the edge between node n and its parent, we can test with logarithmic delay if $\overline{v}$ in ρ_n. This works similar to the enumeration algorithm 6 without *foreach* loops and parametrized by the tuple $\vec{t}$ as shown in algorithm 4 in chapter 3.

Summarizing, to efficiently enumerate query results and process updates we need to store a T-reduct plus a set of indices on its GMRs. The data structure containing these elements is called a (T, N)-*representation*.

Definition 24 ((T, N)-representation)**.** Let *db* be a database. A (T, N)-representation ((T, N)-rep for short) of *db* is composed by a T-reduct of *db* and, for each node n with parent p, the following set of indices:

- If n belongs to N, then we store an index P_n on ρ_n by $(\mathit{pred}(p \to n), \mathit{var}(p), \mathit{var}(n))$.
- If n is a node with a sibling m, then we store an index S_n on ρ_n by $(\mathit{pred}(p), \mathit{var}(m), \mathit{var}(p))$.

Together with the notion of (T, N)-rep, Algorithms 6 and 7 provide a framework for dynamic query evaluation. By constructing the T-reduct and set of indices (and their update procedures) one can process free-connex acyclic GCQs under updates. Naturally, to implement such framework one needs to devise indices for a particular set of predicates. For example, DYN is an instantiation to the class of NCQs, and in the previous section we showed how to instantiate this framework for a GCQ based on equalities and inequalities. Next, we present the general set of indices required to process free-connex acyclic GCQs with equalities and inequalities.

IEDyn. For queries that have only inequality predicates, the instantiation of a (T, N)-representation of *db* contains a T-reduct of *db* and, for each node n with parent p, the following data structures:

- If $n \in N$, the index P_n on ρ_n from Definition 24 is obtained by doing two things. (1) First, group ρ_n according to the variables in $\mathit{var}(n) \cap \mathit{var}(p)$. Then, per group, sort the tuples according to the variables of $\mathit{var}(n)$ mentioned in $\mathit{pred}(p \to n)$ (if any).(2) Create a hash table that maps each tuple $\vec{t} \in \pi_{\mathit{var}(n) \cap \mathit{var}(p)}(\rho_n)$ to its corresponding group in ρ_n. If $\mathit{var}(n) \cap \mathit{var}(p)$ is empty this hash table is omitted.
- If n has a sibling m, the index S_n of Definition 24 is obtained by doing two things. (1) First, group ρ_n according to the variables in $\mathit{var}(n) \cap \mathit{var}(m)$. Then, per group, sort the tuples according to the variables of $\mathit{var}(n)$ mentioned in $\mathit{pred}(p)$ (if any). (2) Create a hash table mapping each $\vec{t} \in \pi_{\mathit{var}(n) \cap \mathit{var}(m)}(\rho_n)$ to the corresponding group in $\vec{s} \in \rho_n$. If $\mathit{var}(n) \cap \mathit{var}(m)$ is empty this hash table is omitted.

In Section 4.3 we illustrated how to use these data structures. Effectively, in Figure 4.7 I_{ρ_r} and I_{ρ_s} are examples of S_n, used for update propagation, while $I_{\rho_{\{y,z,w\}}}$ is an example of P_n, used for enumeration.

Note that the example query from Section 4.3 has at most one inequality between each pair of atoms. This causes each edge in T to consist of at most one inequality. As such, when creating the index P_n for a node $n \in N$, the reduct ρ_n will be sorted per group according to at most one variable. This is important for enumeration delay because, as exemplified in Section 4.3, we can then find compatible tuples by first the corresponding group and then iterating over the sorted group from the start and stopping when the first non-compatible tuple is found. When there are multiple inequalities per pair of atoms then we will need to sort according to multiple variables under some lexicographic order. This causes enumeration delay to become logarithmic since then compatible tuples will intermingle with non-compatible tuples, and a binary search is necessary to find the next batch of compatible tuples in the group.

We call IEDYN the algorithm for processing free-connex acyclic GCQs with equalities and inequalities.

Theorem 4. *Let Q be a GCQ in which all predicates are equalities and inequalities. Let (T, N) be a binary and sibling-closed GJT pair compatible with Q. Given a database db over at(Q), a (T, N)-rep $\mathcal{D}$ of db, under* IEDYN *Algorithm 6 enumerates $Q(db)$ with delay $O(|N| \cdot \log(|db|))$. Also, given an update u under* IEDYN *Algorithm 7 transforms $\mathcal{D}$ into a (T, N)-rep of $db + u$ in time $O(|T| \cdot M^2 \cdot \log(M))$, where $M = |db| + |u|$.*

Proof. Let us first prove the enumeration bounds. It is immediate to see that for every node $n \in T$ the GMR ρ_n satisfies $|\rho_n| \leq |db|$, given that ρ_n is defined as a series of semijoins based on *db* (or, equivalently, because every internal node has a guard). Therefore, according to Proposition 19 the enumeration delay is $O(|N| \cdot f(|db|))$ where N is the connex subset of T and f is the delay provided by the index P_n. Now, from the description of IEDYN these indices are implemented as hash tables that map each tuple $\vec{t}$ in $\pi_{var(p) \cap var(n)} \rho_n$ to a lexicographically sorted set containing $\rho_n \ltimes_{pred(p \to n)} \vec{t}$, where (p, n) is a parent-child pair. Therefore, given a tuple $\vec{t} \in \rho_p$ we can enumerate $\rho_n \ltimes_{pred(p \to n)} \vec{t}$ by first projecting $\vec{t}$ over $var(n)$ and then iterating over all tuples satisfying $pred(p \to n)$. Since these predicates are only inequalities, each group can be kept sorted lexicographically and, as mentioned earlier, enumeration can be achieved with logarithmic delay. It follows from Prop. 19 that the enumeration delay is $O(|N| \cdot \log(|db|))$.

Now we discuss update time. As can be seen in Algorithm 7, for each parent-child pair $(p, n) \in T$ we need to compute either $\pi_{var(p)}(\rho_m \bowtie_{pred(p \to n)} \Delta_n)$ or $\pi_{var(p)} \sigma_{pred(p)}(\Delta_n)$, depending on whether or not n has a sibling m. If n does not have a sibling, computing $\pi_{var(p)} \sigma_{pred(p)}(\Delta_n)$ can be done directly by sorting Δ_n lexicographically, enumerating those tuples satisfying $pred(p)$ (with logarithmic delay), and finally projecting over $var(p)$. This takes time in $O(|\Delta_n| \cdot \log(|\Delta_n|)$, which is clearly contained in $O(M^2 \cdot \log(M))$ since $|\Delta_n| \leq M$. The more involved case is when n has a sibling m and we need to compute $\pi_{var(p)}(\rho_m \bowtie_{pred(p)} \Delta_n)$. Here we first sort Δ_n lexicographically. Then, for every tuple $\vec{t}$ in $\pi_{var(p)} \rho_m$ compute $\pi_{var(p)}(\vec{t} \bowtie_{pred(p)} \Delta_n)$. Note that this can be done in time $O(|\Delta_n| \cdot \log(|\Delta_n|))$ since from the constructed data structures we can enumerate $\Delta_n \ltimes_{pred(p)} \vec{t}$ with logarithmic delay. Because the previous procedure needs to be performed for each $\vec{t} \in \rho_n$, this can be done in time $O(|\rho_n| \cdot |\Delta_n| \cdot \log(|\Delta_n|))$ and therefore in time $O(M^2 \cdot \log(M))$. Note that here we ignore the sorting steps as well as the maintenance of the corresponding GMRs as those steps are clearly $O(M \cdot \log(M))$. Finally, since we need to perform the procedure described above once per each parent-child pair, the entire routine takes at most $O(|T| \cdot M^2 \cdot \log(M))$. □

From the previous result we can see that for the general case of equalities and inequalities we already have a procedure that can be quadratic in the size of the database.However, if we restrict the use of inequalities in a particular way, we can speed up both update processing and enumeration delay.

Theorem 5. *Let Q, T and N be defined as in Theorem 4, and assume that for each $p \in T$ it is the case that $|pred(p)| \leq 1$. Given a database db over at(Q), a (T, N)-rep $\mathcal{D}$ of db, under* IEDYN *Algorithm 6 enumerates $Q(db)$ with delay $O(|N|)$. Also, given an update u under* IEDYN *Algorithm 7 transforms $\mathcal{D}$ into a (T, N)-rep of $db + u$ in time $O(|T| \cdot M \cdot \log(M))$, where $M = |db| + |u|$.*

Proof. The main observation to prove this result is that when there is a single predicate, a lexicographically sorted set is totally sorted by a single attribute. Regarding enumeration, this implies that given a parent-child pair (p, n) and a tuple $\vec{t} \in \pi_{var(n)} \rho_p$, we can enumerate $\rho_n \ltimes_{pred(P)} \vec{t}$ with constant delay. The reason behind this is that the index P_n maps $\vec{t}$ to a totally sorted set, and therefore we can start from the largest/smallest value of the relevant attribute, and iterate over all tuples decreasingly/increasingly until we find a tuple that does not satisfy the inequality. At that point we are certain that we have visited all tuples satisfying the inequality.

The update processing can also be improved by a similar argument, although the modification is slightly more involved. Assume again that we have a parent-child pair (p, n) and want to compute $\pi_{var(p)}(\rho_m \bowtie_{pred(p)} \Delta_n)$, where m is the sibling of n. We do so

efficiently as follows. Recall that the index S_m groups ρ_m by $var(n) \cap var(m)$ and sorts each group by the variables involved in $pred(p)$. We construct an index over Δ_n with the same characteristics, which is achieved by a vanilla implementation in $O(|\Delta_n| \cdot \log(|\Delta_n|))$. Again, since $pred(p)$ contains at most a single inequality, each group will be sorted by a single variable and hence totally sorted. Assume now that m is a guard of p. Since by definition $\rho_m \ltimes_{pred(p)} \Delta_n = \sigma_{pred(p)}(\rho_m \ltimes \Delta_n)$, to compute this join it is sufficient to find for each tuple $\vec{t}$ in ρ_m the matching tuples in the corresponding group of Δ_n. However, a naive implementation would take $O(M^2)$, since for such $\vec{t}$ we might iterate over a potentially linear set of tuples in Δ_m. This can be avoided by considering the following two observations:

1. Given a tuple $\vec{t}$ in ρ_m, since m is a guard of p we only need to compute the multiplicity associated to $\vec{t}$ in $\sigma_{pred(p)}(\pi_{var(p)}(\rho_m \ltimes \Delta_n))$, which can be computed as $\rho_m(\vec{t}) \cdot \sum_{\vec{s} \in \Delta_n \ltimes_{pred(p)} \vec{t}} \Delta_n(\vec{s})$.

2. Let $\vec{t_1}$ and $\vec{t_2}$ be two tuples belonging to the same group in ρ_m. Assume $pred(p) = a < b$, with $a \in var(n)$ and $b \in var(m)$. Then, if $\vec{t_1}(a) < \vec{t_2}(a)$ we have that $\Delta_n \ltimes_{pred(p)} \vec{t_2}$ is a subset of $\Delta_n \ltimes_{pred(p)} \vec{t_1}$.

By these two facts, if we iterate in order over the tuples $\vec{t}$ of each group of ρ_m, and we iterate simultaneously in order over the tuples $\vec{s}$ in the group of Δ_n corresponding to $\vec{t}$ (which can be done with constant delay), we can compute the corresponding multiplicities incrementally, visiting each tuple in Δ_n only once. Therefore, this join can be computed in linear time in M and the most expensive part of this procedure is to actually construct and maintain the sorted groups, an $O(M \cdot \log(M))$ procedure. It is easy to see that this can be generalized to any inequality, and that in the case in which n is a guard of p it suffices to swap the roles of ρ_m and Δ_n. We conclude that in this case IEDYN updates the corresponding (T, N)-representation in $O(M \cdot \log(M))$. □

4.5 Experimental Evaluation

In this section, we present the experimental evaluation of GDYN. We do so by first presenting the experimental setup and the actual implementation details in section 4.5.1, and then discuss the evaluation results in section 4.5.2. We evaluate GDYN over GCQs under updates for both runtime memory footprint, runtime throughput and the enumeration of query results. Moreover, we evaluate against competing systems on multiple dimensions as we will discuss in the following sections.

4.5.1 Experimental Setup

In this subsection, we first present practical implementation details. Then we present the query and the data streams used for evaluation followed by brief description of the competing systems against which we evaluate IEDYN. Finaly, we present the experimental setup.

Practical Implementation. We instantiate GDYN to the case of equalities and inequalities (i.e IEDYN in Section 4.3) and we have implemented IEDYN as a query compiler that generates executable code in the Scala programming language. The generated code instantiates a (T, N)-rep and defines *trigger functions* that are used for maintaining the T-rep under updates. Our implementation is basic in the sense that we use Scala off-the-shelf collection libraries (notably `MutableTreeMap`) to implement the required indices. Faster implementations with specialized code for the index structures are certainly possible.

Our implementation supports two modes of operation: *push-based* and *pull-based*. In both modes, the system maintains the T-rep under updates. In the *push-based mode* the system generates, on its output stream, the delta result $\Delta Q(db, u)$ after each single-tuple update u. To do so, it uses a modified version of enumeration (Algorithm 6) that we call *delta enumeration*. Similarly to how Algorithm 6 enumerates $Q(db)$, delta enumeration enumerates $\Delta Q(db, u)$ with constant delay (if Q has at most one inequality per pair of atoms) resp. logarithmic delay (otherwise). To do so, it uses both (1) the T-reduct GMRs ρ_n and (2) the delta GMRs $\Delta\rho_n$ that are computed by Algorithm 7 when processing u. In this case, however, one also needs to index the $\Delta\rho_n$ similarly to ρ_n. In the *pull-based mode*, in contrast, the system only maintains the T-rep under updates but does not generate any output stream. Nevertheless, at any time a user can call the enumeration (Algorithm 6) procedure to obtain the current output.

We have described in Section 4.4 how IEDYN can process free-connex acyclic GCQs under updates. It should be noted that our implementation also supports the processing of general acyclic GCQs that are not necessarily free-connex. This is done using the following simple strategy. Let Q be acyclic but not free-connex. First, compute a free-connex acyclic approximation Q_F of Q. Q_F can always be obtained from Q by extending the set of output variables of Q. In the worst case, we need to add all variables, and Q_F becomes the full join underlying Q. Then, use IEDYN to maintain a T-rep for Q_F. When operating in push-based mode, for each update u, we use the T-representation to delta-enumerate $\Delta Q_F(db, u)$ and project each resulting tuple to materialize $\Delta Q(db, u)$ in an array. Subsequently, we copy this array to the output. Note that the materialization of $\Delta Q(db, u)$ here is necessary since the delta enumeration on T can produce duplicate tuples after projection. When operating in pull-based mode, we materialize $Q(db)$ in an array, and use delta enumeration of Q_F to maintain the array under updates. Of course, under this strategy, we require $\Omega(\|Q(db)\|)$ space in the worst case, just like (H)IVM would, but we avoid the (partial) materialization of delta queries. Note the distinction between the two modes: in push-based mode $\Delta Q(db, u)$ is materialized (and discarded once the output is generated), while in pull-based mode $Q(db)$ is materialized upon requests.

Queries and Streams. In contrast to the setting for equi-join queries where systems can be compared based on industry-strength benchmarks such as TPC-H and TPC-DS, there is no established benchmark suite for inequality-join queries.

We evaluate IEDYN on the GCQ queries listed in table 4.1. Here, queries Q_1–Q_6 are full join queries (i.e., queries without projections). Among these, Q_1, Q_3 and Q_4 are cross products with inequality predicates, while Q_2, Q_5 and Q_6 have at least one equality in addition to the inequality predicates. Queries Q_1 and Q_2 are binary join queries, while Q_3–Q_6 are multi-way join queries. Queries Q_7–Q_{12} project over the result of queries Q_4–Q_6. Among these, Q_7–Q_9 are free-connex acyclic while Q_{10}–Q_{12} acyclic but not free-connex.

We evaluate these queries on streams of updates where each update consists of a single tuple insertion. The database is always empty when we start processing the update stream. We synthetically generate two kinds of update streams: *randomly-ordered* and *temporally-ordered* update streams. In *randomly-ordered* update streams, insertions can

Query	Expression
Q_1	$R(a,b,c) \bowtie S(d,e,f) \vert a < d$
Q_2	$R(a,b,c,k) \bowtie S(d,e,f,k) \vert a < d$
Q_3	$R(a,b,c) \bowtie S(d,e,f) \bowtie T(g,h,i) \vert a < d \wedge e < g$
Q_4	$R(a,b,c) \bowtie S(d,e,f) \bowtie T(g,h,i) \vert a < d \wedge d < g$
Q_5	$R(a,b,c,k) \bowtie S(d,e,f,k) \bowtie T(g,h,i) \vert a < d \wedge d < g$
Q_6	$R(a,b,c) \bowtie S(d,e,f,k) \bowtie T(g,h,i,k) \vert a < d \wedge d < g$
Q_7	$\pi_{a,b,d,e,f,g,h}(Q_4)$
Q_8	$\pi_{a,d,e,f,g,h,k}(Q_5)$
Q_9	$\pi_{d,e,f,g,h,k}(Q_6)$
Q_{10}	$\pi_{b,c,e,f,h,i}(Q_4)$
Q_{11}	$\pi_{b,c,e,f,h,i}(Q_5)$
Q_{12}	$\pi_{b,c,e,f,h,i}(Q_6)$

Table 4.1: Queries for experimental evaluation.

occur in any order. In contrast, *temporally-ordered* update streams guarantee that any attribute that participates in an inequality in the query has a larger value than the same attribute in any of the previously inserted tuples. Randomly-ordered update streams are useful for comparing against systems that allow processing of out-of-order tuples; temporally-ordered update streams are useful for comparison against systems that assume events arrive always with increasing timestamp values. Examples of systems that process temporally-ordered streams are automaton-based CER systems.

A random update stream of size N for a query with k relations is generated as follows. First, we generate N/k tuples with random attribute values for each relation. Then, we insert tuples in the update stream by uniformly and randomly selecting them without repetitions. This ensures that there are N/k insertions from each relation in the stream. To utilize the same update stream for evaluating each system we compare to, each stream is stored in a file. We choose the values for equality join attributes uniformly at random from 1 to 200, except for the scalability and selectivity experiments in Section 4.5.2 where the interval depends on the stream size.

Temporally-ordered streams are generated similarly, but when a new insertion tuple is chosen, a new value is inserted in the attributes that are compared through inequalities. This value is larger than the corresponding values of previously inserted tuples. All attributes hold integer values, except for attributes c and i which contain string values.

Competitors. We compare IEDYN with DBToaster (DBT) [KAK+14], Esper (E) [Esp], SASE (SE) [WDR06, ZDI14, ADGI08], Tesla (T) [CM10, CM12a], and ZStream (Z) [MM09] using *memory footprint*, *update processing time*, and *enumeration delay* as comparison metrics. The competing systems differ in their mode of operation (push-based vs pull-based) and some of them only support temporally-ordered streams.

DBToaster is a state-of-the-art implementation of HIVM. It operates in pull-based mode, and can deal with randomly-ordered update streams. DBToaster is particularly meticulous in that it materializes only useful views, and therefore it is an interesting implementation for comparison. DBToaster has been extensively tested on equi-join queries and has proven to be more efficient than a commercial database management system, a commercial stream processing system and an IVM implementation [KAK+14]. DBToaster compiles given SQL statements into executable trigger programs in different programming languages. We compare against those generated in Scala from the DBToaster Release 2.2[1], and it uses actors[2] to generate events from the input files. During our experiments, however, we have found that this creates unnecessary memory overhead. For a fair memory-wise comparison, we have therefore removed these actors.

[1] https://dbtoaster.github.io/

[2] https://doc.akka.io/docs/akka/2.5/

Esper is a CER engine with a relational model based on Stanford STREAM [ABB+16]. It is push-based, and can deal with randomly-ordered update streams. We use the Java-based open source[3] for our comparisons. Esper processes queries expressed in the Esper event processing language (EPL).

SASE is an automaton-based CER system. It operates in push-based mode, and can deal with temporally-ordered update streams only. We use the publicly available Java-based implementation of SASE[4]. This implementation does not support projections. Furthermore, since SASE requires queries to specify a match semantics (any match, next match, partition contiguity) but does not allow combinations of such semantics, we can only express queries Q_1, Q_2, and Q_4 in SASE. Hence, we compare against SASE for these queries only. To be coherent with our semantics, the corresponding SASE expressions use the any match semantics [ADGI08].

Tesla/T-Rex is also an automaton-based CER system. It operates in push-based mode only, and supports temporally-ordered update streams only. We use the publicly available C-based implementation[5]. This implementation operates in a publish-subscribe model where events are published by clients to the server, known as TRexServer. Clients can subscribe to receive recognized composite events. Tesla cannot deal with queries involving inequalities on multiple attributes e.g. Q_3, therefore, we do not show results for Q_3. Since Tesla works in a decentralized manner, we measure the update processing time by logging the time at the Tesla TRexServer from the start of the stream being processed until the end.

ZStream is a CER system based on a relational internal architecture. It operates in push-based mode, and can deal with temporally-ordered update streams only. ZStream is not available publicly. Hence, we have created our own implementation following the lazy evaluation algorithm of ZStream described in their original paper [MM09]. This paper does not describe how to treat projections, and as such we compare against ZStream only for full join queries Q_1–Q_6.

The query expressions for all of these systems are presented in the appendix section A.2. For Esper the queries are written in its respective language *Event Processing Language* (EPL), for SASE and Tesla/TRex the queries are written in their respective rules languages, and for DBToaster we have used the SQL standard syntax.

Setup. Our experiments are run on an 8-core 3.07 GHz machine running Ubuntu with GNU/Linux 3.13.0-57-generic. To compile the different systems or generated trigger programs, we have used GCC version 4.8.2, Java 1.8.0_101, and Scala version 2.12.4. Each query is evaluated 10 times to measure update processing delay, and two times to measure memory footprint. We present the average over those runs. Each time a query is evaluated, 20 GB of main memory are freshly allocated to the program. To measure the memory footprint for Scala/Java based systems, we invoke the JVM system calls every 10 updates and consider the maximum value. For C/C++ base systems we use the GNU/Linux *time* command to measure memory usage. Experiments that measure memory footprint are always run separately of the experiments that measure processing time.

[3] http://www.espertech.com/esper/esper-downloads/
[4] https://github.com/haopeng/sase
[5] https://github.com/deib-polimi/TRex

4.5.2 Experimental Evaluation

Before presenting experimental results we make some remarks. First, when we compare against another system we run IEDYN in the operation mode supported by the competitor. For push-based systems we report the time required to both process the entire update stream, and generate the changes to the output after each update. When comparing against a pull-based system, the measured time includes only processing the entire update stream. We later report the speed with which the result can be generated from the underlying representation of the output (a T-representation in the case of IEDYN). When comparing against a system that supports randomly-ordered update streams, we only report comparisons using streams of this type. We have also looked at temporally-ordered streams for these systems, but the throughput of the competing systems is similar (fluctuating between 3% and 12%) while that of IEDYN significantly improves (fluctuating between 35% and 50%) because insertions to sorted lists become constant instead of logarithmic.

