Factorisation in Relational Databases

Jakub Závodný
Merton College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Trinity 2013

Abstract

We study representation systems for relational data based on relational algebra expressions with unions, products, and singleton relations. Algebraic factorisation using the distributivity of product over union allows succinct representation of many-to-many relationships; further succinctness is brought by sharing repeated subexpressions. We show that these techniques are especially applicable to results of conjunctive queries.

In the first part of the dissertation we derive tight asymptotic size bounds for two flavours of factorised representations of results of conjunctive queries. Any conjunctive query is characterised by rational parameters that govern the factorisability of its results independently of the database instance. We relate these parameters to fractional edge covers and fractional hypertree decompositions.

Factorisation naturally extends from relational data to its provenance. We characterise conjunctive queries by tight bounds on their readability, which captures how many times each input tuple is used to contribute to an output tuple, and we define syntactically the class of queries with bounded readability.

In the second part of the dissertation we describe FDB, a relational database engine that uses factorised representations at the physical layer to reduce data redundancy and boost query performance. We develop algorithms for optimisation and evaluation of queries with selection, projection, join, aggregation and order-by clauses on factorised representations. By introducing novel operators for factorisation restructuring and a new optimisation objective to maintain intermediate and final results succinctly factorised, we allow query evaluation with lower time complexity than on flat relations. Experiments show that for data sets with many-to-many relationships, FDB can outperform relational engines by orders of magnitude.

Acknowledgements

I would like to express most grateful thanks to my supervisor Dan Olteanu for accepting a lost doctoral student that I was and giving me so much support from day one. This dissertation owes much to his patient guidance, honest opinions, energy and positive attitude in the face of deadlines. I would also like to thank Nurzhan Bakibayev and Tomáš Kočiský for contributing to the implementation of FDB, my dissertation examiners Georg Gottlob and Jan Van den Bussche, and anonymous reviewers of my papers, for hundreds of comments that improved this work, and Robert Fink for providing technical wizardry and sanity checks when they were most needed.

I feel incredibly lucky to have spent four years as a doctoral student in Oxford, and I would like to thank everyone who has kept its magical spirit of doing things well; deeply and honestly. Personally let me thank František Simančík for displaying unfaltering diligence, Daniel Bundala for his supply of wise books and research topics, Irena & Sirup for most enjoyable programming sessions and technical discussions, and Rastislav Lenhardt for showing us how to stay on top of life while moving forward. My friends far away proved an invaluable source of refreshing perspectives and genuine genius, and helped me stay on track more than they may realise: thank you Busáč, Buggo, Martinko, Ondrej, Špilik, Tomáš. Finally but most importantly, I would like to thank my sister Eva and my parents for their patience, support and love.

Contents

1 Introduction **1**

1.1 Contributions 6

1.2 Organisation and Published Work 8

2 Preliminaries **11**

2.1 Relational Databases 11

2.2 Hypergraphs 15

3 Factorised Representations of Relations **19**

3.1 Factorised Representations 22

3.2 F-trees and D-trees 29

3.3 Factorisability of Query Results 38

3.4 Size Bounds 43

3.5 Succinctness Gap and Tree Decompositions 56

3.6 Computing F-representations and D-representations 70

3.7 Related Work 76

3.8 Conclusion and Directions for Future Work 78

4 Factorisation and Readability of Provenance **81**

4.1 Provenance Polynomials and Annotated Databases 85

4.2 Factorisation of Provenance Using F-Trees 88

4.3 Readability of Provenance 91

4.4 Dichotomy for Readability 95

4.5 Related Work 99
4.6 Conclusion and Directions for Future Work 101

5 FDB: A Factorised Database Engine **103**
5.1 Query Evaluation 113
5.2 Query Optimisation 132
5.3 Experimental Evaluation 138
5.4 Related Work 155
5.5 Conclusion and Directions for Future Work 159

Bibliography **160**

A Deferred Proofs **167**

Chapter 1

Introduction

For over four decades, the relational data model has been the prevalent formal framework for modelling and managing tabular data. In the relational model, tables containing data entries are modelled by relations, their rows are modelled by tuples, and their columns are called attributes. Queries can be made against the relations in relational calculus, relational algebra or a derived practical query language such as SQL [Cod70, AHV95]. The role of a database management system is to store data in a computer and answer queries issued by the user. Modelling data abstractly by relations at a logical level allows for complete freedom in its physical representation and storage in the database system, as long as the answers to queries issued by the user match those specified by the model. The principal challenge in developing database systems is to design efficient data structures that represent the relations, and algorithms that evaluate the queries.

Methods of representing and storing relational data are therefore of great importance for database systems. A pervasive use scenario for relational data is one where the tuples describe entities or relationships of a certain type and the attributes describe their properties. The number of attributes is small and fixed while the number of rows is variable and potentially very large. The principal paradigm for storing relational data is hence per row: data pertaining to each tuple is stored together, so that any single tuple can be retrieved or modified quickly even if nonlocal data access is expensive. This approach is adopted by major standard database systems such as PostgreSQL, Oracle and IBM DB2.

Other storage paradigms have been developed to suit other data characteristics and

access patterns: vertical partitioning and column stores where data for each column is stored separately to facilitate access to a given column for many rows at a time [AMH08], horizontal partitioning techniques that distribute data into partitions to store them in separate logical or physical locations [ANY04], and adaptive systems that combine different approaches based on the current workload [CMWM09]. Caching of most frequently accessed portions of the data in lowest latency storage is used to reduce the overall average access latency, and indexing is used to quickly find tuples containing a given key. Past research and development has produced a vast variety of data storage and access methods, all representing relational data at the logical level as defined by the relational model [RG03]. The choice of an appropriate storage method at the physical level depends among others on the use scenario: parameters of the stored data, the query workload, and the architecture of the computer system on which the database system runs.

Apart from data access and query processing speed, a key property of a data representation system is its succinctness: the amount of physical storage it requires to store the represented data. To reduce the storage requirements, traditional compression methods may be used to compress individual data values [IW94]. Compression across several data values can further reduce storage requirements if the values repeat, such as dictionary methods, or run-length encoding in column stores for columns with a small range of possible values [AMF06]. Designing a database system involves a trade-off between succinctness and efficient data access or query processing. We would like compact, non-redundant representations of relational data that support efficient query evaluation.

In this dissertation we study a class of succinct representation systems for relational data based on algebraic factorisation of relations allowed by the distributivity of the relational Cartesian product over union. Factorised representations address the redundancies of many-to-many relationships, such as in results of conjunctive queries. Query-aware factorisation of results can dramatically reduce the representation size while allowing for fast data access and efficient query evaluation algorithms.

Example 1.1. Consider a simplified database depicted in Figure 1, used by a pizzeria to store the ingredient composition of different pizzas on offer (in the relation Pizzas depicted

Pizzas

pizza	item
Margherita	base
Margherita	tomato
Margherita	cheese
Capricciosa	base
Capricciosa	tomato
Capricciosa	cheese
Capricciosa	ham
Capricciosa	mushrooms
Hawaii	base
Hawaii	tomato
Hawaii	cheese
Hawaii	ham
Hawaii	pineapple

Orders

customer	pizza
Mario	Capricciosa
Mario	Margherita
Pietro	Hawaii
Lucia	Hawaii
Lucia	Capricciosa
Giulio	Margherita
Giulio	Hawaii
Flavia	Margherita

Figure 1.1: An example database with a table of pizza ingredients and a table of pizza orders by customers.

to the left) and the orders of pizzas by customers (in the relation Orders depicted to the right). The relation Pizzas has two attributes named pizza and item, and each tuple of the relation states that a given pizza contains a given item, e.g., the last tuple states that pizza Hawaii contains pineapple. The relation can be seen as a union of all its tuples (more precisely, a union of relations containing the individual tuples), and each tuple is a product of two singleton relations (relations over one attribute that contain a single data value):

$$\begin{aligned}\text{Pizzas} = {} & \langle\text{Margherita}\rangle \times \langle\text{base}\rangle \cup \langle\text{Margherita}\rangle \times \langle\text{tomato}\rangle \cup \langle\text{Margherita}\rangle \times \langle\text{cheese}\rangle \cup \\ & \langle\text{Capricciosa}\rangle \times \langle\text{base}\rangle \cup \langle\text{Capricciosa}\rangle \times \langle\text{tomato}\rangle \cup \langle\text{Capricciosa}\rangle \times \langle\text{cheese}\rangle \cup \\ & \langle\text{Capricciosa}\rangle \times \langle\text{ham}\rangle \cup \langle\text{Capricciosa}\rangle \times \langle\text{mushrooms}\rangle \cup \langle\text{Hawaii}\rangle \times \langle\text{base}\rangle \cup \\ & \langle\text{Hawaii}\rangle \times \langle\text{tomato}\rangle \cup \langle\text{Hawaii}\rangle \times \langle\text{cheese}\rangle \cup \langle\text{Hawaii}\rangle \times \langle\text{ham}\rangle \cup \\ & \langle\text{Hawaii}\rangle \times \langle\text{pineapple}\rangle,\end{aligned}$$

where $\langle \text{v} \rangle$ denotes the singleton relation containing the value v. Using the distributivity of product over union, the relation Pizzas can be factorised e.g. as

$$\begin{aligned}\text{Pizzas} = {} & (\langle\text{Margherita}\rangle \cup \langle\text{Capricciosa}\rangle \cup \langle\text{Hawaii}\rangle) \times (\langle\text{base}\rangle \cup \langle\text{tomato}\rangle \cup \langle\text{cheese}\rangle) \cup \\ & \langle\text{Capricciosa}\rangle \times (\langle\text{ham}\rangle \cup \langle\text{mushrooms}\rangle) \cup \langle\text{Hawaii}\rangle \times (\langle\text{ham}\rangle \cup \langle\text{pineapple}\rangle).\end{aligned}$$

This nested factorisation encodes the same relation Pizzas as the flat union of products, but the factorisation contains only 12 singletons while the union of products contains 26 singletons: each product has two singletons and corresponds to one of the 13 tuples. □

The relationship between a flat relation and its factorised representation is on a par with the relationship between a logic function in disjunctive normal form and its equivalent nested formula obtained by algebraic factorisation. A given relation R can have many equivalent factorisations, all representing the same relation R, but with different nesting structures and different sizes. An important task for a database system that represents relations internally in factorised form is to quickly choose a small, or smallest, equivalent factorisation of a given relation.

An example of an extremely factorisable relation is a product of n relations with two tuples each, whose size is exponential in n but whose product-of-unions factorisation has size linear in n. Not all relations have exponentially small factorisations, in fact, by a simple counting argument, most relations are incompressible by more than a constant factor. The succinctness of factorisations is enabled by expressing combinations of groups of values symbolically instead of listing all possible combinations as in the flat representation (whether stored by row or by column). A naturally occurring class of relations exhibiting such combinations of groups are the results of relational joins.

Example 1.2. Consider the simplified pizzeria database from Example 1.1, with a table Pizzas containing the ingredients to different pizzas and a table Orders containing the orders of different pizzas by customers. A natural join query Orders $\bowtie_{\text{pizza}}$ Pizzas finds a list of all ingredients ordered in pizzas by all customers. For the example database from

Figure 1, the query result (shown truncated) is:

customer	pizza	item
Mario	Capricciosa	base
Mario	Capricciosa	tomato
Mario	Capricciosa	cheese
Mario	Capricciosa	ham
Mario	Capricciosa	mushrooms
Mario	Margherita	base
Mario	Margherita	tomato
	...	

For each pizza the result contains a tuple with each combination of an ingredient of that pizza and a customer who bought that pizza. These combinations can be factored into a product of a union of ingredients and a union of customers as follows:

$$\begin{aligned}&\langle\text{Capricciosa}\rangle \times (\langle\text{Mario}\rangle \cup \langle\text{Lucia}\rangle)\times\\&\quad(\langle\text{base}\rangle \cup \langle\text{tomato}\rangle \cup \langle\text{cheese}\rangle \cup \langle\text{ham}\rangle \cup \langle\text{mushrooms}\rangle) \cup\\&\langle\text{Margherita}\rangle \times (\langle\text{Mario}\rangle \cup \langle\text{Giulio}\rangle \cup \langle\text{Flavia}\rangle) \times (\langle\text{base}\rangle \cup \langle\text{tomato}\rangle \cup \langle\text{cheese}\rangle) \cup\\&\langle\text{Hawaii}\rangle \times (\langle\text{Pietro}\rangle \cup \langle\text{Lucia}\rangle \cup \langle\text{Giulio}\rangle)\times\\&\quad(\langle\text{base}\rangle \cup \langle\text{tomato}\rangle \cup \langle\text{cheese}\rangle \cup \langle\text{ham}\rangle \cup \langle\text{pineapple}\rangle).\end{aligned}$$

This factorisation of the example query result contains 24 singletons, compared to the 102 singletons in the flat list of all 34 tuples. Indeed, for any input database, the result of the join Orders $\bowtie_{\text{pizza}}$ Pizzas admits a factorisation with the same nesting structure, where the ingredients and the customers for each pizza are factored out. □

Designing a database system that successfully exploits the factorisability of join query results inevitably faces the challenges of determining the nesting structures that join query results admit, assessing and comparing the succinctness of these factorisations, and classifying join queries by their factorisability.

While a nested factorisation of a relation or a query result may reveal its internal structure and be of independent interest to the user, in this work we consider factorisations solely as a representation system for relational data: we envisage a database system using factorisations to represent relations at the physical level while presenting the data as flat relations to the user. Any such database system must allow for fast conversion between relations and their factorised representations. In cases where factorisation is used to succinctly represent a query result, evaluating the query directly in factorised form presents an opportunity to outperform relational systems that compute the large flat result explicitly.

A workable database system must also balance the succinctness of factorisations and the ease of data access and processing. Reducing storage requirements is of limited use unless the database system can perform a range of query processing tasks: quickly access or update the individual tuples of the stored relation, enumerate them in an order specified by the user, and perform query operators such as selections, joins or aggregates on the represented data. Each of the required tasks presents a challenge to find algorithms that evaluate the operation on the underlying relation even though the list of tuples is not directly available, but also an opportunity to take advantage of factorisation succinctness and outperform the relational methods by the virtue of processing a smaller representation of the data.

1.1 Contributions

This dissertation addresses the questions of factorisation succinctness and the challenges of query processing on factorisations as outlined above. The main contributions are the introduction of factorised representations, the characterisation of factorisability of conjunctive query results and of their provenance, and the development of algorithms for evaluation and optimisation of conjunctive queries with aggregation and ordering. We demonstrate experimentally that in some scenarios database systems can benefit from representing relations in factorised form, achieving improvements of several orders of magnitude in both storage requirements and processing time.

Factorised Representations and Size Bounds

The central subject of this dissertation is the idea of representing relational data in factorised form. We formally define the formalism of factorised representations for relations, extend it to factorised representations with definitions, and show the soundness and completeness of both representation systems. Under mild conditions it is shown that the tuples of a factorised representation (with or without definitions) can be enumerated with delay linear in the size of the tuples, which is optimal.

We propose factorisation schemas, called f-trees and d-trees, that define nesting structures for factorisations suitable for representing conjunctive query results. For any conjunctive query Q, we characterise the f-trees and d-trees that always factorise the results of Q, and calculate parameters $s(Q)$ and $s^{\uparrow}(Q)$ that characterise tight asymptotic bounds on the sizes of the factorisations without and with definitions respectively. By relating f-trees and d-trees to fractional hypertree decompositions [GM06] of Q we quantify the relationships between $s(Q)$, $s^{\uparrow}(Q)$ and existing decomposition widths of Q.

Factorisation and Readability of Provenance

The study of factorisation of query results can be naturally extended to factorisation of provenance, we consider in particular the factorisation of provenance polynomials [GKT07] for results of conjunctive queries. In addition to extending size bounds to factorisations of provenance polynomials, we characterise query results by asymptotic bounds on their readability. The readability of a polynomial is the minimal k such that each factorisation of the polynomial contains at least k occurrences of some variable. Our characterisation subsumes prior results on read-once provenance [DS04] and extends them to a dichotomy between queries with bounded and unbounded readability.

FDB: A Factorised Database Engine

The second part of the dissertation develops the principal application of factorised representations of relations: their use in database systems to internally represent relations and query results, and thereby decrease storage requirements and boost query processing. We

present algorithms to compute results of conjunctive queries directly in factorised form. For equi-join queries the algorithms are quasilinear in the output size and they outperform query evaluation with flat relational output whenever the factorisation is asymptotically smaller than the flat relational result.

For factorisations without definitions we build a framework of operators that execute atomic selection, projection, join and aggregation queries, together with novel operators that modify the factorisation nesting structure but leave the represented relation unchanged. We present quasi-optimal algorithms to execute each of the operators. We show that any query with selections, projections, joins, aggregation and ordering can be compiled into a sequence of such operators, we discuss new optimisation objectives of choosing succinct representations for intermediate results that arise in the presence of factorisations, and present both exact and heuristic approaches to query optimisation.

The presented techniques for optimisation and evaluation of queries on factorised representations of relational data are implemented in a prototype system FDB (Factorised DataBase), which we evaluate experimentally. We demonstrate that the succinctness gap between flat relations and their factorised representations follow the theoretical results and can be several orders of magnitude in practice. The performance gap follows the succinctness gap when input to queries is factorised, when the output is allowed in factorised form, or in the cases of limit queries. We show that FDB outperforms major open-source relational engines in such settings and discuss the use scenarios in which database systems benefit from using factorised representation for relational data.

1.2 Organisation and Published Work

This chapter is intended to provide a high-level overview of the context, subject and contributions of this dissertation. Chapter 2 defines the basic terminology and notation. Chapter 3 defines the framework of factorised representations, establishes the asymptotic bounds on their size and their relationship to other query decomposition measures, and presents an algorithm for computing factorised representations of conjunctive query results. Chapter 4 extends these notions to the factorisation of query result provenance and presents the read-

ability dichotomy. Chapter 5 describes query evaluation and optimisation algorithms for factorised representations, presents their experimental evaluation in the FDB query engine, and analyses its results. Each chapter begins with an introduction that presents a high-level view of the contents of the chapter, together with explanatory and intuitive examples, and ends with a discussion of related and prior work. To preserve the flow of the text, long technical proofs or parts of proofs that do not add major insights are deferred to the appendix.

Parts of the work presented in this dissertation have been published or submitted for publication to peer-reviewed journals and conferences, jointly with my supervisor Dan Olteanu and other collaborators. This thesis is based on the work with Dan Olteanu on factorised representations presented at ICDT 2012 [OZ12], where factorised representations of relations were first introduced. The extended results on factorised representations and size bounds as they appear in Chapter 3 have been submitted for publication in TODS [OZ13]. The ideas behind Chapter 4 were first outlined at the TaPP workshop [OZ11] and further developed in [OZ12]. Results on FDB presented in Chapter 5 were published in PVLDB publications [BOZ12b, BOZ12a, BKOZ13] co-authored with Dan Olteanu, Nurzhan Bakibayev (who contributed the bulk of the implementation of FDB) and Tomáš Kočiský (who collaborated on results in Section 5.1.6).

Chapter 2

Preliminaries

2.1 Relational Databases

Throughout this dissertation we consider relational databases with named attributes, we express queries in relational algebra and interpret them under set semantics. In this section we outline the basic definitions and notation used, for a comprehensive overview of relational databases we refer the reader to standard textbooks [AHV95].

Relations and Databases

An *attribute* A is any symbol from a fixed set **att** of attributes and a *schema* $\mathcal{S}$ is a set of attributes. A *tuple* t over a schema $\mathcal{S}$ is a mapping from $\mathcal{S}$ to a fixed domain **dom**. A *relation* $\mathbf{R}$ over a schema $\mathcal{S}$ is a set of tuples over $\mathcal{S}$. (Occasionally we will discuss the case of bag semantics, in which case we consider a relation as a multiset, also called a bag, of tuples.) A *database* $\mathbf{D}$ is a collection of named relations. The size $|\mathbf{R}|$ of a relation $\mathbf{R}$ is the number of its tuples; the size $|\mathbf{D}|$ of a database $\mathbf{D}$ is the sum of the sizes of its relations.

Conjunctive Query Syntax.

In the first part of the thesis we consider *conjunctive queries* written in relational algebra form:

$$\pi_{\mathcal{P}}(\sigma_{\psi}(R_1 \times \ldots \times R_n)),$$

where $R_1, \ldots, R_n$ are distinct relation symbols over disjoint schemas $\mathcal{S}_1, \ldots, \mathcal{S}_n$ whose union $\mathcal{S}$ is called the schema of the query, ψ is a conjunction of equalities of the form $A_1 = A_2$ with attributes A_1 and A_2 from $\mathcal{S}$, and the projection list $\mathcal{P}$ is a subset of $\mathcal{S}$.

The equivalence class of an attribute A in Q is the set consisting of A and of all attributes of $\mathcal{S}$ that are transitively equal to A in the selection condition ψ of Q. The attributes in $\mathcal{P}$ are called the *head* attributes. If $\mathcal{P} = \mathcal{S}$ we can drop the projection $\pi_{\mathcal{P}}$ and Q is called an *equi-join* query. The equi-join of Q is the equi-join query $\hat{Q} = \sigma_\psi(R_1 \times \ldots \times R_n)$. The *size* of Q is $|Q| = n$.

Conjunctive Query Semantics

The result $Q(\mathbf{D})$ of the query $Q = \pi_{\mathcal{P}}(\sigma_\psi(R_1 \times \ldots \times R_n))$ on a database $\mathbf{D}$ containing relations $\mathbf{R}_1, \ldots, \mathbf{R}_n$ is a relation of schema $\mathcal{P}$ consisting of all tuples t for which there exists a tuple t' over $\mathcal{S} = \bigcup_i \mathcal{S}_i$ such that

- t is a restriction of t' to $\mathcal{P}$,
- for each $i = 1, \ldots, n$, the restriction of t' to $\mathcal{S}_i$ is in the relation $\mathbf{R}_i$ of $\mathbf{D}$, and
- $t \models \psi$, i.e., for all equalities $A_1 = A_2$ in ψ we have $t(A_1) = t(A_2)$.

Although the relation symbols are distinct and have disjoint schemas, they may be mapped to the same database relation. We thus consider queries with self-joins, though avoid the explicit use of aliases and renaming operators in the algebraic expressions. We assume without loss of generality that this correspondence between relation symbols, as well as a correspondence between the attributes of relation symbols mapped to the same database relation, are given together with the query. The database $\mathbf{D}$ then only contains one relation instance for each set of relation symbols mapped to the same database relation.

Natural Joins

In cases where precise attribute naming is unimportant, such as in examples, we occasionally make the selection condition implicit using the less cluttered natural join notation

$$R_1(u_1) \bowtie R_2(u_2) \bowtie \cdots \bowtie R_n(u_n),$$

where each u_i is a list of variables of the same size as $\mathcal{S}_i$. Such expressions have the usual semantics, the above expression is equivalent to the query $\pi_{\mathcal{P}}\sigma_{\psi}(R_1 \times \cdots \times R_n)$ where ψ equates all attributes marked by the same variable names in the join expression, and $\mathcal{P}$ contains one attribute corresponding to each distinct variable name.

Selections with Constants

Beyond conjunctive queries we will make use of the relational algebra selection operator with a condition involving constants. Formally, besides equalities of attributes $A_i = A_j$ we allow conditions of the form $A_i \, \theta \, c$, where A_i is an attribute, c is a constant from the domain, and θ is a binary operator. We often use the shorthand $\sigma_{\mathcal{S}=t}$ where t is a tuple of schema $\mathcal{S}$ to mean the selection σ_{φ} with $\varphi = \bigwedge_{A \in \mathcal{S}} A = t(A)$.

Query Restrictions

For a conjunctive query $Q = \pi_{\mathcal{P}}(\sigma_{\psi}(R_1 \times \ldots \times R_n))$, a database $\mathbf{D}$, and a set $S \subseteq \mathcal{P}$, an *S-restriction* is a query-database pair $(Q_S, \mathbf{D}_S)$ defined as follows.

The query Q_S is the equi-join $\sigma_{\psi_S}(R_1^S \times \cdots \times R_n^S)$, where ψ_S and R_i^S are ψ and R_i respectively, restricted to attributes equivalent to S under ψ (i.e., to the union of equivalence classes of attributes in S). We consider all relation symbols R_i^S as referring to a separate relation instance even if some of the R_i referred to the same relation instance.

The database $\mathbf{D}_S$ consists of the relation instances $\mathbf{R}_i^S$ that are obtained by projecting each relation instance $\mathbf{R}_i$ from $\mathbf{D}$ onto the attributes in the transitive closure of S. In case Q has self-joins, we create a separate copy of each relation instance for each relation symbol referring to it before taking the projection. (Different relation symbols referring to the same relation instance may have different attributes in the transitive closure of S.)

The size of the query result $Q_S(\mathbf{D}_S)$ is related to the projection of $Q(\mathbf{D})$ to S as follows.

Lemma 2.1. *Let Q be a query, $\mathbf{D}$ be a database, and $(Q_S, \mathbf{D}_S)$ be the S-restriction for a subset S of the set of head attributes of Q. Then, $|\pi_S(Q(\mathbf{D}))| \leq |Q_S(\mathbf{D}_S)|$.*

Attribute Dependencies

Two disjoint collections $\mathcal{A}$ and $\mathcal{B}$ of attributes of a relation $\mathbf{R}$ are called *independent conditioned on* another collection of attributes $\mathcal{C}$ if $\mathbf{R}$ is a natural join $\mathbf{R}_A \bowtie_{\mathcal{C}} \mathbf{R}_B$ of two relations $\mathbf{R}_A$ and $\mathbf{R}_B$ with attributes including $\mathcal{A}$ and $\mathcal{B}$ respectively. $\mathcal{A}$ and $\mathcal{B}$ are *independent* if they are independent conditioned on the empty set. If two attributes are not conditionally independent, they are *dependent*.

In case the relation $\mathbf{R}$ is the result of a conjunctive query Q, we can deduce specific dependency information by static analysis of Q. For a query Q, two head attributes A and B are *Q-dependent* if any of the following statements hold:

- they belong to the same relation in Q,
- there is a chain of relations $R_1, \ldots, R_k$ in Q such that A is in the schema of R_1, B is in the schema of R_k, and each successive R_i and R_{i+1} are joined on an attribute that does not belong to the projection list $\mathcal{P}$ and neither does any equivalent attribute,
- A is equivalent to A' and B is equivalent to B' where A' and B' are Q-dependent by either of the two statements above.

Lemma 2.2. *If the attributes A and B are Q-dependent for a query Q, then there exists a database $\mathbf{D}$ for which A and B are dependent in the relation $Q(\mathbf{D})$. If A and B are not Q-dependent then for all databases $\mathbf{D}$, A and B are not dependent in the relation $Q(\mathbf{D})$.*

Aggregation and Ordering Queries

Conjunctive queries are composed of the standard relational algebra operations for product, equality selection and projection. In the second part of the dissertation we extend the class of queries under consideration by selection operators with constants and additional relational algebra operators for aggregation, ordering, and limit. The ordering and limiting operators assume that each relation is an ordered set of tuples.

- The aggregation operator $\varpi_{G;\alpha_1 \leftarrow F_1, \ldots, \alpha_k \leftarrow F_k}$ groups the input tuples by the attributes in the set G and then applies the aggregation functions F_1 to F_k on the tuples within each group. For each group of input tuples the output contains a single tuple with

matching values of attributes in G, and the results of the functions F_1 to F_k on that group in new attributes named α_1 to α_k, respectively. We consider standard aggregation functions: sum, count, min, and max; avg can be modelled by a pair of (sum, count) aggregation functions.