It is also important to remark that some executions of the competing systems failed either because they required more than 20GB of main memory or they took more than 1500 seconds. If an execution requires more than 20GB, we report the processing time elapsed until the exception was raised. If an execution is still running after 1500 seconds, we stop it and report its maximum memory usage while running.

Full join queries. Figure 4.8 compares the update processing time of IEDYN against the competing systems for full join queries Q_1–Q_6. In particular, we show IEDYN (IE) VS Z,DBT,E,T, and SE on full join queries. The X-axis shows stream sizes and the y-axis shows update delay in $seconds$. The symbols *, +, and ′ show that DBT ran out of memory, Z ran out of memory, and T was stopped after 1500 seconds, respectively. We have grouped experiments that are run under comparable circumstances: in the top row experiments are conducted for push-based systems on temporally-ordered update streams (SE, T, Z); in the second row push-based systems on randomly-ordered update streams (E), and in the bottom row pull-based systems on randomly-ordered update streams (DBT). We observe that all of the competing systems have large processing times even for very small update stream sizes, and that for some systems execution even failed. All of these behaviors are due to the low selectivity of joins on this dataset. Table 4.5.2 shows the output size of each query for the largest stream sizes reported in Figure 4.8. We report on streams that generate outputs of different sizes below.

Query	|Stream|	|Output|
Q_1	12k	18,017k
Q_2	12k	3.8k
Q_3	2.7k	178,847k
Q_4	2.7k	90,425k
Q_5	21k	411,669k
Q_6	21k	297,873k
Q_7	2.7k	114,561k
Q_8	21k	411,669k
Q_9	21k	99,043k
Q_{10}	2.7k	114,561k
Q_{11}	21k	294,139k
Q_{12}	21k	297,873k

Table 4.2: Maximum output sizes per query, k=1000.

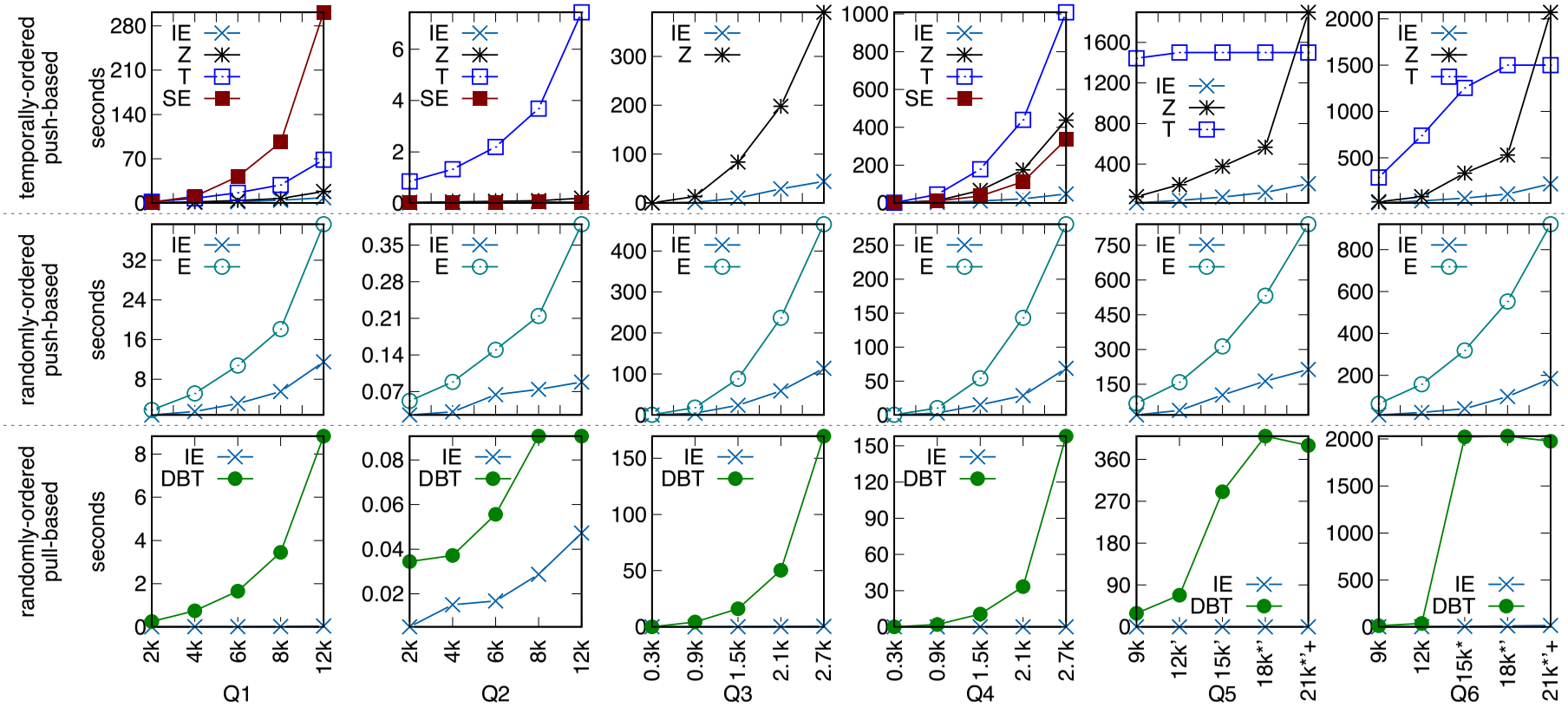


Figure 4.8: IEDyn (IE) VS (Z,DBT,E,T, SE) on full join queries. The X-axis shows stream sizes and the y-axis update delay in $seconds$ (*: DBT out of memory, +: Z out of memory, $'$: T was stopped after 1500 seconds)

Figure 4.8 is complemented by Figures 4.9, 4.10, and 4.11 where we plot the processing time and memory footprint used by IEDYN as a percentage of the corresponding usage in the competing systems. Both, SE and Z support temporally ordered streams, however, SE supports only queries Q_1, Q_2, and Q_4 and Z supports Q_1–Q_6, therefore in Figure 4.10 we show SE and in 4.11 we show Z. Note that IEDYN significantly outperforms the competing systems on all full join queries. Specifically, it outperforms DBT up to one order of magnitude in processing time and up to two orders of magnitude in memory footprint. It outperforms T up to two orders of magnitude in processing time, and more than one order of magnitude in memory footprint. Moreover, for these queries, even in push-based mode IEDYN can support the enumeration of query results from its data structures at any time while competing push-based systems have no such support. Hence, IEDYN is not only more efficient but also provides more functionality.

Projections. Figure 4.9 depicts results of IEDYN against E, DBT, and T for all queries. In this figure, the symbols * and ′ indicate that DBT ran out of memory and T was stopped after 1500 seconds (maximum allocated time to each query), respectively. Results in this figure show that IEDYN significantly outperforms both E and DBT on free-connex queries Q_7–Q_9: two orders of magnitude improvement over the throughput of T and more than twofold improvement over that of E. Memory usage is also significantly less: one order of magnitude over E on the larger datasets for Q_7, and a consistent twofold improvement over T. Similarly, IEDYN outperforms DBT on free-connex queries Q_7 and Q_8 in time and memory by one and two orders of magnitude, respectively.

For non-free-connex queries Q_{10}–Q_{12}, IEDYN continues to outperform E, T, and DBT in terms of processing time. In memory footprint IEDYN outperforms E for Q_{10} and Q_{12}. Compared to DBT, IEDYN still improves on memory footprint on non-free-connex queries, though less significantly. In contrast, IEDYN largely improves memory usage over T on larger datasets, even on non-free-connex queries.

Result enumeration. We know from Section 4.4 that T-reps maintained by IEDYN feature constant delay enumeration (CDE). This theoretical notion, however, hides a constant factor that could decrease performance in practice when compared to full materialization. In Figure 4.12, we show the practical application of CDE in IEDYN and compare against DBT which materializes the full query results. We plot the time required to enumerate the result from IEDYN's T-rep as a fraction of the time required to enumerate the result from DBT's materialized views. As can be seen from the figure, both enumeration times are comparable on average.

Note that we do not compare enumeration time for push-based systems, since for these systems the time required for delta enumeration is already included in the update processing time reported in Figures 4.8, 4.9 (bottom), 4.10 and 4.11.

Selective inequality joins. We execute IEDYN over the datasets that are uniformly distributed and analyze its performance. On uniformly distributed datasets, the inequality joins yield large query results. One could argue that this might not be realistic. To address this problem, we generated datasets with probability distributions that are parametrized by a *selectivity* s, such that the expected number of output tuples is s percent of the cartesian product of all relations in the query.

Our results depicted in Figure 4.13 show that IEDYN not only outperforms existing systems on less selective inequality joins; we also perform better on very selective inequality joins consistently. For super selective inequality joins the measurements come similar to what we observe for equality joins, which we investigated in detail in our previous work on equality joins [IUV17].

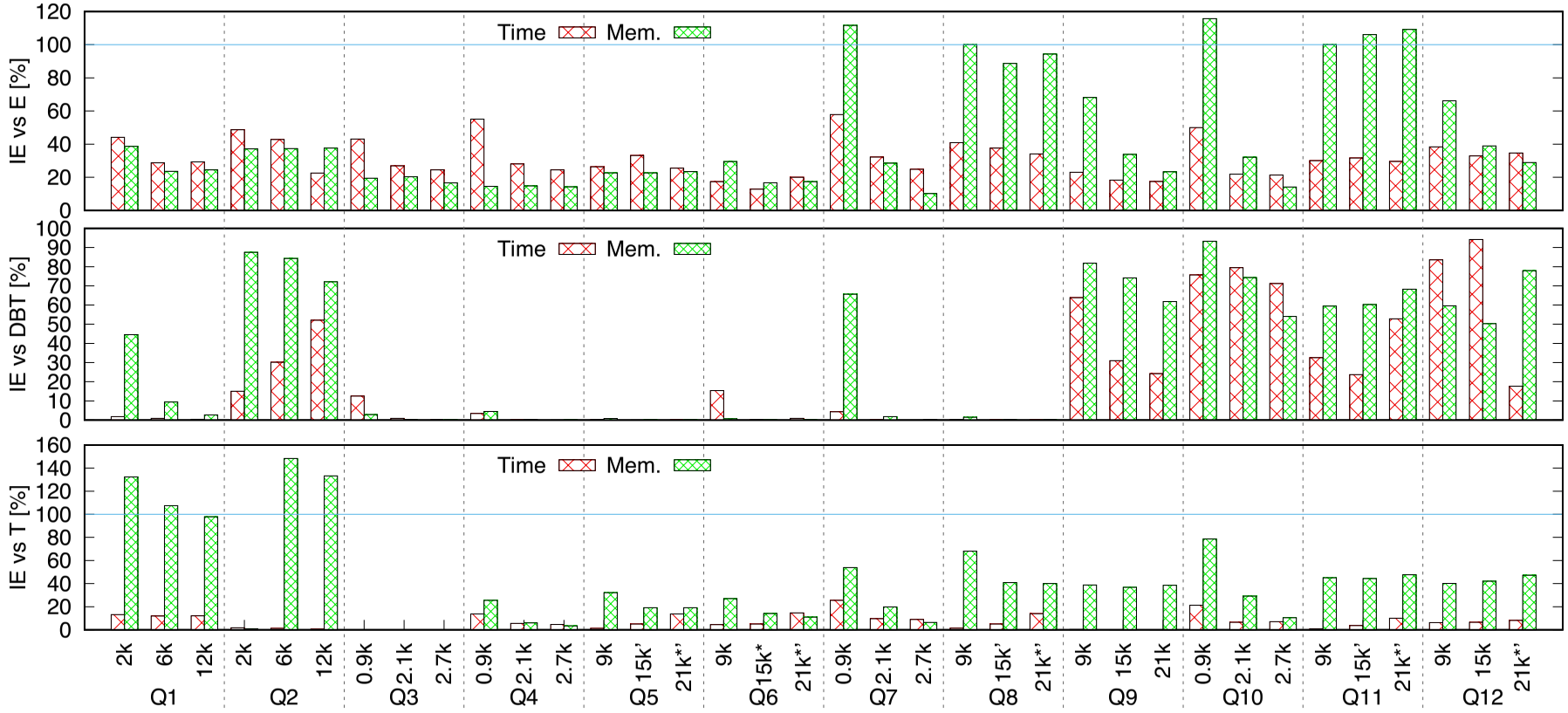


Figure 4.9: IEDYN (IE) VS (E, DBT, T) fulljoin and projection queries, (*: DBT ran out of memory, ′: T was stopped after 1500 seconds)

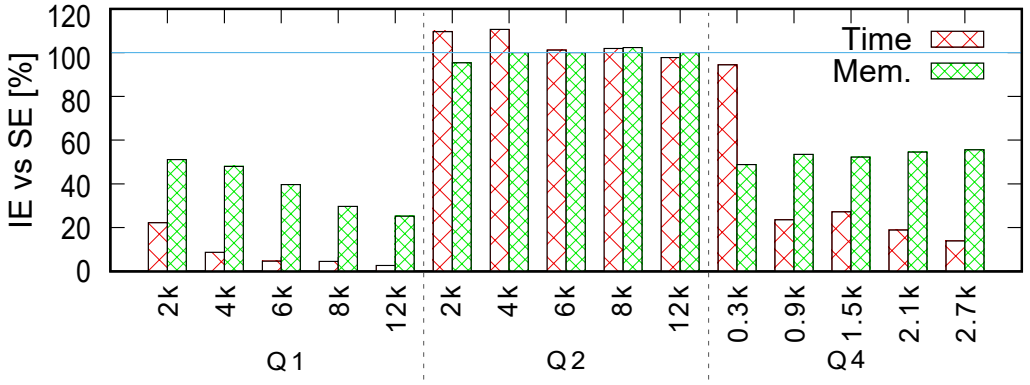


Figure 4.10: IEDYN (IE) VS SE on temporally ordered datasets

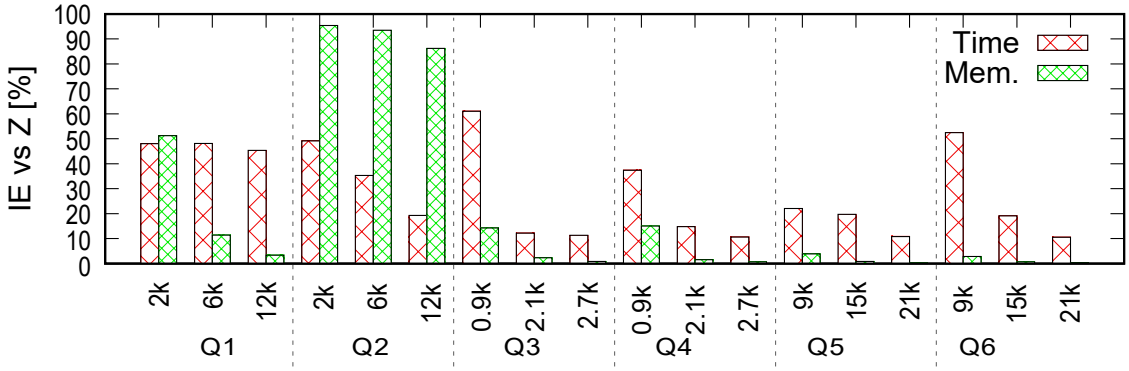


Figure 4.11: IEDYN (IE) VS Z on temporally ordered datasets

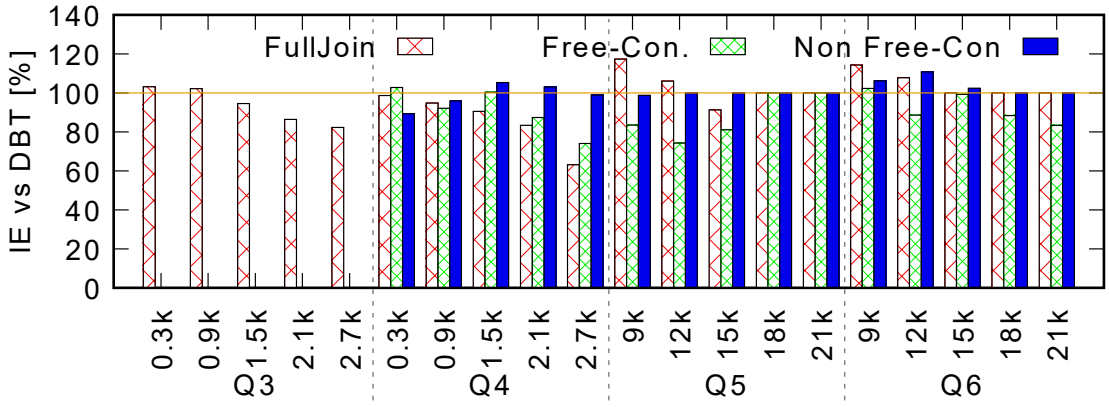


Figure 4.12: Enumeration of query results: IEDYN vs DBT, different bars for Q_4, Q_5, Q_6 show their projected versions

Scalability. To present that IEDYN performs consistently on varying sizes of input data streams, we report the stream processing delay and the memory footprint each time 10%(shown as number of tuples in millions) of the stream is processed in Figure 4.14. These results show that IEDYN has linearly increasing memory footprint as well as update delay as the stream size advances. We show results for representative queries Q_4, Q_5, Q_7, and Q_8.

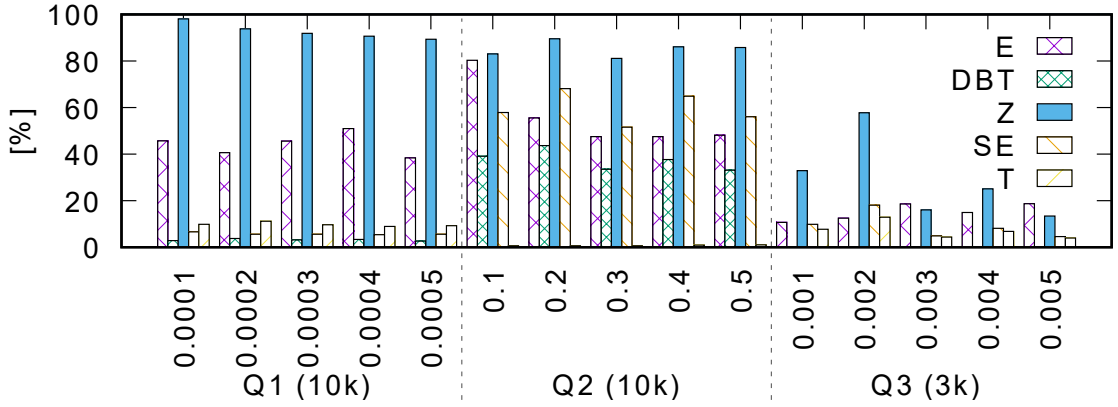


Figure 4.13: IEDYN as percentage of (E, DBT, SE, Z, T) for higher join selectivities. X-axis shows queries with tuples per relation and selectivities

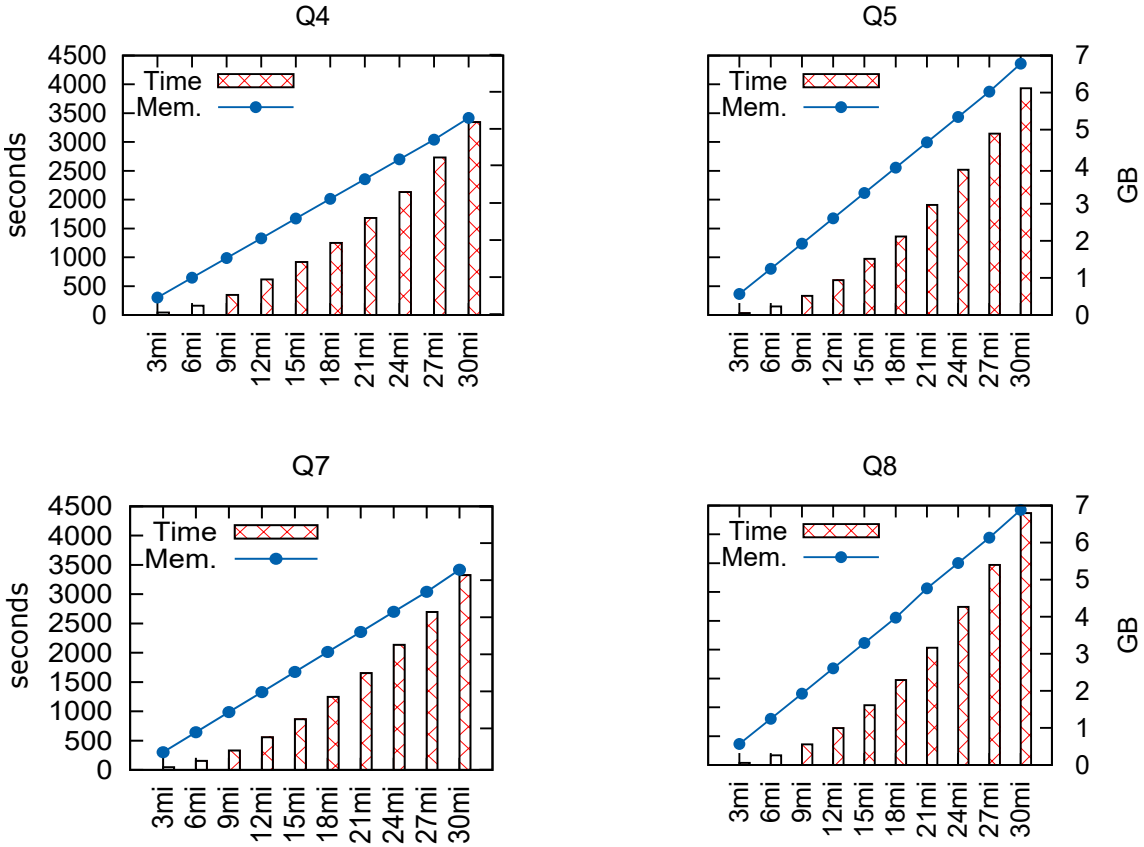


Figure 4.14: IEDYN scalability ($mi = 1,000,000$)

4.6 Conclusion

We presented the GDYN framework - an extension of the DYN framework - to work for queries that feature arbitrary θ-joins. We presented GJTs for GCQs, and showed the IEDYN algorithm with an extensive example that works for queries that features equalities and inequalities. There, we showed that, for any GCQ that features at most one inequality predicate between a pair of relations, IEDYN can process updates with log-

linear delay, supports constant time lookups in query result, and supports enumeration of query results with constant delay. Next, we presented the GDyn framework that works for arbitrary θ-joins and features the properties P_1^* through P_2^* of DCLRs. We experimentally showed that GDyn can process queries that appear in traditional BI systems as well as in IFP systems. In particular, we compared against state of the art BI and CER systems on multiple dimensions such as: pull and push based systems, systems that process streams in temporal order and in random order, and systems that are based on relational model and automaton model. We also showed for representative queries that IEDyn is scalable. In the next chapter, we present the algorithm to test a GCQ for acyclicity and computation of GJTs for GCQs that are acyclic.