- The ordering operator o_G orders lexicographically the input relation by the list G of attributes, where each attribute is followed by $\uparrow$ or $\downarrow$ for ascending or descending order, respectively; by default, the order is ascending and omitted.
- The limiting operator λ_k outputs the first k input tuples in the input order.

SQL

Conjunctive queries implement SQL queries of the form

$$\texttt{select distinct } \mathcal{P} \texttt{ from } R_1, \ldots, R_n \texttt{ where } \psi.$$

The additional operators implement SQL queries with `group by` clauses and aggregation functions in the list $\mathcal{P}$, and with `order by` and `limit` clauses. SQL `having` clauses, which are conjunctions of conditions involving aggregate functions, attributes in the group-by list and constants, are readily supported. Such a clause can be implemented by adding its aggregate functions to the aggregate operator and by adding on top of the query a selection operator whose condition is that of the clause.

Computational Model

We use the uniform-cost RAM model where the values of the domain **dom** as well as the pointers into the database are of constant size.

2.2 Hypergraphs

Hypergraphs

A *hypergraph* $H = (V, E)$ consists of a set V and a set E of nonempty subsets of V. The elements of V are called *nodes* or *vertices*, the elements of E are called *edges* or *hyperedges*.

If each edge of a hypergraph contains exactly two vertices, the hypergraph is a graph. For any graph or hypergraph H, $V(H)$ denotes its set of vertices and $E(H)$ its set of edges.

Query Hypergraphs

Let Q be an equi-join query. The *hypergraph of* Q has one vertex for each equivalence class of attributes of Q and one edge for each relation symbol R containing all vertices with attributes of R.

Edge Covers and Independent Sets

Let $H = (V, E)$ be a hypergraph. An *edge cover* of H is a set of edges $C \subseteq E$ that covers all nodes of H, i.e., $\bigcup C = V$. The edge cover number of H is the size of the smallest edge cover of H. An *independent set* in H is a set of nodes $S \subseteq V$ such that no edge of H contains more than one node from S. The independence number of H is the size of the largest independent set of H.

The *fractional edge cover number* of $H = (V, E)$ is the cost of an optimal solution to the linear program with variables $\{x_e\}_{e \in E}$:

$$\begin{array}{lll} \text{minimise} & \sum_{e \in E} x_e & \\ \text{subject to} & \sum_{e : v \in e} x_e \geq 1 & \text{for each node } v, \\ & x_e \geq 0 & \text{for each edge } e. \end{array}$$

The linear program assigns a nonnegative weight x_e to each edge e. The conditions state that each vertex of H must be covered by edges whose sum of the weights is greater than 1. The objective is to minimise the total weight of all edges. By restricting the variables x_e to the values 0 and 1 we obtain the standard non-weighted version of edge cover by interpreting x_e as the indicator function of the event $e \in C$. In the fractional version the variables can hold any nonnegative real number, though the optimal solution is always rational.

The *fractional independence number* of H is the cost of a maximal solution to the linear program with variables $\{y_v\}_{v \in V}$ assigning weights to vertices of H and constraining the total weight of all vertices in each edge to be at most one; the fractional relaxation of the

independent set problem.

$$\begin{array}{lll} \text{maximise} & \sum_{v\in V} y_v & \\ \text{subject to} & \sum_{v:v\in e} y_v \leq 1 & \text{for each edge } e, \\ & x_v \geq 0 & \text{for each node } v. \end{array}$$

This linear program is dual to the linear program for fractional edge cover number, and by the linear programming duality theorem the optimal solutions of both programs have equal total weight [SU11]. The fractional edge cover number and the fractional independence numbers are therefore equal for any hypergraph H, we denote them by $\rho^*(H)$.

Tree Decompositions and Fractional Hypertree Decompositions

The notions of tree decompositions and various hypertree decompositions of hypergraphs, and the associated decomposition widths, are used to identify tractable classes of equip-join queries, constraint satisfaction problems, and other generally intractable problems [GLS99, GM06]. In this work we use the notions of tree decompositions and fractional hypertree decompositions for hypergraphs as defined in [GM06].

A *tree decomposition* of a hypergraph H is a pair $(T, (B_t)_{t\in V(T)})$ where

- T is a tree, and
- $(B_t)_{t\in V(T)}$ is a family of sets of vertices of H, called *bags*, labelled by the vertices of T, such that each edge of H is contained in some bag B_t, and for each vertex v of H the set $\{t : B_t \ni v\}$ of bags containing v is connected in T.

A *fractional hypertree decomposition* of H is a triple $(T, (B_t)_{t\in V(T)}, (\gamma_t)_{t\in V(T)})$, such that $(T, (B_t))$ is a tree decomposition and

- $(\gamma_t)_{t\in V(T)}$ is a family of *weight functions* $E(H) \mapsto [0, \infty)$ such that for each $t \in V(T)$, γ_t covers all vertices of B_t, i.e. $\sum_{e\ni v} \gamma_t(e) \geq 1$ for all $v \in B_t$.

The weight of a weight function γ_t is $\text{weight}(\gamma_t) = \sum_{e\in E(H)} \gamma_t(e)$ and the width of the decomposition is $\max_{t\in V(T)} \text{weight}(\gamma_t)$. The *fractional hypertree width* of H, $\text{fhw}(H)$, is the minimum possible width of a fractional hypertree decomposition of H.

In a fractional hypertree decomposition of a hypergraph H, each weight function γ_t must

be a fractional edge cover of the hypergraph H restricted to the vertices of B_t. Since we are primarily interested in fractional hypertree decompositions of minimum possible width, for a given tree decomposition $(T, (B_t))$ we often consider each γ_t to be an optimal fractional edge cover of B_t, and hence obtain a minimum-width extension of $(T, (B_t))$ into a fractional hypertree decomposition $(T, (B_t), (\gamma_t))$. By the *fractional width* of a tree decomposition we mean the width of its minimal fractional extension; note that $\mathrm{fhw}(H)$ is the minimal possible fractional width of a tree decomposition of H. This treatment also allows us to sidestep the concept of hypertree decomposition and its variations.

Chapter 3

Factorised Representations of Relations

In this chapter we introduce and study two compact representation systems for relational data based on algebraic factorisation. Both representation systems are defined using relational algebra expressions with unions, products, and singleton relations (i.e., relations with one tuple over one attribute): f-representations, which employ algebraic factorisation using distributivity of product over union, and d-representations, which are f-representations where further succinctness is brought by explicit sharing of repeated subexpressions. The relationship between a relation encoded as a list of tuples and an equivalent factorised representation is on a par with the relationship between logic functions in disjunctive normal form and their equivalent nested formulas obtained by algebraic factorisation. Similarly, the relationship between f-representations and d-representations is on a par with the relationship between formulas and circuits for logic functions.

Example 3.1. Consider the relation $R = \{(a, b, c) \in \mathbb{N}^3 : 1 \leq a < b < c \leq 5\}$ with tuples $(1, 2, 3)$, $(1, 2, 4)$, etc. If we write $\langle x \rangle$ for the singleton relation $\{x\}$, the tuples of R can be written as $\langle 1\rangle\times\langle 2\rangle\times\langle 3\rangle$, $\langle 1\rangle\times\langle 2\rangle\times\langle 4\rangle$, etc., and the relation R can be expressed by the flat

relational algebra expression

$$\begin{aligned} R = {} & \langle 1\rangle\times\langle 2\rangle\times\langle 3\rangle \;\cup\; \langle 1\rangle\times\langle 2\rangle\times\langle 4\rangle \;\cup\; \langle 1\rangle\times\langle 2\rangle\times\langle 5\rangle \;\cup\; \langle 1\rangle\times\langle 3\rangle\times\langle 4\rangle \;\cup\; \langle 1\rangle\times\langle 3\rangle\times\langle 5\rangle \cup \\ & \langle 1\rangle\times\langle 4\rangle\times\langle 5\rangle \;\cup\; \langle 2\rangle\times\langle 3\rangle\times\langle 4\rangle \;\cup\; \langle 2\rangle\times\langle 3\rangle\times\langle 5\rangle \;\cup\; \langle 2\rangle\times\langle 4\rangle\times\langle 5\rangle \;\cup\; \langle 3\rangle\times\langle 4\rangle\times\langle 5\rangle. \end{aligned}$$

A more succinct factorised representation of R would be for example

$$R = \langle 1\rangle\times\langle 2\rangle\times(\langle 3\rangle\cup\langle 4\rangle\cup\langle 5\rangle) \;\cup\; (\langle 1\rangle\cup\langle 2\rangle)\times\langle 3\rangle\times(\langle 4\rangle\cup\langle 5\rangle) \;\cup\; (\langle 1\rangle\cup\langle 2\rangle\cup\langle 3\rangle)\times\langle 4\rangle\times\langle 5\rangle.$$

Using d-representations, which use definitions to label shared subexpressions, the representation could be further compacted to

$$\begin{aligned} & X := \langle 1\rangle\cup\langle 2\rangle; \\ & Y := \langle 4\rangle\cup\langle 5\rangle; \\ & R = \langle 1\rangle\times\langle 2\rangle\times(\langle 3\rangle\cup Y) \;\cup\; X\times\langle 3\rangle\times Y \;\cup\; (X\cup\langle 3\rangle)\times\langle 4\rangle\times\langle 5\rangle. \end{aligned}$$

□

Both representation formalisms are complete for relational data in the sense that they can represent any relation instance. Moreover, they allow for fast retrieval of tuples of the represented relation: tuples can be enumerated with the same time complexity as listing them from the relation (time per tuple linear in tuple size, which is constant with respect to data complexity). Factorised representations can nevertheless be exponentially more succinct than traditional flat representations of relations as lists of tuples, e.g., in the presence of join dependencies and multi-valued dependencies in the relations. Results of conjunctive queries exhibit such dependencies and can be predictably factorised with an exponential succinctness gap that can be inferred from the query syntax alone. The nesting structures of succinct factorisations of query results can also be inferred from the query, and factorisations of query results can be computed directly from the input database, without first computing the result in flat relational form.

These representation systems naturally capture as a special case lossless decompositions defined by join dependencies, as investigated in the context of normal forms in database

design [AHV95], conditional independence in Bayesian networks [Pea89], and succinct representations of provenance polynomials of query results used for efficient computation in probabilistic databases [OH08, SDG10], as described in Chapter 4. It also captures product decompositions of relations as studied in the context of incomplete information [OKA08], as well as factorisations of relational data representing large, sparse feature matrices recently used to scale up machine learning algorithms [Ren13]. Tilings of relations by closed n-sets can also be naturally expressed as factorised representations [CBRB09], and the more general task of identifying products of unions (under the name of closed n-sets or formal concepts) in relations is well studied in the context of formal concept analysis and knowledge discovery [SWW98]. These existing decomposition techniques can be straightforwardly used to supply data in factorised form.

These representations lie at the foundation of a new kind of database systems, with relations at the logical layer and succinct, factorised representations at the physical layer. In the following chapters of this dissertation we introduce the main-memory query engine FDB that employs f-representations and supports the evaluation of select-project-join queries with aggregates and ordering, and show that f-representations can boost the performance of relational query processing by orders of magnitude.

In this chapter we formally define factorised representations and study their theoretical foundations, establishing the following properties:

- Factorised representations form a complete representation system for relational data. The tuples of a factorised representation with or without definitions can be enumerated with delay linear in the size of its schema and thus constant with respect to data complexity (Section 3.1).

- We introduce classes of factorised representations with the same nesting structures: these are so-called f-trees for f-representations, and d-trees for d-representations (Section 3.2). For a given conjunctive query, we can infer which f-trees and d-trees factorise all possible results of that query (Section 3.3).

- For any conjunctive query Q, there exist rational numbers $s(Q)$ and $s^{\uparrow}(Q)$ such that for any input database $\mathbf{D}$, the result $Q(\mathbf{D})$ has an f-representation of size $O(|\mathbf{D}|^{s(Q)})$

and a d-representation of size $O(|\mathbf{D}|^{s^{\uparrow}(Q)})$. These bounds complement the known bound $O(|\mathbf{D}|^{\rho^*(Q)})$ for the size of the flat relational result $Q(\mathbf{D})$, where $\rho^*(Q)$ is the fractional edge cover of an equi-join query Q. Our size bounds are asymptotically optimal within the class of factorisations defined by f-trees and d-trees (Section 3.4).

- Factorised representations for results of conjunctive queries can be computed directly from the query and the input database. For equi-join queries we give worst-case optimal algorithms: an f-representation of $Q(\mathbf{D})$ can be computed in time $O(|\mathbf{D}|^{s(Q)} \log |\mathbf{D}|)$ and a d-representation in time $O(|\mathbf{D}|^{s^{\uparrow}(Q)} \log |\mathbf{D}|)$ with respect to data complexity (Section 3.6).

- For results of equi-join queries, we quantify the succinctness gap between flat relations, f-representations, and d-representations, using the corresponding parameters $\rho^*(Q)$, $s(Q)$ and $s^{\uparrow}(Q)$. We show $1 \leq s^{\uparrow}(Q) \leq s(Q) \leq \rho^*(Q) \leq |Q|$, where the factor between $s(Q)$ and $s^{\uparrow}(Q)$ is at most logarithmic in the size of the schema, while the factor between $\rho^*(Q)$ and $s(Q)$ can be as large as $|Q|$ (Section 3.5).

- Finally, factorisation of equi-join query results using f-trees and d-trees is closely related to path decompositions and tree decompositions of the query. We give a two-way translation between d-trees and tree decompositions showing that $s^{\uparrow}(Q)$ equals the fractional hypertree width of Q, and a one-way translation from f-trees to path decompositions showing that $s(Q)$ is at least the fractional hyperpath width of Q (Section 3.5).

3.1 Factorised Representations

In this section we introduce the subject of this work, two succinct representation systems for relational data. The basic idea is to represent relations symbolically as expressions in a fragment of relational algebra consisting of union, Cartesian product, and so-called singleton relations, which are unary relations with one tuple. We call such representations *factorised representations* or *f-representations*, since they employ algebraic factorisation to nest products and unions and hence express combinations of values symbolically. Further

succinctness can be achieved by introducing symbolic references into the representations, so that repeated subexpressions can be defined only once and referred to several times. Factorised representations with definitions are called *d-representations*.

3.1.1 Factorised Representations

Factorised representations of relations are defined as typed relational algebra expressions consisting of unions, Cartesian products, and singleton relations [OZ12].

Definition 3.1. A *factorised representation*, or f-representation for short, over a schema $\mathcal{S}$ is a relational algebra expression of one of the following forms.

- $\emptyset$, representing the empty relation over schema $\mathcal{S}$,
- $\langle\rangle$, representing the relation consisting of the nullary tuple, if $\mathcal{S} = \emptyset$,
- $\langle A\!:\!a \rangle$, representing the unary relation with a single tuple with value a, if $\mathcal{S} = \{A\}$ and a is a value in the domain $\mathcal{D}$,
- $(E_1 \cup \cdots \cup E_n)$, representing the union of the relations represented by E_i, where each E_i is an f-representation over $\mathcal{S}$,
- $(E_1 \times \cdots \times E_n)$, representing the Cartesian product of the relations represented by E_i, where each E_i is an f-representation over some schema $\mathcal{S}_i$ such that $\mathcal{S}$ is the disjoint union of all $\mathcal{S}_i$.

The expressions $\langle A\!:\!a \rangle$ are called *singletons of type A* (or A-singletons for short) and the expression $\langle\rangle$ is called the nullary singleton. □

Any f-representation E over a schema $\mathcal{S}$ represents a relation $[\![E]\!]$ over $\mathcal{S}$, called the relation of E. Different f-representations can represent the same relation. Two f-representations E_1 and E_2 over the same schema are *equivalent* if $[\![E_1]\!] = [\![E_2]\!]$.

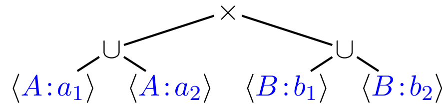
Example 3.2. The f-representation $(\langle A\!:\!a_1 \rangle \cup \langle A\!:\!a_2 \rangle) \times (\langle B\!:\!b_1 \rangle \cup \langle B\!:\!b_2 \rangle)$ over the schema $\{A, B\}$ represents the relation $\{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$ over $\{A, B\}$. □

Any relation has at least one f-representation, the so-called *flat* f-representation that is a (possibly empty) union of products of singletons, where each product of singletons represents a distinct tuple in the relation. This property of the representation system is called completeness.

Proposition 3.1. *Factorised representations form a complete representation system for relational data.*

Any f-representation has a parse tree whose internal nodes are unions and products, and whose leaves are singletons or empty relations. Relational algebra expressions and their parse trees are equivalent ways of describing f-representations, we will use them interchangeably in this dissertation. Relational algebra expressions are better suited for readability in text; their parse trees are better suited for formal analysis, proofs and algorithms. Parentheses in relational expressions will be omitted where this helps clarity.

Example 3.3. The f-representation $(\langle A\!:\!a_1\rangle \cup \langle A\!:\!a_2\rangle) \times (\langle B\!:\!b_1\rangle \cup \langle B\!:\!b_2\rangle)$ has the parse tree



□

3.1.2 Factorised Representations with Definitions

We now introduce the representation system of factorised representations with definitions, called d-representations. While f-representations eliminate redundancy in relations by expressing products of unions of expressions symbolically instead of listing all possible products of the expressions, they may still contain multiple copies of the same expression that cannot be removed by further factorisation. This redundancy can be eliminated by defining (and physically storing) the subexpression only once and referring to this definition (using a pointer to the single stored copy) at each of its occurrences in the representation.

Definition 3.2. A *factorised representation with definitions*, or d-representation for short, is a set of named expressions $\{(N_1, D_1), \ldots, (N_n, D_n)\}$, where each named expression (N_i, D_i) consists of a name N_i and an expression D_i with products, unions, singletons and names of other expressions. Formally, an expression over a schema $\mathcal{S}$ is of one of the following forms:

- $\emptyset$, representing the empty relation over $\mathcal{S}$,
- $\langle\rangle$, representing the relation consisting of the nullary tuple, if $\mathcal{S} = \emptyset$,
- $\langle A\!:\!a\rangle$, representing the unary relation with a single tuple with value a, if $\mathcal{S} = \{A\}$ and a is a value in the domain $\mathcal{D}$,

- $(E_1 \cup \cdots \cup E_n)$ representing the union of the relations represented by E_i, where each E_i is an expression over $\mathcal{S}$,
- $(E_1 \times \cdots \times E_n)$, representing the Cartesian product of the relations represented by E_i, where each E_i is an expression over schema $\mathcal{S}_i$ such that $\mathcal{S}$ is the disjoint union of all $\mathcal{S}_i$.
- a name N_i of another expression D_i over $\mathcal{S}$, representing the same relation as D_i.

The schema of a d-representation is the schema of its first expression D_1. We require that each D_i only contains names N_j with $j > i$, and that each name N_j with $j > 1$ is used at least once. In this thesis, the name of an expression D_i will always be ${}^{\uparrow}D_i$. □

Any d-representation D over a schema $\mathcal{S}$ represents a relation $[\![D]\!]$ over $\mathcal{S}$. For any d-representation D consisting of expressions $D_1, \ldots, D_n$, we can start with the root expression D_1 and repeatedly replace the names ${}^{\uparrow}D_j$ by the expressions D_j until we obtain a single expression without names, i.e., an f-representation. This f-representation is called the *traversal* of D, and D represents the same relation as its traversal.

Any f-representation E can be identified with the d-representation $\{E\}$. Therefore, d-representations also form a complete representation system for relational data.

Proposition 3.2. *Factorised representations with definitions form a complete representation system for relational data.*

Just as any f-representation has a parse tree, any d-representation has a directed acyclic parse graph, which can be constructed as a collection of the parse trees of the constituent expressions, in which any leaf corresponding to a name ${}^{\uparrow}D_j$ is identified with the root of the parse tree of expression D_j. The traversal of a d-representation viewed as a parse graph is constructed by listing its nodes in depth-first order without marking them as visited (thus possibly listing some nodes multiple times). More precisely, for any node N with edges to nodes $C_1, \ldots, C_n$, let traversal(N) be a tree with a copy of N as a root and traversal(C_1), ..., traversal(C_n) as children subtrees; then *traversal* of a d-representation D is traversal(root(D)).

Similar to f-representations, we use relational expressions with definitions and parse graphs as interchangeable ways to describe d-representations.

Example 3.4. Consider the relation $\mathbf{R}_n$ over schema $\{A_1, \dots, A_n\}$ whose tuples are all binary sequences $(a_1, \dots, a_n)$ with no two consecutive zeros. The d-representation consisting of

$$D_{1,0} = \langle A_1 : 0 \rangle, \quad D_{1,1} = \langle A_1 : 1 \rangle \qquad \text{and}$$
$$D_{k,0} = {}^{\uparrow}D_{k-1,1} \times \langle A_k : 0 \rangle, \quad D_{k,1} = ({}^{\uparrow}D_{k-1,0} \cup {}^{\uparrow}D_{k-1,1}) \times \langle A_k : 1 \rangle \qquad \text{for } k = 2, \dots, n,$$

and root $D = {}^{\uparrow}D_{n,0} \cup {}^{\uparrow}D_{n,1}$, represents the relation $\mathbf{R}_n$.[1] This can be seen by showing inductively over k that $D_{k,d}$ represents the relation $\sigma_{A_k=d}(\mathbf{R}_k)$. □

Since d-representations include all f-representations, all definitions and results for arbitrary d-representations mentioned in the sequel also apply to f-representations. Here we introduce several notions for d-representations (and f-representations alike) that are used later.

Definition 3.3. A d-representation is *normal* if

- it contains no empty relation or nullary singleton, unless it is itself the empty relation or the nullary singleton,
- all its products are at least binary, and
- no child of a union is a union.

□

Definition 3.4. By expanding a (non-empty, non-nullary) normal f-representation using the distributivity of product over union, we obtain an equivalent flat f-representation that is a union of products of non-nullary singletons, which we call *monomials*. The monomials of a normal d-representation are the monomials of its traversal. A normal f-representation or d-representation is *deterministic* if its monomials are all distinct. □

For a normal d-representation the monomials correspond to the tuples of the relation obtained by interpreting the d-representation under bag semantics. For a deterministic d-representation D the monomials are all distinct and hence they also correspond to the tuples of its relation $[\![D]\!]$ under set semantics.

[1] In Definition 3.2 the expressions D_i of a d-representation are indexed by natural numbers $i = 1, \dots, n$, but any partial order with a least element is sufficient and can be re-indexed by consecutive naturals.

3.1.3 Representation Size

The size of a d-representation is determined by the total length of the expressions, when stored as a set of factorised expressions and by the number of its nodes plus the number of its edges, when stored as a parse graph. Both measures are within a constant factor of each other; we will next use the former one. A further size measure is the number of singleton nodes in the representation.

Definition 3.5. The size $|E|$ of a d-representation E is the total number of its singletons, empty set symbols, unions, products, and names of other expressions. The number of singletons in E is denoted by $\|E\|$. □

Although any relation has a flat f-representation, nested f-representations can be exponentially more succinct than their equivalent flat f-representations, where the exponent is the size of the schema.

Example 3.5. The f-representation $(\langle A_1 : 0\rangle \cup \langle A_1 : 1\rangle) \times \ldots \times (\langle A_n : 0\rangle \cup \langle A_n : 1\rangle)$ has $2n$ singletons and size $4n-1$, while any equivalent flat f-representation has 2^{n+1} singletons and size $4 \cdot 2^n - 1$. □

By deduplicating common subexpressions, d-representations can represent relations even more succinctly than their traversal f-representations.

Example 3.6. The d-representation D from Example 3.4 has size $O(n)$, while the size of its traversal is exponential in n since only the singleton $\langle A_1 : 1\rangle$ occurs F_n times (F_n denotes the nth Fibonacci number). □

We will further study representation succinctness in Sections 3.4 and 3.5.

3.1.4 Constant-delay Enumeration of Encoded Tuples

Examples 3.5 and 3.6 show that f-representations and d-representations can be exponentially smaller than the relations they represent. The records of a relation encoded as a normal d-representation, or its f-representation traversal, can nevertheless be enumerated with the same complexity as listing them from the relation.

Proposition 3.3. *The monomials of a normal d-representation D over a schema $\mathcal{S}$ can be enumerated with $O(|\mathcal{S}|)$ delay and space.*

Proof. We assume that the d-representation is stored as a parse graph, but any representation allowing constant-time enumeration of elements of unions and products is sufficient.

Enumeration algorithm. We explore the d-representation D using depth-first search. For each union, we follow the edge to its first child, and construct a list $\mathcal{L}$ of nodes visited in pre-order. We then repeat the following.

1. We output the product of all singletons in $\mathcal{L}$.
2. We find the last union node U in the list $\mathcal{L}$ for which its child C in $\mathcal{L}$ is not its last child. If such U exists, we remove all nodes after C from $\mathcal{L}$ and replace C by the next child of U. If such U does not exist, we terminate.
3. We explore D using depth-first search. For each union in $\mathcal{L}$, we follow the edge to its child in $\mathcal{L}$, and for each union not in $\mathcal{L}$ we follow the edge to its first child. We update the list $\mathcal{L}$ to a list of nodes visited in pre-order during the search.

Termination. If we order the nodes of traversal(D) in pre-order, by each repetition of step (3) the list $\mathcal{L}$ becomes lexicographically greater. Since there are finitely many possible lists $\mathcal{L}$, the algorithm terminates.

Correctness. Each monomial of a given normal d-representation D is a product of the singletons reached by recursively exploring D, choosing one child at each union and all children at each product. Any choice of children at the unions of D corresponds to a monomial of D, therefore, each product output in step (1) is a monomial of D. Conversely, for any monomial m of D, consider the choice of children at each union that generates m, and let $\mathcal{L}_m$ be the pre-ordered list of nodes visited by a depth-first search of D that only follows the chosen children at each union. During the execution of the enumeration algorithm, the first union of $\mathcal{L}$ will cycle through its children, so eventually the child of the first union in $\mathcal{L}$ will be identical to $\mathcal{L}_m$. The first time this happens, all other unions in $\mathcal{L}_m$ will have their first child in $\mathcal{L}_m$. The next union in $\mathcal{L}$ is then the same as in $\mathcal{L}_m$, and at some point its child in $\mathcal{L}$ will be identical to $\mathcal{L}_m$. By induction we can show that at some point, all unions in $\mathcal{L}$ will have the same children in $\mathcal{L}$ as in $\mathcal{L}_m$, hence $\mathcal{L}$ and $\mathcal{L}_m$ will be

identical, and m will be output in the next execution of step (1). Moreover, since $\mathcal{L}$ strictly increases under lexicographic order, each monomial is only output once.

Delay and Space. For any choice of children at the unions, we reach exactly $|\mathcal{S}|$ singletons; one for each attribute. Since the products are at least binary, we reach at most $|\mathcal{S}|-1$ of them, and since there are no directly nested unions, the number of reached unions is $O(|\mathcal{S}|)$. The size of $\mathcal{L}$ is therefore $O(|\mathcal{S}|)$, and steps (1) and (2) of the algorithm take time $O(|\mathcal{S}|)$. The initial depth-first search takes time linear in the number of explored nodes, which is $O(|\mathcal{S}|)$. The same holds for the depth-first search in step (3). Its choices of children at unions are at first dictated by the list of nodes $\mathcal{L}$, but the nodes in $\mathcal{L}$ are listed in pre-order, so each can be accessed in constant time during the search. $\square$

Since the monomials of a normal deterministic d-representation D correspond to the distinct tuples of its relation $[\![D]\!]$, Proposition 3.3 implies the following.