5

JOIN TREE COMPUTATION

So far, we have discussed algorithms DYN and GDYN to process updates on DCLRs and enumerate query results. As discussed in chapters 3 and 4, these DCLRs are based on Generalized Join Trees (GJTs) and GJTs are defined for Generalized Conjunctive Queries (GCQs) that are acyclic. In this chapter, we discuss how to test GCQs for acyclicity (respectively free-connex acyclicity) and present our algorithms to compute a compatible GJT pair (T, N) for a GCQ Q if Q is acyclic.

The canonical algorithm for checking acyclicity of normal conjunctive queries is the GYO algorithm (from Graham-Yu-Ozsoyoglu) [AHV95], also known as the *GYO-reduction*. Our algorithm for checking acyclicity is a generalization of the GYO algorithm that checks free-connex acyclicity in addition to normal acyclicity and deals with GCQs featuring θ-join predicates instead of NCQs that have equality joins only.

This chapter is structured as follows. We first recall the classical GYO-reduction algorithm in Section 5.1. Then we introduce our extended GYO-reduction algorithm for GCQs in Section 5.2. We prove it correct in Section 5.3.

5.1 Classical GYO

The classical GYO algorithm operates on *hypergraphs*. A *hypergraph* H is a set of non-empty hyperedges. Recall from chapter 4 Section 4.1 that a hyperedge is just a finite set of variables. Every GCQ is naturally associated to a hypergraph as follows.

Definition 25. Let Q be a GCQ. The hypergraph of Q, denoted $hyp(Q)$, is the hypergraph

$$hyp(Q) = \{\overline{x} \mid r(\overline{x}) \text{ atom of } Q, \overline{x} \neq \emptyset\}.$$

The GYO algorithm checks acyclicity of a normal conjunctive query Q by constructing $hyp(Q)$ and repeatedly removing *ears* (defined as follows) from this hypergraph. If ears can be removed until only the empty hypergraph remains, then the query is acyclic; otherwise it is cyclic.

An *ear* in a hypergraph H is a hyperedge e for which we can divide its variables into two groups: (1) those that appear exclusively in e, and (2) those that are contained in another hyperedge f of H. A variable that appears exclusively in a single hyperedge is also called an *isolated variable*. Thus, ear removal corresponds to executing the following two reduction operations.

- Remove isolated variables: select a hyperedge e in H and remove isolated variables from it; if e becomes empty, remove e it altogether from H.
- Subset elimination: remove hyperedge e from H if there exists another hyperedge f for which $e \subseteq f$.

The *GYO-reduction* of a hypergraph is the hypergraph that is obtained by executing these operations until no further operation is applicable. The following result is standard; see e.g., [AHV95] for a proof.

Proposition 20. *An NCQ Q is acyclic, if and only if, the GYO-reduction of $hyp(Q)$ is the empty hypergraph.*

5.2 GYO-reduction for GCQs

In order to extend the GYO-reduction for the purpose of checking free-connex acyclicity (not simply acyclicity) of GCQs (not simply standard NCQs), we will: (1) redefine the notion of being an ear to take into account the predicates specified in a query Q; and (2) transform the GYO-reduction into a two-stage procedure. The first stage allows to check that a connex set with exactly $out(Q)$ can exist while the first and second stage combined check that the query is acyclic.

Our modified GYO-reduction algorithm operates on *hypergraph triplets* instead of hypergraphs, which are defined as follows.

Definition 26. A *hypergraph triplet* (or simply *triplet*) is a triple $\mathcal{H} = (hyp(\mathcal{H}), out(\mathcal{H}), pred(\mathcal{H}))$ with $hyp(\mathcal{H})$ a hypergraph, $out(\mathcal{H})$ a hyperedge, and $pred(\mathcal{H})$ a set of predicates.

Intuitively, the variables in $out(\mathcal{H})$ will correspond to the output variables of a query and the set $pred(\mathcal{H})$ will contain predicates that need to be taken into account when removing ears. Every GCQ is therefore naturally associate to a hypergraph triplet as follows.

Definition 27. The hypergraph triplet of a GCQ Q, denoted $\mathcal{H}(Q)$, is the triplet

$$(hyp(Q), out(Q), pred(Q))$$

In order to extend the notion of an ear, we require the following preliminary definitions. Let $\mathcal{H}$ be a hypergraph triplet. Variables that occur in $out(\mathcal{H})$ or in at least two hyperedges in $hyp(\mathcal{H})$ are called *equijoin variables* of $\mathcal{H}$. We denote the set of all equijoin variables of $\mathcal{H}$ by $jv(\mathcal{H})$ and abbreviate $jv_{\mathcal{H}}(e) = e \cap jv(\mathcal{H})$. A variable x is *isolated* in $\mathcal{H}$ if it is not an equijoin variable and is not mentioned in any predicate, i.e., if $x \notin jv(\mathcal{H})$ and $x \notin var(pred(\mathcal{H}))$. We denote the set of isolated variables of $\mathcal{H}$ by $isol(\mathcal{H})$ and abbreviate $isol_{\mathcal{H}}(e) = e \cap isol(\mathcal{H})$. The *extended variables* of hyperedge e in $\mathcal{H}$, denoted $ext_{\mathcal{H}}(e)$ is the set of all variables of predicates that mention some variable in e, except the variables in e themselves:

$$ext_{\mathcal{H}}(e) = \bigcup\{var(\theta) \mid \theta \in pred(\mathcal{H}), var(\theta) \cap e \neq \emptyset\} \setminus e.$$

Finally, a hyperedge e is a *conditional subset* of hyperedge f w.r.t. $\mathcal{H}$, denoted $e \sqsubseteq_{\mathcal{H}} f$, if $jv_{\mathcal{H}}(e) \subseteq f$ and $ext_{\mathcal{H}}(e \setminus f) \subseteq f$. We omit subscripts from our notation if the triplet is clear from the context.

Example 10. *In Figure 5.1 we depict several hypergraph triplets. There, hyperedges in $\mathcal{H}$ are depicted by colored regions and variables in $out(\mathcal{H})$ are underlined. We use dashed lines to connect variables that appear together in a predicate. So, in $\mathcal{H}_1$, we have predicates θ_1, θ_2 with $var(\theta_1) = \{t, v\}$ and $var(\theta_2) = \{x, y\}$. Now consider triplet $\mathcal{H}_1$ in particular. It is the hypergraph triplet $\mathcal{H}(Q)$ for the following GCQ Q:*

$$Q = \pi_{t,u,z,w}(r_1(s,t,u) \bowtie r_2(t,u) \bowtie r_3(u,w,x) \bowtie r_4(s,v) \bowtie r_5(w,z,y) \mid t < v \wedge x < y)$$

Moreover, $jv(\mathcal{H}_1) = \{s,t,u,w,z\}$ and $isol(\mathcal{H}_1) = \emptyset$. Furthermore, $ext_{\mathcal{H}_1}(\{v\}) = \{t\}$ since the predicate $\theta_1 = t < v$ shares variables with $\{v\}$. Finally $jv_{\mathcal{H}_1}(\{s,v\}) = \{s\} \subseteq \{s,t,u\}$ and $ext_{\mathcal{H}_1}(\{s,v\} \setminus \{s,t,u\}) = ext_{\mathcal{H}_1}(\{v\}) = \{t\} \subseteq \{s,t,u\}$. Therefore, $\{s,v\} \sqsubseteq_{\mathcal{H}_1} \{s,t,u\}$. Similarly, $\{t,u\} \sqsubseteq_{\mathcal{H}_1} \{s,t,u\}$.

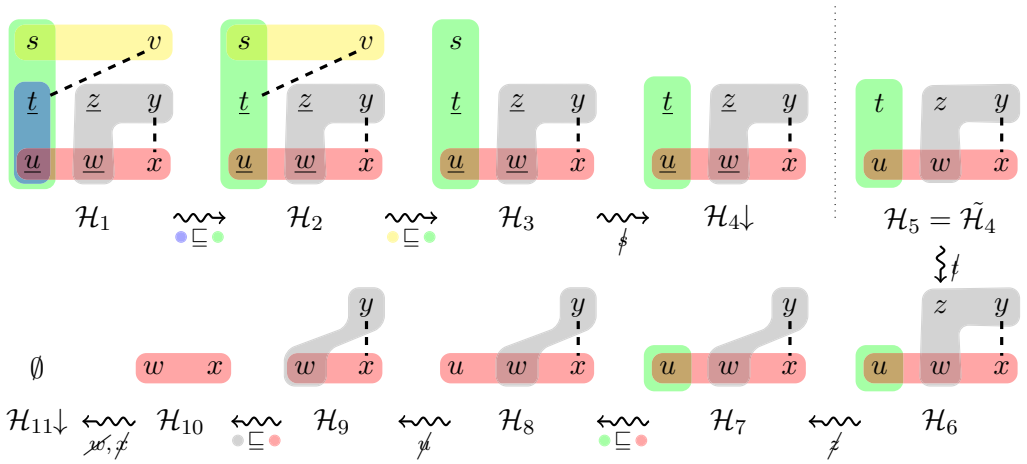


Figure 5.1: Illustration of GYO-reduction for GCQs. Colored regions depict hyperedges. Variables in *out* are underlined. Variables occurring in the same predicate are connected by dashed lines.

Given these definitions, we are now ready to re-define ears as follows.

Definition 28. A hyperedge e is an *ear in triplet* $\mathcal{H}$ if $e \in hyp(\mathcal{H})$ and either

1. we can divide its variables into two groups: (a) those that are isolated and (b) those that form a conditional subset of another hyperedge $f \in hyp(\mathcal{H}) \setminus \{e\}$; or
2. e consists only of non-join variables, i.e., $jv(e) = \emptyset$ and $ext(e) = \emptyset$.

Note that case (2) allows for $\theta \in pred(\mathcal{H})$ with $var(\theta) \subseteq e$. We call predicates that are covered by a hyperedge in this sense *filters* because they correspond to filtering tuples in a single GMR, instead of θ-joining two GMRs. If, in case (2), there is no filter θ with $var(\theta) \subseteq e$, then $e = isol_{\mathcal{H}}(e)$. Similar to the classical GYO reduction, we can view ear removal as a rewriting process on triplets, where we consider the following three reduction operations.

- (ISO) Remove isolated variables: select a hyperedge $e \in hyp(\mathcal{H})$ and remove a non-empty set $X \subseteq isol_{\mathcal{H}}(e)$ from it. If e becomes empty, remove it from $hyp(\mathcal{H})$.
- (CSE) Conditional subset elimination: remove hyperedge e from $hyp(\mathcal{H})$ if it is a conditional subset of another hyperedge f in $hyp(\mathcal{H})$. Also update $pred(\mathcal{H})$ by removing all predicates θ with $var(\theta) \cap (e \setminus f) \neq \emptyset$.

- (FLT) Filter elimination: select $e \in hyp(\mathcal{H})$ and a non-empty subset of predicates $\Theta \subseteq pred(\mathcal{H})$ with $var(\Theta) \subseteq e$, and remove all predicates in Θ from $pred(\mathcal{H})$.

We write $\mathcal{H} \rightsquigarrow \mathcal{I}$ to denote that triplet $\mathcal{I}$ is obtained from triplet $\mathcal{H}$ by application of a single such operation, and $\mathcal{H} \rightsquigarrow^* \mathcal{I}$ to denote that $\mathcal{I}$ is obtained by a sequence of zero or more of such operations.

Example 11. *For the hypergraph triplets illustrated in Figure 5.1 we have $\mathcal{H}_1 \rightsquigarrow \mathcal{H}_2 \rightsquigarrow \mathcal{H}_3 \rightsquigarrow \mathcal{H}_4$ and $\mathcal{H}_5 \rightsquigarrow \mathcal{H}_6 \rightsquigarrow \mathcal{H}_7 \rightsquigarrow \mathcal{H}_8 \rightsquigarrow \mathcal{H}_9 \rightsquigarrow \mathcal{H}_{10} \rightsquigarrow \mathcal{H}_{11}$. For each reduction,it is illustrated in the figure which set of isolated variables is removed, or which conditional subset is removed.*

We write $\mathcal{H}\!\downarrow$ to denote $\mathcal{H}$ is in *normal form*, i.e., that no operation is applicable on triplet $\mathcal{H}$. Note that, because each operation removes at least one variable, hyperedge, or predicate, we will always reach a normal form after a finite number of operations. Furthermore, while multiple different reduction steps may be applicable on a given triplet $\mathcal{H}$, the order in which we apply them does not matter:

Proposition 21 (Confluence)**.** *Assume that $\mathcal{H} \rightsquigarrow^* \mathcal{I}_1$ and $\mathcal{H} \rightsquigarrow^* \mathcal{I}_2$ then there exists $\mathcal{J}$ such that $\mathcal{I}_1 \rightsquigarrow^* \mathcal{J}$ and $\mathcal{I}_2 \rightsquigarrow^* \mathcal{J}$.*

Because the proof is technical but not overly enlightening, we defer it to section 5.4. A direct consequence is that normal forms are unique: if $\mathcal{H} \rightsquigarrow^* \mathcal{I}_1\!\downarrow$ and $\mathcal{H} \rightsquigarrow^* \mathcal{I}_2\!\downarrow$ then $\mathcal{I}_1 = \mathcal{I}_2$.

Let $\mathcal{H}$ be a triplet. The residual of $\mathcal{H}$, denoted $\tilde{\mathcal{H}}$, is the triplet $(hyp(\mathcal{H}), \emptyset, pred(\mathcal{H}))$, i.e., the triplet where $out(\mathcal{H})$ is set to $\emptyset$. A triplet is *empty* if it equals $(\emptyset, \emptyset, \emptyset)$.

Our main result in this section is the following. It states that, to check whether a GCQ Q is free-connex acyclic it suffices to start from $\mathcal{H}(Q)$ and do two stages of reductions: the first from $\mathcal{H}(Q)$ until a normal form $\mathcal{I}\!\downarrow$ is reached, and the second from the latter's residual, $\tilde{\mathcal{I}}$, until another normal form $\mathcal{J}$ is reached.[1]

Theorem 6. *Let Q be a GCQ. Assume $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\!\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\!\downarrow$. Then the following hold.*

1. *Q is acyclic if, and only if, $\mathcal{J}$ is the empty triplet.*
2. *Q is free-connex acyclic if, and only if, $\mathcal{J}$ is the empty triplet and $var(hyp(\mathcal{I})) = out(Q)$.*
3. *For every GJT T of Q and every connex subset N of T it holds that $var(hyp(\mathcal{I})) \subseteq var(N)$.*

We devote Section 5.3 to the proof.

Example 12. *Fig. 5.1 illustrates the two-stage sequence of reductions starting from $\mathcal{H}(Q)$ with Q the GCQ of Example 10. Note that $\mathcal{H}(Q) = \mathcal{H}_1$ and $\mathcal{H}_5$ is the residual of $\mathcal{H}_4$. Because we end with the empty triplet, Q is acyclic but not free-connex since $out(Q) \subsetneq var(\mathcal{H}_4)$.*

Theorem 6 gives us a decision procedure for checking free-connex acyclicity of GCQ Q. From its proof in Section 5.3, we can actually derive an algorithm for constructing a compatible GJT pair for Q. At its essence, this algorithm starts with the set of atoms appearing in Q, and subsequently uses the sequence of reduction steps from Theorem 6 to construct a GJT from it, at the same time checking free-connex acyclicity. Every reduction step causes new nodes to be added to the partial GJT constructed so far. We will refer to such partial GJTs as *Generalized Join Forests* (GJF).

[1] Note that because we set $out(\mathcal{I}) = \emptyset$ when taking the residual, new variables may become isolated and therefore more reductions steps may be possible on $\tilde{\mathcal{I}}$ even though $\mathcal{I}$ itself was in normal form.

Definition 29 (GJF)**.** A *Generalized Join Forest* is a set F of pairwise disjoint GJTs such that for all distinct trees $T_1, T_2 \in F$ we have $\mathit{var}(T_1) \cap \mathit{var}(T_2) = \mathit{var}(n_1) \cap \mathit{var}(n_2)$ where n_1 and n_2 are the roots of T_1 and T_2.

Every GJF encodes a hypergraph as follows.

Definition 30. The hypergraph $\mathit{hyp}(F)$ associated to GJF F is the hypergraph that has one hyperedge for every non-empty root node in F,

$$\mathit{hyp}(F) = \{\mathit{var}(n) \mid n \text{ root node in } F, \mathit{var}(n) \neq \emptyset\}.$$

The GJT construction algorithm does not manipulate hypergraph triplets directly. Instead, it manipulates *GJF triplets*. A GJF triplet is defined like a hypergraph triplet, except that it has a GJF instead of a hypergraph.

Definition 31. A *GJF triplet* is a triple $\mathbb{F} = ((\mathbb{F}), \mathit{out}(\mathbb{F}), \Theta_{\mathbb{F}})$ with $(\mathbb{F})$ a GJF, $\mathit{out}(\mathbb{F})$ a hyperedge, and $\Theta_{\mathbb{F}}$ a set of predicates. Every GJF triplet $\mathbb{F}$ induces a hypergraph triplet $\mathcal{H}(\mathbb{F}) = (\mathit{hyp}((\mathbb{F})), \mathit{out}(\mathbb{F}), \Theta_{\mathbb{F}})$.

The algorithm for constructing a GJT pair compatible with a given GCQ Q is now shown in Algorithm 8. It starts in line 2 by initializing the GJF triplet $\mathbb{F}$ to

$$\mathbb{F} = ((Q), \mathit{out}(Q), \mathit{pred}(Q)$$

. Here, (Q) is the GJF obtained by creating, for every atom $r(\overline{x})$ that occurs $k > 0$ times in Q, k corresponding leaf nodes labeled by $r(\overline{x})$. In Lines 3–4, Algorithm 8 then performs the first phase of reduction steps of Theorem 6. To this end, it checks whether a reduction operation is applicable to $\mathcal{H}(\mathbb{F})$ and, if so, *enacts* this operation by modifying $\mathbb{F}$ as follows.

- (ISO). If the reduction operation on the hypergraph triplet $\mathcal{H}(\mathbb{F})$ were to remove a non-empty subset X of isolated variables from hyperedge e, then $\mathbb{F}$ is modified as follows. Let $n_1, \ldots, n_k$ be all the root nodes in $(\mathbb{F})$ that are labeled by e. Merge the corresponding trees into one tree by creating a new node n with $\mathit{var}(n) = e$ and attaching $n_1, \ldots, n_k$ as children to it with $\mathit{pred}(n \to n_i) = \emptyset$ for $1 \leq i \leq k$. Then, enact the removal of X by creating a new node p with $\mathit{var}(p) = e \setminus X$ and attaching n as child to it with $\mathit{pred}(p \to n) = \emptyset$.
- (CSE) If the reduction operation on $\mathcal{H}(\mathbb{F})$ were to remove a hyperedge e because it is a conditional subset of another hyperedge f, then $\mathbb{F}$ is modified as follows. Let $n_1, \ldots, n_k$ (resp. $m_1, \ldots, m_l$) be all the root nodes in $(\mathbb{F})$ that are labeled by e (resp. f), and let $T_1, \ldots, T_k$ (resp. $U_1, \ldots, U_l$) be their corresponding trees. Similar to the previous case, merge the T_i (resp. U_j) into a single tree with new root n labeled by e (resp. m labeled by f). Then enact the removal of e by creating a new node p with $\mathit{var}(p) = f$ and attaching n and m as children with $\mathit{pred}(p \to n) = \{\theta \in \mathit{pred}(\mathbb{F}) \mid \mathit{var}(\theta) \cap (e \setminus f) \neq \emptyset\}$ and $\mathit{pred}(p \to m) = \emptyset$.
- (FLT) If the reduction operation on $\mathcal{H}(\mathbb{F})$ were to remove non-empty set of predicates Θ because there exists a hyperedge e with $\mathit{var}(\Theta) \subseteq e$, then $\mathbb{F}$ is modified as follows. Let $n_1, \ldots, n_k$ be all the root nodes in $(\mathbb{F})$ that are labeled by e. Merge the corresponding trees into one tree by creating a new root n labeled by e, and attaching $n_1, \ldots, n_k$ as children with $\mathit{pred}(n \to n_i) = \Theta$. Enact the removal of Θ by removing all $\theta \in \Theta$ from $\Theta(\mathbb{F})$.

It is straightforward to check that these modifications of the forest triplet $\mathbb{F}$ faithfully enact the corresponding operations on $\mathcal{H}(\mathbb{F})$, in the following sense.

Algorithm 8 Compute a GJT pair

1: **Input:** A GCQ Q.
2: $\mathbb{F} \leftarrow ((Q), \mathit{out}(Q), \mathit{pred}(Q))$
3: **while** a reduction step is applicable to $\mathcal{H}(\mathbb{F})$ **do**
4: enact the reduction on $\mathbb{F}$
5: $X \leftarrow$ set of all root nodes in $\mathbb{F}$
6: set $\mathit{pred}(\mathbb{F}) := \emptyset$
7: **while** a reduction step is applicable to $\mathcal{H}(\mathbb{F})$ **do**
8: enact the reduction on $\mathbb{F}$
9: **if** $\mathcal{H}(\mathbb{F})$ is not the empty triplet **then**
10: **error** "Q is not acyclic"
11: **else**
12: $T \leftarrow$ tree obtained by connecting all root nodes of $\mathbb{F}$'s forest to a new root, labeled by $\emptyset$
13: $N \leftarrow$ all nodes in X and their ancestors in T
14: **return** (T, N)

Lemma 15. *Let $\mathbb{F}$ be a forest triplet and assume $\mathcal{H}(\mathbb{F}) \rightsquigarrow \mathcal{I}$. Let $\mathbb{G}$ be the result of enacting this reduction operation on $\mathbb{F}$. Then $\mathbb{G}$ is a valid forest triplet and $\mathcal{H}(\mathbb{G}) = \mathcal{I}$.*

We continue the explanation of Algorithm 8. In line 5, Algorithm 8 records the set of root nodes obtained after the first stage of reductions. It then updates $\mathbb{F}$ by setting $\mathit{out}(\mathbb{F}) = \emptyset$ in line 6 and continues with the second stage of reductions in lines 7–8. It then employs Theorem 6 to check acyclicity of Q. If Q is not acyclic, it reports this in lines 9–10. If Q is acyclic, then we know by Theorem 6 that $\mathcal{H}(\mathbb{F})$ has become the empty triplet. Note that $\mathcal{H}(\mathbb{F})$ can be empty only if all the roots of $\mathbb{F}$'s join forest are labeled by the empty set of variables. As such, we can transform this forest into a join tree T by linking all of these roots to a new unique root, also labeled $\emptyset$. This is done in line 12. In line 13, the set of nodes N is computed, and consists of all nodes identified at the end of the first stage (line 5) plus all of their parents in T.