Corollary 3.1 (Proposition 3.3)**.** *The tuples of a normal deterministic d-representation D over a schema $\mathcal{S}$ can be enumerated with $O(|\mathcal{S}|)$ delay and space.*

Note that $O(|\mathcal{S}|)$ delay is optimal since each tuple has size $O(|\mathcal{S}|)$. With respect to data complexity (where the schema size is constant), we can thus enumerate the tuples with constant delay and space similarly to listing them from a set.

3.2 F-trees and D-trees

In this section we define classes of normal f-representations and d-representations with the same nesting structures, called f-trees and d-trees respectively. We give exact conditions under which a relation admits an f-representation (or d-representation) over a given f-tree (respectively d-tree) and precisely characterise such representations if they exist.

In subsequent sections we define f-trees and d-trees over which *any* result of a given query admits representation, infer representation size bounds using the f-trees and d-trees, and give algorithms to compute representations over f-trees and d-trees.

3.2.1 F-trees for F-representations

We first introduce f-trees, which define the schemas as well as the nesting structures of f-representations.

Definition 3.6. An *f-tree* over a possibly empty schema $\mathcal{S}$ is a rooted forest with each node labelled by a non-empty subset of $\mathcal{S}$ such that each attribute of $\mathcal{S}$ occurs in exactly one node. □

We next define how an f-tree dictates the nesting structure of an f-representation.

Definition 3.7. We say that an f-representation E is over a given f-tree $\mathcal{T}$ if it satisfies the following:

- If $\mathcal{T}$ is empty, then $E = \emptyset$ or $E = \langle\rangle$.
- If $\mathcal{T}$ is a single node labelled by $\{A_1, \dots, A_k\}$, then

  $$E = \bigcup_a \langle A_1 : a\rangle \times \cdots \times \langle A_k : a\rangle$$

  where the union $\bigcup_a$ is over a collection of distinct values a.
- If $\mathcal{T}$ is a single tree with a root labelled by $\{A_1, \dots, A_k\}$ and a non-empty forest $\mathcal{U}$ of children, then

  $$E = \bigcup_a \langle A_1 : a\rangle \times \cdots \times \langle A_k : a\rangle \times E_a$$

  where each E_a is an f-representation over $\mathcal{U}$ and the union $\bigcup_a$ is over a collection of distinct values a.
- If $\mathcal{T}$ is a forest of trees $\mathcal{T}_1, \dots, \mathcal{T}_k$, then

  $$E = E_1 \times \cdots \times E_k$$

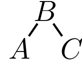
  where each E_i is an f-representation over $\mathcal{T}_i$.

□

The shape of the f-tree specifies a hierarchy of attributes by which we group the tuples of the represented relation in the f-representation. We group the tuples of the relation by the values of the attributes labelling the root, factor out the common value in each group, and then continue recursively on each group using the attributes lower in the f-tree.

Branching into several subtrees denotes (conditional) independence of attributes in the different subtrees; this leads to a product of f-representations over the individual subtrees. For each node, all attributes labelling the node have equal values in the represented relation.

Example 3.7. Consider a relation with schema $\{A, B, C\}$ and domain $\mathcal{D} = \{1, \ldots, 5\}$ that represents the inequalities $A < B < C$, as in Example 3.1. An f-representation of this relation over the following f-tree is given next.



$$
\begin{aligned}
&\langle B\!:\!2\rangle \times \langle A\!:\!1\rangle && \times (\langle C\!:\!3\rangle \cup \langle C\!:\!4\rangle \cup \langle C\!:\!5\rangle) \;\cup \\
&\langle B\!:\!3\rangle \times (\langle A\!:\!1\rangle \cup \langle A\!:\!2\rangle) && \times (\langle C\!:\!4\rangle \cup \langle C\!:\!5\rangle) \;\cup \\
&\langle B\!:\!4\rangle \times (\langle A\!:\!1\rangle \cup \langle A\!:\!2\rangle \cup \langle A\!:\!3\rangle) && \times \langle C\!:\!5\rangle.
\end{aligned}
$$

□

For a given f-tree $\mathcal{T}$ over a schema $\mathcal{S}$, not all relations over $\mathcal{S}$ have an f-representation over $\mathcal{T}$ since the subexpressions over subtrees that are siblings in $\mathcal{T}$ must appear in a product and this may not be possible for all relations. However, in case an f-representation over a given f-tree exists, it is unique up to the commutativity of product and union, and we characterise it precisely in the following section.

Example 3.8. The relation $\{\langle 1, 1, 1\rangle, \langle 2, 1, 2\rangle\}$ over schema $\{A, B, C\}$ does not admit an f-representation over the f-tree from Example 3.7, since any such f-representation must essentially be of the form $\langle B\!:\!1\rangle \times E_A \times E_C$, where E_A is a union of A-values and E_C is a union of C-values. □

Proposition 3.4. *Any f-representation over an f-tree is normal and deterministic.*

Proof. The normality condition is syntactic and easy to prove: from Definition 3.7 of an f-representation E over an f-tree $\mathcal{T}$, E contains no $\emptyset$ or $\langle\rangle$ unless it is $\emptyset$ or $\langle\rangle$ itself, there are no directly nested unions, and all expressions in Definition 3.7 can be parsed so that all products have at least two arguments.

Determinism can be proven by bottom-up induction over $\mathcal{T}$. For any node $\mathcal{A}$, any f-representation over $\mathcal{A}$ is deterministic as it is a union of $\langle\mathcal{A}{:}a\rangle$ for distinct values a. For a single tree $\mathcal{T}$ with root $\mathcal{A}$ and forest of children $\mathcal{U}$, any f-representation E_a over $\mathcal{U}$ is deterministic by the induction hypothesis, and hence the f-representations $\langle\mathcal{A}{:}a\rangle \times E_a$ are all deterministic and disjoint for different a. Hence any f-representation over $\mathcal{T}$ of the form $\bigcup_a \langle\mathcal{A}{:}a\rangle \times E_a$ is also deterministic. Finally, any f-representation over a forest $\mathcal{T}_1, \dots, \mathcal{T}_k$ is a product of f-representations over the $\mathcal{T}_i$. Each of these is deterministic by the induction hypothesis, and the product of deterministic f-representations over disjoint schemas is deterministic. □

We next introduce notation concerning f-trees. For any node $\mathcal{A}$ of an f-tree $\mathcal{T}$, $\mathcal{T}_\mathcal{A}$ denotes the subtree of $\mathcal{T}$ rooted at $\mathcal{A}$. By a *subtree* of $\mathcal{T}$ we mean a subtree of the form $\mathcal{T}_\mathcal{A}$ for some $\mathcal{A}$. By a *forest* of $\mathcal{T}$ we mean a set $\{\mathcal{T}_\mathcal{B}\}$ of all children $\mathcal{B}$ of some node in $\mathcal{T}$ or of all roots $\mathcal{B}$ of $\mathcal{T}$.We denote by $\text{anc}(\mathcal{A})$ the set of all attributes at nodes that are ancestors of $\mathcal{A}$ in $\mathcal{T}$, and by $\text{path}(\mathcal{A})$ the set of attributes at the ancestors of $\mathcal{A}$ and at $\mathcal{A}$. We overload the function anc to also retrieve the set of ancestor attributes of attributes, subtrees and forests. The node containing an attribute A or attributes $\{A_i\}$ is denoted by $\mathcal{A}$ and we speak interchangeably of an f-tree node and of its set of attributes. For any node $\mathcal{A} = \{A_1, \dots, A_k\}$, we use the shorthand $\langle\mathcal{A}{:}a\rangle$ for the product $\langle A_1{:}a\rangle \times \cdots \times \langle A_k{:}a\rangle$. Finally, we use shorthands such as $\pi_\mathcal{A}$, $\pi_{\mathcal{T}_\mathcal{A}}$ or $\pi_\mathcal{U}$ to mean the projection on the attributes of a node $\mathcal{A}$, a subtree $\mathcal{T}_\mathcal{A}$ or a forest $\mathcal{U}$, and $\text{anc}(\mathcal{A}) = t$ to mean the selection condition $\wedge_{B\in\text{anc}(\mathcal{A})} B = t(B)$, which enforces that tuples agree with t on the attributes $\text{anc}(\mathcal{A})$.

Example 3.9. In the left f-tree in Figure 3.1, $\text{path}(C)$ is the union of all attribute sets at nodes on the root-to-leaf path ending at C: $\text{path}(C) = \mathcal{A}\cup\mathcal{B}\cup\mathcal{C} = \{A_R, A_S, A_T, B_R, B_S, C\}$. The tree $\mathcal{T}_\mathcal{B}$ has root $\mathcal{B} = \{B_R, B_S\}$ and children $\mathcal{C} = \{C\}$ and $\mathcal{D} = \{D\}$. □

3.2.2 Characterisation of F-representations over F-trees

For any relation $\mathbf{R}$ and any f-tree $\mathcal{T}$ over the same schema, we now explicitly define an f-representation $\mathcal{T}(\mathbf{R})$. We prove that if $\mathbf{R}$ admits an f-representation over $\mathcal{T}$, it is unique and equal to $\mathcal{T}(\mathbf{R})$.

We first define f-representations $E(\mathbf{R}, \mathcal{X}, t)$ over subtrees or forests $\mathcal{X}$ of $\mathcal{T}$ and (context) tuples t over $\text{anc}(\mathcal{X})$ attributes. They are computed from relations $\pi_{\mathcal{X}}\sigma_{\text{anc}(\mathcal{X})=t}\mathbf{R}$, which are vertical-horizontal partitions of $\mathbf{R}$, but may not necessarily represent them exactly. We then specify conditions under which each $E(\mathbf{R}, \mathcal{X}, t)$ indeed represents the relation $\pi_{\mathcal{X}}\sigma_{\text{anc}(\mathcal{X})=t}\mathbf{R}$ and show how their composition into the f-representation $\mathcal{T}(\mathbf{R})$ represents the relation $\mathbf{R}$.

Definition 3.8. Let $\mathcal{T}$ be a f-tree and $\mathbf{R}$ a relation over the same schema. Let $\mathcal{T}(\mathbf{R})$ be the expression $E(\mathbf{R}, \mathcal{T}, \langle\rangle)$, where for any subtree or forest $\mathcal{X}$ in $\mathcal{T}$, and any tuple t over $\text{anc}(\mathcal{X})$, the expression $E(\mathbf{R}, \mathcal{X}, t)$ is defined recursively as follows.

- For any leaf $\mathcal{A}$,

$$E(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t) = \bigcup_{a \in A} \langle \mathcal{A}\!:\!a \rangle,$$

- and for any subtree $\mathcal{T}_{\mathcal{A}}$ with root $\mathcal{A}$ and children $\mathcal{T}_1, \ldots, \mathcal{T}_k$,

$$E(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t) = \bigcup_{a \in A} \langle \mathcal{A}\!:\!a \rangle \times E(\mathbf{R}, \{\mathcal{T}_1, \ldots, \mathcal{T}_k\}, t \times \langle \mathcal{A}\!:\!a \rangle),$$

  where $A = \pi_{A}\sigma_{\text{anc}(\mathcal{A})=t}\mathbf{R}$ and $A_1 \in \mathcal{A}$.

- For each non-empty forest $\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$,

$$E(\mathbf{R}, \{\mathcal{T}_1, \ldots, \mathcal{T}_k\}, t) = E(\mathbf{R}, \mathcal{T}_1, t) \times \cdots \times E(\mathbf{R}, \mathcal{T}_k, t).$$

- For $\mathcal{T}$ empty, $E(\mathbf{R}, \mathcal{T}, \langle\rangle)$ is $\emptyset$ if $\mathbf{R} = \emptyset$ and $\langle\rangle$ otherwise.

□

The f-representation $\mathcal{T}(\mathbf{R})$ represents the relation $\mathbf{R}$ if (i) the attributes at the same node of $\mathcal{T}$ always have equal values, and (ii) each time the recursive definition of $E(\mathbf{R}, \mathcal{X}, t)$ encounters branching in $\mathcal{T}$, the respective partition of the relation can be written as a product of relations represented by the individual branches. We next formalise this intuition.

Definition 3.9. An f-tree $\mathcal{T}$ is *valid* for a relation $\mathbf{R}$ over the same schema if

- for each node $\mathcal{A}$ of $\mathcal{T}$, the attributes of $\mathcal{A}$ have equal values in all tuples of $\mathbf{R}$, and
- for each forest $\mathcal{U} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$ of $\mathcal{T}$ and each tuple $t \in \pi_{\text{anc}(\mathcal{U})}(\mathbf{R})$, the relation $\pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t}(\mathbf{R}))$ is a product of projections $\pi_{\mathcal{T}_i}(\sigma_{\text{anc}(\mathcal{U})=t}(\mathbf{R}))$ to $\mathcal{T}_i$.

□

We can now characterise which f-trees can factorise a given relation.

Proposition 3.5. *A relation* $\mathbf{R}$ *has an f-representation over an f-tree* $\mathcal{T}$ *iff* $\mathcal{T}$ *is valid for* $\mathbf{R}$*. Any f-representation of* $\mathbf{R}$ *over* $\mathcal{T}$ *is equal to* $\mathcal{T}(\mathbf{R})$ *up to commutativity of product and union.*

Example 3.10. For the relation $\mathbf{R} = \{(a, b, c) : a < b < c\}$ over $\{1, \ldots, 5\}$, and the f-tree $\mathcal{T}$ with root $\mathcal{B}$ and children $\mathcal{A}$ and $\mathcal{C}$, as given in Example 3.7, we have

$$E(\mathbf{R}, \mathcal{A}, \langle B:2\rangle) = \langle A:1\rangle, \qquad E(\mathbf{R}, \mathcal{C}, \langle B:2\rangle) = \langle C:3\rangle \cup \langle C:4\rangle \cup \langle C:5\rangle, \qquad \text{and}$$
$$E(\mathbf{R}, \{\mathcal{A}, \mathcal{C}\}, \langle B:2\rangle) = \langle A:1\rangle \times (\langle C:3\rangle \cup \langle C:4\rangle \cup \langle C:5\rangle),$$

and similarly for $E(\mathbf{R}, \{\mathcal{A}, \mathcal{C}\}, \langle B:3\rangle)$ and $E(\mathbf{R}, \{\mathcal{A}, \mathcal{C}\}, \langle B:4\rangle)$. The expression $E(\mathbf{R}, \mathcal{T}, \langle\rangle)$ is the union $\bigcup_{b=2}^{4} E(\mathbf{R}, \{\mathcal{A}, \mathcal{C}\}, \langle B:b\rangle)$. Moreover, the relation $\pi_{\{\mathcal{A},\mathcal{C}\}}(\sigma_{\mathcal{B}=2}\mathbf{R})$ is a product of projections $\langle A:1\rangle$ and $(\langle C:3\rangle \cup \langle C:4\rangle \cup \langle C:5\rangle)$ to $\mathcal{A}$ and $\mathcal{C}$ respectively, and similarly for $\mathcal{B} = 3$ and $\mathcal{B} = 4$. Therefore, $\mathcal{T}$ is valid for $\mathbf{R}$, and $\mathcal{T}(\mathbf{R}) = E(\mathbf{R}, \mathcal{T}, \langle\rangle)$ is the unique f-representation of $\mathbf{R}$ over $\mathcal{T}$, fully shown in Example 3.7.

For $\mathbf{R}' = \{\langle 1, 1, 1\rangle, \langle 2, 1, 2\rangle\}$ from Example 3.8 and the same f-tree $\mathcal{T}$,

$$E(\mathbf{R}', \mathcal{A}, \langle B:1\rangle) = \langle A:1\rangle \cup \langle A:2\rangle, \qquad E(\mathbf{R}', \mathcal{C}, \langle B:2\rangle) = \langle C:1\rangle \cup \langle C:2\rangle, \qquad \text{and}$$
$$E(\mathbf{R}', \{\mathcal{A}, \mathcal{C}\}, \langle B:1\rangle) = (\langle A:1\rangle \cup \langle A:2\rangle) \times (\langle C:1\rangle \cup \langle C:2\rangle),$$

with $E'(\mathbf{R}, \mathcal{T}, \langle\rangle) = E'(\mathbf{R}, \{\mathcal{A}, \mathcal{C}\}, \langle B:1\rangle)$. However, the relation $\pi_{\{\mathcal{A},\mathcal{C}\}}(\sigma_{\mathcal{B}=1}\mathbf{R}') = \langle A:1\rangle \times \langle C:1\rangle \cup \langle A:2\rangle \times \langle C:2\rangle$ is not the product of projections to A and C, so $\mathcal{T}$ is not valid for $\mathbf{R}'$ and $[\![\mathcal{T}(\mathbf{R}')]\!] \neq \mathbf{R}'$. □

3.2.3 D-trees for D-representations

D-trees are f-trees where we make explicit the dependencies of attributes. This dependency information can be effectively used to avoid repetitions of expressions in d-representations, hence increasing the succinctness of the representation.

Definition 3.10. A d-tree $\mathcal{T}^{\uparrow}$ is an f-tree $\mathcal{T}$ in which each node $\mathcal{A}$ is annotated by a set of attributes $\text{key}(\mathcal{A})$ such that

- $\text{key}(\mathcal{A}) \subseteq \text{anc}(\mathcal{A})$,

- $\text{key}(\mathcal{A})$ is a union of nodes, and
- for any child $\mathcal{B}$ of $\mathcal{A}$, $\text{key}(\mathcal{B}) \subseteq \text{key}(\mathcal{A}) \cup \mathcal{A}$.

□

The set $\text{key}(\mathcal{A})$ specifies the ancestor attributes of $\mathcal{A}$ on which the attributes in the subtree rooted at $\mathcal{A}$ may depend. Naturally, if $\mathcal{B}$ is a child of $\mathcal{A}$ and $\mathcal{T}_A$ may only depend on $\text{key}(\mathcal{A})$, then $\mathcal{T}_\mathcal{B}$ may only depend on $\text{key}(\mathcal{A}) \cup \mathcal{A}$.

In the sequel, we denote by $\mathcal{T}^\uparrow$ a d-tree and by $\mathcal{T}$ its underlying f-tree. All notation for f-trees carries over to d-trees. We also define $\text{key}(\mathcal{T}_\mathcal{A}) = \text{key}(\mathcal{A})$ for any subtree $\mathcal{T}_\mathcal{A}$, and $\text{key}(\mathcal{U}) = \bigcup_i \text{key}(\mathcal{T}_i)$ for any forest $\mathcal{U}$ of subtrees $\{\mathcal{T}_i\}$.

We define the d-representation $\mathcal{T}^\uparrow(\mathbf{R})$ for a d-tree $\mathcal{T}^\uparrow$ and relation $\mathbf{R}$ similar to the f-representation $\mathcal{T}(\mathbf{R})$, except now the expressions $D(\mathbf{R}, \mathcal{X}, t)$ only depend on $t \in \text{key}(\mathcal{X})$ instead of $t \in \text{anc}(\mathcal{X})$.

Definition 3.11. Let $\mathcal{T}^\uparrow$ be a d-tree and $\mathbf{R}$ a relation over the same schema. We define $\mathcal{T}^\uparrow(\mathbf{R})$ to be the set of expressions $D(\mathbf{R}, \mathcal{X}, t)$ for all subtrees or forests $\mathcal{X}$ in $\mathcal{T}$ and all $t \in \pi_{\text{key}(\mathcal{X})}(\mathbf{R})$, where the expressions $D(\mathbf{R}, \mathcal{X}, t)$ are defined as follows:

- For any leaf $\mathcal{A}$,

$$D(\mathbf{R}, \mathcal{A}, t) = \textstyle\bigcup_{a \in A} \langle \mathcal{A} {:} a \rangle,$$

- and for any subtree $\mathcal{T}_\mathcal{A}$ with root $\mathcal{A}$ and children $\mathcal{T}_1, \ldots, \mathcal{T}_k$,

$$D(\mathbf{R}, \mathcal{T}_\mathcal{A}, t) = \textstyle\bigcup_{a \in A} \langle \mathcal{A} {:} a \rangle \times {}^\uparrow D(\mathbf{R}, \mathcal{U}, \pi_{\text{key}(\{\mathcal{T}_1, \ldots, \mathcal{T}_k\})}(t \times \langle \mathcal{A} {:} a \rangle)),$$

  where $A = \pi_{A_1} \sigma_{\text{key}(\mathcal{A}) = t} \mathbf{R}$ and $A_1 \in \mathcal{A}$.

- For any non-empty forest $\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$,

$$D(\mathbf{R}, \{\mathcal{T}_1, \ldots, \mathcal{T}_k\}, t) = {}^\uparrow D(\mathbf{R}, \mathcal{T}_1, \pi_{\text{key}(\mathcal{T}_1)} t) \times \cdots \times {}^\uparrow D(\mathbf{R}, \mathcal{T}_k, \pi_{\text{key}(\mathcal{T}_k)} t).$$

If $\mathcal{T}$ is empty, then $\mathcal{T}^\uparrow(\mathbf{R})$ is the set consisting of the expression $D(\mathbf{R}, \mathcal{T}, \langle\rangle)$ that is $\emptyset$ if $\mathbf{R} = \emptyset$ and $\langle\rangle$ otherwise. □

The definition of $\mathcal{T}^\uparrow(\mathbf{R})$ is the same as $\mathcal{T}(\mathbf{R})$ except that its expressions are not inlined recursively but only referenced in other expressions, and all expressions $E(\mathbf{R}, \mathcal{X}, t)$

whose context t agree on the values of $\text{key}(\mathcal{X})$ have been replaced by a single expression $D(\mathbf{R}, \mathcal{X}, \pi_{\text{key}(\mathcal{X})}t)$. If all replaced expressions $E(\mathbf{R}, \mathcal{X}, t)$ were indeed equal to $D(\mathbf{R}, \mathcal{X}, \pi_{\text{key}(\mathcal{X})}t)$, then the traversal of $\mathcal{T}^{\uparrow}(\mathbf{R})$ is $\mathcal{T}(\mathbf{R})$ and they represent the same relation. We next specify a precise condition on $\mathbf{R}$ and $\mathcal{T}^{\uparrow}$ when this happens.

Definition 3.12. A d-tree $\mathcal{T}^{\uparrow}$ is *valid* for a relation $\mathbf{R}$ if

- $\mathcal{T}$ is valid for $\mathbf{R}$ and
- for any node $\mathcal{A}$ and any tuples t_1, t_2 over $\text{anc}(\mathcal{A})$ such that $\pi_{\text{key}(\mathcal{A})}(t_1) = \pi_{\text{key}(\mathcal{A})}(t_2)$, it holds that $\pi_{\mathcal{T}_{\mathcal{A}}}(\sigma_{\text{anc}(\mathcal{A})=t_1}(\mathbf{R})) = \pi_{\mathcal{T}_{\mathcal{A}}}(\sigma_{\text{anc}(\mathcal{A})=t_2}(\mathbf{R}))$.

The first condition ensures that $\mathcal{T}(\mathbf{R})$ represents $\mathbf{R}$ and the second condition ensures that the traversal of $\mathcal{T}^{\uparrow}(\mathbf{R})$ is $\mathcal{T}(\mathbf{R})$. This is formalised in the following proposition.

Proposition 3.6. *If $\mathcal{T}^{\uparrow}$ is valid for $\mathbf{R}$, then $\mathcal{T}^{\uparrow}(\mathbf{R})$ is a d-representation of $\mathbf{R}$ and its traversal is $\mathcal{T}(\mathbf{R})$.*

A d-representation is normal (deterministic) if its traversal is normal (respectively deterministic). Since any f-representation $\mathcal{T}(\mathbf{R})$ is normal and deterministic, by Proposition 3.6 it follows that any d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ is normal and deterministic.

If $\mathcal{T}$ is a valid f-tree for a relation $\mathbf{R}$, the annotation $\text{key}(\mathcal{A}) = \text{anc}(\mathcal{A})$ for all nodes $\mathcal{A}$ yields a valid d-tree $\mathcal{T}^{\uparrow}$, the subexpressions $D(\mathbf{R}, \mathcal{X}, t)$ of $\mathcal{T}^{\uparrow}(\mathbf{R})$ correspond one-to-one to the subexpressions $E(\mathbf{R}, \mathcal{X}, t)$ of $\mathcal{T}(\mathbf{R})$, and each $D(\mathbf{R}, \mathcal{X}, t)$ is used at most once, so there is no sharing of subexpressions. F-representation over f-trees are thus a special case of d-representations over d-trees.

Example 3.11. Consider a relation $\mathbf{R} = \{(a, b, c) : 1 \leq a \neq b \neq c \leq 4\}$ and the f-tree $\mathcal{T}_2$ with root $\mathcal{A}$, child $\mathcal{B}$, and its child $\mathcal{C}$. The corresponding f-representation is (singleton types omitted)

$$\begin{aligned}
\mathcal{T}_2(\mathbf{R}) = \langle 1\rangle \times \big(&\langle 2\rangle \times (\langle 1\rangle \cup \langle 3\rangle \cup \langle 4\rangle) \,\cup\, \langle 3\rangle \times (\langle 1\rangle \cup \langle 2\rangle \cup \langle 4\rangle) \,\cup\, \langle 4\rangle \times (\langle 1\rangle \cup \langle 2\rangle \cup \langle 3\rangle)\big) \cup \\
&\langle 2\rangle \times \big(\langle 1\rangle \times (\langle 2\rangle \cup \langle 3\rangle \cup \langle 4\rangle) \,\cup\, \langle 3\rangle \times (\langle 1\rangle \cup \langle 2\rangle \cup \langle 4\rangle) \,\cup\, \langle 4\rangle \times (\langle 1\rangle \cup \langle 2\rangle \cup \langle 3\rangle)\big) \cup \\
&\langle 3\rangle \times \big(\langle 1\rangle \times (\langle 2\rangle \cup \langle 3\rangle \cup \langle 4\rangle) \,\cup\, \langle 2\rangle \times (\langle 1\rangle \cup \langle 3\rangle \cup \langle 4\rangle) \,\cup\, \langle 4\rangle \times (\langle 1\rangle \cup \langle 2\rangle \cup \langle 3\rangle)\big) \cup \\
&\langle 4\rangle \times \big(\langle 1\rangle \times (\langle 2\rangle \cup \langle 3\rangle \cup \langle 4\rangle) \,\cup\, \langle 2\rangle \times (\langle 1\rangle \cup \langle 3\rangle \cup \langle 4\rangle) \,\cup\, \langle 3\rangle \times (\langle 1\rangle \cup \langle 2\rangle \cup \langle 4\rangle)\big),
\end{aligned}$$

where e.g. the expressions $E(\mathbf{R}, \mathcal{C}, \langle 1, 2\rangle) = E(\mathbf{R}, \mathcal{C}, \langle 3, 2\rangle) = E(\mathbf{R}, \mathcal{C}, \langle 3, 2\rangle) = (\langle 1\rangle \cup$

$\langle 3\rangle \cup \langle 4\rangle)$ are equal because the possible values of C are determined by the value of B, independent of A.