We will prove in Section 5.3 that Algorithm 8 is correct, in the following sense.

Theorem 7. *Given a GCQ Q, Algorithm 8 reports an error if Q is cyclic. Otherwise, it returns a sibling-closed GJT pair (T, N) with T a GJT for Q. If Q is free-connex acyclic, then (T, N) is compatible with Q. Otherwise, $\mathit{out}(Q) \subsetneq \mathit{var}(N)$, but $\mathit{var}(N)$ is minimal in the sense that for every other GJT pair (T', N') with T' a GJT for Q we have $\mathit{var}(N) \subseteq \mathit{var}(N')$.*

It is straightforward to check that this algorithm runs in polynomial time in the size of Q.

Example 13. *In Fig. 5.2, we show a GJT T and use this GJT to illustrate a number of GJFs $F_1, \ldots, F_{10}$ in the following way: let level 1 be the leaf nodes, level 2 the parents of the leaves, and so on. Then we take GJF F_i to be the set of all trees rooted at nodes at level i, for $1 \leq i \leq 10$, and with each level i, we mention the set of remaining predicates θ_i for $1 \leq i \leq k$ where k is the number of predicates in Q. Nodes (resp. predicates with each F_i) labeled by "$\bullet$" in Fig. 5.2 indicates that the node (and hence tree, resp. predicates) was already present in F_{i-1} and did not change. These should hence not be interpreted as new nodes (resp. predicates changed). With this coding of forests, it is easy to see that for all $1 \leq i \leq 9$, $F_i = \mathit{hyp}(\mathcal{H}_i)$ with $\mathcal{H}_i$ illustrated in Fig. 5.1. (Note here that the hypergraph of residual of $\mathcal{H}_4$ i.e. $\mathcal{H}_5$ is the same as $\mathcal{H}_4$, hence we do not show the corresponding F_5.) Furthermore, $\mathit{pred}(F_i) = \mathit{pred}(Q) \setminus \mathit{pred}(\mathcal{H}_i)$ with*

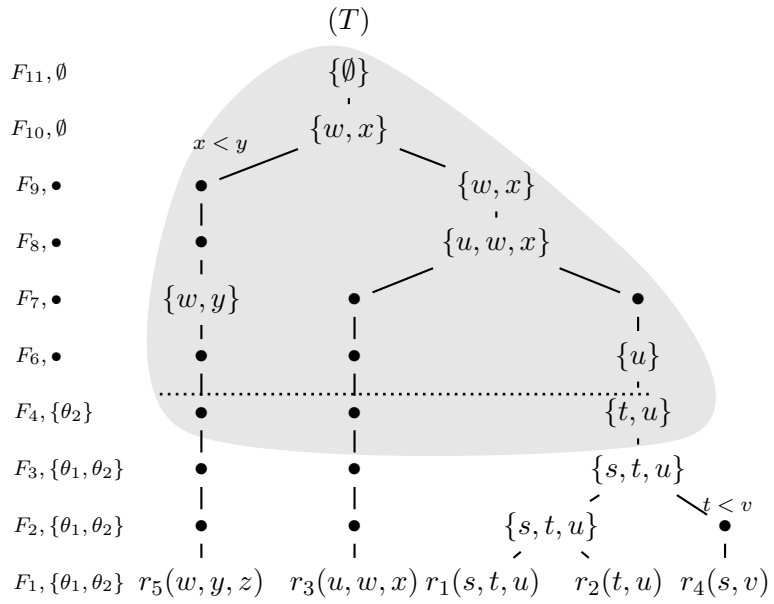


Figure 5.2: GJT Construction by GYO-reduction.

Q the GCQ from Example 10. As such, the tree illustrates the sequence of GJF triplets that is obtained by enacting the hypergraph reductions illustrated in Fig. 5.1. For example, let $\mathbb{F}_1 = (F_1, out(Q), pred(Q)$. After enacting the removal of hyperedge $\{t, u\}$ from $\mathcal{H}_1$ to obtain $\mathcal{H}_2$ we obtain $\mathbb{F}_2 = (F_2, out(Q), pred(Q))$. Here, F_2 is obtained by merging the single-node trees (i.e. labelled by the atoms in Q) $\{s, t, u\}$ and $\{t, u\}$ in to a single tree with root $\{s, t, u\}$. The shaded area illustrate the nodes in the connex subset N computed by Algorithm 8.

We stress that Algorithm 8 is non-deterministic in the sense that the pair (T, N) returned depends on the order in which the reduction operations are performed.

5.3 Correctness of GYO for GCQs

In this section, we prove that algorithm 8 is correct by proving the Theorems 6 and 7 i.e. soundness and completeness by means of a sequence of propositions.

5.3.1 Soundness and Completeness

In this subsection, we first present with a series of propositions the soundness and completeness corollaries and then present the proofs of theorems 6 and 7. To start with, we present the following proposition that we will require to prove theorem 7.

Proposition 22. *Let Q be a GCQ. Further, assume that $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\downarrow$. If $\mathcal{J}$ is the empty triplet, then, when run on Q, Algorithm 8 returns a GJT pair (T, N) such that T is a GJT for Q, N is sibling-closed, and $var(N) = var(hyp(\mathcal{I}))$.*

Proof. Assume that $\mathcal{J}$ is the empty triplet. Algorithm 8 starts in line 3 by initializing $\mathbb{F} = ((Q), out(Q), pred(Q))$. Clearly, $\mathcal{H}(\mathbb{F}) = \mathcal{H}(Q)$ at this point. Algorithm 8 subsequently

modifies $\mathbb{F}$ throughout its execution. Let $\mathbb{H}$ denote the initial version of $\mathbb{F}$; let $\mathbb{I}$ denote the version of $\mathbb{F}$ when executing line 5; let $\tilde{\mathbb{I}}$ denote the version of $\mathbb{F}$ after executing line 6 and let $\mathbb{J}$ denote the version of $\mathbb{F}$ when executing line 9. By repeated application of Lemma 15 we know that $\mathcal{H}(Q) = \mathcal{H}(\mathbb{H}) \rightsquigarrow^* \mathcal{H}(\mathbb{I})$. Furthermore, $\mathcal{H}(\mathbb{I})$ is in normal form. Since also $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\downarrow$ and normal forms are unique, $\mathcal{H}(\mathbb{I}) = \mathcal{I}$. Therefore, $\mathcal{H}(\tilde{\mathbb{I}}) = \tilde{\mathcal{I}}$. Again by repeated application of Lemma 15 we know that $\tilde{\mathcal{I}} = \mathcal{H}(\tilde{\mathbb{I}}) \rightsquigarrow^* \mathcal{H}(\mathbb{J})$. Moreover, $\mathcal{H}(\mathbb{J})$ is in normal form. Since also $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\downarrow$ and normal forms are unique, $\mathcal{H}(\mathbb{J}) = \mathcal{J}$. Since $\mathcal{J}$ is empty, will execute lines 12–14. Since $\mathcal{J}$ is the empty hypergraph triplet, every root of every tree in $(\mathbb{J})$ must be labeled by $\emptyset$. By definition of join forests, no two distinct trees in $(\mathbb{J})$ hence share variables. As such, the tree T obtained in line 12 by linking all of these roots to a new unique root, also labeled $\emptyset$, is a valid GJT.

We claim that T is a GJT for Q. Indeed, observe that $\mathit{at}(T) = \mathit{at}(Q)$ and the number of times that an atom occurs in Q equals the number of times that it occurs as a label in T. This is because initially $(\mathbb{H}) = (Q)$ and by enacting reduction steps we never remove nor add nodes labeled by atoms. Furthermore $\mathit{pred}(T) = \mathit{pred}(Q)$. This is because initially $\mathit{pred}(\mathbb{H}) = \mathit{pred}(Q)$ yet $\Theta_{\mathbb{J}}$ is empty. This means that, for every $\theta \in \mathit{pred}(Q)$, there was some reduction step that removed θ from the set of predicates of the current GJF triplet $\mathbb{F}$. However, when enacting reduction steps we only remove predicates after we have added them to $(\mathbb{F})$. Therefore, every predicate in $\mathit{pred}(Q)$ must occur in T. Conversely, during enactment of reduction steps we never add predicates to $(\mathbb{F})$ that are not in $\Theta_{\mathbb{F}}$, so all predicates in T are also in $\mathit{pred}(Q)$. Thus, T is a GJT for Q.

It remains to show that N is a sibling-closed connex subset of T and $\mathit{var}(\mathit{hyp}(\mathcal{I})) = \mathit{var}(N)$. To this end, let X be the set of all root nodes of $(\mathbb{I}))$, as computed in Line 5. Since $\mathbb{J}$ is obtained from $\tilde{\mathbb{I}}$ by a sequence of reduction enactments, and since such enactments only add new nodes and never delete them, M is a subset of nodes of $(\mathbb{J})$ and therefore also of T. As computed in Line 13, N consists of X and all ancestors of nodes of X in T. Then N is a connex subset of T by definition. Moreover, since enactments of reduction steps can only merge existing trees or add new parent nodes (never new child nodes), N must also be sibling-closed. Furthermore, since $\mathcal{H}(\mathbb{I}) = \mathcal{I}$, $\mathit{hyp}((\mathbb{I})) = \mathit{hyp}(\mathcal{I})$. Thus, $\mathit{var}(X) = \mathit{var}(\mathit{hyp}(\mathbb{I})) = \mathit{var}(\mathit{hyp}(\mathcal{I}))$.Then, since X is the frontier of N and N is sibling-closed we have $\mathit{var}(N) = \mathit{var}(X) = \mathit{var}(\mathit{hyp}(\mathcal{I}))$ by Lemma 9. □

Next, we present the following soundness corollary.

Corollary 1 (Soundness)**.** *Let Q be a GCQ. Assume that $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\downarrow$.*

1. *If $\mathcal{J}$ is the empty triplet then Q is acyclic.*
2. *If $\mathcal{J}$ is the empty triplet and $\mathit{var}(\mathit{hyp}(\mathcal{I})) = \mathit{out}(Q)$ then Q is free-connex acyclic.*

To also show completeness, we will interpret a GJT T for a GCQ Q as a "parse tree" that specifies the two-stage sequence of reduction steps that can be done on $\mathcal{H}(Q)$ to reach the empty triplet. Not all GJTs will allows us to do so easily, however, and we will therefore restrict our attention to those GJTs that are *canonical*.

Definition 32 (Canonical)**.** A GJT T is *canonical* if:

1. its root is labeled by $\emptyset$;
2. every leaf node n is the child of an internal node m with $\mathit{var}(n) = \mathit{var}(m)$;
3. for all internal nodes n and m with $n \neq m$ we have $\mathit{var}(n) \neq \mathit{var}(m)$; and

4. for every edge $m \to n$ and all $\theta \in \mathit{pred}(m \to n)$ we have $\mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) \neq \emptyset$.

A connex subset N of T is *canonical* if every node in it is interior in T. A GJT pair (T, N) is canonical if both T and N are canonical.

The following proposition shows that we may restrict our attention to canonical GJT pairs without loss of generality.

Proposition 23. *For every GJT pair there exists an equivalent canonical pair.*

Proof. Let T be a GJT. The proof proceeds in three steps. *Step 1.* Let T_1 be the GJT obtained from T by (i) removing all predicates from T, and (ii) creating a new root node r that is labeled by $\emptyset$ and attaching the root of T to it, labeled by the empty set of predicates. T_1 satisfies the first canonicality condition, but is not equivalent to T because it has none of T's predicates. Now re-add the predicates in T to T_1 as follows. For each edge $m \to n$ in T and each predicate $\theta \in \mathit{pred}_T(m \to n)$, if $\mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) \neq \emptyset$ then add θ to $\mathit{pred}_{T_1}(m \to n)$. Otherwise, if $\mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) = \emptyset$, do the following. First observe that, by definition of GJTs, $\mathit{var}(\theta) \subseteq \mathit{var}(n) \cup \mathit{var}(m)$. Because $\mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) = \emptyset$ this implies $\mathit{var}(\theta) \subseteq \mathit{var}(m)$. Because we disallow nullary predicates, $\mathit{var}(m) \neq \emptyset$. Let a be the first ancestor of m in T_1 such that $\mathit{var}(\theta) \not\subseteq \mathit{var}(a)$. Such an ancestor exists because the root of T_1 is labeled $\emptyset$. Let b be the child of a in T_1. Since a is the first ancestor of m with $\mathit{var}(\theta) \not\subseteq \mathit{var}(a)$, $\mathit{var}(\theta) \subseteq \mathit{var}(b)$. Therefore, $\mathit{var}(\theta) \subseteq \mathit{var}(b) \cup \mathit{var}(a)$ and $\mathit{var}(\theta) \cap (\mathit{var}(b) \setminus \mathit{var}(a)) \neq \emptyset$. As such, add θ to $\mathit{pred}_{T_1}(a \to b)$. After having done this for all predicates in T, T_1 becomes equivalent to T, and satisfies canonicality conditions (1) and (3). Then take take $N_1 = N \cup \{r\}$. Clearly, N_1 is a connex subset of T_1 and $\mathit{var}(N) = \mathit{var}(N')$. Therefore, (T_1, N_1) is equivalent to (T, N).

Step 2. Let T_2 be obtained from T_1 by adding, for each leaf node l in T_1 a new interior node n_l labeled by $\mathit{var}(l)$ and inserting it in-between l and its parent in T_1. I.e., if l has parent p in T_1 then we have $p \to n_l \to l$ in T_2 with $\mathit{pred}_{T_2}(p \to n_l) = \mathit{pred}_{T_1}(p \to n)$ and $\mathit{pred}_{T_2}(n_l \to l) = \emptyset$.[2] Furthermore, let N_2 be the connex subset of T_2 obtained by replacing every leaf node l in N_1 by its newly inserted node n_l. Clearly, $\mathit{var}(N_2) = \mathit{var}(N_1) = \mathit{var}(N)$ because $\mathit{var}(l) = \mathit{var}(n_l)$ for every leaf l of T_1. By our construction, (T_2, N_2) is equivalent to (T, N); T_2 satisfies canonicality conditions (1), (2), and (4); and N_2 is canonical.

Step 3. It remains to enforce condition (3). To this end, observe that, by the connectedness condition of GJTs, T_2 violates canonicality condition (3) if and only if there exist internal nodes m and n where m is the parent of n such that $\mathit{var}(m) = \mathit{var}(n)$. In this case, we call n a culprit node. We will now show how to obtain an equivalent pair (U, M) that removes a single culprit node; the final result is then obtained by iterating this reasoning until all culprit nodes have been removed.

The culprit removal procedure is essentially the reverse of the binarization procedure of Fig. 4.5. Concretely, let n be a culprit node with parent m and let $n_1, \ldots, n_k$ be the children of n in T_2. Let U be the GJT obtained from T_2 by removing n and attaching all children n_i of n as children to m with edge label $\mathit{pred}_U(m \to n_i) = \mathit{pred}_{T_2}(n \to n_i)$, for $1 \leq i \leq k$. Because $\mathit{var}(n) = \mathit{var}(m)$, the result is still a valid GJT. Moreover, because $\mathit{var}(n) = \mathit{var}(m)$ and T_2 satisfied condition (4), we had $\mathit{pred}_{T_2}(m \to n) = \emptyset$, so no predicate was lost by the removal of n. Finally, define M as follows. If $n \in N_2$, then set $M = N_2 \setminus \{n\}$, otherwise set $M = N_2$. In the former case, since N_2 is connex and $n \in N_2$, m must also be in N_2. It is hence in M. Therefore, in both cases, $\mathit{var}(N) = \mathit{var}(N_2) = \mathit{var}(M)$. Furthermore, it is straightforward to check that M is a connex subset of U. Finally, since N_2 consisted only of interior nodes of T_2, M consists only of interior nodes of U and hence remains canonical. □

[2] Note that all leafs have a parent since the root of T_1 is an interior node labeled by $\emptyset$.

We also require the following auxiliary notions and insights. First, if (T, N) is a GJT pair, then define the *hypergraph associated to* (T, N), denoted $hyp(T, N)$, to be the hypergraph formed by node labels in N,

$$hyp(T, N) = \{var_T(n) \mid n \in N, var_T(n) \neq \emptyset\}.$$

Further, define $pred(T, N)$ to be the set of all predicates occurring on edges between nodes in N. For a hyperedge $\overline{z}$, define the *hypergraph triplet* of (T, N) w.r.t. $\overline{z}$, denoted $\mathcal{H}(T, N, \overline{z})$ to be the hypergraph triplet $(hyp(T, N), \overline{z}, pred(T, N))$.

The following technical Lemma shows that we can use canonical pairs as "parse" trees to derive a sequence of reduction steps.

Lemma 16. *Let (T, N_1) and (T, N_2) be canonical GJT pairs with $N_2 \subseteq N_1$. Then*

$$\mathcal{H}(T, N_1, \overline{z}) \overset{*}{\rightsquigarrow} \mathcal{H}(T, N_2, \overline{z})$$

for every $\overline{z} \subseteq var(N_2)$.

To prove this lemma, we first require a number of auxiliary results.

We first make the following observations (i.e. lemma 17, lemma 18, and lemma 19) regarding canonical GJT pairs.

Lemma 17. *Let (T, N) be a canonical GJT pair, let n be a frontier node of N and let m be the parent of n in T.*

1. *$x \notin var(N \setminus \{n\})$, for every $x \in var(n) \setminus var(m)$.*
2. *$hyp(T, N \setminus \{n\}) = hyp(T, N) \setminus \{var(n)\})$.*
3. *$\theta \notin pred(m \to n)$, for every $\theta \in pred(T, N \setminus \{n\})$*
4. *$pred(T, N \setminus \{n\}) = pred(T, N) \setminus pred(m \to n)$.*
5. *$pred(m \to n) = \{\theta \in pred(T, N) \mid var(\theta) \cap (var(n) \setminus var(m)) \neq \emptyset\}$.*
6. *$pred(T, N \setminus \{n\}) = \{\theta \in pred(T, N) \mid var(\theta) \cap (var(n) \setminus var(m)) = \emptyset\}$.*

Proof. (1) Let $x \in var(n) \setminus var(m)$ and let c be a node in $N \setminus \{n\}$. Clearly the unique undirected path between c and n in T must pass through m. Because $x \notin var(m)$ it follows from the connectedness condition of GJTs that also $x \notin var(c)$. As such, $x \notin var(N \setminus \{n\})$.

(2) The $\supseteq$ direction is trivial. For the $\subseteq$ direction, assume that $m \in N \setminus \{n\}$ with $var(m) \neq \emptyset$. Then clearly $m \in N$ and hence $var(m) \in hyp(T, N)$. Furthermore, because N is canonical, both m and n are interior nodes in T. Then, because T is canonical and $m \neq n$ we have $var(m) \neq var(n)$. Therefore, $var(m) \in hyp(T, N) \setminus \{var(n)\}$.

(3) Let $\theta \in pred(T, N \setminus n)$. Then θ occurs on the edge between two nodes in $N \setminus n$, say $m' \to n'$. By definition of GJTs, $var(\theta) \subseteq var(n') \cup var(m') \subseteq var(N \setminus \{n\})$. Now suppose for the purpose of contradiction that also $\theta \in pred(m \to n)$. Because T is nice, there is some $x \in var(\theta) \cap (var(n) \setminus var(m)) \neq \emptyset$. Hence, by (1), $x \notin var(N \setminus \{n\})$, which contradicts $var(\theta) \subseteq var(N \setminus \{n\})$.

(4) Clearly, $pred(T, N) \setminus pred(m \to n) \subseteq pred(T, N \setminus \{n\})$. The converse inclusion follows from (3).

(5) The $\subseteq$ direction follows from the fact that m and n are in N, and T is nice. To also see $\supseteq$, let $\theta \in \mathit{pred}(T, N)$ with $\mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) \neq \emptyset$. There exists $x \in \mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m))$. By (1), $x \notin \mathit{var}(N \setminus \{n\})$. Therefore, θ cannot occur between edges in $N \setminus \{n\}$ in T. Since it nevertheless occurs in $\mathit{pred}(T, N)$, it must hence occur in $\mathit{pred}(m \to n)$.

(6) Follows directly from (4) and (5). □

Lemma 18. *Let (T, N) be a canonical GJT pair, let n be a frontier node of N and let m be the parent of n in T. Let $\overline{z} \subseteq \mathit{var}(N \setminus \{n\})$.*

1. $\mathit{var}(n) \sqsubseteq_{\mathcal{H}(T,N,\overline{z})} \mathit{var}(m)$.
2. $x \notin \mathit{jv}(\mathcal{H}(T, N, \overline{z}))$, *for every* $x \in (\mathit{var}(n) \setminus \mathit{var}(m))$.

Proof. For reasons of parsimony, let $\mathcal{H} = \mathcal{H}(T, N, \overline{z})$. We first prove (2) and then (1).

(2) Let $x \in \mathit{var}(n) \setminus \mathit{var}(m)$. By Lemma 17(1), $x \notin \mathit{var}(N \setminus \{n\})$. Therefore, x occurs in $\mathit{var}(n)$ in $\mathcal{H}$ and in no other hyperedge. Furthermore, because $\overline{z} \subseteq \mathit{var}(N \setminus \{n\})$, also $x \notin \overline{z}$. Hence $x \notin \mathit{jv}_{\mathcal{H}}(\mathit{var}(n))$.

(1) We need to show that $\mathit{jv}_{\mathcal{H}}(\mathit{var}(n)) \subseteq \mathit{var}(m)$ and $\mathit{ext}_{\mathcal{H}}(\mathit{var}(n) \setminus \mathit{var}(m)) \subseteq \mathit{var}(m)$. Let $x \in \mathit{jv}_{\mathcal{H}}(\mathit{var}(n))$. By contraposition of (2), we know that $x \notin (\mathit{var}(n) \setminus \mathit{var}(m))$. Therefore, $x \in \mathit{var}(m)$ and thus $\mathit{jv}_{\mathcal{H}}(\mathit{var}(n)) \subseteq \mathit{var}(m)$. To show $\mathit{ext}_{\mathcal{H}}(\mathit{var}(n) \setminus \mathit{var}(m)) \subseteq \mathit{var}(m)$, let $y \in \mathit{ext}_{\mathcal{H}}(\mathit{var}(n) \setminus \mathit{var}(m))$. Then $y \notin \mathit{var}(n) \setminus \mathit{var}(m)$ and there exists $\theta \in \mathit{pred}(T, N)$ with $\mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) \neq \emptyset$ and $y \in \mathit{var}(\theta)$. By Lemma 17(5), $\theta \in \mathit{pred}_T(m \to n)$. Thus, $y \in \mathit{var}(m) \cup \mathit{var}(n)$. Since also $y \notin \mathit{var}(n) \setminus \mathit{var}(m)$, it follows that $y \in \mathit{var}(m)$. Therefore, $\mathit{ext}_{\mathcal{H}}(\mathit{var}(n) \setminus \mathit{var}(m)) \subseteq \mathit{var}(m)$. □

Lemma 19. *Let (T, N) be a canonical GJT pair and let n be a frontier node of N. Then*

$$\mathcal{H}(T, N, \overline{z}) \overset{*}{\rightsquigarrow} \mathcal{H}(T, N \setminus \{n\}, \overline{z})$$

for every $\overline{z} \subseteq \mathit{var}(N \setminus \{n\})$.