We can set $\text{key}(\mathcal{A}) = \emptyset$, $\text{key}(\mathcal{B}) = \mathcal{A}$ and $\text{key}(\mathcal{C}) = \mathcal{B}$ to obtain the d-tree $\mathcal{T}_2^{\uparrow}$, for which

$$
\begin{aligned}
D(\mathbf{R}, \mathcal{C}, \langle B\!:\!1\rangle) &:= \langle 2\rangle \cup \langle 3\rangle \cup \langle 4\rangle,\\
D(\mathbf{R}, \mathcal{C}, \langle B\!:\!2\rangle) &:= \langle 1\rangle \cup \langle 3\rangle \cup \langle 4\rangle,\\
D(\mathbf{R}, \mathcal{C}, \langle B\!:\!3\rangle) &:= \langle 1\rangle \cup \langle 2\rangle \cup \langle 4\rangle,\\
D(\mathbf{R}, \mathcal{C}, \langle B\!:\!4\rangle) &:= \langle 1\rangle \cup \langle 2\rangle \cup \langle 3\rangle,\\
D(\mathbf{R}, \mathcal{T}_{2\mathcal{B}}, \langle A\!:\!1\rangle) &:= \langle 2\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!2\rangle) \;\cup\; \langle 3\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!3\rangle) \;\cup\; \langle 4\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!4\rangle),\\
D(\mathbf{R}, \mathcal{T}_{2\mathcal{B}}, \langle A\!:\!2\rangle) &:= \langle 1\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!1\rangle) \;\cup\; \langle 3\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!3\rangle) \;\cup\; \langle 4\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!4\rangle),\\
D(\mathbf{R}, \mathcal{T}_{2\mathcal{B}}, \langle A\!:\!3\rangle) &:= \langle 1\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!1\rangle) \;\cup\; \langle 2\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!2\rangle) \;\cup\; \langle 4\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!4\rangle),\\
D(\mathbf{R}, \mathcal{T}_{2\mathcal{B}}, \langle A\!:\!4\rangle) &:= \langle 1\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!1\rangle) \;\cup\; \langle 2\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!2\rangle) \;\cup\; \langle 3\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{C}, \langle B\!:\!3\rangle) \qquad \text{and}\\
D(\mathbf{R}, \mathcal{T}_2, \langle\rangle) &:= \langle 1\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{T}_{2\mathcal{B}}, \langle A\!:\!1\rangle) \;\cup\; \langle 2\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{T}_{2\mathcal{B}}, \langle A\!:\!2\rangle)\\
&\quad \cup\; \langle 3\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{T}_{2\mathcal{B}}, \langle A\!:\!3\rangle) \;\cup\; \langle 4\rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{T}_{2\mathcal{B}}, \langle A\!:\!4\rangle).
\end{aligned}
$$

Then $\mathcal{T}_2^{\uparrow}(\mathbf{R})$ consists of the above expressions with root $D(\mathbf{R}, \mathcal{T}_2, \langle\rangle)$. By shortening the full-blown expression names $D(\mathbf{R}, \mathcal{X}, t)$, the d-representation can be written as

$$
\begin{aligned}
D_0 =& \langle 1\rangle \times \big(\langle 2\rangle \times {}^{\uparrow}D_2 \;\cup\; \langle 3\rangle \times {}^{\uparrow}D_3 \;\cup\; \langle 4\rangle \times {}^{\uparrow}D_4\big) \;\cup\; \langle 2\rangle \times \big(\langle 1\rangle \times {}^{\uparrow}D_1 \;\cup\; \langle 3\rangle \times {}^{\uparrow}D_3 \;\cup\; \langle 4\rangle \times {}^{\uparrow}D_4\big) \cup\\
&\langle 3\rangle \times \big(\langle 1\rangle \times {}^{\uparrow}D_1 \;\cup\; \langle 2\rangle \times {}^{\uparrow}D_2 \;\cup\; \langle 4\rangle \times {}^{\uparrow}D_4\big) \;\cup\; \langle 4\rangle \times \big(\langle 1\rangle \times {}^{\uparrow}D_1 \;\cup\; \langle 2\rangle \times {}^{\uparrow}D_2 \;\cup\; \langle 3\rangle \times {}^{\uparrow}D_3\big),\\
D_1 :=& \langle 2\rangle \cup \langle 3\rangle \cup \langle 4\rangle,\\
D_2 :=& \langle 1\rangle \cup \langle 3\rangle \cup \langle 4\rangle,\\
D_3 :=& \langle 1\rangle \cup \langle 2\rangle \cup \langle 4\rangle,\\
D_4 :=& \langle 1\rangle \cup \langle 2\rangle \cup \langle 3\rangle,
\end{aligned}
$$

which makes apparent the sharing of subexpressions D_1 to D_4 in contrast to the f-representation $\mathcal{T}_2(\mathbf{R})$. $\square$

3.3 Factorisability of Query Results

In this section we study the factorisability of results of conjunctive queries using f-representations and d-representations over f-trees and respectively d-trees. In particular, for any conjunctive query Q, we characterise the f-trees (d-trees) over which *all results* of Q admit an f-representation (and respectively a d-representation). We consider without loss of generality f-trees and d-trees whose nodes correspond bijectively to the equivalence classes of head attributes in the input query Q; a detailed justification is given in Remark 3.1.

3.3.1 F-trees for Queries

In Chapter 2 we defined the notion of Q-dependent attributes for a conjunctive query Q, and showed that Q-dependent attributes can be dependent in results of Q. We use this property to characterise the f-trees that factorise all results of Q.

Proposition 3.7. *Let Q be a conjunctive query and $\mathcal{T}$ be an f-tree whose nodes are equivalence classes of attributes of Q. Then, $Q(\mathbf{D})$ has an f-representation over $\mathcal{T}$ for any database $\mathbf{D}$ iff any two Q-dependent nodes lie along a root-to-leaf path in $\mathcal{T}$.*

Proof. We first show that the path condition is necessary. Let $\mathcal{A}$ and $\mathcal{B}$ be nodes that do not lie along a root-to-leaf path in $\mathcal{T}$; they lie in sibling subtrees $\mathcal{T}_a$ and $\mathcal{T}_b$ of some forest $\mathcal{U}$ of $\mathcal{T}$. If $Q(\mathbf{D})$ has an f-representation over $\mathcal{T}$, then in this f-representation, for any c over anc$(\mathcal{U})$ the fragment $\pi_{\mathcal{U}}\sigma_{\text{anc}(\mathcal{U})=c}Q(\mathbf{D})$ is represented by a single expression $E(\mathbf{R},\mathcal{U},c)$, and hence $\sigma_{\text{anc}(\mathcal{U})=c}Q(\mathbf{D})$ is a product of its projections $\pi_{\mathcal{U}}\sigma_{\text{anc}(\mathcal{U})=c}Q(\mathbf{D})$ and $\pi_{\mathcal{T}\setminus\mathcal{U}}\sigma_{\text{anc}(\mathcal{U})=c}Q(\mathbf{D})$. Moreover, for any c, the relation $\pi_{\mathcal{U}}\sigma_{\text{anc}(\mathcal{U})=c}Q(\mathbf{D})$ is a product of its projections to the individual subtrees, including $\pi_{\mathcal{T}_a}Q(\mathbf{D})$ and $\pi_{\mathcal{T}_b}Q(\mathbf{D})$. Therefore,

$$Q(\mathbf{D}) = \pi_{\mathcal{T}_a\cup\text{anc}(\mathcal{U})}Q(\mathbf{D}) \bowtie_{\text{anc}(\mathcal{U})} \pi_{\mathcal{T}_b\cup\text{anc}(\mathcal{U})}Q(\mathbf{D}) \bowtie_{\text{anc}(\mathcal{U})} \pi_{\mathcal{T}\setminus\mathcal{T}_a\setminus\mathcal{T}_b}Q(\mathbf{D}),$$

so $\mathcal{A}$ and $\mathcal{B}$ are independent conditioned on anc$(\mathcal{U})$ in $Q(\mathbf{D})$, and hence they are not Q-dependent. It follows that any two Q-dependent nodes do lie on a root-to-leaf path in $\mathcal{T}$.

Conversely, we prove that if any two dependent nodes lie along a root-to-leaf path,

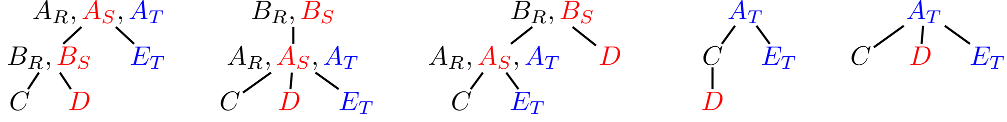


Figure 3.1: Left to right: two valid f-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ and one invalid f-tree for query Q_1 in Example 3.12. A valid and an invalid f-tree for the query $\pi_{A_T,C,D,E_T}Q_1$ in Example 3.12.

then for any forest $\mathcal{U}$ of subtrees $\mathcal{T}_j$ in $\mathcal{T}$, and any tuple $t \in \pi_{\text{anc}(\mathcal{U})}(Q(\mathbf{D}))$, the relation $\pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t}(Q(\mathbf{D})))$ is a product of its projections to the subtrees $\mathcal{T}_j$. Let T_j be the set of attributes in $\mathcal{T}_j$ together with all equivalent and dependent attributes. Then any relation with attributes in some T_j has all its attributes in T_j or equivalent to $\text{anc}(\mathcal{U})$, and hence $\sigma_{\text{anc}(\mathcal{U})=t}(\times_i \mathbf{R}_i)$ is a product of its projections $\pi_{\mathcal{T}_j}\sigma_{\text{anc}(\mathcal{U})=t}(\times_i \mathbf{R}_i)$, $\text{anc}(\mathcal{U})$, and the remaining attributes. Thus

$$\pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t}(Q(\mathbf{D}))) = \pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t}(\pi_{\mathcal{P}}(\sigma_{\psi}(\times_i \mathbf{R}_i)))) = \pi_{\mathcal{U}}(\sigma_{\psi}(\sigma_{\text{anc}(\mathcal{U})=t}(\times_i \mathbf{R}_i)))$$

is a product of its projections to $\mathcal{T}_j$. □

The intuition behind Proposition 3.7 is as follows. Setting aside the condition that attributes in a node have equal values, an f-tree $\mathcal{T}$ is valid for a relation $\mathbf{R}$ if any two sibling subtrees in $\mathcal{T}$ are independent conditioned on their common ancestors in $\mathcal{T}$. Since Q-dependent attributes are dependent in some results of Q, they cannot lie in sibling subtrees of $\mathcal{T}$, i.e., they must lie on a root-to-leaf path.

The condition in Proposition 3.7 is called the *path condition*. Any f-tree satisfying the path condition is *valid* for the query Q: we call it an f-tree of Q. Proposition 3.7 shows that an f-tree is valid for a query Q if and only if it is valid for all possible results of Q.

In the simpler case of an equi-join query Q (thus without projection), two nodes are Q-dependent whenever they contain attributes that belong to the same relation. The path condition for an equi-join query then states that the attributes of any relation must lie along a root-to-leaf path in $\mathcal{T}$.

Example 3.12. Consider the relations R, S and T over schemas $\{A_R, B_R, C\}$, $\{A_S, B_S, D\}$ and $\{A_T, E_T\}$ respectively and the equi-join query $Q_1 = \sigma_{\psi}(R \times S \times T)$ with $\psi = (A_R =$

$A_S = A_T, B_R = B_S$). The first and second f-trees in Figure 3.1 are valid for Q_1. The third f-tree is invalid since the attributes A_S and D are both from relation S and hence Q_1-dependent, but are not on a common root-to-leaf path.

Consider now the query $\pi_{A_T,C,D,E_T} Q_1$. The attribute class $\{B_R, B_S\}$ is entirely projected out, so the attributes of R and S are now Q_1-dependent, and hence the corresponding nodes $\{A_T\}, \{C\}, \{D\}$ are Q_1-dependent. The relation T induces the dependency of the nodes $\{A_T\}$ and $\{E_T\}$. The fourth f-tree in Figure 3.1 satisfies the path condition and hence is valid for our query, while the fifth f-tree, which is obtained by removing the attributes projected away from the first f-tree in Figure 3.1, is not valid. $\square$

Remark 3.1. *Proposition 3.7 only considers f-trees whose node labels coincide with the attribute classes of Q. For other f-trees of the same schema as Q, the characterisation can be extended as follows. If two attributes not equivalent in Q label the same node in $\mathcal{T}$, then $Q(\mathbf{D})$ does not always have an f-representation over $\mathcal{T}$. If two attributes equivalent in Q are in different nodes of $\mathcal{T}$ and they have no equivalent common ancestor, then $Q(\mathbf{D})$ also need not have an f-representation over $\mathcal{T}$. The f-trees left out are those where several nodes have attributes from the same class, and for each class, among the nodes containing attributes of that class, there is one which is an ancestor of the others. For any such f-tree, $Q(\mathbf{D})$ always has an f-representation over $\mathcal{T}$, but the f-tree constructed by pushing up all attributes of a class to the top-most node labelled by an attribute in that class defines f-representations with smaller or equal size. For the purpose of this work, Proposition 3.7 thus characterises all interesting f-trees.* $\square$

Remark 3.2. *In this section we establish the equivalence of the semantic notion of validity of f-trees (an f-tree $\mathcal{T}$ is valid for a query Q if $\mathcal{T}$ factorises all results of Q) and a syntactic notion of validity ($\mathcal{T}$ is valid for Q if it satisfies the path constraint). Therefore, whenever we speak about the f-trees of a query Q, it does not matter whether we consider Q as a query expression as defined in Chapter 2 (a syntactic notion) or up to query equivalence (a semantic notion). In particular we later define the parameters $s(Q)$ in terms of the f-trees of Q; the results in this section imply that $s(Q)$ is invariant under query equivalence under set semantics.*

3.3.2 D-Trees for Queries

We now extend our characterisation of f-trees, over which all results of a given query Q admit an f-representation, d-trees and d-representations: The underlying f-tree of the d-tree must be valid for Q, and for any node $\mathcal{B}$ of $\mathcal{T}^{\uparrow}$, the subtree $\mathcal{T}_{\mathcal{B}}$ may only be dependent on those ancestors of $\mathcal{B}$ that are in the set $\text{key}(\mathcal{B})$.

Definition 3.13. A d-tree $\mathcal{T}^{\uparrow}$ is *valid* for a query Q if the f-tree $\mathcal{T}$ is valid for Q and there is no node $\mathcal{B}$ with an ancestor $\mathcal{A} \not\subseteq \text{key}(\mathcal{B})$ and a descendant $\mathcal{C}$ such that $\mathcal{A}$ and $\mathcal{C}$ are Q-dependent. □

Proposition 3.8. *Let Q be a conjunctive query and let $\mathcal{T}^{\uparrow}$ be a d-tree whose nodes are equivalence classes of attributes of Q. Then $Q(\mathbf{D})$ has a d-representation over $\mathcal{T}^{\uparrow}$ for any database $\mathbf{D}$ iff $\mathcal{T}^{\uparrow}$ is valid for Q.*

Consider a query Q and a valid f-tree $\mathcal{T}$. We can obtain a valid d-tree by defining $\text{key}(\mathcal{B}) = \text{anc}(\mathcal{B})$ for all nodes $\mathcal{B}$, since then no node $\mathcal{B}$ has an ancestor $\mathcal{A} \not\subseteq \text{key}(\mathcal{B})$, and the validity condition holds vacuously. Furthermore, the nodes of $\text{anc}(\mathcal{B})$ can be divided into those that are Q-dependent on some node from $\mathcal{T}_{\mathcal{B}}$, and those that are not. Each node Q-dependent on a node in $\mathcal{T}_{\mathcal{B}}$ must be in $\text{key}(\mathcal{B})$ for the d-tree to be valid, yet the others need not be in $\text{key}(\mathcal{B})$. As we show in the following section, by shrinking the set of keys at a node, the resulting d-representation decreases in size. Therefore, we will often consider the minimal possible keys of an f-tree.

Definition 3.14. The *minimal d-tree* of an f-tree $\mathcal{T}$ for a query Q is the d-tree $\mathcal{T}^{\uparrow}$ where for each node $\mathcal{B}$, $\text{key}(\mathcal{B})$ is the set of all nodes of $\text{anc}(\mathcal{B})$ which are Q-dependent on some node from $\mathcal{T}_{\mathcal{B}}$. □

Example 3.13. Consider the query Q_2 from Example 3.15, whose hypergraph is depicted in Figure 3.2 left, and the d-tree $\mathcal{T}_4^{\uparrow}$ depicted in Figure 3.2 right. The f-tree $\mathcal{T}_4$ is a single root-to-leaf path, so it is valid for Q_2. The only node N in $\mathcal{T}_4^{\uparrow}$ for which $\text{key}(N) \neq \text{anc}(N)$ is the leaf $\mathcal{E}$; no attributes from $\mathcal{B}$ are in $\text{key}(\mathcal{E})$. However, no attributes from the subtree $\mathcal{T}_{\mathcal{E}} = \mathcal{E}$ are Q-dependent on $\mathcal{B}$. Therefore, $\mathcal{T}_4^{\uparrow}$ is also valid for Q_2. In fact, it is the minimal d-tree of its underlying f-tree $\mathcal{T}_4$.

3.3.3 Extensions of F-trees and D-trees

We present an alternative characterisation of the d-trees of a conjunctive query Q via d-trees of its equi-join $\hat{Q}$. Given an d-tree $\hat{\mathcal{T}}^{\uparrow}$ of $\hat{Q}$, a first approach to obtain a d-tree $\mathcal{T}^{\uparrow}$ of Q is to remove the attributes from $\hat{\mathcal{T}}^{\uparrow}$ that are projected away in Q. A problem arises when all attributes of a node are projected away: the node remains without attributes, the expressions of the corresponding union would not be labelled by distinct singletons, and the resulting d-representation may encode duplicate products of singletons and hence cease to be deterministic. Removing an empty node from the d-tree is also not always feasible as illustrated by Example 3.12: the attributes in its children subtrees may become dependent, which invalidates the d-tree. However, removing an empty leaf never invalidates the d-tree. This observation leads to an alternative characterisation of d-trees of arbitrary conjunctive queries.

Definition 3.15. An *extension* of a d-tree $\mathcal{T}^{\uparrow}$ (f-tree $\mathcal{T}$) of a conjunctive query Q is a d-tree $\hat{\mathcal{T}}^{\uparrow}$ (f-tree $\hat{\mathcal{T}}$) of the equi-join $\hat{Q}$ of Q such that $\mathcal{T}^{\uparrow}$ ($\mathcal{T}$) can be obtained from $\hat{\mathcal{T}}^{\uparrow}$ ($\hat{\mathcal{T}}$) by erasing the non-head attributes in Q and repeatedly removing leaf nodes. □

Proposition 3.9. *Let Q be a conjunctive query. A d-tree $\mathcal{T}^{\uparrow}$ and an f-tree $\mathcal{T}$ are valid for Q iff there exists an extension $\hat{\mathcal{T}}^{\uparrow}$ of $\mathcal{T}^{\uparrow}$, and respectively an extension $\hat{\mathcal{T}}$ of $\mathcal{T}$.*

Proof. First we prove the claim for f-trees. Let $\mathcal{T}$ be an f-tree valid for Q, so that any two Q-dependent nodes lie on a single root-to-leaf path. If we add equivalent non-head attributes to the existing nodes, their Q-dependence does not change. The relations of Q can be partitioned into equivalence classes of relations that are either joined by attributes that are all projected out, or connected by a chain of thus joined relations. For any such class $\mathcal{R}$ of relations, any two of their attributes are Q-dependent, and hence all of them lie on a single root-to-leaf path (no two of the attributes can lie in sibling subtrees). Let L be a lowest node on that path with an attribute from $\mathcal{R}$. Add a path of nodes under L, each node labelled by an equivalence class of non-head attributes from $\mathcal{R}$. Then for any relation $R \in \mathcal{R}$, the attributes of R lie on a single root-to-leaf path extending the path ending at L. If we do this for all such classes $\mathcal{R}$ of relations, the path constraint will be satisfied for all relations of Q. Moreover, the obtained tree $\hat{\mathcal{T}}$ will be an extension of $\mathcal{T}$.

Conversely suppose that there exists an f-tree $\hat{\mathcal{T}}$ valid for $\hat{Q}$ which is an extension of $\mathcal{T}$. We will show by contradiction that $\mathcal{T}$ is valid for Q: suppose that $\mathcal{T}$ is invalid, i.e., that two Q-dependent nodes $\mathcal{A}$ and $\mathcal{B}$ are in sibling subtrees $\mathcal{T}_A$ and $\mathcal{T}_B$. Since $\mathcal{A}$ and $\mathcal{B}$ are Q-dependent, their respective nodes in $\hat{\mathcal{T}}$ (call them $\mathcal{C}_0$ and $\mathcal{C}_k$) contain dependent attribtues. Therefore, there exist relations $R_1, \ldots, R_k$ such that R_i and R_{i+1} are joined on attributes from a node $\mathcal{C}_i$ of $\hat{\mathcal{T}}$ labelled only by non-head attributes of Q, and R_1 has an attribute in $\mathcal{C}_0$ and R_k an attribute in $\mathcal{C}_k$. Since $\hat{\mathcal{T}}$ is valid for $\hat{Q}$, the attributes of each R_i lie on a single root-to-leaf path. However, since $\mathcal{C}_0 \in \mathcal{T}_A$ and $\mathcal{C}_k \in \mathcal{T}_B$, there must exist a relation R_i for which $\mathcal{C}_i \in \mathcal{T}_A$ and $\mathcal{C}_{i+1} \in \mathcal{T}_B$. This is a contradiction to all attributes of R_i lying on a single root-to-leaf path in $\mathcal{T}$.

Finally we prove the claim for d-trees. If a d-tree $\mathcal{T}^{\uparrow}$ is valid, then its f-tree $\mathcal{T}$ is valid and it has an extension $\hat{\mathcal{T}}$ by the above. By defining key$(\mathcal{A})$ for each new node $\mathcal{A}$ to be the set of ancestors of $\mathcal{A}$ that are dependent on $\mathcal{A}$, we create a valid d-tree $\hat{\mathcal{T}}^{\uparrow}$. Conversely, if there exists an extension $\hat{\mathcal{T}}^{\uparrow}$ of $\mathcal{T}^{\uparrow}$ then $\hat{T}$ is an extension of $\mathcal{T}$ and hence $\mathcal{T}$ is valid for Q, and the keys key$(\mathcal{A})$ on $\mathcal{T}$ make it a valid d-tree for Q since the same keys key$(\mathcal{A})$ make the extension $\hat{\mathcal{T}}^{\uparrow}$ valid for $\hat{Q}$. □

Example 3.14. The fourth f-tree from the left in Figure 3.1 is valid for the query $\pi_{A_T,C,D,E_T} Q_1$ and can be extended to an f-tree $\hat{\mathcal{T}}$ for Q_1 by adding a leaf with attributes B_R, B_S under D, and adding the attributes A_R, A_S to the node labelled by A_T. The f-tree $\hat{\mathcal{T}}$ then satisfies the condition from Proposition 3.9.

The fifth f-tree in Figure 3.1 cannot be extended to an f-tree valid for Q_1, since the leaf B_R, B_S would have to be a descendant of C and also a descendant of D. □

3.4 Size Bounds

The main result of this section is the following characterisation of conjunctive queries based on the size of f-representations and d-representations of their results. This is a restatement of Theorems 3.3, 3.4, 3.5 and 3.7, proved later in this section.

Theorem 3.1. *For any non-Boolean query $Q = \pi_{\mathcal{P}} \sigma_{\psi}(R_1 \times \cdots \times R_n)$ there is a rational*

number $s(Q)$ *such that:*

- *For any database* $\mathbf{D}$*,* $Q(\mathbf{D})$ *admits an f-representation with size* $O(|\mathcal{P}| \cdot |\mathbf{D}|^{s(Q)})$*.*
- *For any f-tree* $\mathcal{T}$ *of* Q*, there exist arbitrarily large databases* $\mathbf{D}$ *for which the f-representation of* $Q(\mathbf{D})$ *over* $\mathcal{T}$ *has size* $\Omega((|\mathbf{D}|/|Q|)^{s(Q)})$*.*

There is also a rational number $s^{\uparrow}(Q)$ *such that:*

- *For any database* $\mathbf{D}$*,* $Q(\mathbf{D})$ *admits a d-representation with size* $O(|\mathcal{P}|^2 \cdot |\mathbf{D}|^{s^{\uparrow}(Q)})$*.*
- *For any d-tree* $\mathcal{T}^{\uparrow}$ *of* Q*, there exist arbitrarily large databases* $\mathbf{D}$ *for which the d-representation of* $Q(\mathbf{D})$ *over* $\mathcal{T}^{\uparrow}$ *has size* $\Omega((|\mathbf{D}|/|Q|)^{s^{\uparrow}(Q)})$*.*

In this section we only consider non-Boolean queries; results of Boolean queries can be represented by either the nullary tuple or the empty relation, both of size 1.

The corresponding upper and lower bounds from Theorem 3.1 meet with respect to data complexity. For a fixed query Q and any database $\mathbf{D}$, $Q(\mathbf{D})$ admits an f-representation with size $O(|\mathbf{D}|^{s(Q)})$ but any f-tree defines infinitely many f-representations of size $\Omega(|\mathbf{D}|^{s(Q)})$. Similarly, $Q(\mathbf{D})$ admits a d-representation of size $O(|\mathbf{D}|^{s^{\uparrow}(Q)})$ but any d-tree defines infinitely many d-representations of size $\Omega(|\mathbf{D}|^{s^{\uparrow}(Q)})$. We also lift the lower bounds to the language of all f-representations over f-trees and all d-representations over d-trees. The following is a restatement of Theorems 3.6 and 3.8.

Theorem 3.2. *For any fixed non-Boolean query* Q*, there exist arbitrarily large databases* $\mathbf{D}$ *for which any f-representation of the result* $Q(\mathbf{D})$ *over any f-tree has size* $\Omega(|\mathbf{D}|^{s(Q)})$*, and there exist arbitrarily large databases* $\mathbf{D}$ *for which any d-representation of the result* $Q(\mathbf{D})$ *over any d-tree has size* $\Omega(|\mathbf{D}|^{s^{\uparrow}(Q)})$*.*

A precise definition of the parameters $s(Q)$ and $s^{\uparrow}(Q)$ characterising the query Q is given later in this section, and their relationship to each other and to other known measures will be explored in the following section.

To prove Theorems 3.1 and 3.2, we first bound the size $|D|$ of a d-representation D in terms of the number of its singletons $\|D\|$. We then derive an expression for the exact number of singletons $\|\mathcal{T}^{\uparrow}(\mathbf{R})\|$ of a d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ as a function of the relation $\mathbf{R}$ and the d-tree $\mathcal{T}^{\uparrow}$. We then derive upper and lower bounds on this number in case $\mathbf{R}$ is a

query result $Q(\mathbf{D})$, as functions of the query Q and the size $|\mathbf{D}|$ of the input database. We adapt these results for the special case of f-representations over f-trees.

In the following, all formal statements referring to queries Q and d-trees $\mathcal{T}^{\uparrow}$ or f-trees $\mathcal{T}$ assume universal quantification over all conjunctive queries Q and all d-trees $\mathcal{T}^{\uparrow}$ of Q or f-trees $\mathcal{T}$ of Q, unless explicitly stated otherwise.