Proof. For reasons of parsimony, let us abbreviate $\mathcal{H}_1 = \mathcal{H}(T, N, \overline{z})$ and $\mathcal{H}_2 = \mathcal{H}(T, N \setminus \{n\}, \overline{z})$. We make the following case analysis.

Case (1): Node n is the root in N. Because the root of a canonical tree is labeled by $\emptyset$ we have $\mathit{var}(n) = \emptyset$. Since n is a frontier node of N, $N = \{n\}$. Thus, $\mathit{hyp}(T, N) = \emptyset$ and $\mathit{hyp}(T, N \setminus \{n\}) = \emptyset$. Furthermore, $\mathit{pred}(T, N) = \mathit{pred}(T, N \setminus \{n\}) = \emptyset$ and $\overline{z} \subseteq \mathit{var}(N \setminus \{n\}) = \mathit{var}(\emptyset) = \emptyset$. As such, both $\mathcal{H}_1$ and $\mathcal{H}_2$ are the empty triplet $(\emptyset, \emptyset, \emptyset)$. Therefore $\mathcal{H}_1 \rightsquigarrow^* H_2$.

Case (2): n has parent m in N and $\mathit{var}(m) \neq \emptyset$. Then $\mathit{var}(n) \neq \emptyset$ since in a canonical tree the root node is the only interior node that is labeled by the empty hyperedge. Therefore, $\mathit{var}(n) \in \mathit{hyp}(T, N)$, $\mathit{var}(m) \in \mathit{hyp}(T, N)$, and $\mathit{var}(n) \sqsubseteq_{\mathcal{H}_1} \mathit{var}(m)$ by Lemma 18(1). We can hence apply reduction (CSE) to remove $\mathit{var}(n)$ from $\mathit{hyp}(\mathcal{H}_1)$ and all predicates that intersect with $\mathit{var}(n) \setminus \mathit{var}(m)$ from $\mathit{pred}(\mathcal{H}_1)$. By Lemma 17(2) and 17(6) the result is exactly $\mathcal{H}_2$:

$$\begin{aligned} &\mathit{hyp}(\mathcal{H}_2) \\ &= \mathit{hyp}(T, N \setminus \{n\}) \end{aligned}$$

$$= \mathit{hyp}(T, N) \setminus \{\mathit{var}(n)\} = \mathit{hyp}(\mathcal{H}_1) \setminus \{\mathit{var}(n)\}$$

$$\begin{aligned}
&\mathit{pred}(\mathcal{H}_2)\\
&= \mathit{pred}(T, N \setminus \{n\})\\
&= \{\theta \in \mathit{pred}(T, N) \mid \mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) = \emptyset\}\\
&= \{\theta \in \mathit{pred}(\mathcal{H}_1) \mid \mathit{var}(\theta) \cap (\mathit{var}(n) \setminus \mathit{var}(m)) = \emptyset\}
\end{aligned}$$

Case (3): n has parent m in N and $\mathit{var}(m) = \emptyset$. Then $\mathit{var}(n) \neq \emptyset$ since since in a canonical tree the root node is the only interior node that is labeled by the empty hyperedge. By definition of GJTs, it follows that for every $\theta \in \mathit{pred}(m \to n)$ we have $\mathit{var}(\theta) \subseteq \mathit{var}(n) \cup \mathit{var}(m) = \mathit{var}(n)$. In other words: all $\theta \in \mathit{pred}(m \to n)$ are filters. As such, we can use reduction (FLT) to remove all predicates in $\mathit{pred}(m \to n)$ from $\mathcal{H}_1$. This yields a triplet $\mathcal{I}$ with the same hypergraph as $\mathcal{H}_1$, same set of output variables as $\mathcal{H}_1$, and

$$\begin{aligned}
\mathit{pred}(\mathcal{I}) &= \mathit{pred}(\mathcal{H}_1) \setminus \mathit{pred}_T(m \to n)\\
&= \mathit{pred}(T, N) \setminus \mathit{pred}_T(m \to n)\\
&= \mathit{pred}(T, N \setminus \{n\}) = \mathit{pred}(\mathcal{H}_2),
\end{aligned}$$

where the third equality is due to Lemma 17(4). We claim that every variable in e is isolated in $\mathcal{I}$. From this the result follows, because then we can apply (ISO) to remove the entire hyperedge $\mathit{var}(e)$ from $\mathit{hyp}(\mathcal{I}) = \mathit{hyp}(\mathcal{H}_1)$ while preserving $\mathit{out}(\mathcal{I})$ and $\mathit{pred}(\mathcal{I})$. The resulting triplet hence equals $\mathcal{H}_2$. To see that $e \subseteq \mathit{isol}(\mathcal{I})$, observe that no predicate in $\mathit{pred}(\mathcal{I}) = \mathit{pred}(T, N \setminus \{n\})$ shares a variable with $\mathit{var}(n) = (\mathit{var}(n) \setminus \mathit{var}(m))$ by Lemma 17(6). Therefore $\mathit{var}(n) \cap \mathit{var}(\mathit{pred}(\mathcal{I})) = \emptyset$. Furthermore, $\mathit{var}(n) \cap \mathit{jv}(\mathcal{I}) = \emptyset$ because $\mathit{jv}(\mathcal{I}) = \mathit{jv}(\mathcal{H}_1)$ and no $x \in \mathit{var}(n) = \mathit{var}(n) \setminus \mathit{var}(m)$ is in $\mathit{jv}(\mathcal{H}_1)$ by Lemma 18(2). □

Now we present the proof of lemma 16 (restated) by induction in the following.

Lemma 16. *Let (T, N_1) and (T, N_2) be canonical GJT pairs with $N_2 \subseteq N_1$. Then*

$$\mathcal{H}(T, N_1, \overline{z}) \overset{*}{\rightsquigarrow} \mathcal{H}(T, N_2, \overline{z})$$

for every $\overline{z} \subseteq \mathit{var}(N_2)$.

Proof. By induction on k, the number of nodes in $N_1 \setminus N_2$. In the base case where $k = 0$, the result trivially holds since then $N_1 = N_2$ and the two triplets are identical. For the induction step, assume that $k > 0$ and the result holds for $k - 1$. Because both N_1 and N_2 are connex subsets of the same tree T, there exists a node $n \in N_1$ that is a frontier node in N_1, and which is not in N_2. Then define $N_1' = N_1 \setminus \{n\}$. Clearly (T, N_1') is again canonical, and $|N_1' \setminus N_2| = k - 1$. Therefore, $\mathcal{H}(T, N_1', \overline{z}) \rightsquigarrow^* \mathcal{H}(T, N_2, \overline{z})$ by induction hypothesis. Furthermore, by $\mathcal{H}(T, N_1, \overline{z}) \rightsquigarrow^* \mathcal{H}(T, N_1', \overline{z})$ by Lemma 19, from which the result follows. □

We will also require the following insight for completeness.

Lemma 20. *Let H_1 and H_2 be two hypergraphs such that for all $e \in H_2$ there exists $f \in H_1$ such that $e \subseteq f$. Then $(H_1 \cup H_2, \overline{z}, \Theta) \rightsquigarrow^* (H_1, \overline{z}, \Theta)$, for every hyperedge $\overline{z}$ and set of predicates Θ.*

Proof. The proof is by induction on k, the number of hyperedges in $H_2 \setminus H_1$. In the base case where $k = 0$, the result trivially holds since $H_1 \cup H_2 = H_1$ and the two triplets are hence identical. For the induction step, assume that $k > 0$ and the result holds for $k - 1$. Fix some $e \in H_2 \setminus H_1$ and define $H_2' = H_2 \setminus \{e\}$. Then $|H_2' \setminus H_1| = k - 1$. We show that $(H_1 \cup H_2, \overline{z}, \Theta) \rightsquigarrow^* (H_1 \cup H_2', \overline{z}, \Theta)$, from which the result follows since $(H_1 \cup H_2', \overline{z}, \Theta) \rightsquigarrow^* (H_1, \overline{z}, \Theta)$ by induction hypothesis. To this end, we observe that there exists $f \in H_1 \setminus \{e\}$ with $e \subseteq f$. Therefore, $jv_{(H_1 \cup H_2, \overline{z}, \Theta)}(e) \subseteq e \subseteq f$. Moreover, $e \setminus f = \emptyset$. Therefore, $ext_{(H_1 \cup H_2, \overline{z}, \Theta)}(e \setminus f) = \emptyset \subseteq f$. Thus $e \sqsubseteq_{(H_1 \cup H_2, \overline{z}, \Theta)} f$. We may therefore apply (CSE) to remove e from $H_1 \cup H_2$, yielding $H_1 \cup H_2'$. Since no predicate shares variables with $e \setminus f = \emptyset$ this does not modify Θ. Therefore, $(H_1 \cup H_2, \overline{z}, \Theta) \rightsquigarrow^* (H_1 \cup H_2', \overline{z}, \Theta)$. □

With these tools in hand, we are ready to prove completeness.

Proposition 24. *Let Q be a GCQ, let T be a GJT for Q and let N be a connex subset of T with $out(Q) \subseteq var(N)$. Assume that $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\!\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\!\downarrow$. Then $\mathcal{J}$ is the empty triplet and $var(hyp(\mathcal{I})) \subseteq var(N)$.*

Proof. By Proposition 23 we may assume without loss of generality that (T, N) is a canonical GJT pair. Let A be the set of all of T's interior nodes. Clearly, A is a connex subset of T and $var(A) \subseteq var(Q)$. Furthermore, because for every atom $r(\overline{x})$ in Q there is a leaf node l in T labeled by $r(\overline{x})$ (as T is a GJT for Q), which has a parent interior node n_l labeled $\overline{x}$ (because T is canonical), also $var(Q) \subseteq var(A)$. Therefore, $var(A) = var(Q)$. By the same reasoning, $hyp(Q) \subseteq hyp(T, A)$. Therefore, $hyp(T, A) = hyp(T, A) \cup hyp(Q)$. Furthermore, because every interior node in a GJT has a guard descendant, and the leaves of T are all labeled by atoms in Q, we know that for every node $n \in A$ there exists some hyperedge $f \in hyp(Q)$ such that $var(n) \subseteq var(f)$. In addition, we claim that $pred(T, A) = pred(Q)$. Indeed, $pred(T, A) \subseteq pred(Q)$ since T is a GJT for Q. The converse inclusion follows from canonicality properties (2) and (4): because leaf nodes in a canonical GJT have a parent labeled by the same hyperedge, there can be no predicates on edges to leaf nodes in T. Thus, all predicates in T are on edges between interior nodes, i.e., in $pred(T, A)$. Then, because every predicate in Q appears somewhere in T (since T is a GJT for Q), we have $pred(Q) \subseteq pred(T, A)$. From all of the observations made so far and Lemma 20, we obtain:

$$\begin{aligned}
\mathcal{H}(T, A, out(Q)) &= (hyp(T, A), out(Q), pred(T, A)) \\
&= (hyp(T, A) \cup hyp(Q), out(Q), pred(T, A)) \\
&\overset{*}{\rightsquigarrow} (hyp(Q), out(Q), pred(T, A)) \\
&= (hyp(Q), out(Q), pred(Q)) \\
&= \mathcal{H}(Q)
\end{aligned}$$

Thus $\mathcal{H}(T, A, out(Q)) \rightsquigarrow^* \mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}$. Furthermore, because (T, N) is also canonical with $N \subseteq A$ and $out(Q) \subseteq var(N)$ we have $\mathcal{H}(T, A, out(Q)) \rightsquigarrow^* \mathcal{H}(T, N, out(Q))$ by Lemma 16. Then, because reduction is confluent (Proposition 21) we obtain that $\mathcal{H}(T, N, out(Q))$ and $\mathcal{I}$ can be reduced to the same triplet. Because $\mathcal{I}$ is in normal form, necessarily $\mathcal{H}(T, N, out(Q)) \rightsquigarrow^* \mathcal{I}$. Since reduction steps can only remove nodes and hyperedges (and never add them), $var(hyp(\mathcal{I})) \subseteq var(N)$.

It remains to show that $\mathcal{J}$ is the empty triplet. Hereto, we first verify the following. For any two hypergraph triplets $\mathcal{U}$ and $\mathcal{V}$, if $\mathcal{U} \rightsquigarrow^* \mathcal{V}$ then also $\tilde{\mathcal{U}} \rightsquigarrow^* \tilde{\mathcal{V}}$. From this, $\mathcal{H}(T, A, out(Q)) \rightsquigarrow^* \mathcal{I}$, and the fact that $\mathcal{H}(T, A, \emptyset)$ is the residual of $\mathcal{H}(T, A, out(Q))$ we conclude $\mathcal{H}(T, A, \emptyset) \rightsquigarrow^* \tilde{\mathcal{I}}$. Then, because $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}$, it follows that $\mathcal{H}(T, A, \emptyset) \rightsquigarrow^* \mathcal{J}$. Let r be T's root node, which is labeled by $\emptyset$ since T in canonical. Then clearly $\{r\}$ is a connex subset of T. By Lemma 16, $\mathcal{H}(T, A, \emptyset) \rightsquigarrow^* \mathcal{H}(T, \{r\}, \emptyset)$. Now observe that the hypergraph of $\mathcal{H}(T, \{r\}, \emptyset)$ is empty, and its predicate set is also empty. Therefore, $\mathcal{H}(T, \{r\}, \emptyset)$ is the empty hypergraph triplet. In particular, it is in normal form. But, since $\mathcal{J}$ is also in normal form and normal forms are unique, $\mathcal{J}$ must also be the empty triplet. □

Given the above discussion, now we present the following completeness corollary.

Corollary 2 (Completeness)**.** *Let Q be a GCQ. Assume that $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\downarrow$.*

1. *If Q is acyclic, then $\mathcal{J}$ is the empty triplet.*
2. *If Q is free-connex acyclic, then $\mathcal{J}$ is the empty triplet and $var(hyp(\mathcal{I})) = out(Q)$.*
3. *For every GJT T of Q and every connex subset N of T it holds that $var(hyp(\mathcal{I})) \subseteq var(N)$.*

Proof. (1) Since Q is acyclic, there exists a GJT T for Q. Let N be the set of all of T's nodes. Then N is a connex subset of T and $out(Q) \subseteq var(N) = var(Q)$. The result then follows from Proposition 24.

(2) Since Q is free-connex acyclic, there exists a GJT pair (T, N) compatible with Q. In particular, $var(N) = out(Q)$. By Proposition 24, $\mathcal{J}$ is the empty triplet, and $var(hyp(\mathcal{I})) \subseteq var(N) = out(Q)$. It remains to show $out(Q) \subseteq var(hyp(\mathcal{I}))$. First verify the following: A reduction step on a hypergraph triplet $\mathcal{H}$ never removes any variable in $out(\mathcal{H})$ from $hyp(\mathcal{H})$, nor does it modify $out(\mathcal{H})$. Then, since $out(\mathcal{H}(Q)) = out(Q) \subseteq var(Q) \subseteq var(hyp(\mathcal{H}(Q))))$, and $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}$ we obtain $out(Q) \subseteq var(hyp(\mathcal{I}))$.

(3) Follows directly from Proposition 24. □

Now we present the proofs of theorems 6 and 7

Theorem 6. *Let Q be a GCQ. Assume $\mathcal{H}(Q) \rightsquigarrow^* \mathcal{I}\downarrow$ and $\tilde{\mathcal{I}} \rightsquigarrow^* \mathcal{J}\downarrow$. Then the following hold.*

1. *Q is acyclic if, and only if, $\mathcal{J}$ is the empty triplet.*
2. *Q is free-connex acyclic if, and only if, $\mathcal{J}$ is the empty triplet and $var(hyp(\mathcal{I})) = out(Q)$.*
3. *For every GJT T of Q and every connex subset N of T it holds that $var(hyp(\mathcal{I})) \subseteq var(N)$.*

Theorem 6 follows directly from Corollaries 1 and 2.

Theorem 7. *Given a GCQ Q, Algorithm 8 reports an error if Q is cyclic. Otherwise, it returns a sibling-closed GJT pair (T, N) with T a GJT for Q. If Q is free-connex acyclic, then (T, N) is compatible with Q. Otherwise, $out(Q) \subsetneq var(N)$, but $var(N)$ is minimal in the sense that for every other GJT pair (T', N') with T' a GJT for Q we have $var(N) \subseteq var(N')$.*

Theorem 7 follows from Theorem 6 and Proposition 22.

5.4 Proof of Confluence of GYO for GCQs

In this subsection, we present the proof of the confluence of the GYO-reduction for GCQs i.e. the proof of proposition 21. Because no infinite sequences of reduction steps are possible, it suffices to demonstrate local confluence:

Proposition 25. *If $\mathcal{H} \rightsquigarrow \mathcal{I}_1$ and $\mathcal{H} \rightsquigarrow \mathcal{I}_2$ then there exists $\mathcal{J}$ such that both $\mathcal{I}_1 \rightsquigarrow^* \mathcal{J}$ and $\mathcal{I}_2 \rightsquigarrow^* \mathcal{J}$.*

Indeed, it is a standard result in the theory of rewriting systems that confluence (Lemma 21) and local confluence (lemma 25) coincide when infinite sequences of reductions steps are impossible

Before proving Lemma 25, we observe that the property of being isolated or being a conditional subset is preserved under reductions, in the following sense.

Lemma 21. *Assume that $\mathcal{H} \rightsquigarrow \mathcal{I}$. Then $pred(\mathcal{I}) \subseteq pred(\mathcal{H})$ and for every hyperedge e we have $ext_{\mathcal{I}}(e) \subseteq ext_{\mathcal{H}}(e)$, $jv_{\mathcal{I}}(e) \subseteq jv_{\mathcal{H}}(e)$, and $isol_{\mathcal{H}}(e) \subseteq isol_{\mathcal{I}}(e)$. Furthermore, if $e \sqsubseteq_{\mathcal{H}} f$ then also $e \sqsubseteq_{\mathcal{I}} f$.*

Proof. First observe that $pred(\mathcal{I}) \subseteq pred(\mathcal{H})$, since reduction operators only remove predicates. This implies that $ext_{\mathcal{I}}(e) \subseteq ext_{\mathcal{H}}(e)$ for every hyperedge e. Furthermore, because reduction operators only remove hyperedges and never add them, it is easy to see that $jv_{\mathcal{H}}(e) \subseteq jv_{\mathcal{I}}(e)$. Hence, if $x \in isol_{\mathcal{H}}(e)$ then $x \notin jv_{\mathcal{H}}(e) \supseteq jv_{\mathcal{I}}(e)$ and $x \notin var(pred(\mathcal{H})) \supseteq var(pred(\mathcal{I}))$. Therefore, $x \in isol_{\mathcal{I}}(e)$. As such, $isol_{\mathcal{I}}(e) \subseteq isol_{\mathcal{H}}(e)$.

Next, assume that $e \sqsubseteq_{\mathcal{H}} f$. We need to show that $jv_{\mathcal{I}}(e) \subseteq f$ and $ext_{\mathcal{I}}(e \setminus f) \subseteq f$. The first condition follows since $jv_{\mathcal{I}}(e) \subseteq jv_{\mathcal{H}}(e) \subseteq f$ where the last inclusion is due to $e \sqsubseteq_{\mathcal{H}} f$. The second also follows since $ext_{\mathcal{I}}(e \setminus f) \subseteq ext_{\mathcal{H}}(e \setminus f) \subseteq f$ where the last inclusion is due to $e \sqsubseteq_{\mathcal{H}} f$. □

Proof of Proposition 25. If $\mathcal{I}_1 = \mathcal{I}_2$ then it suffices to take $\mathcal{J} = \mathcal{I}_1 = \mathcal{I}_2$. Therefore, assume in the following that $\mathcal{I}_1 \neq \mathcal{I}_2$. Then, necessarily $\mathcal{I}_1$ and $\mathcal{I}_2$ are obtained by applying two different reduction operations on $\mathcal{H}$. We make a case analysis on the types of reductions applied.

(1) Case (ISO, ISO): assume that $\mathcal{I}_1$ is obtained by removing the non-empty set $X_1 \subseteq isol_{\mathcal{H}}(e_1)$ from hyperedge e_1, while $\mathcal{I}_2$ is obtained by removing non-empty $X_2 \subseteq isol_{\mathcal{H}}(e_2)$ from e_2 with $X_1 \neq X_2$. There are two possibilities.

(1a) $e_1 \neq e_2$. Then e_2 is still a hyperedge in $\mathcal{I}_2$ and e_1 is still a hyperedge in $\mathcal{I}_1$. By Lemma 21, $isol_{\mathcal{H}}(e_1) \subseteq isol_{\mathcal{I}_2}(e_1)$ and $isol_{\mathcal{H}}(e_2) \subseteq isol_{\mathcal{I}_1}(e_2)$. Therefore, we can still remove X_2 from $\mathcal{I}_1$ by means of rule ISO, and similarly remove X_1 from $\mathcal{I}_2$. Let $\mathcal{J}_1$ (resp. $\mathcal{J}_2$) be the result of removing X_2 from $\mathcal{I}_1$ (resp. $\mathcal{I}_2$). Then $\mathcal{J}_1 = \mathcal{J}_2$ (and hence equals triplet $\mathcal{J}$):

$$\begin{aligned} hyp(\mathcal{J}_1) &= hyp(\mathcal{H}) \setminus \{e_1, e_2\} \cup \{e_1 \setminus X_1 \mid e_1 \setminus X_1 \neq \emptyset\} \quad \cup \{e_2 \setminus X_2 \mid e_2 \setminus X_2 \neq \emptyset\} \\ &= hyp(\mathcal{J}_2) \\ pred(\mathcal{J}_1) &= pred(\mathcal{H}) = pred(\mathcal{J}_2) \end{aligned}$$

(1b) $e_1 = e_2$. We show that $X_2 \setminus X_1 \subseteq isol_{\mathcal{I}_1}(e_1 \setminus X_1)$ and similarly $X_1 \setminus X_2 \subseteq isol_{\mathcal{I}_1}(e_2 \setminus X_1)$. This suffices because we can then apply ISO to remove $X_2 \setminus X_1$ from $\mathcal{I}_1$ and $X_1 \setminus X_2$ from $\mathcal{I}_2$. In both cases, we reach the same triplet as removing $X_1 \cup X_2 \subseteq isol_{\mathcal{H}}(e_1)$ from $\mathcal{H}$.[3]

To see that $X_2 \setminus X_1 \subseteq isol_{\mathcal{I}_1}(e_1 \setminus X_1)$, let $x \in X_2 \setminus X_1$. We need to show $x \notin jv_{\mathcal{I}_1}(e_1 \setminus X_1)$ and $x \notin var(pred(\mathcal{I}_1))$. Because $x \in X_2 \subseteq isol_{\mathcal{H}}(e_1)$ we know $x \notin jv_{\mathcal{H}}(e_1)$. Then, since $x \notin X_1$, also $x \notin jv_{\mathcal{H}}(e_1 \setminus X_1)$. By Lemma 21, $jv_{\mathcal{I}_1}(e_1 \setminus X_1) \subseteq jv_{\mathcal{H}}(e_1 \setminus X_1)$. Therefore, $x \notin jv_{\mathcal{I}_1}(e_1 \setminus X_1)$. Furthermore, because $x \in isol_{\mathcal{H}}(e_1)$ we know $x \notin var(pred(\mathcal{H}))$. Since $var(pred(\mathcal{I}_1)) \subseteq var(pred(\mathcal{H}))$ by Lemma 21, also x *not* $\in var(pred(\mathcal{I}_1))$.