3.4.1 Size and Number of Singletons

The size $|D|$ of a d-representation is defined to be the number of its singletons $\|D\|$ plus the number of empty set symbols, unions, products and references to other expressions. For d-representations over d-trees, we show that the size is not significantly larger than the number of singletons.

Lemma 3.1. *For any d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ we have $|\mathcal{T}^{\uparrow}(\mathbf{R})| = O(\|\mathcal{T}^{\uparrow}(\mathbf{R})\| \cdot |\mathcal{T}|)$.*

Proof. By Definition 3.11, the d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ consists of factorised expressions of the form

$$D(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t) = \bigcup_{a \in A} \langle \mathcal{A} : a \rangle \times {}^{\uparrow}D(\mathbf{R}, \mathcal{U}, t') \qquad \text{for subtrees } \mathcal{T}_{\mathcal{A}}, \text{ and}$$
$$D(\mathbf{R}, \mathcal{U}, t) = {}^{\uparrow}D(\mathcal{T}_1, t'_1) \times \cdots \times {}^{\uparrow}D(\mathcal{T}_k, t'_k) \qquad \text{for forests } \mathcal{U},$$

where in the first expression the reference ${}^{\uparrow}D(\mathbf{R}, \mathcal{U}, t')$ is omitted if $\mathcal{U}$ is empty, $\langle \mathcal{A} : a \rangle$ is the product of singletons $\langle A_i : a \rangle$ for all $A_i \in \mathcal{A}$, and $\mathcal{T}_1, \ldots, \mathcal{T}_k$ are the trees comprising the forest $\mathcal{U}$. The exact definition of tuples t' and t'_i is not important. The size of each expression $\mathbf{D}(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t)$ is $O(|\mathcal{A}|)$ and the size of each expression $D(\mathbf{R}, \mathcal{U}, t)$ is $O(k) = O(|\mathcal{T}|)$. Moreover, each expression $D(\mathbf{R}, \mathcal{U}, t)$ where $\mathcal{U}$ is a forest is referenced in at least one $D(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t)$ where $\mathcal{T}_{\mathcal{A}}$ is a subtree, apart from a possible root expression $D(\mathbf{R}, \mathcal{T}, \langle\rangle) = {}^{\uparrow}D(\mathbf{R}, \mathcal{T}_1, \langle\rangle) \times \cdots \times {}^{\uparrow}D(\mathbf{R}, \mathcal{T}_k, \langle\rangle)$ if the d-tree is itself a forest. Therefore, to each product of singletons $\langle \mathcal{A} : a \rangle$ in $\mathcal{T}^{\uparrow}(\mathbf{R})$ we can associate its expression $D(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t)$ and the therein referenced expression $D(\mathbf{R}, \mathcal{U}, t)$ (if any) as shown above, of total size $O(|\mathcal{A}| + |\mathcal{T}|)$. Per each singleton this amounts to $O(|\mathcal{T}|)$. These expressions cover the entire d-representation up to the possible root fragment of size $O(|\mathcal{T}|)$. It follows that the size of $\mathcal{T}^{\uparrow}(\mathbf{R})$ is $O(\|\mathcal{T}^{\uparrow}(\mathbf{R})\| \cdot |\mathcal{T}|)$. □

For f-representations over f-trees we can tighten the bound as follows.

Lemma 3.2. *For any f-representation $\mathcal{T}(\mathbf{R})$ we have $|\mathcal{T}(\mathbf{R})| = O(\|\mathcal{T}(\mathbf{R})\|)$.*

Proof. In any f-representation $\mathcal{T}(\mathbf{R})$, each union symbol and each product symbol is followed by a singleton, so there are at most as many unions and products as singletons. Since there are no empty set symbols, the result follows. □

3.4.2 Counting Singletons in Representations

Consider first any f-representation $\mathcal{T}(\mathbf{R})$. For any attribute A in the root of $\mathcal{T}$, $\mathcal{T}(\mathbf{R})$ contains one occurrence of the singleton $\langle A\!:\!a\rangle$ for each A-value a in $\mathbf{R}$. For any attribute B in a child of the root, and for each A-value a, $\mathcal{T}(\mathbf{R})$ contains a subexpression over $\mathcal{T}_B$ which contains a singleton $\langle B\!:\!b\rangle$ for each B-value b in $\sigma_{A=a}\mathbf{R}$. Continuing top-down along $\mathcal{T}$, we deduce that for any attribute C, each singleton $\langle C\!:\!c\rangle$ appears once for each combination of values of the ancestor attributes of C, with which it contributes to some tuple of $\mathbf{R}$. Similarly, in any d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$, the singleton $\langle C\!:\!c\rangle$ appears once for each combination of values of $\text{key}(C)$ with which it contributes to some tuple of $\mathbf{R}$.

We next formalise the above observation and express the exact number of singletons in the d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ as a function of $\mathbf{R}$ and $\mathcal{T}^{\uparrow}$.

Lemma 3.3. *Let $\mathcal{T}^{\uparrow}(\mathbf{R})$ be the d-representation of a relation $\mathbf{R}$ over a non-empty d-tree $\mathcal{T}^{\uparrow}$, A be an attribute of $\mathbf{R}$, and x be a value.*

- *The number of occurrences of the singleton $\langle A\!:\!x\rangle$ in $\mathcal{T}^{\uparrow}(\mathbf{R})$ is $|\pi_{\text{key}(A)}\sigma_{A=x}\mathbf{R}|$.*
- *The number of occurrences of A-singletons in $\mathcal{T}^{\uparrow}(\mathbf{R})$ is $|\pi_{\text{key}(A)\cup\mathcal{A}}\mathbf{R}|$.*
- *The number of singletons in $\mathcal{T}^{\uparrow}(\mathbf{R})$ is $\|\mathcal{T}^{\uparrow}(\mathbf{R})\| = \sum_{A\in schema(\mathbf{R})} |\pi_{\text{key}(A)\cup\mathcal{A}}\mathbf{R}|$.*

Proof. The singleton $\langle A\!:\!x\rangle$ occurs in expressions of the form $D(\mathbf{R}, \mathcal{T}^{\uparrow}{}_{\mathcal{A}}, t)$ for the node $\mathcal{A}$ that contains A. In particular, by Definition 3.11 it occurs exactly once for each $t \in \pi_{\text{key}(\mathcal{A})}\mathbf{R}$ such that $x \in \sigma_{\text{key}(\mathcal{A})=t}\mathbf{R}$. These are exactly the $t \in \pi_{\text{key}(A)}\sigma_{A=x}\mathbf{R}$, so the number of occurrences of $\langle A\!:\!x\rangle$ is $|\pi_{\text{key}(A)}\sigma_{A=x}\mathbf{R}|$. The total number of occurrences of A-singletons is

thus

$$
\begin{aligned}
&\textstyle\sum_{x\in\pi_A(\mathbf{R})} |\pi_{\text{key}(A)}\sigma_{\mathcal{A}=x}(\mathbf{R})| \\
=&\textstyle\sum_{x\in\pi_A(\mathbf{R})} |\pi_{\text{key}(A)\cup\mathcal{A}}\sigma_{\mathcal{A}=x}(\mathbf{R})| \\
=&\textstyle\sum_{x\in\pi_A(\mathbf{R})} |\sigma_{\mathcal{A}=x}\pi_{\text{key}(A)\cup\mathcal{A}}(\mathbf{R})| \\
=&| \cup_{x\in\pi_A(\mathbf{R})} \sigma_{\mathcal{A}=x}\pi_{\text{key}(A)\cup\mathcal{A}}(\mathbf{R})| \\
=&|\pi_{\text{key}(A)\cup\mathcal{A}}(\mathbf{R})|.
\end{aligned}
$$

Finally, if $\mathcal{T}^\uparrow$ is non-empty then there are no nullary singletons in $\mathcal{T}^\uparrow(\mathbf{R})$, so $\|\mathcal{T}^\uparrow(\mathbf{R})\|$ is the number of typed singletons of all types, which is $\sum_{A\in\text{schema}(\mathbf{R})} |\pi_{\text{key}(A)\cup\mathcal{A}}\mathbf{R}|$. □

An analogous result for f-representations follows by noting that $\mathcal{T}(\mathbf{R}) = \mathcal{T}^\uparrow(\mathbf{R})$ when $\text{key}(\mathcal{A}) = \text{anc}(\mathcal{A})$ for all nodes $\mathcal{A}$ of the d-tree $\mathcal{T}^\uparrow$, and that $\text{anc}(A) \cup \mathcal{A} = \text{path}(A)$.

Corollary 3.2 (Lemma 3.3)**.** *Let $\mathcal{T}(\mathbf{R})$ be the f-representation of a relation $\mathbf{R}$ over a non-empty f-tree $\mathcal{T}$, A be an attribute of $\mathbf{R}$, and a be a value.*

- *The number of occurrences of the singleton $\langle A\!:\!a\rangle$ in $\mathcal{T}(\mathbf{R})$ is $|\pi_{\text{anc}(A)}\sigma_{A=a}\mathbf{R}|$.*
- *The number of occurrences of A-singletons in $\mathcal{T}(\mathbf{R})$ is $|\pi_{\text{path}(A)}\mathbf{R}|$.*
- *$\|\mathcal{T}(\mathbf{R})\| = \sum_{A\in schema(\mathbf{R})} |\pi_{\text{path}(A)}\mathbf{R}|$.*

3.4.3 Upper Bounds

Lemma 3.3 gives an exact expression for the number of singletons in a d-representation $\mathcal{T}^\uparrow(\mathbf{R})$ in terms of the relation $\mathbf{R}$ and the d-tree $\mathcal{T}^\uparrow$. In case $\mathbf{R}$ is a query result $Q(\mathbf{D})$, and $\mathcal{T}^\uparrow$ is valid for Q, we can quantify the number of singletons in the d-representation $\mathcal{T}^\uparrow(Q(\mathbf{D}))$ directly in terms of the database size $|\mathbf{D}|$.

Recall from Lemma 3.3 that for any attribute A in $\mathcal{T}^\uparrow$, the number of singletons of type A in $\mathcal{T}^\uparrow(Q(\mathbf{D}))$ is $|\pi_{\text{key}(A)\cup\mathcal{A}}Q(\mathbf{D})|$. By Lemma 2.1, we can bound it from above using the $(\text{key}(A) \cup \mathcal{A})$-restriction of Q and $\mathbf{D}$.

Corollary 3.3 (Lemmata 2.1 and 3.3)**.** *For any database $\mathbf{D}$, the number of occurrences of A-singletons in the d-representation $\mathcal{T}^\uparrow(Q(\mathbf{D}))$ is at most $|Q_{\text{key}(A)\cup\mathcal{A}}(\mathbf{D}_{\text{key}(A)\cup\mathcal{A}})|$.*

Proof. By Lemma 3.3, the number of occurrences of A-singletons in $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ is equal to $|\pi_{\text{key}(A)\cup\mathcal{A}}(Q(\mathbf{D}))|$, and by Lemma 2.1, this is at most $|Q_{\text{key}(A)\cup\mathcal{A}}(\mathbf{D}_{\text{key}(A)\cup\mathcal{A}})|$. □

We can further estimate the size of any $|Q_S(\mathbf{D}_S)|$ as a function of $|\mathbf{D}_S|$ and Q_S. Intuitively, if we can cover all attributes of the query Q_S by $k \leq |Q_S|$ of its relations, then $|Q_S(\mathbf{D}_S)|$ is at most the product of the sizes of these k relations, which is at most $|\mathbf{D}_S|^k$. The edge cover number $\rho(Q_S)$ is the minimal possible k such that Q_S can be covered by k relations; the best possible upper bound on $|Q_S(\mathbf{D}_S)|$ that we can obtain by this reasoning is thus $|\mathbf{D}_S|^{\rho(Q_S)}$. The following lemma further improves the upper bound by lifting the edge cover number $\rho(Q_S)$ the fractional edge cover number $\rho^*(Q_S)$. (See Chapter 2 for the definition of $\rho(Q)$ and $\rho^*(Q)$.)

Lemma 3.4 (Adapted from [AGM08])**.** *For any equi-join query Q and database $\mathbf{D}$, we have $|Q(\mathbf{D})| \leq M^{\rho^*(Q)} \leq |\mathbf{D}|^{\rho^*(Q)}$, where M is the size of the largest relation in $\mathbf{D}$.*

Proof. For any solution $\{x_{R_i}\}$ to the fractional edge cover linear program we have $|Q(\mathbf{D})| \leq \prod_i |\mathbf{R}_i|^{x_{R_i}}$ [AGM08]. By considering an optimal solution, it follows that

$$|Q(\mathbf{D})| \leq \textstyle\prod_i |\mathbf{R}_i|^{x_{R_i}} \leq \prod_i M^{x_{R_i}} = M^{\sum_i x_{R_i}} = M^{\rho^*(Q)} \leq |\mathbf{D}|^{\rho^*(Q)}.$$

□

Together with Corollary 3.3, this yields the following bound.

Lemma 3.5. *For any database $\mathbf{D}$, the number of occurrences of A-singletons in the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ is at most $|\mathbf{D}|^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})}$.*

Lemma 3.5 gives an upper bound on the number of occurrences of singletons of any attribute. We can obtain an upper bound on the total number of occurrences of singletons in the f-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ by summing these bounds over all attributes of Q. A simpler bound is obtained by estimating each of the summands by the largest one. A single bound for all possible d-trees of the query Q is obtained by considering the one with the smallest bound.

Definition 3.16. Let Q be a conjunctive query. For any d-tree $\mathcal{T}^{\uparrow}$ of Q, define

$$s^{\uparrow}(\mathcal{T}^{\uparrow}) = \max\{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}) \mid A \in \mathcal{P}\}$$

to be the maximum possible $\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})$ over all head attributes A of Q, and

$$s^{\uparrow}(Q) = \min\{s^{\uparrow}(\mathcal{T}^{\uparrow}) \mid \mathcal{T}^{\uparrow} \text{ is a d-tree of } Q\}$$

to be the minimum possible $s^{\uparrow}(\mathcal{T}^{\uparrow})$ over all d-trees $\mathcal{T}^{\uparrow}$ of Q. □

Corollary 3.4 (Lemma 3.5).

- *For any database* $\mathbf{D}$*, the number of singletons in* $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ *is at most* $|\mathcal{P}| \cdot |\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})}$.
- *For any database* $\mathbf{D}$*, there exists a d-representation of* $Q(\mathbf{D})$ *with at most* $|\mathcal{P}| \cdot |\mathbf{D}|^{s^{\uparrow}(Q)}$ *singletons.*

Using Lemma 3.1, we can turn bounds on the number of singletons into size bounds.

Theorem 3.3. *The size of* $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ *is* $O(|\mathcal{P}|^2 \cdot |\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})})$*, and for any database* $\mathbf{D}$ *there exists a d-representation of* $Q(\mathbf{D})$ *with size* $O(|\mathcal{P}|^2 \cdot |\mathbf{D}|^{s^{\uparrow}(Q)})$.

Analogous upper bounds can be shown for the sizes of f-representations over f-trees. The number of A-singletons in an f-representation $\mathcal{T}(\mathbf{R})$ is $|\pi_{\text{path}(\mathcal{A})}\mathbf{R}|$ by Corollary 3.2. This is at most $|Q_{\text{path}(\mathcal{A})}(\mathbf{D}_{\text{path}(\mathcal{A})})|$ by Lemma 2.1, which is at most $|\mathbf{D}|^{\rho^*(Q_{\text{path}(\mathcal{A})})}$ by Lemma 3.4. Similarly to d-trees, we maximise this value over all head attributes of $\mathcal{T}$ to obtain bounds for f-representations over $\mathcal{T}$, and then minimise over all f-trees $\mathcal{T}$ of Q to obtain bounds for f-representations of results of Q.

Definition 3.17. Let Q be a conjunctive query. For any f-tree $\mathcal{T}$ of Q, define

$$s(\mathcal{T}) = \max\{\rho^*(Q_{\text{path}(\mathcal{A})}) \mid A \in \mathcal{P}\}$$

to be the maximum possible $\rho^*(Q_{\text{path}(\mathcal{A})})$ over all head attributes A of Q, and

$$s(Q) = \min\{s(\mathcal{T}) \mid \mathcal{T} \text{ is an f-tree of } Q\}$$

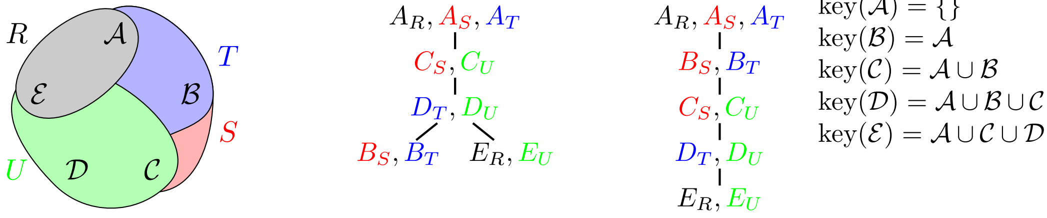


Figure 3.2: Left to right: Hypergraph of query Q_2 from Example 3.15 with nodes $\mathcal{A} = \{A_R, A_S, A_T\}$, $\mathcal{B} = \{B_S, B_T\}$, $\mathcal{C} = \{C_S, C_U\}$, $\mathcal{D} = \{D_T, D_U\}$, and $\mathcal{E} = \{E_R, E_U\}$; f-trees $\mathcal{T}_3$ and $\mathcal{T}_4$ of the query Q_2; and keys of nodes of the f-tree $\mathcal{T}_4$ turning it into a d-tree $\mathcal{T}_4^{\uparrow}$.

to be the minimum possible $s(\mathcal{T})$ over all f-trees $\mathcal{T}$ of Q. □

Corollary 3.5 (Theorem 3.3). *The number of singletons in $\mathcal{T}(Q(\mathbf{D}))$ is at most $|\mathcal{P}| \cdot |\mathbf{D}|^{s(\mathcal{T})}$, and for any database $\mathbf{D}$, there exists an f-representation of $Q(\mathbf{D})$ with at most $|\mathcal{P}| \cdot |\mathbf{D}|^{s(Q)}$ singletons.*

Using Lemma 3.2, we obtain the bounds on f-representation size. The difference of a factor of $|\mathcal{P}|$ compared to d-representations is due to the tighter bound on f-representation size expressed in the number of singletons.

Theorem 3.4. *The size of $\mathcal{T}(Q(\mathbf{D}))$ is $O(|\mathcal{P}| \cdot |\mathbf{D}|^{s(\mathcal{T})})$, and for any database $\mathbf{D}$ there exists an f-representation of $Q(\mathbf{D})$ with size $O(|\mathcal{P}| \cdot |\mathbf{D}|^{s(Q)})$.*

Example 3.15. Consider a database with relations R, S, T, and U with schemas $\{A_R, E_R\}$, $\{A_S, B_S, C_S\}$, $\{A_T, B_T, D_T\}$ and $\{C_U, D_U, E_U\}$ respectively, and the query $Q_2 = \sigma_\psi(R \times S \times T \times U)$, with $\psi = (A_R = A_S = A_T, B_S = B_T, C_S = C_U, D_T = D_U, E_R = E_U)$. The hypergraph of Q_2 is depicted in Figure 3.2 left. The attribute classes of Q_2 can be covered by the two relations S and U, so $\rho^*(Q_2^{\mathcal{E}}) \leq 2$. On the other hand, the attribute classes $\{B_S, B_T\}$ and $\{E_R, E_U\}$ have no relations in common, so their corresponding conditions $x_S + x_T \geq 1$ and $x_R + x_U \geq 1$ imply $\rho^*(Q_2^{\mathcal{E}}) \geq 2$. It follows that $\rho^*(Q_2) = 2$, so the result $Q_2(\mathbf{D})$ has size at most $|\mathbf{D}|^2$ for any database $\mathbf{D}$.

First consider the simpler case of f-representations over the f-trees in Figure 3.2. We compute $|\mathcal{T}_3(Q_2(\mathbf{D}))|$, where $\mathcal{T}_3$ is the left f-tree in Figure 3.2. The nodes with largest paths are $\mathcal{B} = \{B_S, B_T\}$ and $\mathcal{E} = \{E_R, E_U\}$. Consider the query restriction $Q_2^{\mathcal{B}}$. We need at least two relations to cover all attributes of $Q_2^{\mathcal{B}}$, so the edge cover number of $Q_2^{\mathcal{B}}$ is 2. However,

in the fractional edge cover linear program, we can assign $x_S = x_T = x_U = 1/2$ and $x_R = 0$. The covering conditions are satisfied, since each attribute class is covered by two of the relations S, T, U. The cost of this solution is $3/2$. It is in fact the optimal solution, so $\rho^*(Q_2^{\mathcal{B}}) = 3/2$. For $Q_2^{\mathcal{E}}$, the optimal solution is $x_U = 2/3$ and $x_R = x_S = x_T = 1/3$ with total cost $\rho^*(Q_2^{\mathcal{E}}) = 5/3$, and hence $s(\mathcal{T}_3) = 5/3$. It follows that the factorisation $\mathcal{T}_4(Q_2(\mathbf{D}))$ has at most $11 \cdot |\mathbf{D}|^{5/3}$ singletons, which is asymptotically smaller than the number of singletons $11 \cdot |\mathbf{D}|^2$ in the flat result.

The succinctness of representations over $\mathcal{T}_3$ is achieved by storing values of $\mathcal{B}$ and $\mathcal{E}$ independently for each combination of values of $\mathcal{A}, \mathcal{C}$ and $\mathcal{D}$, as represented by $\mathcal{B}$ and $\mathcal{E}$ lying in different branches of $\mathcal{T}_3$ under $\mathcal{A}, \mathcal{C}$ and $\mathcal{D}$. For comparison, in the right f-tree $\mathcal{T}_4$ in Figure 3.2, path($\mathcal{E}$) contains all attributes of Q_2. Hence $\rho^*(Q_2^{\mathcal{E}}) = \rho^*(Q_2) = 2$, so $s(\mathcal{T}_4) = 2$ and f-representations over the f-tree $\mathcal{T}_4$ present no asymptotic saving in space compared to flat representations.

Consider now the d-tree $\mathcal{T}_4^{\uparrow}$, whose underlying f-tree is $\mathcal{T}_4$ and the node keys are as defined in Figure 3.2 right. Now $\text{key}(\mathcal{E}) \cup \mathcal{E} = \mathcal{A} \cup \mathcal{C} \cup \mathcal{D} \cup \mathcal{E}$ is a strict subset of path($\mathcal{E}$), and $\rho^*(Q_{\text{key}(\mathcal{E}) \cup \mathcal{E}})$ equals $5/3$, strictly less than $\rho^*(Q_{\text{path}(\mathcal{E})}) = 2$. For all other nodes N the value $\rho^*(Q_{\text{key}(N) \cup N})$ is at most $5/3$, so d-representations over $\mathcal{T}_4^{\uparrow}$ have size at most $11 \cdot |\mathbf{D}|^{5/3}$. The succinctness of d-representations over $\mathcal{T}_4^{\uparrow}$ compared to f-representations over $\mathcal{T}_4$ is achieved by storing a union of $\mathcal{E}$-values only once for each combination of values from $\text{key}(\mathcal{E}) = \mathcal{A} \cup \mathcal{C} \cup \mathcal{D}$, and referencing this same expression for each different value of $\mathcal{B}$.

For the case of Q_2, it turns out that $\mathcal{T}_3$ is an optimal f-tree and $\mathcal{T}_4^{\uparrow}$ is an optimal d-tree, so $s(Q_2) = s^{\uparrow}(Q_2) = 5/3$. It is not necessarily true that $s(Q) = s^{\uparrow}(Q)$, in Section 3.5 we show examples of queries with $s^{\uparrow}(Q) \ll s(Q)$. □

3.4.4 Lower Bounds

We next show that the upper bound on the d-representation size is best possible in the following sense. For any non-Boolean query Q and any d-tree $\mathcal{T}^{\uparrow}$ of Q, there are arbitrarily large databases for which the size of the d-representation of the query result over $\mathcal{T}^{\uparrow}$ asymptotically meets the upper bound in terms of data complexity.

By Lemma 3.3, the number of singletons of type A in $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ is $|\pi_{\text{key}(A) \cup \mathcal{A}} Q(\mathbf{D})|$,

and Lemma 2.1 bounds any $|\pi_S(Q(\mathbf{D}))|$ from above by $|Q_S(\mathbf{D}_S)|$, where $(Q_S, \mathbf{D}_S)$ is the S-restriction of Q and $\mathbf{D}$. The following result provides a corresponding lower bound.

Lemma 3.6. *For any subset S of head attributes of a query Q and any database $\mathbf{D}_S$ over the schema of Q_S with largest relation of size M, there exists a database $\mathbf{D}$ such that $\mathbf{D}_S$ is the S-restriction of $\mathbf{D}$, with size $M \leq |\mathbf{D}| \leq |\mathbf{D}_S|$ and $|\pi_S(Q(\mathbf{D}))| \geq |Q_S(\mathbf{D}_S)|$.*

Proof. Each relation symbol R_i^S in Q_S is the relation symbol R_i restricted to the attributes of S^*. Construct a database $\mathbf{D}'$ by extending each relation in $\mathbf{D}_S$ with the removed attributes: for each attribute in the schema of Q but not Q_S, we allow a single value 1, and extend each tuple in each relation by this value for these new attributes. For relations in Q but with no attributes in Q_S, the relation instance in $\mathbf{D}_S$ is $\{\langle\rangle\}$, so $\mathbf{D}'$ will consist of a single tuple with value 1 in each attribute. There is a one-to-one correspondence between the tuples of $\mathbf{D}'$ and $\mathbf{D}_S$, so $|\mathbf{D}'| = |\mathbf{D}_S|$.

Finally we merge the relation instances in $\mathbf{D}'$ which should be equal due to self-joins. We construct the database $\mathbf{D}$ as follows. For any class $\{R_{i_1}, \ldots, R_{i_m}\}$ of relation symbols which refer to the same relation, replace the relation instances $\mathbf{R}_{i_1}, \ldots, \mathbf{R}_{i_m}$ in $\mathbf{D}'$ by a single relation instance $\mathbf{R} = \bigcup_j \mathbf{R}_{i_j}$ in $\mathbf{D}$, and interpret each of the relation symbols R_{i_j} by $\mathbf{R}$. By construction the largest relation in $\mathbf{D}$ is at least as large as the largest relation in $\mathbf{D}_S$, so $M \leq |\mathbf{D}|$. By the union bound we have $|\mathbf{D}| \leq |\mathbf{D}'| = |\mathbf{D}_S|$, and

$$
\begin{aligned}
|\pi_S(Q(\mathbf{D}))| &= |\pi_S(\pi_{\mathcal{P}}(\sigma_\psi(R_1 \times \cdots \times R_n)))(\mathbf{D})| \\
&\geq |\pi_S(\pi_{\mathcal{P}}(\sigma_\psi(R_1 \times \cdots \times R_n)))(\mathbf{D}')| && (2) \\
&= |\pi_S(\sigma_\psi(R_1 \times \cdots \times R_n))(\mathbf{D}')| \\
&= |\pi_{S^*}(\sigma_\psi(R_1 \times \cdots \times R_n))(\mathbf{D}')| && (4) \\
&= |\pi_{S^*}(\sigma_{\psi_S}(R_1 \times \cdots \times R_n))(\mathbf{D}')| && (5) \\
&= |\sigma_{\psi_S}(\pi_{S^*} R_1 \times \cdots \times \pi_{S^*} R_n)(\mathbf{D}')| \\
&= |Q_S(\mathbf{D}_S)|.
\end{aligned}
$$

Inequality (2) holds because each relation of $\mathbf{D}'$ is a subset of the corresponding relation of $\mathbf{D}$, equality (4) holds because each attribute in S^* is equivalent to some attribute in S, and

equality (5) holds because in $\mathbf{D}'$ the values in all attributes outside S^* are equal. $\square$

In a first attempt to make the lower bound $|Q_S(\mathbf{D}_S)|$ as large as possible while keeping $|\mathbf{D}_S|$ small, we pick k attribute classes of Q_S and let each of them attain N different values. If each relation has attributes from at most one of these classes and size at most N, then $\mathbf{D}_S$ has size $|Q_S| \cdot N$ but the result $Q_S(\mathbf{D}_S)$ has size N^k. The picked k attribute classes correspond to an independent set of k nodes in the hypergraph of Q_S.