[3]Should $X_2 \setminus X_1$ be empty, we don't actually need to do anything on $\mathcal{I}_1$: $X_1 \cup X_2$ is already removed from it. A similar remark holds for $\mathcal{I}_2$ when $X_1 \setminus X_2$ is empty.

$X_1 \setminus X_2 \subseteq \mathit{isol}_{\mathcal{I}_1}(e_2 \setminus X_1)$ is shown similarly.

(2) Case (CSE, CSE): assume that $\mathcal{I}_1$ is obtained by removing hyperedge e_1 because it is a conditional subset of hyperedge f_1, while $\mathcal{I}_2$ is obtained by removing e_2, conditional subset of f_2. Since $\mathcal{I}_1 \neq \mathcal{I}_2$ it must be $e_1 \neq e_2$. We need to further distinguish the following cases.

(2a) $e_1 \neq f_2$ and $e_2 \neq f_1$. In this case, e_2 and f_2 remain hyperedges in $\mathcal{I}_1$ while e_1 and f_1 remain hyperedges in $\mathcal{I}_2$. Then, by Lemma 21, $e_2 \sqsubseteq_{\mathcal{I}_1} f_2$ and $e_1 \sqsubseteq_{\mathcal{I}_2} f_2$. Let $\mathcal{J}_1$ (resp. $\mathcal{J}_2$) be the triplet obtained by removing e_2 from $\mathcal{I}_1$ (resp. e_1 from $\mathcal{I}_2$). Then $\mathcal{J}_1 = \mathcal{J}_2$ since clearly $\mathit{out}(\mathcal{J}_1) = \mathit{out}(\mathcal{J}_2)$ and

$$\mathit{hyp}(\mathcal{J}_1) = \mathit{hyp}(\mathcal{H}) \setminus \{e_1, e_2\} = \mathit{hyp}(\mathcal{J}_2)$$
$$\mathit{pred}(\mathcal{J}_1) = \{\theta \in \mathit{pred}(\mathcal{H}) \mid \mathit{var}(\theta) \cap (e_1 \setminus f_1) = \emptyset, \qquad \mathit{var}(\theta) \cap (e_2 \setminus f_2) = \emptyset\} = \mathit{pred}(\mathcal{J}_2)$$

From this the result follows by taking $\mathcal{J} = \mathcal{J}_1 = \mathcal{J}_2$.

(2b) $e_1 \neq f_2$ but $e_2 = f_1$. Then $e_1 \sqsubseteq_{\mathcal{H}} e_2$ and $e_2 \sqsubseteq_{\mathcal{H}} f_2$ with $f_2 \neq e_1$. It suffices to show that $e_1 \sqsubseteq_{\mathcal{H}} f_2$ and $e_1 \setminus f_2 = e_1 \setminus f_1$, because then (CSE) due to $e_1 \sqsubseteq_{\mathcal{H}} f_1$ has the same effect as CSE on $e_1 \sqsubseteq_{\mathcal{H}} f_2$, and we can apply the reasoning of case (2a) because $e_1 \neq f_2$ and $e_2 \neq f_2$.

We first show $e_1 \setminus f_2 = e_1 \setminus f_1$. Let $x \in e_1 \setminus f_2$ and suppose for the purpose of contradiction that that $x \in e_2 = f_1$. Then, since $e_1 \neq e_2$, $x \in \mathit{jv}(e_2) \subseteq f_2$ where the last inclusion is due to $e_2 \sqsubseteq_{\mathcal{H}} f_2$. Hence, $e_1 \setminus f_2 \subseteq e_1 \setminus f_1$. Conversely, let $x \in e_1 \setminus f_1$. Since $f_1 = e_2$, $x \notin e_2$. Suppose for the purpose of contradiction that $x \in f_2$. Because $e_1 \neq f_2$, $x \in \mathit{jv}_{\mathcal{H}}(e_1) \subseteq e_2$ where the last inclusion is due to $e_1 \sqsubseteq_{\mathcal{H}} e_2$. Therefore, $e_2 \setminus f_1 = e_1 \setminus f_2$.

To show that $e_1 \sqsubseteq_{\mathcal{H}} f_2$, let $x \in \mathit{jv}_{\mathcal{H}}(e_1)$. Because $e_1 \sqsubseteq_{\mathcal{H}} e_2$, $x \in e_2$. Because x occurs in two distinct hyperedges in $\mathcal{H}$, also $x \in \mathit{jv}_{\mathcal{H}}(e_2)$. Then, because $e_2 \sqsubseteq_{\mathcal{H}} f_2$, $x \in f_2$. Hence $\mathit{jv}_{\mathcal{H}}(e_1) \subseteq f_2$. It remains to show $\mathit{ext}_{\mathcal{H}}(e_1 \setminus f_2) \subseteq f_2$. To this end, let $x \in \mathit{ext}_{\mathcal{H}}(e_1 \setminus f_2)$ and suppose for the purpose of contradiction that $x \notin f_2$. By definition of *ext* there exists $\theta \in \mathit{pred}(\mathcal{H})$ and $y \in \mathit{var}(\theta) \cap (e_1 \setminus f_2)$ such that $x \in \mathit{var}(\theta) \setminus (e_1 \setminus f_2)$. In particular, $y \notin f_2$. Since $e_1 \setminus f_2 = e_1 \setminus e_2$, $y \in \mathit{var}(\theta) \cap (e_1 \setminus e_2)$ and $x \in \mathit{var}(\theta) \setminus (e_1 \setminus e_2)$. Thus, $x \in \mathit{ext}_{\mathcal{H}}(e_1 \setminus e_2)$. Then, since $e_1 \sqsubseteq_{\mathcal{H}} e_2$, $x \in e_2$. Thus, $x \in e_2 \setminus f_2$ since $x \notin f_2$. Hence $x \in \mathit{var}(\theta) \cap (e_2 \setminus f_2)$. Furthermore, since $y \notin e_2$ also $y \notin e_2 \setminus f_2$. Hence, $y \in \mathit{var}(\theta) \setminus (e_2 \setminus f_2)$. But then θ shows that $y \in \mathit{ext}_{\mathcal{H}}(e_2 \setminus f_2)$. Then, by because $e_2 \sqsubseteq_{\mathcal{H}} f_2$, also $y \in f_2$ which yields the desired contradiction.

(2c) $e_1 = f_2$ but $e_2 \neq f_1$. Similar to case (2b).

(2d) $e_1 = f_2$ and $e_2 = f_1$. Then $e_1 \sqsubseteq_{\mathcal{H}} e_2$ and $e_2 \sqsubseteq_{\mathcal{H}} e_1$ and $e_1 \neq e_2$. Let $\mathcal{K}_1$ (resp. $\mathcal{K}_2$) be the triplet obtained by applying (FLT) to remove all $\theta \in \mathit{pred}(\mathcal{I}_1)$ (resp. $\theta \in \mathit{pred}(\mathcal{I}_2)$ for which $\mathit{var}(\theta) \subseteq \mathit{var}(e_2)$ (resp. $(\mathit{var}(\theta) \subseteq \mathit{var}(e_2)$. Furthermore, let $\mathcal{J}_1$ (resp. $\mathcal{J}_2$) be the triplet obtained by applying ISO to removing $\mathit{isol}_{\mathcal{I}_1}(e_2)$ from $\mathcal{K}_1$ (resp. removing $\mathit{isol}_{\mathcal{I}_2}(e_1)$ from $\mathcal{K}_2$). Here, we take $\mathcal{J}_1 = \mathcal{K}_1$ if $\mathit{isol}_{\mathcal{K}_1}(e_2)$ is empty (and similarly for $\mathcal{J}_2$). Then clearly $\mathcal{H} \rightsquigarrow \mathcal{I}_1 \rightsquigarrow^* \mathcal{K}_1 \rightsquigarrow^* J_1$ and $\mathcal{H} \rightsquigarrow \mathcal{I}_2 \rightsquigarrow^* \mathcal{K}_2 \rightsquigarrow^* \mathcal{J}_2$. The result then follows by showing that $\mathcal{J}_1 = \mathcal{J}_2$. Towards this end, first observe that $\mathit{out}(\mathcal{J}_1) = \mathit{out}(\mathcal{K}_1) = \mathit{out}(\mathcal{I}_1) = \mathit{out}(\mathcal{H}) = \mathit{out}(\mathcal{I}_2) = \mathit{out}(\mathcal{K}_2) = \mathit{out}(\mathcal{J}_2)$. Next, we show that $\mathit{pred}(\mathcal{J}_1) = \mathit{pred}(\mathcal{J}_2)$. We first observe that $\mathit{pred}(\mathcal{J}_1) = \mathit{pred}(\mathcal{K}_1)$ and $\mathit{pred}(\mathcal{J}_2) = \mathit{pred}(\mathcal{K}_2)$ since the ISO operation does not remove predicates. Then observe that

$$\begin{aligned}\mathit{pred}(\mathcal{K}_1) &= \{\theta \in \mathit{pred}(\mathcal{I}_1) \mid \mathit{var}(\theta) \not\subseteq \mathit{var}(e_2)\} \\ &= \{\theta \in \mathit{pred}(\mathcal{H}) \mid \mathit{var}(\theta) \cap (e_1 \setminus e_2) = \emptyset \qquad \text{and} \qquad \mathit{var}(\theta) \not\subseteq e_2\},\end{aligned}$$

$$\begin{aligned} \mathit{pred}(\mathcal{K}_2) &= \{\theta \in \mathit{pred}(\mathcal{I}_2) \mid \mathit{var}(\theta) \not\subseteq e_1\} \\ &= \{\theta \in \mathit{pred}(\mathcal{H}) \mid \mathit{var}(\theta) \cap (e_2 \setminus e_1) = \emptyset \qquad \text{and} \qquad \mathit{var}(\theta) \not\subseteq e_1\}. \end{aligned}$$

We only show the reasoning for $\mathit{pred}(\mathcal{K}_1) \subseteq \mathit{pred}(\mathcal{K}_2)$, the other direction being similar. Let $\theta \in \mathit{pred}(\mathcal{K}_1)$. Then $\mathit{var}(\theta \cap (e_1 \setminus e_2) = \emptyset$ and $\mathit{var}(\theta) \not\subseteq e_2$. Since $\mathit{var}(\theta) \not\subseteq e_2$ there exists $y \in \mathit{var}(\theta) \setminus e_2$. Then, because $\mathit{var}(\theta) \cap (e_1 \setminus e_2) = \emptyset$, $y \notin e_1$. Thus, $\mathit{var}(\theta) \not\subseteq e_1$. Now, suppose for the purpose of obtaining a contradiction, that $\mathit{var}(\theta) \cap (e_2 \setminus e_1) \neq \emptyset$. Then take $z \in \mathit{var}(\theta) \cap (e_2 \setminus e_1)$. But then $y \in \mathit{ext}_{\mathcal{H}}(e_2 \setminus e_1)$. Hence, $y \in e_1$ because $e_2 \sqsubseteq_{\mathcal{H}} e_1$, which yields the desired contradiction with $y \notin e_2$. Therefore, $\mathit{var}(\theta) \cap (e_2 \setminus e_1) = \emptyset$, as desired. Hence $\theta \in \mathit{pred}(\mathcal{K}_2)$.

It remains to show that $\mathit{hyp}(\mathcal{J}_1) = \mathit{hyp}(\mathcal{J}_2)$. To this end, first observe

$$\begin{aligned} \mathit{hyp}(\mathcal{J}_1) &= \mathit{hyp}(\mathcal{K}_1) \setminus \{e_2\} \cup \{e_2 \setminus \mathit{isol}_{\mathcal{K}_1}(e_2)\}, \\ &= \mathit{hyp}(\mathcal{H}) \setminus \{e_1\} \setminus \{e_2\} \cup \{e_2 \setminus \mathit{isol}_{\mathcal{K}_1}(e_2)\}, \\ \mathit{hyp}(\mathcal{J}_2) &= \mathit{hyp}(\mathcal{K}_2) \setminus \{e_1\} \cup \{e_1 \setminus \mathit{isol}_{\mathcal{K}_2}(e_1)\} \\ &= \mathit{hyp}(\mathcal{H}) \setminus \{e_2\} \setminus \{e_1\} \cup \{e_1 \setminus \mathit{isol}_{\mathcal{K}_2}(e_1)\}. \end{aligned}$$

Clearly, $\mathit{hyp}(\mathcal{J}_1) = \mathit{hyp}(\mathcal{J}_2)$ if $e_2 \setminus \mathit{isol}_{\mathcal{K}_1}(e_2) = e_1 \setminus \mathit{isol}_{\mathcal{K}_2}(e_1)$.

We only show $e_2 \setminus \mathit{isol}_{\mathcal{K}_1}(e_2) \subseteq e_1 \setminus \mathit{isol}_{\mathcal{K}_2}(e_1)$, the other inclusion being similar. Let $x \in e_2 \setminus \mathit{isol}_{\mathcal{K}_1}(e_2)$. Since $x \notin \mathit{isol}_{\mathcal{K}_1}(e_2)$ one of the following hold.

- $x \in \mathit{out}(\mathcal{K}_1)$. But then, $x \in \mathit{out}(\mathcal{K}_1) = \mathit{out}(\mathcal{I}_1) = \mathit{out}(\mathcal{H}) = \mathit{out}(\mathcal{I}_2) = \mathit{out}(\mathcal{K}_2)$. In particular, x is an equijoin variable in $\mathcal{H}$ and $\mathcal{K}_{\in}$. Then $x \in \mathit{jv}_{\mathcal{H}}(e_2) \subseteq e_1$ because $e_2 \sqsubseteq_{\mathcal{H}} e_1$. From this and the fact that x remains an equijoin variable in $\mathcal{K}_2$, we obtain $x \in e_1 \setminus \mathit{isol}_{\mathcal{K}_2}(e_1)$.
- x occurs in e_2 and in some hyperedge g in $\mathcal{K}_1$ with $g \neq e_2$. Since e_1 is not in $\mathcal{K}_1$ also $g \neq e_1$. Since every hyperedge in $\mathcal{K}_1$ is in $\mathcal{I}_1$ and every hyperedge in $\mathcal{I}_1$ is in $\mathcal{H}$, also g is in $\mathcal{H}$. But then, x occurs in two distinct hyperedges in $\mathcal{H}$, namely e_2 and g, and hence $x \in \mathit{jv}_{\mathcal{H}}(e_2) \subseteq e_1$ because $e_2 \sqsubseteq_{\mathcal{H}} e_1$. However, because x also occurs in g which must also be in $\mathcal{I}_2$ and therefore also in $\mathcal{K}_2$, x also occurs in two distinct hyperedges in $\mathcal{K}_2$, namely e_1 and g. Therefore, $x \in \mathit{jv}_{\mathcal{I}_2}(e_1)$ and hence $x \in e_1 \setminus \mathit{isol}_{\mathcal{I}_2}(e_1)$, as desired.
- $x \in \mathit{var}(\mathit{pred}(\mathcal{K}_1))$. Then there exists $\theta \in \mathit{pred}(\mathcal{K}_1)$ such that $x \in \mathit{var}(\theta)$. Since $\mathit{pred}(\mathcal{K}_1) = \mathit{pred}(\mathcal{K}_2)$, $\theta \in \mathit{pred}(\mathcal{K}_2)$. As such, $\theta \in \mathit{pred}(\mathcal{H})$, $\mathit{var}(\theta) \cap (e_2 \setminus e_1) = \emptyset$, and $\mathit{var}(\theta) \not\subseteq e_1$. But then, since $x \in \mathit{var}(\theta)$; $x \in e_2$; and $\mathit{var}(\theta) \cap (e_2 \setminus e_1) = \emptyset$, it must be the case that $x \in e_1$. As such, $x \in e_1$ and $x \in \mathit{var}(\mathcal{K}_2)$. Hence $x \in e_1 \setminus \mathit{isol}_{\mathcal{K}_2}(e_1)$.

(3) Case (ISO, CSE): assume that $\mathcal{I}_1$ is obtained by removing the non-empty set of isolated variables $X_1 \subseteq \mathit{isol}_{\mathcal{H}}(e_1)$ from e_1, while $\mathcal{I}_2$ is obtained by removing hyperedge e_2, conditional subset of hyperedge f_2. We may assume w.l.o.g. that $e_1 \neq \mathit{isol}_{\mathcal{H}}(e_1)$: if $e_1 = \mathit{isol}_{\mathcal{H}}(e_1)$ then the ISO operation removes the complete hyperedge e_1. However, because no predicate in $\mathcal{H}$ shares any variable with e_1, it is readily verified that $e_1 \sqsubseteq_{\mathcal{H}} e_2$ and thus the removal of e_1 can also be seen as an application of CSE on e_1[4], and we are hence back in case (2).

[4]Note that, since e_1 does not share variables with any predicate, the CSE operation also does not remove any predicates from $\mathcal{H}_1$, similar to the ISO operation and hence yields $\mathcal{I}_1$.

Now reason as follows. Because $e_2 \sqsubseteq_{\mathcal{H}} f_2$ and because isolated variables of e_1 occur in no other hyperedge in $\mathcal{H}$, it must be the case that $e_2 \cap X_1 = \emptyset$. In particular, e_1 and e_2 must hence be distinct. Therefore, $e_1 \in \mathit{hyp}(\mathcal{I}_2)$ and $e_2 \in \mathit{hyp}(\mathcal{I}_1)$. By Lemma 21, we can apply ISO on $\mathcal{I}_2$ to remove X_1 from e_1. It then suffices to show that e_2 remains a conditional subset of some hyperedge f_2' in $\mathcal{I}_1$ with $e_2 \setminus f_2 = e_2 \setminus f_2'$. Indeed, we can then use ECQ to remove e_2 from $\mathit{hyp}(\mathcal{I}_1)$ as well as predicates θ with $\mathit{var}(\theta) \cap (e_2 \setminus f_2) \neq \emptyset$ from $\mathit{pred}(\mathcal{I}_1)$. This clearly yields the same triplet as the one obtained by removing X_1 from e_1 in $\mathcal{I}_2$. We need to distinguish two cases.

(3a) $f_2 \neq e_1$. Then $f_2 \in \mathit{hyp}(\mathcal{I}_1)$ and hence $e_2 \sqsubseteq_{\mathcal{I}_1} f_2$ by Lemma 21. We hence take $f_2' = f_2$.

(3b) $f_2 = e_1$. Then we take $f_2' = e_1 \setminus X$. Since $e_1 \neq \mathit{isol}_{\mathcal{H}}(e_1)$ it follows that $e_1 \setminus X_1 \neq \emptyset$. Therefore, $f_2' = e_1 \setminus X_1 \in \mathit{hyp}(\mathcal{I}_1)$. Furthermore, since $X \subseteq \mathit{isol}_{\mathcal{H}}(e_1)$, no variable in X is in any other hyperedge in $\mathcal{H}$. In particular $X \cap e_2 = \emptyset$. Therefore, $e_2 \setminus f_2' = e_2 \setminus (e_1 \setminus X) = (e_2 \setminus e_1) \cup (e_2 \cap X) = e_2 \setminus e_1 \setminus e_1 = e_2 \setminus f_2$. It remains to show that $e_2 \sqsubseteq_{\mathcal{I}_1} e_1 \setminus X_1$.

- $\mathit{jv}_{\mathcal{I}_1}(e_2) \subseteq e_1 \setminus X_1$. Let $x \in \mathit{jv}_{\mathcal{I}_1}(e_2)$. By Lemma 21, $x \in \mathit{jv}_{\mathcal{I}_1}(e_2) \subseteq \mathit{jv}_{\mathcal{H}}(e_2) \subseteq e_1$ where the last inclusion is due to $e_2 \sqsubseteq_{\mathcal{H}} e_1$. In particular, x is an equijoin variable in $\mathcal{H}$. But then it cannot be an isolated variable in any hyperedge. Therefore, $x \notin X_1$.
- $\mathit{ext}_{\mathcal{I}_1}(e_2 \setminus e_1) \subseteq e_1 \setminus X$. Let $x \in \mathit{ext}_{\mathcal{I}_1}(e_2 \setminus e_1)$. Then $x \in \mathit{ext}_{\mathcal{I}_1}(e_2 \setminus e_1) \subseteq \mathit{ext}_{\mathcal{H}}(e_2 \setminus e_1) \subseteq e_1$ where the first inclusion is by Lemma 21 and the second by $e_2 \sqsubseteq_{\mathcal{H}} e_1$. Then, because $x \in \mathit{ext}_{\mathcal{H}}(e_2 \setminus e_1)$ it follows from the definition of *ext*, that x occurs in some predicate in $\mathit{pred}(\mathcal{H})$. However, X is disjoint with $\mathit{var}(\mathit{pred}(\mathcal{H}))$ since it consist only of isolated variables. Therefore, $x \notin X$.