Similar to the upper bound, we can strengthen the above lower bound by lifting independent sets to a weighted version. Since the linear programs for the (fractional) edge cover and the independent set problems are dual, this lower bound meets the upper bound from Section 3.4.3. The following result forms the basis of our argument.

Lemma 3.7 ([AGM08])**.** *For any equi-join query Q without self-joins, there exist arbitrarily large databases $\mathbf{D}$ such that $|Q(\mathbf{D})| \geq (|\mathbf{D}|/|Q|)^{\rho^*(Q)}$.*

We now use Lemmata 3.3, 3.6 and 3.7 to construct databases $\mathbf{D}$ with lower bounds on the number of A-singletons in the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$.

Lemma 3.8. *There exist arbitrarily large databases $\mathbf{D}$ such that the number of A-singletons in $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ is at least $(|\mathbf{D}|/|Q|)^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})}$.*

Proof. By Lemma 3.7 applied to $Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}$, there exist arbitrarily large databases $\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}$ such that $|Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}})| \geq (|\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}|/|Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}|)^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})}$. By Lemma 3.6, there exists a database $\mathbf{D}$ with $|\mathbf{D}| \leq |\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}|$ such that $|\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}(Q(\mathbf{D}))| \geq |Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}})|$. Moreover, $\mathbf{D}$ is at least as large as the largest relation of $\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}$, so $\mathbf{D}$ also gets arbitrarily large. By Lemma 3.3, the number of A-singletons in $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ is

$$\begin{aligned}|\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}(Q(\mathbf{D}))| &\geq |Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}})| \\ &\geq (|\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}|/|Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}|)^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})} \\ &\geq (|\mathbf{D}|/|Q|)^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})}.\end{aligned}$$

$\square$

We now lift Lemma 3.8 from A-singletons to all singletons in $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ by considering the attribute A for which the lower bound $(|\mathbf{D}|/|Q|)^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})}$ is the largest.

Corollary 3.6 (Lemma 3.8). *There exist arbitrarily large databases* $\mathbf{D}$ *for which* $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ *has at least* $(|\mathbf{D}|/|Q|)^{s^{\uparrow}(\mathcal{T}^{\uparrow})}$ *singletons.*

Since the size of a d-representation is at least the number of its singletons, we also have the following.

Theorem 3.5. *There exist arbitrarily large databases* $\mathbf{D}$ *for which* $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ *has size* $\Omega((|\mathbf{D}|/|Q|)^{s^{\uparrow}(\mathcal{T}^{\uparrow})}) = \Omega((|\mathbf{D}|/|Q|)^{s^{\uparrow}(Q)})$.

Theorem 3.5 can be generalised to a lower bound for the representation size of results of a given query in the language of d-representations over d-trees, regardless of the d-tree chosen.

Theorem 3.6. *For a fixed query* Q*, there exist arbitrarily large databases* $\mathbf{D}$ *for which any d-representation of the result* $Q(\mathbf{D})$ *over any d-tree has size* $\Omega(|\mathbf{D}|^{s^{\uparrow}(Q)})$.

Proof. To prove this theorem we need to strengthen the requirements on the sizes of database examples witnessing the lower bounds in Lemma 3.7 and Theorem 3.5. The changes are of a technical nature and the full proofs of the adapted versions are deferred to the electronic appendix.

Lemma 3.7, adapted. For any equi-join query Q without self-joins, there exist constants b_Q, c_Q such that for any sufficiently large N, there exists a database $\mathbf{D}$ of size $N \leq |\mathbf{D}| \leq b_Q \cdot N$ such that $|Q(\mathbf{D})| \geq c_Q \cdot |\mathbf{D}|^{\rho^*(Q)}$.

Theorem 3.5, adapted. For any query Q there exist constants b_Q, c_Q such that for any sufficiently large N and for any d-tree $\mathcal{T}^{\uparrow}$ of Q, there exists a database $\mathbf{D}_{\mathcal{T}^{\uparrow}}$ of size $N \leq |\mathbf{D}_{\mathcal{T}^{\uparrow}}| \leq b_Q \cdot N$ such that $|\mathcal{T}^{\uparrow}(Q(\mathbf{D}_{\mathcal{T}^{\uparrow}}))| \geq c_Q \cdot |\mathbf{D}_{\mathcal{T}^{\uparrow}}|^{s^{\uparrow}(Q)}$.

For any N sufficiently large let $\mathbf{D}_{\mathcal{T}^{\uparrow}}$ be as in the adapted version of Theorem 3.5. Construct the database $\mathbf{D}$ as a disjoint union of $\mathbf{D}_{\mathcal{T}^{\uparrow}}$ for all d-trees $\mathcal{T}^{\uparrow}$ of Q. (Label each data element in $\mathbf{D}_{\mathcal{T}^{\uparrow}}$ by $\mathcal{T}^{\uparrow}$, so that the corresponding relations of $\mathbf{D}_{\mathcal{T}^{\uparrow}}$ are disjoint, and for each relation symbol of Q construct a relation instance in $\mathbf{D}$ by taking a union of the corresponding relation instances in all $\mathbf{D}_{\mathcal{T}^{\uparrow}}$.) The result $Q(\mathbf{D})$ is a disjoint union of the results $Q(\mathbf{D}_{\mathcal{T}^{\uparrow}})$, and for any d-tree $\mathcal{T}^{\uparrow}$ the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ contains the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}_{\mathcal{T}^{\uparrow}}))$, so its size is at least $c_Q \cdot |\mathbf{D}_{\mathcal{T}^{\uparrow}}|^{s^{\uparrow}(Q)}$. The size of each $\mathbf{D}_{\mathcal{T}^{\uparrow}}$

is at most $b_Q \cdot N$, so the size of $\mathbf{D}$ is at most $d \cdot b_Q \cdot N$, where d is the number of d-trees of Q. Therefore, for any d-tree $\mathcal{T}^{\uparrow}$ the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ has size at least $b_Q \cdot (|\mathbf{D}|/(c \cdot d))^{s^{\uparrow}(Q)}$, which is $\Omega(|\mathbf{D}|^{s^{\uparrow}(Q)})$ for a fixed Q. □

For a fixed query, the upper and lower bounds on the size of d-representations of query results meet asymptotically. The fractional versions of the minimum edge cover number for the upper bounds and of the maximum independent set number for the lower bounds are essential for the tightness result, since their integer versions need not be equal. The parameter $s^{\uparrow}(Q)$ thus completely characterises queries by the representability of their results within the class of d-representations defined by d-trees.

Analogous lower bounds can be deduced for the special case of f-representations over f-trees.

Lemma 3.9. *For any f-tree $\mathcal{T}$ of Q there exist arbitrarily large databases $\mathbf{D}$ for which $\mathcal{T}(Q(\mathbf{D}))$ has at least $(|\mathbf{D}|/|Q|)^{s(\mathcal{T})}$ singletons.*

Theorem 3.7. *For any f-tree $\mathcal{T}$ of Q there exist arbitrarily large databases $\mathbf{D}$ for which $\mathcal{T}(Q(\mathbf{D}))$ has size $\Omega((|\mathbf{D}|/|Q|)^{s(\mathcal{T})}) = \Omega((|\mathbf{D}|/|Q|)^{s(Q)})$.*

Theorem 3.8. *For a fixed query Q, there exist arbitrarily large databases $\mathbf{D}$ for which the any f-representation of the result $Q(\mathbf{D})$ over any f-tree has size $\Omega(|\mathbf{D}|^{s(Q)})$.*

Example 3.16. Let us continue Example 3.15 and consider the query Q_2 and the left f-tree $\mathcal{T}_3$ from Figure 3.2. The hypergraph of $Q_2^{\mathcal{E}}$, has maximum independent set of size 1, since any two nodes share a common edge. We can trivially construct databases $\mathbf{D}$ for which the number of $\mathcal{E}$-singletons is linear in $\mathbf{D}$, yet this is much smaller than the lower bound given by Lemma 3.9. The fractional relaxation of the maximum independent set problem allows to increase the optimal cost to 5/3, thus meeting $\rho^*(Q_2^{\mathcal{E}})$ by duality of linear programming. In this relaxation we assign nonnegative values to the attribute classes, so that the sum of values in each relation is at most one. By assigning $y_{\mathcal{A}} = 2/3$ and $y_{\mathcal{C}} = y_{\mathcal{D}} = y_{\mathcal{E}} = 1/3$, the sum in each relation is exactly one, and the total cost is 5/3. This is used in the proofs of Lemmas 3.7 and 3.8 to construct arbitrarily large databases $\mathbf{D}$ for which the number of $\mathcal{E}$-singletons in $\mathcal{T}_3(Q_2(\mathbf{D}))$ is at least $(|\mathbf{D}|/|Q_2|)^{5/3} = (|\mathbf{D}|/4)^{5/3}$.

One such database $\mathbf{D}$ would contain the relations $\mathbf{R} = [4] \times [2]$, $\mathbf{S} = [4] \times [2] \times [1]$, $\mathbf{T} = [4] \times [2] \times [1]$ and $\mathbf{U} = [2] \times [2] \times [2]$. Here $[N]$ denotes $\{1, \ldots, N\}$ and the attributes of each relation are ordered alphabetically. Each relation has size 8 and the database $\mathbf{D}$ has size $32 = 8 \times |Q_2|$. The result $Q_2(\mathbf{D})$ corresponds to the relation where $A_R = A_S = A_T \in [4]$, $B_S = B_T = 1$, $C_S = C_U \in [2]$, $D_T = D_U \in [2]$ and $E_R = E_U \in [2]$, and any combination of these values is allowed. Its size is $|Q_2(\mathbf{D})| = 32 = (32/4)^{5/3} = (|\mathbf{D}|/|Q_2|)^{5/3}$. By replacing powers of 2 in this example by powers of larger integers, we can create arbitrarily large database examples with $|Q_2(\mathbf{D})| = (|\mathbf{D}|/|Q_2|)^{5/3}$.

Since all f-trees $\mathcal{T}$ for Q_2 have $s(\mathcal{T}) \geq s(Q_2) = 5/3$, the results in this subsection show that for any such f-tree $\mathcal{T}$ we can find databases $\mathbf{D}$ for which the size of $\mathcal{T}(Q_2(\mathbf{D}))$ is at least $(|\mathbf{D}|/|Q_2|)^{5/3} = (|\mathbf{D}|/4)^{5/3}$. □

3.5 Succinctness Gap and Tree Decompositions

In this section we compare and quantify the succinctness of flat relational representations, f-representations over f-trees and d-representations over d-trees in representing equi-join query results. The succinctness of these representation systems for query results is characterised by the parameters $\rho^*(Q)$, $s(Q)$ and $s^{\uparrow}(Q)$ of the asymptotic size bounds introduced in the previous section. Recall that for a given equi-join query Q, $\rho^*(Q)$, $s(Q)$ and $s^{\uparrow}(Q)$ are the smallest numbers such that for any database $\mathbf{D}$, the result $Q(\mathbf{D})$ has

- a flat representation of size $O(|\mathbf{D}|^{\rho^*(Q)})$,
- an f-representation over an f-tree with size $O(|\mathbf{D}|^{s(Q)})$,
- a d-representation over a d-tree with size $O(|\mathbf{D}|^{s^{\uparrow}(Q)})$.

In this section we study the relationships of the parameters $\rho^*(Q)$, $s(Q)$ and $s^{\uparrow}(Q)$ to each other and to known parameters of fractional hypertree width and fractional hyperpath width.

We first show that d-trees are closely related to tree decompositions and that the parameter $s^{\uparrow}(Q)$ equals the fractional hypertree width $\text{fhw}(Q)$ of Q (Corollary 3.7). Similarly but to a smaller extent, f-trees are related to path decompositions, and the parameter $s(Q)$ is greater or equal to the fractional hyperpath width $\text{fhpw}(Q)$ of Q (Corollary 3.8). Together

$$1 \overset{(1)}{\leq} \underbrace{s^{\uparrow}(Q) = \text{fhw}(Q) \overset{(2)}{\leq} \text{fhpw}(Q) \overset{(3)}{\leq} s(Q)}_{\text{factor } O(\log |\mathcal{S}|)} \overset{(4)}{\leq} \rho^*(Q) \overset{(5)}{\leq} |Q|$$

Figure 3.3: The hierarchy of parameters for non-empty equi-join queries Q. Each inequality may express a gap asymptotically as large as permitted by the remaining constraints. In particular, inequalities (2) and (3) may express a gap with a factor of $\Omega(\log |\mathcal{S}|)$, and inequalities (1), (4) and (5) a factor of $\Omega(|Q|)$.

with the trivial observation that $s(Q) = \min_{\text{f-tree } \mathcal{T} \text{ of } Q}(\max_{\mathcal{A} \in V(\mathcal{T})}(\rho^*(Q_{\text{path}(\mathcal{A})}))) \leq \rho^*(Q)$ we obtain the hierarchy of inequalities of parameters summarised in Figure 3.3.

We also quantify the gaps between these parameters. The parameter $s(Q)$ is bounded above by $O(\text{fhw}(Q) \cdot \log |\mathcal{S}|)$, where $\mathcal{S}$ is the schema of Q (Proposition 3.14), and this bound is tight: we exhibit a class of queries with $s(Q) = \Omega(\text{fhpw}(Q) \cdot \log |\mathcal{S}|)$ (Propositions 3.16 and 3.17), and from known results on pathwidth it is also easy to exhibit queries with $\text{fhpw}(Q) = \Omega(\text{fhw}(Q) \cdot \log |\mathcal{S}|)$ (Proposition 3.15). The gap between $s(Q)$ and $\rho^*(Q)$ can also be as large as the hierarchy allows; we construct classes of queries with $s(Q) = 1$ while $\rho^*(Q) = |Q|$ (Proposition 3.18). Finally, we note that there exist arbitrarily large queries for which all mentioned parameters are $O(1)$, and queries for which all parameters are $\Omega(|Q|)$ (Proposition 3.19). These results are also summarised in Figure 3.3.

In this section we restrict our attention to equi-join queries as the traditional domain of structural decomposition methods, in which the notions of fractional hypertree decompositions and fractional edge covers relate to size bounds and complexity of evaluation.

3.5.1 D-Trees and Tree Decompositions

We show a close connection between d-trees and fractional hypertree decompositions of the query hypergraph [GM06] for equi-join queries. Any d-tree $\mathcal{T}^{\uparrow}$ of an equi-join query Q can be translated into a fractional hypertree decomposition of Q with width $w = s^{\uparrow}(\mathcal{T}^{\uparrow})$, and any width-$w$ fractional hypertree decomposition of Q can be translated into a d-tree $\mathcal{T}^{\uparrow}$ with $s^{\uparrow}(\mathcal{T}^{\uparrow}) \leq w$. This implies that $s^{\uparrow}(Q)$ coincides with the fractional hypertree width of Q. Let us first recall the definition of a fractional hypertree decomposition of a hypergraph[2].

[2] In this section we speak of a query and its hypergraph interchangeably; by a fractional hypertree decomposition of a query we mean the fractional hypertree decomposition of its hypergraph.

Definition 3.18 ([GM06]). Let H be a hypergraph. A *tree decomposition* of H is a pair $(T, (B_t)_{t \in V(T)})$ where

- T is a tree, and
- $(B_t)_{t \in V(T)}$ is a family of sets of vertices of H, called *bags*, such that each edge of H is contained in some B_t, and for each vertex v of H the set $\{t : B_t \ni v\}$ is connected in T.

A *fractional hypertree decomposition* of H is a triple $(T, (B_t)_{t \in V(T)}, (\gamma_t)_{t \in V(T)})$, where $(T, (B_t))$ is a tree decomposition and

- $(\gamma_t)_{t \in V(T)}$ is a family of *weight functions* $E(H) \mapsto [0, \infty)$ such that for each $t \in V(T)$, γ_t covers all vertices of B_t, i.e. $\sum_{e \ni v} \gamma_t(e) \geq 1$ for all $v \in B_t$.

The weight of a weight function γ_t is $\text{weight}(\gamma_t) = \sum_{e \in E(H)} \gamma_t(e)$ and the width of the decomposition is $\max_{t \in V(T)} \text{weight}(\gamma_t)$. The *fractional hypertree width* of H, $\text{fhw}(H)$, is the minimum possible width of a fractional hypertree decomposition of H. □

In a fractional hypertree decomposition of a hypergraph H, each weight function γ_t must be a fractional edge cover of the hypergraph H restricted to the vertices of B_t. Since we are primarily interested in fractional hypertree decompositions of minimum possible width, for a given tree decomposition $(T, (B_t))$ we often consider each γ_t to be an optimal fractional edge cover of B_t, and hence obtain a minimum-width extension of $(T, (B_t))$ into a fractional hypertree decomposition $(T, (B_t), (\gamma_t))$. By the *fractional width* of a tree decomposition we mean the width of its minimal fractional extension; note that $\text{fhw}(H)$ is the minimal possible fractional width of a tree decomposition of H.

Next we show how any d-tree of an equi-join query Q corresponds to a tree decomposition of Q and vice versa. Intuitively, the vertices of the d-tree correspond to the vertices of the tree decomposition, and the sets $\text{key}(\mathcal{A}) \cup \mathcal{A}$ correspond to the bags $B_{\mathcal{A}}$. Our construction ensures that the fractional width of the corresponding tree decomposition is at most the cost $s^{\uparrow}(\mathcal{T}^{\uparrow})$, of the original d-tree $\mathcal{T}^{\uparrow}$, and vice versa.

Proposition 3.10. *Let $\mathcal{T}^{\uparrow}$ be a d-tree of an equi-join query Q. There exists a fractional hypertree decomposition of Q with width $w = s^{\uparrow}(\mathcal{T}^{\uparrow})$.*

Proof. Let Q be an equi-join query and let $\mathcal{T}^{\uparrow}$ be a d-tree of Q. Consider the pair

$(\mathcal{T}, (B_\mathcal{A})_{\mathcal{A}\in V(\mathcal{T})})$, where $\mathcal{T}$ is the underlying f-tree of $\mathcal{T}^\uparrow$ and the bag $B_\mathcal{A}$ contains the nodes of $\text{key}(\mathcal{A}) \cup \mathcal{A}$ for each node $\mathcal{A}$ of $\mathcal{T}$. We show that it is a tree decomposition of the query Q, with fractional width $s^\uparrow(\mathcal{T}^\uparrow)$.

First we show that each hyperedge of the query Q is contained in some bag $B_\mathcal{B}$. For any relation R of the query Q, the attributes of R lie on a root-to-leaf path in the f-tree $\mathcal{T}$ by the path condition of Proposition 3.7. For the lowest node $\mathcal{B}$ containing an attribute of R, all attributes of R are contained in $\text{path}(\mathcal{B})$. By Definition 3.13 characterising the d-trees of Q, all attributes of R must in fact lie in $\text{key}(\mathcal{B}) \cup \mathcal{B} \subseteq \bigcup B_\mathcal{B}$. Thus the hyperedge corresponding to the relation R is contained in the bag $B_\mathcal{B}$.

Next we show that for any node $\mathcal{B}$ of the query Q the set $\{\mathcal{A} : B_\mathcal{A} \ni \mathcal{B}\}$ is connected in $\mathcal{T}$. Since $\text{key}(\mathcal{A}) \cup \mathcal{A} \subseteq \text{anc}(\mathcal{A}) \cup \mathcal{A}$ for any $\mathcal{A}$, the node $\mathcal{B}$ may only be in $B_\mathcal{A}$ if $\mathcal{B}$ is an ancestor of $\mathcal{A}$ or equal to $\mathcal{A}$, or equivalently, only if $\mathcal{A} \in \mathcal{T}_\mathcal{B}$. Also, by Definition 3.10, $\text{key}(\mathcal{A}) \subseteq \text{key}(\text{parent}(\mathcal{A})) \cup \text{parent}(\mathcal{A})$, so $\text{key}(\mathcal{A}) \cup \mathcal{A} \subseteq \text{key}(\text{parent}(\mathcal{A})) \cup \text{parent}(\mathcal{A}) \cup \mathcal{A}$ and hence $B_\mathcal{A} \subseteq B_{\text{parent}(\mathcal{A})} \cup \{\mathcal{A}\}$, for any node $\mathcal{A}$. Thus if $B_\mathcal{C}$ does not contain $\mathcal{B}$ for some $\mathcal{C} \in \mathcal{T}_\mathcal{B}$, then $B_\mathcal{D}$ will not contain $\mathcal{C}$ for any $\mathcal{D}$ under $\mathcal{C}$. This shows that the set $\{\mathcal{A} : B_\mathcal{A} \ni \mathcal{B}\}$ is a connected subset of $\mathcal{T}$ (in fact, a connected subset of $\mathcal{T}_\mathcal{B}$), and concludes the proof that $(\mathcal{T}, (B_\mathcal{A})_{\mathcal{A}\in V(\mathcal{T})})$ is a tree decomposition of Q.

Finally, each bag $B_\mathcal{A}$ consists of the nodes of $\text{key}(\mathcal{A}) \cup \mathcal{A}$, so the cost of the optimal fractional edge cover $\gamma_\mathcal{A}$ of $B_\mathcal{A}$ is $\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})$, and the width of the corresponding fractional hypertree decomposition is exactly $\max_{\mathcal{A}\in V(Q)}(\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})) = s^\uparrow(\mathcal{T}^\uparrow)$. □

Proposition 3.11. *If there exists a fractional hypertree decomposition of an equi-join query Q with width w, then there exists a d-tree $\mathcal{T}^\uparrow$ of Q such that $s^\uparrow(\mathcal{T}^\uparrow) \leq w$.*

Proof. Let $(T, (B_t), (\gamma_t))$ be a fractional hypertree decomposition of an equi-join query Q. Each bag B_t is a set of vertices of (the hypergraph of) Q, that is, equivalence classes of attributes under the selection condition of Q. We construct the d-tree $\mathcal{T}^\uparrow$ whose nodes are the vertices of Q by mimicking the structure of T. While each node may occur in multiple bags of T; in $\mathcal{T}$ we include each node only once, at its topmost occurrence in T. The formal definition of $\mathcal{T}^\uparrow$ follows.

Construction. For each bag B_t, let $B'_t \subseteq B_t$ be the set of vertices which are not contained

in B_a for any ancestor a of t. Chain the vertices of B'_t into a path, and construct an f-tree $\mathcal{T}$ by replacing each t in T by the path B'_t (all in-edges to t now enter the first node of B'_t and all out-edges from t now exit the last node of B'_t). Each vertex $\mathcal{A}$ lies in a connected subset of bags B_t, so there is exactly one B'_t containing $\mathcal{A}$ and hence exactly one occurrence of $\mathcal{A}$ in $\mathcal{T}$. Finally, construct $\mathcal{T}^\uparrow$ by annotating each node $\mathcal{A}$ of $\mathcal{T}$ with $\text{key}(\mathcal{A}) = \text{anc}(\mathcal{A}) \cap \bigcup B_t$, where t is such that $\mathcal{A} \in B'_t$.

Correctness. First we prove that the f-tree $\mathcal{T}$ is an f-tree of Q. Let A and B be attributes of a relation R, let a and b be such that $\mathcal{A} \in B'_a$ and $\mathcal{B} \in B'_b$, and let t be such that the hyperedge corresponding to R is a subset of B_t, so that $\mathcal{A}, \mathcal{B} \in B_t$. Then both a and b are ancestors of t in T, and hence a and b lie on a root-to-leaf path in T. This implies that $\mathcal{A}$ and $\mathcal{B}$ lie on a root-to-leaf path in $\mathcal{T}$, and shows that the path condition is satisfied.

Next we prove that the d-tree $\mathcal{T}^\uparrow$ is a d-tree of Q, i.e. that for any $\mathcal{A}$, the nodes in the subtree $\mathcal{T}_\mathcal{A}$ can only depend on the vertices from $\text{key}(\mathcal{A})$. Suppose that some node $\mathcal{C}$ from $\mathcal{T}_\mathcal{A}$ depends on some $\mathcal{B}$ from $\text{anc}(\mathcal{A})$. Let a, b, c be such that $\mathcal{A} \in B'_a$, $\mathcal{B} \in B'_b$ and $\mathcal{C} \in B'_c$. Since $\mathcal{A}$ is an ancestor of $\mathcal{C}$ or $\mathcal{A} = \mathcal{C}$, a is an ancestor of c or $a = c$. In any case, if $\mathcal{C} \in B_t$ then t is a descendant of a or $t = a$. Since $\mathcal{B}$ and $\mathcal{C}$ are dependent, they share a hyperedge of Q, and hence there exists a r such that $\mathcal{B}, \mathcal{C} \in B_r$. By the above, r must be a descendant of a or $r = a$. Since $\mathcal{B} \in B'_b \subseteq B_b$ where b is an ancestor of a or equals a, and since the set $\{t : \mathcal{B} \in B_t\}$ is connected in T, we must also have $\mathcal{B} \in B_a$, i.e., $\mathcal{B} \subseteq \bigcup B_a$. Thus $\mathcal{B} \subseteq \text{anc}(\mathcal{A}) \cap \bigcup B_a = \text{key}(\mathcal{A})$, as required.

Finally, for each node $\mathcal{A}$, the set of attributes $\text{key}(\mathcal{A}) \cup \mathcal{A}$ is contained in some $\bigcup B_t$, so $\rho^*(Q_{\text{key}(\mathcal{A}) \cup \mathcal{A}}) \leq \rho^*(Q_{\bigcup B_t}) \leq w$ where w is the width of the original fractional hypertree decomposition. It follows that $s^\uparrow(\mathcal{T}^\uparrow) = \max_\mathcal{A} \rho^*(Q_{\text{key}(\mathcal{A}) \cup \mathcal{A}}) \leq w$. $\square$

The two-way correspondence yields the following equality.

Corollary 3.7 (Propositions 3.10 and 3.11)**.** *For any equi-join query Q, $s^\uparrow(Q) = \text{fhw}(Q)$.*

Proof. Let $\mathcal{T}^\uparrow$ be an optimal d-tree for the equi-join query Q. By Proposition 3.10, $\text{fhw}(Q) \leq s^\uparrow(\mathcal{T}^\uparrow) = s^\uparrow(Q)$. By Proposition 3.11, there exists a d-tree $\mathcal{T}^\uparrow$ such that $s^\uparrow(\mathcal{T}^\uparrow) = \text{fhw}(Q)$, so $s^\uparrow(Q) \leq \text{fhw}(Q)$. The result follows. $\square$

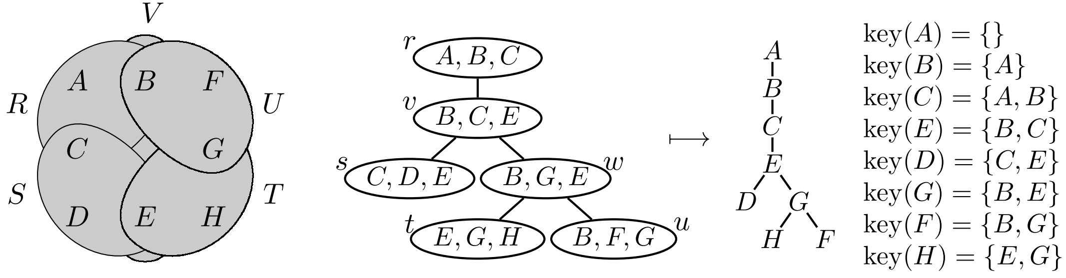


Figure 3.4: Left to right: the hypergraph of query Q_3 from Example 3.17, its tree decomposition of fractional width $\frac{3}{2}$, and a corresponding d-tree $\mathcal{T}^{\uparrow}{}_{Q_3}$ with $s^{\uparrow}(\mathcal{T}^{\uparrow}{}_{Q_3}) = \frac{3}{2}$ as constructed in Proposition 3.11.

Example 3.17. We illustrate the correspondence between tree decompositions and d-trees on the query

$$Q_3 = R(A, B, C) \bowtie S(C, D, E) \bowtie T(E, G, H) \bowtie U(B, F, G) \bowtie V(B, E),$$

for clarity written here as a natural join. The hypergraph of Q_3 has vertices $A, B, \ldots, H$ and edges R, S, T, U, V, as depicted in Figure 3.4 left. A tree decomposition T_{Q_3} of Q_3 with vertices r, s, t, u, v, w is depicted in Figure 3.4 middle, where bags are shown in place of the vertices. The bags B_r, B_s, B_t and B_u can each be covered by a single hyperedge R, S, T and U respectively. The bag $B_v = \{B, C, E\}$ can be covered by assigning weight $\frac{1}{2}$ to each of the hyperedges R, S and V, and $B_w = \{B, G, E\}$ by assigning weight $\frac{1}{2}$ to each of T, U, V. The fractional width of T_{Q_3} is thus $\frac{3}{2}$. The corresponding d-tree $\mathcal{T}^{\uparrow}{}_{Q_3}$ constructed by Proposition 3.11 is depicted in Figure 3.4 right: the nodes of Q are arranged into a tree by their topmost occurrence in the tree decomposition T_{Q_3}, and for each node N the set key(N) contains those ancestors of N which are in the same bag as the topmost occurrence of N.

By applying Proposition 3.10 to $\mathcal{T}^{\uparrow}{}_{Q_3}$, we obtain back a tree decomposition of Q_3 with fractional width $\frac{3}{2}$. It is not equal to the original decomposition T_{Q_3}; it contains an additional vertex A with $B_A = \{A\}$, its child B with $B_B = \{A, B\}$, and then a copy of T_{Q_3} as a subtree under B. (It is not depicted here.) □

3.5.2 F-Trees and Path Decompositions

We draw a connection between f-trees and fractional hyperpath decompositions of equi-join queries, although a looser one than between d-trees and fractional hypertree decompositions. Any f-tree $\mathcal{T}$ of an equi-join query Q can be translated into a fractional hyperpath decomposition of the hypergraph of Q with width $s(\mathcal{T})$, and any width-w fractional hypertree decomposition of Q can be translated into an f-tree $\mathcal{T}$ with $s(\mathcal{T}) \leq w \cdot \log |\mathcal{S}|$, where $\mathcal{S}$ is the schema of Q. It follows that $s(Q)$ is greater or equal to the fractional hyperpath width of Q, but can be greater by at most a factor logarithmic in $|\mathcal{S}|$. The following subsection shows that this logarithmic gap cannot be shrunk.

Definition 3.19. A *path decomposition* of a hypergraph H is a tree decomposition $(T, (B_t))$ of H for which the tree T is a path. A *fractional hyperpath decomposition* of H is a fractional hypertree decomposition $(T, (B_t), (\gamma_t))$ for which T is a path. The *fractional hyperpath width* of a hypergraph H, $\text{fhpw}(H)$, is the minimum possible width of a fractional hyperpath decomposition of H.

Since any fractional hyperpath decomposition is also a fractional hypertree decomposition, $\text{fhw}(H) \leq \text{fhpw}(H)$ for any hypergraph H.

Next we show how any f-tree $\mathcal{T}$ of an equi-join query can be translated into a path decomposition of fractional witdh $s(\mathcal{T})$. Intuitively, each root-to-leaf path in $\mathcal{T}$ corresponds to a bag of the decomposition, and these bags are arranged into a path using some ordering of the leaves of $\mathcal{T}$.

Proposition 3.12. *Let $\mathcal{T}$ be an f-tree of an equi-join query Q. There exists a path decomposition of Q with width $w = s(\mathcal{T})$.*

Proof. Construction. Consider a left-to-right order of the nodes in the f-tree $\mathcal{T}$ induced by any left-to-right order of the children under each node. Let $L_1, \ldots, L_k$ be the leaves of $\mathcal{T}$ in this order, let B_i be the set of nodes of $\mathcal{T}$ (vertices of the hypergraph of Q) on the path from L_i to the root of $\mathcal{T}$, and let P be the path $1 - 2 - \cdots - k$.

Correctness. We show that $(P, (B_i))$ is a path decomposition of H. For each node $\mathcal{A}$, $\mathcal{A} \in B_i$ iff L_i is in the subtree $\mathcal{T}_{\mathcal{A}}$, and hence set of indices i for which B_i contains $\mathcal{A}$ is

a contiguous range of integers, i.e., a connected subset of P. Moreover, for any relation R in Q, the attributes of R lie on a root-to-leaf path in $\mathcal{T}$, so the corresponding nodes are contained in B_i for some index i.

Finally, let γ_i be an optimal fractional edge cover of B_i. Then $(P, (B_i), (\gamma_i))$ is a fractional hyperpath decompostion of Q, and its width is $w = \max_i \rho^*(B_i) = \max_i \rho^*(Q_{\text{path}(L_i)})$. For any non-leaf vertex $\mathcal{A}$ of $\mathcal{T}$, there exists a leaf L_i under $\mathcal{A}$ and $\rho^*(Q_{\text{path}(\mathcal{A})}) \leq \rho^*(Q_{\text{path}(L_i)})$, so in fact $w = \max_{\mathcal{A}} \rho^*(Q_{\text{path}(\mathcal{A})}) = s(\mathcal{T})$. □

Corollary 3.8 (Proposition 3.12)**.** *For any equi-join query Q, $s(Q) \geq \text{fhpw}(Q)$.*

Example 3.18. Consider query Q_3 from Example 3.17 and its f-tree $\mathcal{T}_{Q_3}$ as depicted in Figure 3.4. Proposition 3.12 constructs a path decomposition of Q with bags $B_1 = \{A, B, C, E, D\}$, $B_2 = \{A, B, C, E, G, H\}$ and $B_3 = \{A, B, C, E, G, F\}$. □

The translation of Proposition 3.12 cannot always be reversed. The path decompositions produced by the translation have a special property that for any pair of nodes u and v, the sets $\{t : B_t \ni u\}$ and $\{t : B_t \ni v\}$ are either disjoint or one is contained in the other. A general path decomposition of Q does not have this property and cannot be translated back to an f-tree $\mathcal{T}$ of Q with $s(\mathcal{T})$ equal to the width of the decomposition. However, there exists a translation for which $s(\mathcal{T})$ is by at most a logarithmic factor larger than the width of the original path decomposition. Moreover, such a translation can also be defined for tree decompositions of Q.

We first prove an auxiliary lemma on balanced tree section and then the main result.

Lemma 3.10. *For any tree T there exists a vertex v such that all connected components of $T \setminus v$ have at most $|V(T)|/2$ vertices.*

Proof. Let v be a vertex of T for which the largest connected component of $T \setminus v$ has minimum possible number of vertices. For the sake of contradiction, suppose that $T \setminus v$ has a component C with more than $|V(T)|/2$ vertices, and let c be the vertex in C adjacent to c. Then the sets $C \setminus c$ and $T \setminus C$ are disconnected in $T \setminus c$, and have at most $|V(C)| - 1$ and $|V(T)|/2$ vertices respectively. Therefore, the largest connected component of $T \setminus c$ has less than $|V(C)|$ vertices, a contradiction. □

Proposition 3.13. *If there exists a fractional hypertree decomposition* $(T, (B_t), (\gamma_t))$ *of* Q *with width* w*, then there exists an f-tree* $\mathcal{T}$ *of* Q *such that* $s(\mathcal{T}) \leq w \cdot (\log_2 |V(T)| + 1)$.

Proof. Our construction is related to a known proof that any forest has logarithmic pathwidth [KS93]. For any tree decomposition $(T, (B_t))$ of Q, we rearrange the nodes of its underlying tree to attain height $\log |V(T)|$ (possibly losing the tree decomposition property), and then translate it into an f-tree $\mathcal{T}$ in which each root-to-leaf path will consist of at most $\log |V(T)|$ bags of the tree decomposition.

Construction. Let $(T, (B_t), (\gamma_t))$ be a fractional hypertree decomposition of Q. Construct a rooted tree balance(T) recursively as follows. Let v be a vertex in T such that all connected components of T have at most $|V(T)|/2$ vertices, and let $T_1, \ldots, T_k$ be the connected components of $T \setminus v$. Then balance(T) is a tree with root v and children subtrees balance(T_1), ..., balance(T_k).

Next we repeat on balance(T) the construction of an f-tree from a tree of bags, used in the proof of Proposition 3.11. For each bag B_t, let B'_t be the set of vertices in B_t but not in B_a for any ancestor a of t in balance(T). Chain the vertices of each B'_t into a path, and construct $\mathcal{T}$ by replacing each B_t in balance(T) by the path B'_t.

Correctness. In the tree balance(T) the bags containing a given vertex $\mathcal{A}$ possibly do not form a connected subtree, but the following argument by contradiction shows that there is still only one occurrence of $\mathcal{A}$ in $\mathcal{T}$. If $\mathcal{A} \in B'_x$ and $\mathcal{A} \in B'_y$, then neither of x and y is an ancestor of the other, so they have a least common ancestor p different from x and y, and $\mathcal{A} \notin B_p$. The subtree of balance(T) rooted at p was constructed as balance(T_p) for some connected subtree T_p of T. Since x and y lie in different children subtrees of p in balance(T_p), they are in different connected components of $T_p \setminus p$. Therefore, the set $\{t : B_t \ni \mathcal{A}\}$, containing x and y but not p, is disconnected in T_p and hence also in T. This contradicts $(T, (B_t))$ being a tree decomposition.

Next we show that $\mathcal{T}$ satisfies the path condition. Let A and B be attributes of a relation R, let a and b be such that $\mathcal{A} \in B'_a$ and $\mathcal{B} \in B'_b$. There exists a bag B_t containing all vertices of the hyperedge corresponding to R, in particular, $\mathcal{A}, \mathcal{B} \in B_t$. Then both a and b are ancestors of t in balance(T), and hence $\mathcal{A}$ and $\mathcal{B}$ lie on a root-to-leaf path in $\mathcal{T}$. This

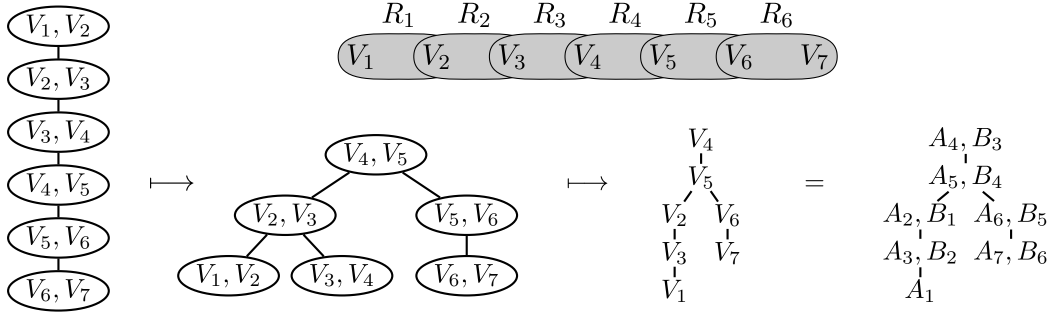


Figure 3.5: Top right: hypergraph of the query Q_6 from Example 3.17. Bottom left to right: a width-1 tree decomposition P (which is also a path decomposition) of the query Q_6, a rearrangement of its bags balance(P), and the resulting f-tree $\mathcal{T}$ as constructed in Proposition 3.13.

completes the proof that $\mathcal{T}$ is an f-tree of Q.

By induction we prove that $\text{depth}(\text{balance}(T)) \leq 1 + \log_2 |V(T)|$: if $|V(T)| = 1$ then $\text{depth}(\text{balance}(T)) = 1$ and if $|V(T)| > 1$ then $\text{depth}(\text{balance}(T)) = 1 + \max_k \text{depth}(\text{balance}(T_k)) \leq 1 + 1 + \log_2 \lfloor |V(T)|/2 \rfloor \leq 1 + \log_2 |V(T)|$.

Finally we prove the bound on $s(\mathcal{T})$. For any attribute A, if $\mathcal{A} \in B'_a$, then $\text{path}(\mathcal{A})$ in the tree $\mathcal{T}$ is contained in the labels of vertices of $\bigcup_{t \in \text{path}(a)} B_t$, where by $\text{path}(a)$ we mean the set containing a and the ancestors of a in balance(T). The weight function $\gamma_A = \sum_{t \in \text{path}(a)} \gamma_t$ covers all vertices in $\bigcup_{t \in \text{path}(a)} B_t$, the weight of each γ_t is at most w, and the size of $\text{path}(a)$ is at most $1 + \log_2 |V(T)|$, so $\rho^*(Q_{\text{path}(\mathcal{A})}) \leq w \cdot (1 + \log_2 |V(T)|)$. Since this holds for any attribute A of Q, we also have $s(\mathcal{T}) \leq w \cdot (1 + \log_2 |V(T)|)$. ☐

Proposition 3.14. *For equi-join queries Q, we have $s(Q) = O(\text{fhw}(Q) \cdot \log |\mathcal{S}|)$.*

Proof. Let $\mathcal{T}^\uparrow$ be an optimal d-tree of Q. The proof of Proposition 3.10 constructs a fractional hypertree decomposition of Q with width $s^\uparrow(\mathcal{T}^\uparrow) = s^\uparrow(Q) = \text{fhw}(Q)$ such that the underlying tree T has $|V(Q)| \leq |\mathcal{S}|$ vertices. The result follows by Proposition 3.13. ☐

Corollary 3.9 (Proposition 3.14)**.** *For equi-join queries Q, we have $s(Q) = O(s^\uparrow(Q) \cdot \log |\mathcal{S}|)$.*

Example 3.19. Consider the chain query of 6 relations

$$Q_6 = \sigma_{B_1=A_2 \wedge B_2=A_3 \wedge \cdots \wedge B_5=A_6}(R_1(A_1, B_1) \times R_2(A_2, B_2) \times \cdots \times R_6(A_6, B_6))$$

same as defined in Definition 3.20. Its hypergraph is a path of vertices $V_1 = A_1$, $V_2 = \{B_1, A_2\}$, ..., $V_6 = \{B_5, A_6\}$, $V_7 = \{B_6\}$ depicted in Figure 3.5.2 top, and it has a straightforward width-1 path decomposition P with bags $\{V_1, V_2\}$, $\{V_2, V_3\}$, ..., $\{V_6, V_7\}$, depicted in Figure 3.5.2 left. This path decomposition can never result as a translation of an f-tree using Proposition 3.12, since e.g. the sets of bags containing V_2 and V_3 are not disjoint, nor one contained in the other.

Proposition 3.13 translates any tree decomposition into an f-tree $\mathcal{T}$ with $s(\mathcal{T})$ only by a logarithmic factor larger than the width of the tree decomposition. For the tree decomposition P, the translation first constructs the tree balance(P) by repeatedly picking out the middle vertex as a root, and then the f-tree $\mathcal{T}$ by keeping the topmost occurrence of each node and removing others, as shown in Figure 3.5.2. The rearrangement $P \mapsto$ balance(P) ensures that each root-to-leaf path in the resulting f-tree $\mathcal{T}$ only contains vertices from a logarithmic number of bags from the original decomposition.

Note that the constructed f-tree $\mathcal{T}$ is not necessarily optimal, we have $s(\mathcal{T}) = 3$ but $s(Q_6) = 2$ as witnessed by the complete binary f-tree $V_4(V_2(V_1, V_3), V_6(V_5, V_7))$. □

3.5.3 Succinctness Gap for D-representations

We show that the logarithmic upper bound on the gap between $s^{\uparrow}(Q)$ and $s(Q)$ is tight by exhibiting a class of queries for which $s(Q) = \Omega(s^{\uparrow}(Q) \cdot \log |\mathcal{S}|)$. First we show that the logarithmic gap exists between fhw(Q) and fhpw(Q), which implies the gap between $s^{\uparrow}(Q)$ and $s(Q)$ since

$$s^{\uparrow}(Q) = \text{fhw}(Q) \leq \text{fhpw}(Q) \leq s(Q).$$

Then we also exhibit a class of queries with a logarithmic gap between fhpw(Q) and $s(Q)$, that is, for which $s(Q) = \Omega(\text{fhpw}(Q) \cdot \log |\mathcal{S}|)$.

The gap between fhw(Q) and fhpw(Q) follows easily from existing results on treewidth and pathwidth.

Proposition 3.15. *There exist arbitrarily large equi-join queries for which* $\text{fhpw}(Q) = \Omega(\text{fhw}(Q) \cdot \log |\mathcal{S}|)$.

Proof. The complete binary tree T_h of height h has $2^h - 1$ vertices, treewidth 1 and pathwidth

$\Omega(h)$ [CDF96]. Pathwidth $\Omega(h)$ implies that any path decomposition of T_h has a bag B with $\Omega(h)$ vertices. Since T_h is a graph and all edges of T_h contain two vertices, two times the weight $\sum_e \gamma(e)$ of any weight function γ on B equals the sum of weights of all vertices $\sum_v \sum_{e \ni v} \gamma(e)$, which is at least $|B|$ if γ covers B. It follows that the weight of γ is $\Omega(h)$ and hence the fractional hyperpath width of T_h is $\Omega(h)$. The fractional hypertree width of T_h is still 1. Therefore, the query Q_h^T whose hypergraph is the tree T_h has $\text{fhw}(Q) = 1$ but $\text{fhpw}(Q) = \Omega(h) = \Omega(\log |2^h - 1|) = \Omega(\log |\mathcal{S}|)$. □

In the remainder of this section we show the gap between $\text{fhpw}(Q)$ and $s(Q)$. A prototypical example for this gap are the chain queries.

Definition 3.20. Consider the relations R_i over schemas $\{A_i, B_i\}$ for $i \in \mathbb{N}$. For any natural number n we define the *chain query* Q_n to be the chain of $n-1$ joins

$$Q_n = \sigma_{B_1=A_2 \wedge B_2=A_3 \wedge \ldots \wedge B_{n-1}=A_n}(R_1 \times \cdots \times R_n).$$

□

The hypergraph of Q_n is a simple path of $n+1$ vertices denoted as $V_1 = \{A_1\}$, $V_2 = \{B_1, A_2\}, \ldots, V_n = \{B_{n-1}, A_n\}, V_{n+1} = \{B_n\}$ connected by the n edges $\{V_i, V_{i+1}\}$ corresponding to the relations R_i for $i = 1, \ldots, n$.

Proposition 3.16. *For any chain query* Q_n*,* $\text{fhpw}(Q_n) = \text{fhw}(Q_n) = s^{\uparrow}(Q_n) = 1$*.*

Proof. Let P be the path $1 - 2 - \cdots - n$ and for each $i = 1, \ldots, n$ define the bag $G_i = \{V_i, V_{i+1}\}$ and the weight function γ_i as $R_i \mapsto 1$ and $R_j \mapsto 0$ for $j \neq i$. Then each hyperedge R_i of Q_n is contained in the bag G_i, for each vertex V_i of Q_n the set of j such that $G_j \ni V_i$ is connected, and each γ_i covers its corresponding bag G_i. Therefore, $(P, (G_i)_{i=1}^n, (\gamma_i)_{i=1}^n)$ is a fractional path decomposition of Q_n. The weight of each γ_i is 1, so the weight of the decomposition is 1. Since Q_n is non-empty, any fractional hypertree decomposition has weight at least 1. It follows that $\text{fhpw}(Q_n) = \text{fhw}(Q_n) = 1$, and by Proposition 3.7, also $s^{\uparrow}(Q_n) = 1$. □

Proposition 3.13 bounds $s(Q_n)$ from above by $s(Q_n) \leq \log_2(n+1)+1$. This bound is also witnessed by the balanced f-tree $\mathcal{T}_n$ constructed by picking the node $V_{\lfloor n/2 \rfloor + 1}$ in the middle of

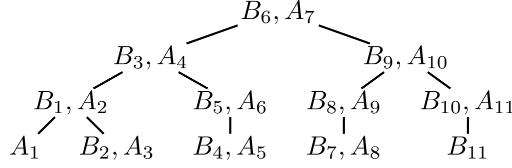


Figure 3.6: The f-tree $\mathcal{T}_{11}$ for Q_{11} with height$(\mathcal{T}_{11}) = 4$ and $s(\mathcal{T}_{11}) = 3$. (From Example 3.20).

the chain query as a root and constructing its two children subtrees by recursively using the two resulting halves of the query: we definitely have $s(\mathcal{T}_n) \leq \text{depth}(\mathcal{T}_n) = \lfloor \log_2(n+1) \rfloor + 1$.

Example 3.20. The f-tree $\mathcal{T}_{11}$ for the chain query Q_{11} (i.e., $n = 11$) is shown in Figure 3.6. Its depth is $\lfloor \log_2 12 \rfloor + 1 = 4$, so definitely $s(\mathcal{T}_{11}) \leq 4$. In fact $s(\mathcal{T}) = 3$, as $\rho^*(Q_{\text{path}(A_1)}) = 3$ and $\rho^*(Q_{\text{path}(A_i)}) \leq 3$ for all other A_i (where $Q_{\text{path}(A_i)}$ is the restriction of Q_{11} to path(A_i)).

The d-tree $\mathcal{T}^\uparrow$ that is a path of nodes $V_1 = \{A_1\}$, $V_2 = \{B_1, A_2\}$, ..., $V_{11} = \{B_{10}, A_{11}\}$ and $V_{12} = \{B_{11}\}$, rooted at V_1, with key$(V_{i+1}) = V_i$ for each i, is a valid d-tree of Q_{11}. Since each key$(V_{i+1}) \cup V_{i+1}$ is covered by the relation R_i, we have $s^\uparrow(\mathcal{T}^\uparrow) = 1$ and hence also $s^\uparrow(Q_{11}) = 1$. □

Next we prove that up to a constant factor, $\mathcal{T}_n$ is optimal for Q_n, and thus the bound of Proposition 3.13 is tight. We first prove a lemma limiting the f-trees among which we need to search for an optimal f-tree. It states that under any node of an f-tree it always pays off to branch into the maximal possible number of branches.

Definition 3.21. An f-tree $\mathcal{T}$ of an equi-join query Q is *maximally branching* if for each node $\mathcal{A}$, the children subtrees of $\mathcal{A}$ correspond to the connected components of the query $Q_{\mathcal{T}_\mathcal{A} \setminus \mathcal{A}}$. □

Note that for *any* f-tree $\mathcal{T}$ of an equi-join query and any node $\mathcal{A}$, each of the connected components of $Q_{\mathcal{T}_\mathcal{A} \setminus \mathcal{A}}$ is wholly contained in one of the children subtrees of $\mathcal{A}$. Otherwise some relation of that connected component would have its attributes in two distinct children subtrees of $\mathcal{A}$, which would violate the path condition. An f-tree $\mathcal{T}$ is maximally branching if the vertices of each connected component of $Q_{\mathcal{T}_\mathcal{A} \setminus \mathcal{A}}$ form a different subtree under $\mathcal{A}$.

Lemma 3.11. *For any equi-join query Q, there exists a maximally branching f-tree $\mathcal{T}_b$ with $s(\mathcal{T}_b) = s(Q)$.*

Proof. Let $\mathcal{T}$ be any f-tree of Q with $s(\mathcal{T}) = s(Q)$. Construct $\mathcal{T}_b$ by splitting $\mathcal{T}$ as much as possible but reflecting the original hierarchy of the nodes in $\mathcal{T}$. Formally, define split($\mathcal{T}$) recursively as follows. For a forest $\mathcal{U}$, $\text{split}(\mathcal{U}) = \bigcup_{\text{tree } \mathcal{T} \text{ in } \mathcal{U}} \text{split}(\mathcal{T})$. For a tree $\mathcal{T}$ with root $\mathcal{A}$ and forest of children subtrees $\mathcal{U}$, let $\{\mathcal{T}_1, \dots, \mathcal{T}_k\} = \text{split}(\mathcal{U})$, let $A = \{i : \mathcal{T}_i \text{ depends on } \mathcal{A}\}$, let $\mathcal{T}_a$ be a tree with root $\mathcal{A}$ and children subtrees $\{\mathcal{T}_i\}_{i \in A}$ and define $\text{split}(\mathcal{T}) = \{\mathcal{T}_i\}_{i \notin A} \cup \mathcal{T}_a$.

By structural induction on $\mathcal{T}$, we can prove that split($\mathcal{T}$) satisfies the path constraint, so $\mathcal{T}_b = \text{split}(\mathcal{T})$ is an f-tree of Q. By construction, $\mathcal{T}_b$ is maximally branching. Also, by structural induction we prove that $Q_{\text{path}_{\text{split}(\mathcal{T})}(\mathcal{A})} \subseteq Q_{\text{path}_{\mathcal{T}}(\mathcal{A})}$ for any node $\mathcal{A}$ in $\mathcal{T}$, so $\rho^*(Q_{\text{path}_{\text{split}(\mathcal{T})}(\mathcal{A})}) \leq \rho^*(Q_{\text{path}_{\mathcal{T}}(\mathcal{A})})$ and hence $s(\mathcal{T}_b) \leq s(\mathcal{T})$. Since $\mathcal{T}$ is optimal, we must in fact have $s(\mathcal{T}_b) = s(\mathcal{T}) = s(Q)$. □

Proposition 3.17. *For any chain query Q_n, $s(Q_n) = \Omega(\log n)$.*

Proof. Let $\mathcal{T}$ be a maximally branching optimal f-tree of Q_n. By top-down induction on $\mathcal{T}$ we can prove that the vertices of any subtree $\mathcal{T}_{\mathcal{A}}$ of $\mathcal{T}$ are $\{V_i\}_{i \in I}$ for a contiguous interval I of integers, and hence that any node in $\mathcal{T}$ has at most two children. It follows that the height of $\mathcal{T}$ is at least $\log_2(n+1)+1$. Since each hyperedge of Q_n covers at most two vertices, $s(Q_n) = s(\mathcal{T}) > (\log_2(n+1)+1)/2 = \Omega(\log n)$. □

3.5.4 Succinctness Gap for F-representations

For non-empty equi-join queries, any f-representation of the query result must be at least linear in the database size, while the result size can be exponential in the query size. We show that there exist queries for which this size gap is attained.