(4): Case (ISO, FLT) Assume that $\mathcal{I}_1$ is obtained by removing the non-empty set $X_1 \subseteq \mathit{isol}_{\mathcal{H}}(e_1)$ from hyperedge e_1, while $\mathcal{I}_2$ is obtained by removing all predicates in the non-empty set $\Theta \subseteq \mathit{pred}(\mathcal{H})$ with $\mathit{var}(\Theta) \subseteq e_2$ for some hyperedge e_2 in $\mathit{hyp}(\mathcal{H})$. Observe that $e_1 \in \mathit{hyp}(\mathcal{I}_2)$. By Lemma 21, $X \subseteq \mathit{isol}_{\mathcal{H}}(e_1) \subseteq \mathit{isol}_{\mathcal{I}_2}(e_1)$. Therefore, we may apply reduction operation (ISO) on $\mathcal{I}_2$ to remove X_1 from e_1. We will now show that, similarly, we may still apply (FLT) on $\mathcal{I}_1$ to remove all predicates in Θ from $\mathit{pred}(\mathcal{I}_1) = \mathit{pred}(\mathcal{H})$. The two operations hence commute, and clearly the resulting triplets in both cases is the same. We distinguish two possibilities. (i) $e_1 \neq e_2$. Then $e_2 \in \mathcal{I}_1$ and, $\mathit{var}(\Theta) \subseteq e_2$ and, since (ISO) does not remove predicates, $\Theta \subseteq \mathit{pred}(\mathcal{H}) = \mathit{pred}(\mathcal{I}_1)$. As such the (FLT) operation indeed applies to remove all predicates in Θ from $\mathit{pred}(\mathcal{I}_1)$. (ii) $e_1 = e_2$. Then, since $X \subseteq \mathit{isol}_{\mathcal{H}}(e_1)$ and isolated variables do no occur in any predicate, $X \cap \mathit{var}(\Theta) = \emptyset$. Then, since $\mathit{var}(\Theta) \subseteq e_2 = e_1$, it follows that also $\mathit{var}(\Theta) \subseteq e_1 \setminus X$. In particular, since we disallow nullary predicates and Θ is non-empty, $e_1 \setminus X \neq \emptyset$. Thus, $e_1 \setminus X \in \mathit{hyp}(\mathcal{I}_1)$ and hence operation (FLT) applies indeed applies to remove all predicates in Θ from $\mathit{pred}(\mathcal{I}_1)$

(5) Case (CSE, FLT): assume that $\mathcal{I}_1$ is obtained by removing hyperedge e_1, conditional subset of e_2 in $\mathcal{H}$, while $\mathcal{I}_2$ is obtained by removing all predicates in the non-empty set $\Theta \subseteq \mathit{pred}(\mathcal{H})$ with $\mathit{var}(\Theta) \subseteq e_3$ for some hyperedge $e_3 \in \mathit{hyp}(\mathcal{H})$. Since the (FLT) operation does not remove any hyperedges, e_1 and e_2 are in $\mathit{hyp}(\mathcal{I}_2)$. Then, since $e_1 \sqsubseteq_{\mathcal{H}} e_2$ also $e_1 \sqsubseteq_{\mathcal{I}_2} e_2$ by Lemma 21. Therefore, we may apply reduction operation (CSE) on $\mathcal{I}_2$ to remove e_1 from $\mathit{hyp}(\mathcal{I}_2)$ as well as all predicates $\theta \in \mathit{pred}(\mathcal{I}_2)$ for which $\mathit{var}(\theta) \cap (e_1 \setminus e_2) \neq \emptyset$. Let $\mathcal{J}_2$ be the triplet resulting from this operation. We will show that, similarly, we may apply (FLT) on $\mathcal{I}_1$ to remove all predicates in $\Theta \cap \mathit{pred}(\mathcal{I}_1)$ from $\mathit{pred}(\mathcal{I}_1)$, resulting in a triplet $\mathcal{J}_1$. Observe that necessarily, $\mathcal{J}_1 = \mathcal{J}_2$ (and hence they form the triplet $\mathcal{J}$). Indeed,

$out(\mathcal{J}_1) = out(\mathcal{I}_1) = out(\mathcal{H}) = out(\mathcal{I}_2) = out(\mathcal{J}_2)$ since reduction operations never modify output variables. Moreover,

$$hyp(\mathcal{J}_1) = hyp(\mathcal{I}_1) = hyp(\mathcal{H}) \setminus \{e_1\} = hyp(\mathcal{I}_2) \setminus \{e_1\} = hyp(\mathcal{J}_2)$$

where the first and third equality is due to fact that (FLT) does not modify the hypergraph of the triplet it operates on. Finally, observe

$$\begin{aligned} pred(\mathcal{J}_1) &= pred(\mathcal{I}_1) \setminus (\Theta \cap pred(\mathcal{I}_1)) \\ &= pred(\mathcal{I}_1) \setminus \Theta \\ &= \{\theta \in pred(\mathcal{H}) \mid var(\theta) \cap (e_1 \setminus e_2) = \emptyset\} \setminus \Theta \\ &= \{\theta \in pred(\mathcal{H}) \setminus \Theta \mid var(\theta) \cap (e_1 \setminus e_2) = \emptyset\} \\ &= \{\theta \in pred(\mathcal{I}_2) \mid var(\theta) \cap (e_1 \setminus e_2) = \emptyset\} \\ &= pred(\mathcal{J}_2) \end{aligned}$$

It remains to show that we may apply (FLT) on $\mathcal{I}_1$ to remove all predicates in $\Theta \cap pred(\mathcal{I}_1)$, resulting in a triplet $\mathcal{J}_1$. There are two possibilities.

- $e_3 \neq e_1$. Then $e_3 \in \mathcal{I}_1$, $\Theta \cap pred((I_1)) \subseteq pred(\mathcal{I}_1))$, and $var(\Theta \cap pred(\mathcal{I}_1)) \subseteq var(\Theta) \subseteq e_3$. Hence the (FLT) operation indeed applies to $\mathcal{I}_1$ to remove all predicates in $\Theta \cap pred(\mathcal{I}_1)$.
- $e_3 = e_1$. In this case we claim that for every $\theta \in \Theta \cap pred(\mathcal{I}_1)$ we have $var(\theta) \subseteq e_2$. As such, $var(\Theta \cap pred(\mathcal{I}_1)) \subseteq e_2$. Since $e_2 \in hyp(\mathcal{I}_1)$ and $\Theta \cap pred(\mathcal{I}_1) \subseteq pred(\mathcal{I}_1)$ we may hence apply (FLT) to remove all predicates in $\Theta \cap pred(\mathcal{I}_1)$ from $\mathcal{I}_1$. Concretely, let $\theta \in \Theta \cap pred(\mathcal{I}_1)$. Because, in order to obtain $\mathcal{I}_1$, (CSE) removes all predicates from $\mathcal{H}$ that share a variable with $e_1 \setminus e_2$, we have $var(\theta) \cap (e_1 \setminus e_2) = \emptyset$. Moreover, because $\theta \in \Theta$, $var(\theta) \subseteq e_1$. Hence $var(\theta) \subseteq e_2$, as desired.

The remaining cases, (CSE, ISO), (FLT, ISO), and (FLT, CSE), are symmetric to case (3), (4), and (5), respectively. □

5.5 Conclusion

In this chapter, we presented the algorithms to compute GJTs for GCQs that are free-connex acyclic. We started with discussion on the classical GYO-reduction algorithm to test a NCQ for acyclicity. Then we presented an extension of the GYO-reduction to test GCQs for acyclicity and free-connexity. We also showed that the GYO-reduction can essentially be used to generate GJTs for GCQs that are acyclic. Moreover, we also presented detailed discussion on the correctness of our algorithms and the generated GJTs.

6

DISCUSSION AND FUTURE DIRECTIONS

In this chapter, we first summarize the research outcomes presented in this thesis in the context of real-time analytics. Then, we discuss the strengths and limitations of the unified model (GDYN) and present possible future areas in which the model can further be investigated.

6.1 Summary

We formulated two main research questions RQ_1 and RQ_2. RQ_1 was related to investigating the possibility of a unified main-memory model for traditional BI systems and IFP systems. In particular, we were interested in investigating a model that supports processing of OLTP, OLAP, CEP, and continuous queries that appear in both BI and IFP systems in a unified framework having same query language, processing framework, and data model. We answered this question by presenting the unified GDYN framework that supports queries appearing in both BI and IFP systems in a main-memory model based on the relational data model and the unified query processing mechanism. Our second research question RQ_2 was related to investigating the possibility to address the trade-off between the runtime memory footprint and update latency in dynamic query evaluation. In particular, the existing incremental evaluation in both IFP and BI approaches pose this trade-off in the following ways: systems that materialize queries (sub)results can reuse these materialized results to process updates fast, however, they incur high memory footprint; while systems that avoid materialization of (sub)results have to recompute them on each update hence incurring high update latency. To address this memory-time trade-off, we formulated DCLRs, the main-memory compact and efficient representations that guarantee specific runtime properties including: 1) enumeration of query results with bounded delay (constant delay for free-connex queries) without materialization, 2) lookups of tuples in query results with bounded delay (constant delay for free-connex queries), 3) takes space linear in the size of the database, and 4) is efficiently maintainable under updates: for NCQs with maximum linear delay, for hierarchical NCQs with constant delay and for GCQs with log-linear delay.

To ensure the properties of DCLRs, we focused on GJTs for queries that are acyclic and these GJTs are the building block of the GDYN framework. We presented algorithms to identify GCQs that are acyclic and to generate GJTs for such queries. To validate the efficiency of GDYN, we implemented the GDYN as query compiler and tested on the publicly available benchmark datasets and synthetic datasets, and compared the results with state of the art BI and IFP systems. We found out that GDYN is not only bridging the gap

between IFP and BI systems, it also addresses the tradeoff between memory footprint and updates processing cost as desired in our research question RQ_2. In particular, we have addressed the tradeoff and have shown that GDYN can be several orders of magnitude more efficient in both memory footprint and update processing.

6.2 Strength, Limitations and Future Directions

6.2.1 Strengths

- Static and dynamic setting: The unified model presented in this thesis presents the best optimal results for NCQs and GCQs in both static and dynamic settings. Moreover, unlike the existing approaches e.g. (H)IVM that pose the space-time trade-off, the GDYN framework addresses this trade-off and presents a practical solution.
- Query plans: Apart from the GDYN framework, we have presented the algorithms to test queries for acyclicity and to generate GJTs for acyclic queries. Our algorithms are generic and take into account the projections and their connexity to generate optimal join trees. This completes the functioning of the framework.
- Broader applicability: The GDYN framework, unlike the other competitive systems, is developed to address the limitations of systems that appear in two different domains - namely BI and IFP systems. As such, GDYN is applicable but not limited to queries that appear in these domains.
- Scalability: We have shown the scalability of the GDYN framework for representative queries. Since GDYN is memory efficient, it can manage to process high volumes of frequent updates without high memory stress. Especially for queries that feature temporal constraints or inequality joins that incur high memory stress, GDYN is highly efficient.

6.2.2 Limitations and Future Directions

- Cyclic queries: the GDYN framework developed in this thesis works well for GCQs that are acyclic and the properties of DCLRs are guaranteed for acyclic queries. Since the GJTs defined in this thesis are based on the notion of width-one GHDs (generalized hyper-tree decompositions), the GDYN framework cannot be utilized with the current properties of DCLRs for queries that are cyclic. One possible and important extension of this work is to look into the possibility to process cyclic queries - especially the Datalog-style queries. Since cyclic queries do not admit join trees that are of width one, it hence remains to define or modify GJTs in such a way that cyclic queries can be processed with optimal guarantees. It also remains an open challenge to develop and extend DCLRs for cyclic queries and this currently remains a limitation for GDYN.
- Parallelism and distribution: In this thesis, we have discussed the evaluation of updates when they occur at a single input (leaf of a GJT) of a query i.e. GDYN is currently designed as an unparallel and stand-alone framework. In the case of updates occurring at multiple input paths i.e. at multiple leaves in GJTs (or multiple input relations in queries), the internal nodes in the GJTs may be accessed and updated by multiple children at the same time. Therefore, it remains an open challenge to build a mechanism to process updates in parallel. Moreover, the evaluation of

queries across distributed systems is another possible extension to the GDYN framework. In particular, one possible approach is to assign each node in a GJT a separate compute node in a distributed system and communicate the data between nodes for computing joins and generating query results. This approach might generate communication stress in the network, however. To address this issue, one can then think of assigning portions (subtrees) of the GJT to compute nodes in the cluster. This, however, remains an open problem of research.

- Aggregates: The GDYN framework currently supports evaluation of simple aggregates (e.g. COUNT, SUM, and AVERAGE) and cannot process aggregates such as MIN and MAX especially in the presence of deletions. Since we work with multi-set semantics (i.e. GMRs) where we maintain multiplicities (COUNTs) of tuples, we can leverage this to compute aggregates SUM and AVERAGE. While we can also compute and maintain MIN and MAX under updates that are only insertions, however, maintaining these aggregates under deletions incur more update evaluation costs. In the case of only insertions, we only need to maintain the current MIN or MAX value for each set of variables in the GROUP BY clause of the query. However, in the case of deletions, maintaining current MIN or MAX do not suffice and we need additional information to answer such queries. For example, in the query $\pi_{a,MIN(d)}(R(a,b) \bowtie S(c,d) \mid a < c)$, apart from sorting relation S on the variable d for each value of the variable c, we need some extra information to compute $min(d)$ for each a in the presence of $a < c$ and deletions. In particular, under each update (deletion) $\Delta R = (a_i, b_i)$ in $R(a,b)$, one needs to get all values $C = \{c_1, \ldots c_k\}$ of variable c in the relation $S(c,d)$ where each $c_i > a_i \; \forall \; c_i \in C$. Then, for each $c_i \in C$, one needs to get the minimum d, sort them, and then choose the minimum. Computing MIN and MAX in this way incurs an additional log-linear cost for each deletion update. To avoid this cost, one can materialize for each a the set of all c and d values, which in turn runs out of the complexity bounds of GJTs. It hence remains an open question to investigate how these aggregates can be evaluated in GDYN under updates, in particular, deletions.

- Multi-query optimization: So far, we have discussed how to process a single query efficiently. One possible are in the future to investigate is the possibility to share pre-computations among queries that share tree (sub)structures. In particular, when two or more queries share a subtree or one query is a sub-query of another query, then it might be interesting to see if we can actually use one subtree (eventually the computations) for all such queries and optimize the overall runtime. This approach in classical databases is known as multi-query optimization. Since GJTs are particular join trees that guarantee certain complexity guarantees, hence it can be an interesting work to further add to the benefits of GDYN framework.

- Fine-grained enumeration: The GDYN framework presented in this thesis is based on GJTs. We have also shown that these GJTs allow for CDE of query results for queries that are free-connex acyclic. An interesting feature of GJTs is that they allow enumeration of sub-trees. In particular, if the query Q_1 is a sub-query of another query Q_2, then the sub-query forms a sub-tree in the GJT of Q_1 and hence, its results can be enumerated from its sub-tree. Intuitively speaking, this is helpful in the particular case of selection of cuboids in data warehousing where a cube can be defined by a GJT and the cuboids in this cubes may form sub-trees. Hence, one can choose cuboids based on the sub-trees by comparing them for being simple and compatible. This, however, needs to be investigated further.

BIBLIOGRAPHY

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[ABB+16] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: the stanford data stream management system. In *Data Stream Management - Processing High-Speed Data Streams*, pages 317–336. 2016.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[ACÇ+03] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD* , pages 147–160, 2008.

[AGM08] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 739–748. IEEE, 2008.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[AKKN12] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of VLDB*, pages 253–278, 2012.

[AMU+17] Alexander Artikis, Alessandro Margara, Martín Ugarte, Stijn Vansummeren, and Matthias Weidlich. Complex event recognition languages: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 7–10, 2017.

[ASAP15] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras. Complex event recognition under uncertainty: A short survey. *Event Processing, Forecasting and Decision-Making in the Big Data Era (EPForDM)*, pages 97–103, 2015.

[BB13] Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques*. PhD thesis, Université de Caen, 2013.

[BDG07a] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings of CSL*, pages 208–222, 2007.

[BDG+07b] L. Brenna, A. J. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. M. White. Cayuga: a high-performance event processing engine. In *Proceedings of the ACM SIGMOD* , pages 1100–1102, 2007.

[BG81] Philip A Bernstein and Nathan Goodman. The power of inequality semijoins. *Information Systems*, 6(4):255–265, 1981.

[BKOZ13] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. Aggregation and ordering in factorised databases. *Proceedings of VLDB*, 6(14):1990–2001, 2013.

[BKS17] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Proceedings of PODS*, 2017. To appear.

[Bui09] Hai-Lam Bui. Survey and comparison of event query languages using practical examples. *Ludwig-Maximilians Universität München thesis*, 2009.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.

[CM10] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 50–61, 2010.

[CM12a] Gianpaolo Cugola and Alessandro Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

[CM12b] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM CSUR*, 44(3):15:1–15:62, 2012.

[CM12c] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM CSUR*, 44(3):15, 2012.

[Cor09] T.H. Cormen. *Introduction to Algorithms, 3rd Edition:*. MIT Press, 2009.

[CR97] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In *International Conference on Database Theory*, pages 56–70. Springer, 1997.

[CY12a] Rada Chirkova and Jun Yang. *Materialized Views*. Now Publishers Inc., Hanover, MA, USA, 2012.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The hipac project: Combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, March 1988.

[DKRC+14] F Dehne, Q Kong, A Rau-Chaplin, H Zaboli, and R Zhou. Scalable real-time olap on cloud architectures. *Journal of Parallel and Distributed Computing*, 2014.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of VLDB* , pages 443–452, 1991.

[DNS+92] David J DeWitt, Jeffrey F Naughton, Donovan A Schneider, Srinivasan Seshadri, et al. *Practical skew handling in parallel joins*. University of Wisconsin-Madison. Computer Sciences Department, 1992.

[DS13] Nilesh Dalvi and Dan Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *Journal of the ACM (JACM)*, 59(6):30:1–30:87, 2013.

[EHS04] Jost Enderle, Matthias Hampel, and Thomas Seidl. Joining interval data in relational databases. In *Proceedings of the ACM SIGMOD*, pages 683–694, 2004.

[Esp] EsperTech. Esper complex event processing engine. `http://www.espertech.com/`.

[FMF13] Nickerson Ferreira, Pedro Martins, and Pedro Furtado. Near real-time with traditional data warehouse architectures: factors and how-to. In *Proceedings of the 17th International Database Engineering & Applications Symposium*, pages 68–75. ACM, 2013.

[FO16] Robert Fink and Dan Olteanu. Dichotomies for queries with negation in probabilistic databases. *ACM Transaction on Database Systems (TODS)*, 41(1):4:1–4:47, 2016.

[For82] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[FRS93] Françoise Fabret, Mireille Regnier, and Eric Simon. *An adaptive algorithm for incremental evaluation of production rules in databases*. Institut national de recherche en informatique et en automatique, 1993.

[FRU+18] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Constant delay algorithms for regular document spanners. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 165–177, 2018.

[GGDD91] Stella Gatziu, Andreas Geppert, Klaus R. Dittrich, and Forschungsbereich Datenbanktechnologie. Integrating active concepts into an object-oriented database system. In *Proceedings of the 3rd Int. Workshop on Database Programming Languages, Nafplion*, pages 399–415. Morgan Kaufmann, 1991.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *ACM SIGMOD Record*, volume 24, pages 328–339. ACM, 1995.

[GLS01] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM (JACM)*, 48(3):431–498, 2001.

[GLS03a] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *Journal of computer and system sciences*, 66(4):775–808, 2003.

[GM14] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, (1):4:1–4:20, 2014.

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of STOC*, pages 21–30, 2015.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of VLDB*, pages 562–573, 1995.

[IUV17] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM SIGMOD*, pages 1259–1274. ACM, 2017.

[IUV+18] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with inequalities under updates. *Proceedings of VLDB*, 11(7):733–745, 2018.

[JPR16] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of PODS*, pages 91–106, 2016.

[KAK+14] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB Journal*, pages 253–278, 2014.

[KLS+17] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Fast and scalable inequality joins. *VLDB Journal*, 26(1):125–150, 2017.

[Koc10] Christoph Koch. Incremental query evaluation in a ring of databases. In *Proceedings of PODS*, pages 87–98, 2010.

[KS11] Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of PODS*, pages 223–234, 2011.

[LGA96] Daniel F. Lieuwen, Narain H. Gehani, and Robert M. Arlein. The ode active database: Trigger semantics and implementation. In *ICDE*, pages 412–420, 1996.

[Mir14] Daniel P Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production System*. Morgan Kaufmann, 2014.

[MM09] Yuan Mei and Samuel Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the ACM SIGMOD*, pages 193–206, 2009.

[NDK16] Milos Nikolic, Mohammad Dashti, and Christoph Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the ACM SIGMOD* , pages 511–526, 2016.

[NO18] Milos Nikolic and Dan Olteanu. Incremental view maintenance with triple lock factorisation benefits. In *Proceedings of the ACM SIGMOD* , 2018. To appear.

[NPRR12] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[NRR13] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *arXiv preprint arXiv:1310.3314*, 2013.

[NRR14] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[OZ12] Dan Olteanu and Jakub Závodnỳ. Factorised representations of query results: Size bounds and readability. In *Proceedings of the 15th International Conference on Database Theory*, pages 285–298. ACM, 2012.

[OZ15] Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):2, 2015.

[Pap03] Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. 2003.

[Pro09] Oracle Complex Event Processing. Lightweight modular application event stream processing in the real world. *An Oracle White Paper*, 2009.

[Pro12] Mark Proctor. Drools: a rule engine for complex event processing. In *Applications of Graph Transformations with Industrial Relevance*, pages 2–2. Springer, 2012.

[SAC+79] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD* , pages 23–34. ACM, 1979.

[Seg13] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of ICDT*, pages 10–20, 2013.

[Seg14] Luc Segoufin. A glimpse on constant delay enumeration. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, pages 13–27, 2014.

[Seg15] Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.

[SMK97] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997.

[SMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009*, 2009.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the ACM SIGMOD* , pages 147–156. ACM, 2014.

[Var82] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of STOC*, pages 137–146, 1982.

[Vel12] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[VZ14] Alejandro Vaisman and Esteban Zimányi. *Data Warehouse Systems: Design and Implementation*. Springer, 2014.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD* , pages 407–418, 2006.

[WM03] Ian Wright and James A. R. Marshall. The execution kernel of rc++: Rete*, a faster rete with treat as a special case. *Int. J. Intell. Games & Simulation*, 2(1):36–48, 2003.

[Y81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of VLDB*, pages 82–94, 1981.

[YK84] Masatoshi Yoshikawa and Yahiko Kambayashi. Processing inequality queries based on generalized semi-joins. In *Proceedings of VLDB*, pages 416–428. Morgan Kaufmann Publishers Inc., 1984.

[ZAL08] Youchan Zhu, Lei An, and Shuangxi Liu. Data updating and query in real-time data warehouse system. In *Computer science and software engineering, 2008 international conference on*, volume 5, pages 1295–1297. IEEE, 2008.

[ZDI14] Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in complex event processing. In *Proce. of SIGMOD*, 2014.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. *ACM SIGMOD Record*, 24(2):316–327, 1995.

[Zut11] Janis Zuters. Near real-time data warehousing with multi-stage trickle and flip. In *International Conference on Business Informatics Research*, pages 73–82. Springer, 2011.

LIST OF FIGURES

1.1 Classification of GCQs and its sub-classes of queries. 12

2.1 Taxonomy of related work 15
2.2 Relations *Customer, Order, and Item* 21
2.3 Relational representation of $Q = Customer \bowtie Order \bowtie Item$ 21
2.4 Factorized representation of $Q = Customer \bowtie Order \bowtie Item$ 21

3.1 Operations on GMRs 28
3.2 Width-one GHDs for $\{R(x,y,z), S(x,y,u), T(y,v,w), U(y,v,p)\}$. T_1 is a traditional join tree, T_3 and T_4 are generalized join trees. In addition, T_4 is simple. 30
3.3 Illustration of T-representations (Example 2). 38
3.4 Illustration of the proof of Proposition 13 53
3.5 DYN usage of resources as a percentage of the resources consumed by DBToaster (lower is better). 58
3.6 Time for enumerating output (lower is better) 60
3.7 Resource utilization v/s % of tuples processed 61

4.1 Operations on GMRs 63
4.2 Two example GJTs. 64
4.3 Illustration of the sibling-closed transform: removal of type-1 violator. The connex sets N and N' are indicated by the shaded areas. 67
4.4 Illustration of the sibling-closed transform: removal of type-2 violator. The connex sets N and N' are indicated by the shaded areas. 69
4.5 Binarizing a k-ary node n. 70
4.6 Example database and its T_2-reduct. 71
4.7 T_2-rep of $db + u$ with T_1 and db as in Fig. 4.6 and u the update containing $(\langle y\colon 2, z\colon 5, w\colon 6\rangle, 2)$ and $(\langle u\colon 4, v\colon 9\rangle, 3)$. 72
4.8 IEDYN (IE) VS (Z,DBT,E,T, SE) on full join queries. The X-axis shows stream sizes and the y-axis update delay in $seconds$ (*: DBT out of memory, +: Z out of memory, ′: T was stopped after 1500 seconds) 86
4.9 IEDYN (IE) VS (E, DBT, T) fulljoin and projection queries, (*: DBT ran out of memory, ′: T was stopped after 1500 seconds) 88
4.10 IEDYN (IE) VS SE on temporally ordered datasets 89
4.11 IEDYN (IE) VS Z on temporally ordered datasets 89
4.12 Enumeration of query results: IEDYN vs DBT, different bars for Q_4, Q_5, Q_6 show their projected versions 89
4.13 IEDYN as percentage of (E, DBT, SE, Z, T) for higher join selectivities. X-axis shows queries with tuples per relation and selectivities 90
4.14 IEDYN scalability ($mi = 1,000,000$) 90

5.1 Illustration of GYO-reduction for GCQs. Colored regions depict hyperedges. Variables in *out* are underlined. Variables occurring in the same predicate are connected by dashed lines. 94
5.2 GJT Construction by GYO-reduction. 98

LIST OF TABLES

1.1 Theoretical results for the different classes of acyclic NCQs in the DYN framework. 12
1.2 Theoretical results for different classes of GCQs in the GDYN framework . 13

2.1 Comparison of GDYN against BI and IFP systems: (×) means the corresponding functionality (reducing update processing time, CDE, or reducing runtime memory footprint) is not addressed, (-) means supports functionality with good performance but degrades because of the memory-time trade-off, (✓) means supports with good performance, and (✓*) for enumeration means CDE when query results are materialized while without "*" means without materialization. 19
2.2 GDYN against f-representations and other approaches: (✓) - CDE is supported, (✓*) - CDE is shown to be supported theoretically, (×) - CDE not supported or not discussed 22
2.3 Acyclicity test and GJT construction for NCQs and GCQs in GDYN against GYO-reduction and Bagan & Goodman (B&G) semi-join inequality reduction: (×) means cyclicity test and join tree construction not supported or discussed, (✓) means supported, and (✓*) means supported partially or with limited support. 25

3.1 Number of tuples in the stream file of each query 56

4.1 Queries for experimental evaluation. 83
4.2 Maximum output sizes per query, k=1000. 85

LIST OF ACRONYMS

- **BI** Business Intelligence
- **CAQ** Conjunctive Aggregate Query
- **CDE** Constant Delay Enumeration
- **CDY** Constant Delay Yannakakis
- **CER** Composite Event Recognition
- **CEP** Complex Event Processing
- **CQ** Conjunctive Query
- **CQL** Continuous Query Language
- **CSE** Conditional Subset Elimination
- **DBT** DBToaster
- **DBMS** Database Management System
- **DCLR** Dynamic Constant Delay Linear Representation
- **DSMS** Data Stream Management System
- **DYN** Dynamic Yannakakis Algorithm
- **EPL** Event Processing Language
- **E** Esper
- **ELT** Extraction-Load-Transform
- **ETL** Extraction-Transform-Load
- **FQ** Full Join Query
- **FLT** Filter Elimination
- **F-representation** Factorized Representations
- **F-IVM** Factorized IVM
- **GCQ** Generalized Conjunctive Query
- **GDYN** Generalized Dynamic Algorithm
- **GHD** Generalized Hypertree Decomposition

- **GJF** Generalized Join Forest
- **GJT** Generalized Join Tree
- **GMR** Generalized Multiset Relation
- **GYO** Graham-Yu-Ozsoyoglu
- **HIVM** Higher-order Incremental View Maintenance
- **IEDYN** Inequality Dynamic Algorithm
- **IFP** Information Flow Processing
- **ISO** Isolated Variables Elimination
- **IVM** Incremental View Maintenance
- **NCQ** Normal Conjunctive Query
- **OLAP** Online Analytical Processing
- **OLTP** Online Transaction Processing
- **RQ** Research Question
- **SE** SASE
- **SQL** Structured Query Language
- **T** Tesla
- **Z** ZStream

A

BENCHMARK QUERIES

A.1 Queries for evaluation of DYN

A.1.1 Full Join Queries

```
FQ1
SELECT *  FROM  orders o, lineitem l, part p , partsupp ps
WHERE  o.orderkey = l.orderkey,  AND  l.partkey = p.partkey
AND  l.partkey = ps.partkey  AND  l.suppkey = ps.suppkey
```

```
FQ2
SELECT *  FROM  lineitem l, orders, customer c, part p , nation n
WHERE  l.orderkey = o.orderkey  AND  o.custkey = c.custkey
AND  l.partkey = p.partkey  AND  c.nationkey = n.nationkey
```

```
FQ3
SELECT *  FROM  orders o, lineitem l,
partsupp ps, supplier s, customer c
WHERE  o.orderkey = l.orderkey  AND
l.suppkey = ps.suppkey  AND
l.suppkey = s.suppkey  AND  o.custkey = c.custkey
```

```
FQ4
SELECT *  FROM  lineitem l, supplier s, partsupp ps
WHERE  l.suppkey = s.suppkey
AND  l.suppkey = ps.suppkey
```

```
FQ5
SELECT * SELECT *  FROM  date_dim dd, store_sales ss, item i
WHERE  ss.s_item_sk = i.i_item_sk
AND  ss.s_date_sk = dd.d_date_sk
```

A.1.2 TPC-H Queries

```
Q1
SELECT l_returnflag, l_linestatus, SUM(l_quantity)
AS sum_qty, SUM(l_extendedprice) AS sum_base_price,
SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS
sum_charge, AVG(l_quantity) AS AVG_qty, AVG(l_extendedprice)
AS AVG_price, AVG(l_discount) AS AVG_disc, count(*) AS count_order
FROM  lineitem  WHERE
l_shipdate <= date '1998-12-01' - interval '108' day
group by  l_returnflag,l_linestatus



Q3
SELECT l_orderkey,SUM(l_extendedprice * (1 - l_discount))
AS revenue,o_orderdate,o_shippriority
FROM customer,orders,lineitem  WHERE c_mktsegment = 'AUTOMOBILE'
AND  c_custkey = o_custkey AND  l_orderkey = o_orderkey
AND  o_orderdate < date '1995-03-13' AND
l_shipdate > date '1995-03-13'
group by l_orderkey,o_orderdate,o_shippriority



Q4
SELECT o_orderpriority, count(*) AS order_count
FROM orders  WHERE o_orderdate >= date '1995-01-01' AND
o_orderdate < date '1995-01-01' + interval '3' month
AND  exists (   SELECT *  FROM          lineitem  WHERE
l_orderkey = o_orderkey AND  l_commitdate < l_receiptdate)
group by o_orderpriority



Q6
SELECT SUM(l_extendedprice * l_discount) AS revenue
FROM lineitem
WHERE  l_shipdate >= date '1994-01-01' AND
l_shipdate < date '1994-01-01' + interval '1' year
AND  l_discount between 0.06 - 0.01  AND
0.06 + 0.01 AND  l_quantity < 24;



Q9
SELECT nation,o_year,SUM(amount) AS sum_profit
FROM (  SELECT  n_name AS nation, extract(year  FROM  o_orderdate)
   AS o_year,
l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity
AS amount
FROM part, supplier, lineitem, partsupp, orders, nation
WHERE s_suppkey = l_suppkey AND ps_suppkey = l_suppkey AND
ps_partkey = l_partkey  AND  p_partkey = l_partkey
```

```
AND  o_orderkey = l_orderkey
AND  s_nationkey = n_nationkey
AND  p_name like '%dim%') AS profit
group by nation,o_year
```

```
Q12
SELECT l_shipmode, SUM(case when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1  else 0 end) AS high_line_count,
SUM(case when o_orderpriority <> '1-URGENT'
AND  o_orderpriority <> '2-HIGH' then 1 else 0 end)
AS low_line_count
FROM orders,lineitem
WHERE o_orderkey = l_orderkey AND
l_shipmode in ('RAIL', 'FOB') AND
l_commitdate < l_receiptdate AND
l_shipdate < l_commitdate AND
l_receiptdate >= date '1997-01-01' AND
l_receiptdate < date '1997-01-01' + interval '1' year
group by l_shipmode
```

```
Q13'
SELECT c_count, COUNT(*) AS custdist
FROM  (
SELECT c.custkey AS c_custkey, COUNT(o.orderkey) AS c_count
FROM  customer c, orders o
WHERE  c.custkey = o.custkey
AND  (o.comment NOT LIKE '%special%requests%')
group by  c.custkey) c_orders
group by  c_count;
```

```
Q16'
SELECT p_brand, p_type, p_size, count(distinct ps_suppkey)
as supplier_cnt
FROM partsupp, part
WHERE  p_partkey = ps_partkey AND p_brand <> 'Brand#34'
AND p_type not like 'LARGE BRUSHED%'
AND p_size in (48, 19, 12, 4, 41, 7, 21, 39)
AND ps_suppkey in ( SELECT s_suppkey FROM supplier
WHERE s_comment not like '%Customer%Complaints%' )
GROUP BY  p_brand, p_type, p_size
```

```
Q18
SELECT c_name,c_custkey,o_orderkey,o_orderdate,
o_totalprice,SUM(l_quantity)
FROM customer,orders,lineitem
WHERE o_orderkey in (SELECT l_orderkey
FROM           lineitem
```

```
group by                l_orderkey having
   SUM(l_quantity) > 314)
AND  c_custkey = o_custkey AND  o_orderkey = l_orderkey
group by
c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice
```

A.1.3 TPC-DS Queries

```
Q3
SELECT dt.d_year, i.i_brand _id, i.i_brand ,
SUM(ss.ss_ext_sales_price) AS sum_agg
FROM  date_dim dt, store_sales ss, item i
WHERE  dt.d_date_sk = ss.ss_sold_date_sk
AND  ss.ss_item_sk = i.i_item_sk
AND  dt.d_moy = 12
AND  i.i_manufact_id = 436
group by dt.d_year, i.i_brand , i.i_brand _id;
```

```
Q7
SELECT i.i_item_id, AVG(ss.ss_quantity)
AS agg1, AVG(ss.ss_list_price)
AS agg2, AVG(ss.ss_coupon_amt) AS agg3,
AVG(ss.ss_sales_price) AS agg4
FROM  store_sales ss, customer_demographics cd,
date_dim d, item i, promotion p
WHERE  ss.ss_item_sk = i.i_item_sk
AND  ss.ss_sold_date_sk = d.d_date_sk
AND  ss.ss_cdemo_sk = cd.cd_demo_sk
AND  ss.ss_promo_sk = p.p_promo_sk
AND  cd.cd_gender = 'F'
AND  cd.cd_marital_status = 'W'
AND  (p.p_channel_email = 'N' OR p.p_channel_event = 'N')
AND  d.d_year = 1998
group by i.i_item_id;
```

```
Q19
SELECT i.i_brand_id, i.i_brand , i.i_manufact_id,
i.i_manufact, SUM(ss.ss_ext_sales_price) AS ext_price
FROM  date_dim dd, store_sales ss, item i,
customer c, customer_address ca, store s
WHERE  dd.d_date_sk = ss.ss_sold_date_sk
AND  ss.ss_item_sk = i.i_item_sk
AND  i.i_manager_id = 7
AND  dd.d_moy = 11
AND  dd.d_year = 1999
AND  ss.ss_customer_sk = c.c_customer_sk
AND  c.c_current_addr_sk = ca.ca_address_sk
AND  ss.ss_store_sk = s.s_store_sk
group by
i.i_br  , i.i_br, _id, i.i_manufact_id, i.i_manufact;
```

```
Q22
SELECT i.i_product_name, i.i_brand ,
i.i_class, i.i_category,
SUM(inv.inv_quantity_on_hand) AS qoh
FROM  date_dim dd, inventory inv, item i,
warehouse wh
WHERE  dd.d_date_sk = inv.inv_date_sk
AND  inv.inv_item_sk = i.i_item_sk
AND  inv.inv_warehouse_sk = wh.w_warehouse_sk
AND  dd.d_month_seq between 1193  AND  1204
group by
i.i_product_name, i.i_brand , i.i_clASs, i.i_category;
```

A.2 Queries for Evaluation of IEDYN

A.2.1 Queries in SQL Syntax for IEDYN and DBToaster

```
Q1: R(a,b,c), S(d,e,f)
SELECT *
FROM    R r, S s
WHERE   r.a < s.d
```

```
Q2: R(a,b,c,k), S(d,e,f,k)
SELECT *
FROM    R r, S s
WHERE   r.a < s.d
AND     r.k = s.k
```

```
Q3: R(a,b,c), S(d,e,f), T(g,h,i)
SELECT *
FROM R r, S s, T t
WHERE r.a < s.d
AND s.e < t.g
```

```
Q4: R(a,b,c), S(d,e,f), T(g,h,i)
SELECT *
FROM R r, S s, T t
WHERE r.a < s.d
AND  s.d < t.g
```

```
Q5: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
SELECT *
FROM R r, S s, T t
WHERE r.a < s.d
AND s.d < t.g
AND r.k = s.k
```

```
Q6: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
SELECT *
FROM R r, S s, T t
WHERE r.a < s.d
AND s.d < t.g
AND t.k = s.k


Q7: R(a,b,c), S(d,e,f), T(g,h,i)
SELECT a,b,d,e,f,g,h
FROM R r, S s, T t
WHERE r.a < s.d
AND  s.d < t.g


Q8: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
SELECT a,d,e,f,g,h,k
FROM R r, S s, T t
WHERE r.a < s.d
AND s.d < t.g
AND r.k = s.k


Q9: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
SELECT d,e,f,g,h,k
FROM R r, S s, T t
WHERE r.a < s.d
AND s.d < t.g
AND t.k = s.k


Q10: R(a,b,c), S(d,e,f), T(g,h,i)
SELECT b,c,e,f,h,i
FROM R r, S s, T t
WHERE r.a < s.d
AND  s.d < t.g


Q11: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
SELECT b,c,e,f,h,i
FROM R r, S s, T t
WHERE r.a < s.d
AND s.d < t.g
AND r.k = s.k


Q12: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
SELECT b,c,e,f,h,i
FROM R r, S s, T t
WHERE r.a < s.d
AND s.d < t.g
AND t.k = s.k
```

A.2.2 Queries in Esper EPL

```
Q1: R(a,b,c), S(d,e,f)
SELECT *
FROM    R.win(10000) as r, S.win(10000) as s
WHERE   r.a < s.d


Q2: R(a,b,c,k), S(d,e,f,k)
SELECT *
FROM    R.win(10000) as r, S.win(10000) as s
WHERE   r.a < s.d
AND     r.k = s.k


Q3: R(a,b,c), S(d,e,f), T(g,h,i)
SELECT *
FROM R.win(10000) as r, S.win(10000) as s, T t
WHERE r.a < s.d
AND s.e < t.g


Q4: R(a,b,c), S(d,e,f), T(g,h,i)
SELECT *
FROM R.win(10000) as r, S.win(10000) as s, T.win(10000) as t
WHERE r.a < s.d
AND  s.d < t.g


Q5: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
SELECT *
FROM R.win(10000) as r, S.win(10000) as s, T.win(10000) as t
WHERE r.a < s.d
AND s.d < t.g
AND r.k = s.k


Q6: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
SELECT *
FROM R.win(10000) as r, S.win(10000) as s, T t
WHERE r.a < s.d
AND s.d < t.g
AND t.k = s.k


Q7: R(a,b,c), S(d,e,f), T(g,h,i)
SELECT a,d,e,f,g,h
FROM R.win(10000) as r, S.win(10000) as s, T.win(10000) as t
WHERE r.a < s.d
AND  s.d < t.g;


Q8: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
SELECT a,d,e,f,g,h,k
FROM R.win(10000) as r, S.win(10000) as s, T.win(10000) as t
WHERE r.a < s.d
AND s.d < t.g
AND r.k = s.k
```

```
Q9: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
SELECT a,d,e,f,g,h,k
FROM R.win(10000) as r, S.win(10000) as s, T.win(10000) as t
WHERE r.a < s.d
AND s.d < t.g
AND t.k = s.j


Q10: R(a,b,c), S(d,e,f), T(g,h,i)
SELECT b,c,e,f,h,i
FROM R.win(10000) as r, S.win(10000) as s, T.win(10000) as t
WHERE r.a < s.d
AND  s.d < t.g


Q11: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
SELECT b,c,e,f,h,i
FROM R.win(10000) as r, S.win(10000) as s, T.win(10000) as t
WHERE r.a < s.d
AND s.d < t.g
AND r.k = s.k


Q12: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
SELECT b,c,e,f,h,i
FROM R.win(10000) as r, S.win(10000) as s, T.win(10000) as t
WHERE r.a < s.d
AND s.d < t.g
AND t.k = s.k
```

A.2.3 Queries in Tesla/TRex Rule Language Syntax

```
Q1: R(a,b,c), S(d,e,f)
Assign 31 => S, 30 => R, 32 => RS
Define RS(a:int, b:int, d:int, e:int, f:int, c:string)
From S() and
each R() within 99999999 from S
Where a:=R.a, b:=R.b, c:=R.c, d:=S.d,e:=S.e, f:=S.f;


Q2: R(a,b,c,k), S(d,e,f,k)\begin{verbatim}
Assign 31 => S, 30 => R, 32 => RS
Define RS(k:int, a:int, b:int, d:int, e:int, f:int, c:string)
From S(k=>$a) and
each R([int]k=$a) within 99999999 from S
Where Where k:=R.k, a:=R.a, b:=R.b, c:=R.c, d:=S.d,e:=S.e, f:=S.f;


Q4: R(a,b,c), S(d,e,f), T(g,h,i)\begin{verbatim}
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST( a:int, b:int, d:int, e:int, f:int, c:string, g:int, h:int,i:string)
From T() and
each S() within 99999999 from T and
each R() within 99999999 from S
Where Where a:=R.a, b:=R.b, c:=R.c, d:=S.d,e:=S.e, f:=S.f, g:=T.g, h:=T.h, i:=T.i;
```

```
Q5: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST(k:int, a:int, b:int, d:int, e:int, f:int, c:string, g:int, h:int,i:string)
From T() and
each S(k=>$a) within 99999999 from T and
each R([int]k=$a) within 99999999 from S
Where Where k:=R.k, a:=R.a, b:=R.b, c:=R.c, d:=S.d,e:=S.e, f:=S.f, g:=T.g, h:=T.h, i:=T.i
```

```
Q6: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST(k:int, a:int, b:int, d:int, e:int, f:int, c:string, g:int, h:int,i:string)
From T(k=>$a) and
each S([int]k=$a) within 99999999 from T and
each R() within 99999999 from S
Where Where k:=S.k, a:=R.a, b:=R.b, c:=R.c, d:=S.d,e:=S.e, f:=S.f, g:=T.g, h:=T.h, i:=T.i
```

```
Q7: R(a,b,c), S(d,e,f), T(g,h,i)
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST( a:int, d:int, e:int, f:int, g:int, h:int)
From T() and
each S() within 99999999 from T and
each R() within 99999999 from S
Where Where a:=R.a, d:=S.d,e:=S.e, f:=S.f, g:=T.g, h:=T.h;
```

```
Q8: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST(k:int, a:int, d:int, e:int, f:int,  g:int, h:int)
From T() and
each S(k=>$a) within 99999999 from T and
each R([int]k=$a) within 99999999 from S
Where Where k:=R.k, a:=R.a, d:=S.d,e:=S.e, f:=S.f, g:=T.g, h:=T.h;
```

```
Q9: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST(k:int, a:int,  d:int, e:int, f:int, g:int, h:int)
From T(k=>$a) and
each S([int]k=$a) within 99999999 from T and
each R() within 99999999 from S
Where Where k:=S.k, a:=R.a,d:=S.d,e:=S.e, f:=S.f, g:=T.g, h:=T.h;
```

```
Q10: R(a,b,c), S(d,e,f), T(g,h,i)
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST( b:int, e:int, f:int, c:string, h:int,i:string)
From T() and
each S() within 99999999 from T and
each R() within 99999999 from S
Where Where  b:=R.b, c:=R.c, e:=S.e, f:=S.f, h:=T.h, i:=T.i;
```

```
Q11: R(a,b,c,k), S(d,e,f,k), T(g,h,i)
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST(  b:int,  e:int, f:int, c:string,h:int,i:string)
From T() and
each S(k=>$a) within 99999999 from T and
each R([int]k=$a) within 99999999 from S
Where Where k:=R.k,  b:=R.b, c:=R.c,e:=S.e, f:=s.f, h:=T.h, i:=T.i;


Q12: R(a,b,c), S(d,e,f,k), T(g,h,i,k)
Assign 31 => S, 30 => R,32 => T, 33 => RST
Define RST(b:int, d:int, e:int, f:int, c:string, h:int,i:string)
From T(k=>$a) and
each S([int]k=$a) within 99999999 from T and
each R() within 99999999 from S
Where Where k:=S.k,  b:=R.b, c:=R.c,e:=S.e, f:=S.f, h:=T.h, i:=T.i;
```

A.2.4 SASE rule language

```
Q1: R(a,b,c), S(d,e,f)
PATTERN SEQ(R r, S s)
WHERE skip-till-any-match
WITHIN 1000000


Q2: R(a,b,c,k), S(d,e,f,k)
PATTERN SEQ(R r, S s)
WHERE partition-contiguity
AND [k]
WITHIN 99999999


Q4: R(a,b,c), S(d,e,f), T(g,h,i)
PATTERN SEQ(R r, S s, T t)
WHERE skip-till-any-match
WITHIN 99999999
```