Proposition 3.18. *There exist arbitrarily large equi-join queries Q such that $s^{\uparrow}(Q) = s(Q) = 1$ and $\rho^*(Q) = |Q|$.*

Proof. The product query $Q = R_1 \times \cdots \times R_n$ over unary relations $R_1, \dots, R_n$ has $\rho^*(Q) = n = |Q|$ but $s(Q) = 1$. □

The product query is a trivial example, but there exist many others. In particular, any equi-join query Q, in which at least one attribute per relation is not involved in joins, has $\rho^*(Q) = |Q|$, yet many such queries still retain small $s(Q)$. For example, queries Q whose

Boolean projections $\pi_{\emptyset}Q$ are hierarchical [DS07] admit an f-tree $\mathcal{T}$ with $s(\mathcal{T}) = 1$. For each root-to-leaf path in such an f-tree there is a relation with attributes in each node of the path. A simple example of a hierarchical query is the join $\sigma_{A_1=\cdots=A_n}(R_1 \times \cdots \times R_n)$, where each R_i is over a schema $\{A_i, B_i\}$.

On the other hand, there exist queries for which $s(Q) = \rho^*(Q)$, and whose results hence cannot benefit from factorisations over f-trees. This happens when no branching is possible in f-trees of Q and all f-trees of Q are paths, so that $Q_{\text{path}(\mathcal{B})} = Q$ for the bottom node $\mathcal{B}$. All f-trees of a query Q are paths iff any two nodes are dependent, i.e., any two attribute classes have attributes from a common relation.

Proposition 3.19. *There exist arbitrarily large equi-join queries Q such that $s^{\uparrow}(Q) = s(Q) = \rho^*(Q) = \Omega(|\mathcal{S}|)$.*

Proof. Consider the relations $R_{i,j}$ for $1 \leq i < j \leq n$ with schemas $\{A^i_{i,j}, A^j_{i,j}\}$. Let $Q = \sigma_{\psi}(\times_{i<j} R_{i,j})$, where ψ equates all attributes with the same superscript. The hypergraph of Q is the complete graph on n nodes, so the possible f-trees of Q are the $n!$ paths of these nodes and the possible d-trees have $\text{key}(N) = \text{anc}(N)$ for all nodes N.

For each such d-tree $\mathcal{T}^{\uparrow}$, the query $Q_{\text{key}(\mathcal{B})\cup\mathcal{B}} = Q_{\text{path}(\mathcal{B})}$ of the bottom node $\mathcal{B}$ includes all nodes of $\mathcal{T}^{\uparrow}$ and hence is equal to Q, and its fractional edge cover number is $\rho^*(Q_{\text{key}(\mathcal{B})\cup\mathcal{B}}) = \rho^*(Q) = \binom{n}{2}\frac{1}{n-1} = \frac{n}{2}$. (An optimal fractional edge cover assigns weight $\frac{1}{n-1}$ to each of the $\binom{n}{2}$ relations.) It follows that $s^{\uparrow}(\mathcal{T}^{\uparrow}) = s(\mathcal{T}) = \rho^*(Q) = \frac{n}{2}$ for any f-tree $\mathcal{T}$ of Q, and hence $s^{\uparrow}(Q) = s(Q) = \rho^*(Q) = \frac{n}{2} = \Omega(|\mathcal{S}|)$. ☐

3.6 Computing F-representations and D-representations

In this section we give algorithms for computing f-representations and d-representations. First we show that for a general relation $\mathbf{R}$ and a d-tree $\mathcal{T}^{\uparrow}$ valid for $\mathbf{R}$, the constructive Definition 3.11 of the d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ can be converted into an algorithm computing $\mathcal{T}^{\uparrow}(\mathbf{R})$ with quasilinear data complexity (Proposition 3.20). The result naturally subsumes the case of f-representations over f-trees and is asymptotically optimal up to the logarithmic factor.

However, there exist queries whose results are exponentially larger than *both* the input database and their succinct f-representations and d-representations. We present algorithms that compute the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ of a query result $Q(\mathbf{D})$ directly from the input database $\mathbf{D}$, query Q and d-tree $\mathcal{T}^{\uparrow}$, without an intermediate computation of the potentially large flat result $Q(\mathbf{D})$. This allows an $o(|Q(\mathbf{D})|)$ time complexity, better than for any possible algorithm computing the flat result. In particular, for an equi-join query Q, we compute the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ with data complexity $O(|\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})}\log|\mathbf{D}|)$ (Proposition 3.22). The algorithm is extended to arbitrary conjunctive queries.

Any algorithm for computing d-representations over d-trees naturally subsumes the computation of f-representations over f-trees, in particular, for any equi-join Q and its f-tree $\mathcal{T}$, we can compute $\mathcal{T}(Q(\mathbf{D}))$ with data complexity $O(|\mathbf{D}|^{s(\mathcal{T})}\log|\mathbf{D}|)$.

3.6.1 Computing D-representations of Relations

We show that Definition 3.11 of the d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ of a given relation $\mathbf{R}$ over a given d-tree $\mathcal{T}^{\uparrow}$ can be easily converted into an algorithm that computes $\mathcal{T}^{\uparrow}(\mathbf{R})$ from $\mathbf{R}$ and $\mathcal{T}^{\uparrow}$ with quasilinear complexity. (We assume a schema of fixed size.)

Proposition 3.20. *Given a relation $\mathbf{R}$ over schema $\mathcal{S}$ and a d-tree $\mathcal{T}^{\uparrow}$ valid for $\mathbf{R}$, the d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ can be computed in time $O(|\mathbf{R}|\cdot\log|\mathbf{R}|\cdot|\mathcal{S}|\cdot|\mathcal{T}|^2)$.*

Proof. By Definition 3.11, the d-representation $\mathcal{T}^{\uparrow}(\mathbf{R})$ consists of the following expressions. For each subtree $\mathcal{T}_{\mathcal{A}}$ and any $t\in\pi_{\text{key}(\mathcal{A})}(\mathbf{R})$ we have

$$D(\mathbf{R},\mathcal{T}_{\mathcal{A}},t)=\textstyle\bigcup_{a\in A}\langle\mathcal{A}\!:\!a\rangle\times{}^{\uparrow}D(\mathbf{R},\mathcal{U},\pi_{\text{key}(\mathcal{U})}(t\times\langle\mathcal{A}\!:\!a\rangle)),$$

where $A=\pi_{\mathcal{A}}\sigma_{\text{key}(\mathcal{A})=t}\mathbf{R}$, and $\mathcal{U}$ is the forest of children of $\mathcal{A}$ (if $\mathcal{U}$ is empty, the expression $D(\mathbf{R},\mathcal{U},\cdot)$ is omitted). For each forest $\mathcal{U}$ of sibling subtrees $\mathcal{T}_1,\dots,\mathcal{T}_k$ and any $t\in\pi_{\text{key}(\mathcal{U})}(\mathbf{R})$ we have

$$D(\mathbf{R},\mathcal{U},t)={}^{\uparrow}D(\mathbf{R},\mathcal{T}_1,\pi_{\text{key}(\mathcal{T}_1)}t)\times\cdots\times{}^{\uparrow}D(\mathbf{R},\mathcal{T}_k,\pi_{\text{key}(\mathcal{T}_k)}t).$$

Algorithm. For any node $\mathcal{A}$, sort the entire relation $\mathbf{R}$ by $\text{key}(\mathcal{A})\cup\mathcal{A}$, so that it is grouped by $\text{key}(\mathcal{A})$, and the groups corresponding to $t\in\pi_{\text{key}(\mathcal{A})}\mathbf{R}$ are grouped by $\mathcal{A}$, the

subgroups corresponding to $\pi_{\mathcal{A}}\sigma_{\text{key}(\mathcal{A})=t}\mathbf{R}$. All expressions $D(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t)$ can be constructed from this information in one linear pass through $\mathbf{R}$. Similarly, for any forest $\mathcal{U}$, group the entire relation by $\text{key}(\mathcal{U})$, and in one pass through $\mathbf{R}$, construct the expression $D(\mathbf{R},\mathcal{U}, t)$ for each $t \in \pi_{\text{key}(\mathcal{U})}(\mathbf{R})$. In the implementation, index the expressions $D(\mathbf{R}, \mathcal{X}, t)$ only by $\mathcal{X}$ and t, as $\mathbf{R}$ is the same for all expressions. To create a parse tree of the d-representation, construct a parse tree of each expression separately, insert all expression names into an associative map, and redirect all edges ending at a reference $\uparrow D$ to the root of D.

Running time. The relation has $|\mathbf{R}|$ tuples of size $|\mathcal{S}|$ each. Sorting the relation thus takes time $O(|\mathbf{R}| \cdot \log |\mathbf{R}| \cdot |\mathcal{S}|)$. Constructing each reference $\uparrow D(\mathbf{R}, \mathcal{X}, t)$ takes time $O(|\mathcal{S}|)$ and there are $O(|\mathcal{T}|)$ references in each expression. There are at most $|\mathbf{R}|$ expressions computed in each linear pass. There is one linear pass for each node and forest in $\mathcal{T}$, so the total running time of the algorithm is $O(|\mathbf{R}| \cdot \log |\mathbf{R}| \cdot |\mathcal{S}| \cdot |\mathcal{T}|^2)$. Each operation in an associative map of $O(|\mathbf{R}| \cdot |\mathcal{T}|^2)$ names takes time $O(\log |\mathbf{R}| + \log |\mathcal{T}|)$, so creating the parse tree does not increase the time complexity. $\square$

3.6.2 Computing D-representations of Equi-Join Query Results

We show that the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ of any equi-join query result can be computed directly from the input database $\mathbf{D}$ and the query Q, with data complexity $O(|\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})} \log |\mathbf{D}|)$. The best asymptotic bound for the size of $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ is $O(|\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})})$, so the algorithm is worst-case optimal up to the logarithmic factor. It is not instance-optimal; for particular databases $\mathbf{D}$ the d-representation may be of size $o(|\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})})$ but the algorithm may still take $\Omega(|\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})})$.

The main idea of the algorithm is to evaluate individual subqueries $Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}$ for each node $\mathcal{A}$ and then stitch their results together into the d-representation of the result of Q. The results of the subqueries $Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}$ represent the largest "non-factorised" fragments of the d-representation and in fact dictate its size as shown in Section 3.4. Each result $Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D})$ can be computed in traditional flat form using one of the known worst-case optimal algorithms in time $O(|\mathbf{D}|^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})})$ [NPRR12, Vel12].

The d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ contains one singleton $\langle\mathcal{A}\!:\!a\rangle$ for each tuple in $\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}(Q(\mathbf{D}))$. (More precisely, for each $t \in \pi_{\text{key}(\mathcal{A})}(Q(\mathbf{D}))$ it contains a union of $\langle\mathcal{A}\!:\!a\rangle$ over $a \in \pi_{\mathcal{A}}\sigma_{\text{key}(\mathcal{A})=t}(Q(\mathbf{D}))$.)

ALGORITHM 1: Computing the d-representation of an equi-join query result.

Data: Equi-join query Q, d-tree $\mathcal{T}^{\uparrow}$, database $\mathbf{D}$.
Result: D-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$.

$R \longleftarrow$ empty d-representation;

```
1 for all nodes 𝒜 in 𝒯 do
    let {𝒯_1, ..., 𝒯_k} = 𝒰 ⟵ forest of children subtrees of 𝒜 in 𝒯;
    R_𝒜 ⟵ Q_key(𝒜)∪𝒜(D) using a worst-case optimal algorithm for equi-joins;
    group R_𝒜 by key(𝒜);
    for t ∈ π_key(𝒜)(R_𝒜) do
        A_t ⟵ π_𝒜 σ_key(𝒜)=t(R_𝒜);
        if 𝒰 = ∅ then D(𝒯_𝒜, t) ⟵ ⋃_{a∈A_t} ⟨𝒜:a⟩;
                else D(𝒯_𝒜, t) ⟵ ⋃_{a∈A_t} ⟨𝒜:a⟩ × ↑D(𝒰, π_key(𝒰)(t × ⟨𝒜:a⟩));
        add expression D(𝒯_𝒜, t) to R;
    end
    group R_𝒜 by key(𝒰);
    for t ∈ π_key(𝒰)(R_𝒜) do
        D(𝒰, t) ⟵ ↑D(𝒯_1, π_key(𝒯_1) t) × ··· × ↑D(𝒯_k, π_key(𝒯_k) t);
        add expression D(𝒰, t) to R;
    end
  end
  if 𝒯 is a forest 𝒯_1, ..., 𝒯_k then add D(𝒯, ⟨⟩) ⟵ ↑D(𝒯_1, ⟨⟩) × ··· × ↑D(𝒯_k, ⟨⟩) to R;
  set D(𝒯, ⟨⟩) as the root of R;
2 for all expressions D in R, bottom-up do
    if D = ⋃_a ⟨𝒜:a⟩ × ↑D_a then
        D ⟵ ⋃_{a:D_a≠∅} ⟨𝒜:a⟩ × ↑D_a;
    else if D = ↑D_1 × ··· × ↑D_k where some D_i = ∅ then
        D ⟵ ∅;
    end
  end
  return D;
```

We first construct a larger d-representation with one singleton $\langle \mathcal{A}{:}a\rangle$ for each tuple in $Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D})$, which contains $\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}(Q(\mathbf{D}))$. Then we identify the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ as a subset of the computed d-representation by removing all its subexpressions that represent the empty relation. The algorithm is given in pseudocode as Algorithm 1. We next prove its correctness and time performance.

Proposition 3.21. *For any equi-join query Q, its d-tree $\mathcal{T}^{\uparrow}$ and database $\mathbf{D}$, Algorithm 1 computes the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$.*

Proof. First we prove that before block 2 in Algorithm 1, the set of expressions R when interpreted as a parse graph contains the d-representation $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ as a subgraph. Since $\mathbf{R}_{\mathcal{A}} = Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}})$ contains $\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}(Q(\mathbf{D}))$ by Lemma 2.1, $\pi_{\text{key}(\mathcal{A})}(\mathbf{R}_{\mathcal{A}})$ con-

tains $\pi_{\text{key}(\mathcal{A})}(Q(\mathbf{D}))$ and similarly $\pi_{\text{key}(\mathcal{U})}(\mathbf{R}_\mathcal{A})$ contains $\pi_{\text{key}(\mathcal{U})}(Q(\mathbf{D}))$ for the forest $\mathcal{U}$ under $\mathcal{A}$, so for each expression $D(\mathbf{R},\mathcal{X},t)$ in $\mathcal{T}^\uparrow(Q(\mathbf{D}))$ as per Definition 3.11, the set R also contains an expression named $D(\mathbf{R},\mathcal{X},t)$. Moreover, all expressions $D(\mathbf{R},\mathcal{U},t)$ in R are precisely as defined by Definition 3.11, and all expressions $D(\mathbf{R},\mathcal{T}_\mathcal{A},t)$ in R contain as a subexpression the one defined by Definition 3.11: they are of the same form, except that the range of their union, $\pi_\mathcal{A}\sigma_{\text{key}(\mathcal{A})=t}(\mathbf{R}_\mathcal{A})$, may be larger than $\pi_\mathcal{A}\sigma_{\text{key}(\mathcal{A})=t}Q(\mathbf{D})$. This shows that $\mathcal{T}^\uparrow(Q(\mathbf{D}))$ as per Definition 3.11 is a subgraph of the parse graph of R.

Next we prove that the set of expressions R before block 2 is a d-representation of the result $Q(\mathbf{D})$. First note that by labelling each subexpression $D(\mathbf{R},\mathcal{X},t)$ with the schema of $\mathcal{X}$, R indeed becomes a d-representation over the schema of $Q(\mathbf{D})$ with root $D(\mathbf{R},\mathcal{T},\langle\rangle)$. By top-down induction over $\mathcal{T}$ it follows that in the traversal of R, each expression $D(\mathbf{R},\mathcal{X},t)$ is multiplied by singletons of all attributes from $\text{anc}(\mathcal{X})$, and those from $\text{key}(\mathcal{X})$ coincide with t. Each singleton $\langle \mathcal{A}\!:\!a\rangle$ in R is in some expression $D(\mathbf{R},\mathcal{T}_\mathcal{A},t)$, where by construction we have $t\times\langle A\!:\!a\rangle\in\mathbf{R}_\mathcal{A}$, and this singleton is multiplied by t in any tuple represented by R. Therefore, for any tuple d represented by R and any node $\mathcal{A}$, $\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}\in\mathbf{R}_\mathcal{A}$, and hence $[\![R]\!]\subseteq\Join_\mathcal{A}\mathbf{R}_\mathcal{A}$. Since each relation R_i has all its attributes included in some $\text{key}(\mathcal{A})\cup\mathcal{A}$, and hence it is unrestricted in $\mathbf{R}_\mathcal{A}$, we can deduce that $[\![R]\!]\subseteq\Join_i\mathbf{R}_i=Q(\mathbf{D})$. Since R contains the d-representation $\mathcal{T}^\uparrow(Q(\mathbf{D}))$, and both are of the same schema, it follows that $[\![R]\!]\supseteq[\![\mathcal{T}^\uparrow(Q(\mathbf{D}))]\!]=Q(\mathbf{D})$. Therefore $[\![R]\!]=Q(\mathbf{D})$.

Finally we prove that after block 2, R equals $\mathcal{T}^\uparrow(Q(\mathbf{D}))$. We have shown above that before block 2, R contains $\mathcal{T}^\uparrow(Q(\mathbf{D}))$, but its unions $D(\mathbf{R},\mathcal{T}_\mathcal{A},t)$ may contain additional terms. Since $[\![D]\!]=[\![\mathcal{T}^\uparrow(Q(\mathbf{D}))]\!]$, all these additional terms must represent the empty relation, otherwise they would contribute additional tuples to $[\![D]\!]$. Block 2 of Algorithm 1 removes exactly these terms and no others, the resulting d-representation R is therefore equal to $\mathcal{T}^\uparrow(Q(\mathbf{D}))$. □

Proposition 3.22. *For any equi-join query Q, its d-tree $\mathcal{T}^\uparrow$ and database $\mathbf{D}$, Algorithm 1 runs in time $O(|\mathbf{D}|^{s^\uparrow(\mathcal{T}^\uparrow)}\cdot\log|\mathbf{D}|\cdot poly(|Q|,|\mathcal{S}|))$.*

Proof. The computation of each $\mathbf{R}_\mathcal{A}$ takes $O(|\mathbf{D}|^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})})=O(|\mathbf{D}|^{s^\uparrow(\mathcal{T}^\uparrow)})$ using a worst-case optimal join algorithm [NPRR12] and the size of $\mathbf{R}_\mathcal{A}$ is also $O(|\mathbf{D}|^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})})=$

$O(|\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})})$. The group-by can then be implemented using a sort in time $O(|\mathbf{R}_{\mathcal{A}}| \cdot \log |\mathbf{R}_{\mathcal{A}}|) = O(|\mathbf{D}|^{s^{\uparrow}(\mathcal{T}^{\uparrow})} \log |\mathbf{D}|)$, all remaining processing takes time linear in $|\mathbf{R}_{\mathcal{A}}|$. If the d-representation is constructed as a parse-graph, the look-up of each expression name in an associative map takes time logarithmic in the total number of expressions, which is $O(|\mathbf{R}_{\mathcal{A}}|)$, so the total time is still quasi-logarithmic in $|\mathbf{R}_{\mathcal{A}}|$. The normalisation procedure implemented in block 2 of the algorithm takes time linear in the result computed thus far, so does not increase the runtime complexity. □

For ease of analysis, in Algorithm 1 we encapsulate the computation of joins $\mathbf{R}_{\mathcal{A}}$, and use their results to construct the d-representation $Q(\mathbf{D})$. It is possible to amalgamate the entire computation into a single multi-way merge-join, as done in [OZ12] for f-representations only.

3.6.3 Computing D-representations of Conjunctive Query Results

The algorithm for equi-join queries can be extended to arbitrary conjunctive queries using d-tree extensions. Recall from Proposition 3.9 that any d-tree of a conjunctive query Q can be extended to a d-tree of the equi-join $\hat{Q}$ of Q.

Proposition 3.23. *Given any conjunctive query Q, a d-tree $\mathcal{T}^{\uparrow}$ of Q and its extension $\hat{\mathcal{T}}^{\uparrow}$, and a database $\mathbf{D}$, we can compute $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$ in time $O(|\mathbf{D}|^{s^{\uparrow}(\hat{\mathcal{T}}^{\uparrow})} \cdot \log |\mathbf{D}|)$ with respect to data complexity.*

Proof. Using the extension d-tree $\hat{\mathcal{T}}^{\uparrow}$, which is a d-tree of the equi-join $\hat{Q}$, we can compute the d-representation $\hat{\mathcal{T}}^{\uparrow}(\hat{Q}(\mathbf{D}))$ in time $O(|\mathbf{D}|^{s^{\uparrow}(\hat{\mathcal{T}}^{\uparrow})} \cdot \log |\mathbf{D}|)$ by Proposition 3.22. Since $\hat{\mathcal{T}}^{\uparrow}$ is an extension of $\mathcal{T}^{\uparrow}$, it contains additional non-head attributes in some nodes, and also additional subtrees and subforests consisting of non-head attributes only. With respect to $\mathcal{T}^{\uparrow}(Q(\mathbf{D}))$, the d-representation $\hat{\mathcal{T}}^{\uparrow}(\hat{Q}(\mathbf{D}))$ therefore contains additional singletons $\langle A\!:\!a \rangle$ for non-head attributes A that are in a node with some head attribute, and additional expressions $E(\mathcal{X}, t)$ for subtrees and subforests $\mathcal{X}$ consisting of non-head attributes only. Both the additional singletons and expressions can be removed from $\hat{\mathcal{T}}^{\uparrow}(\hat{Q}(\mathbf{D}))$ in time linear in its size, so the total runtime is still $O(|\mathbf{D}|^{s^{\uparrow}(\hat{\mathcal{T}}^{\uparrow})} \cdot \log |\mathbf{D}|)$ with respect to data complexity. □

3.7 Related Work

The language of factorisations, in particular factorisations over f-trees, encompasses various representation systems in relational databases, probabilistic and possibilistic databases, or formal concept analysis.

Representations Equivalent to Factorisations over F-trees

Equivalent to the special case of factorised representations over f-trees are generalised hierarchical decompositions (GHDs) and compacted relations over compaction formats. Prior work establishes the correspondences of GHDs to functional and multi-valued dependencies [Del78], and characterises selection conditions with disjunctions that can be performed on the compacted relations in one sequential pass [BRS82], but questions of representation succinctness have not been addressed.

Nested Relations

Nested and non-first normal form relations [Mak77, JS82, AB86] are also closely related to f-representations over f-trees. The nesting structures of nested relations are described by scheme trees [OY87] or Verso formats [AB86] similarly as the nesting structure of f-representations is captured by f-trees. A minor difference to our work is that nested relations allow parts of the schema to be unnested. An unnested portion of the schema is denoted by placing different attributes in a single node of the scheme tree (or Verso format), products of singletons over these attributes can have arbitrary values.

In much of existing work, nested relations are treated as a data model alternative to the first normal form relations, modelling data of hierarchical nature. In the Verso project [AB86, Bid87], a nested relation (called a Verso instance) represents a collection of flat relations over all root-to-node schemas (called the format skeleton) of the Verso format. In the present work an f-representation always represents a relation over the entire schema, obtained by the natural join of all flat relations over root-to-node schemas (called the join closure of the Verso instance). The pre-join semantics of Verso instances allows e.g. for empty subexpressions in some branches of the nested relation, and a query language

with "exists" predicates, but also reduces the number possible formats representing a given relation.

Later work on nested relations [OY87] does consider the representation of a single flat relation by a nested relation, and studies how the presence of join and multi-valued dependencies influences the possible nesting structures of the representing nested relation, thus complementing our results in Section 3.3 which characterise f-trees admissible for the results of a given conjunctive query. For a given set M of multi-valued dependencies, a nested normal form corresponding to M is inferred. In line with Section 3.3, the root-to-leaf paths of the obtained scheme tree correspond to the 4NF decompositions of the original relation with respect to M.

We are not aware of prior work studying the size of nested relations or the succinctness of representation by nested relations.

Representations Subsumed by Factorisations

Various relational representation systems are subsumed by factorised representations of bounded depth. World-set decompositions in incomplete databases [AKO07, OKA08] and OR-objects that represent large spaces of possibilities or choices in design specification [INV91] are equivalent to products of unions of products of singletons. A polynomial-time factorisation algorithm (called prime factorisation of relations) has been proposed for decomposing a relation into a product of unions of products of singletons [OKA08]. Products of unions of singletons are studied under different names (rectangles, bicliques in binary relations, n-sets, formal concepts) and several representation systems are based on unions of products of unions of singletons, such as generalised disjunctive normal forms (GDNFs) studied as succinct presentations of inputs to CSPs [CG10], tilings of databases by bicliques, n-sets or formal concepts [GGM04, CBRB09], and others.

Other Representations and Decompositions

In relational databases, eliminating redundancy caused by join dependencies and multi-valued dependencies is traditionally addressed by normalising the relational schema [Ken83]. Representation systems for relations based on join decompositions include minimal con-

straint networks [Got12], but for these data retrieval (tuple enumeration) is NP-hard. Tuple enumeration is constant time for acyclic queries, in which case the input database together with the query already serve as a compact representation of the result, for general conjunctive queries it is NP-hard [BDG07].

Decompositions of the query hypergraph and the associated parameters measuring the "degree of acyclicity" of the query, such as tree decompositions, fractional and generalised hypertree decompositions, are traditionally used for classifying the tractability and complexity of Boolean queries and constraint satisfaction problems [GLS00, GLS01, GM06].

Factorisation Beyond Relational Data

Representations utilising algebraic factorisation are not restricted to relational data. In the context of relational databases, factorisation can also be applied to provenance polynomials [GKT07] that describe how individual tuples of a query result depend on tuples of the input relations, see Chapter 4. Algebraic and Boolean factorisations were considered in succinct representations of Boolean functions [Bra87] and are closely related to binary decision diagrams, Boolean circuits and other representations of Boolean functions.

3.8 Conclusion and Directions for Future Work

The central concept of this dissertation is that of a factorised representation of a relation. This chapter defines factorised representations in two flavours (with and without definitions), establishes that the represented relation can be enumerated with constant delay under very mild conditions, and studies in depth the representability of conjunctive query results by factorisations. We determine nesting structures that factorise query results, quantify their sizes using rational parameters $s(Q)$ and $s^{\uparrow}(Q)$, and place these factorisability parameters within the picture of other known query decomposability parameters. Query-dependent asymptotic bounds for sizes of factorisations are complemented by algorithms that, for equi-join queries, compute the factorisations within the same time bounds in the worst case. Possible directions for further research include improving these algorithms towards instance-optimality, determining the computational complexity of finding the fac-

torisability parameters $s(Q)$ and $s^{\uparrow}(Q)$, and extending the study of factorisability beyond conjunctive queries and beyond factorisations with regular nesting structures defined by f-trees and d-trees.

Instance-optimal Algorithms

The algorithms for computing f-representations and d-representations of query results presented in this work are proven to be worst-case optimal. Recent results [NNRR13] towards instance-optimal join algorithms with flat relational results can perhaps be extended to the factorised case.

Complexity of Finding $s(Q)$ and $s^{\uparrow}(Q)$

This work introduces the parameters $s(Q)$ and $s^{\uparrow}(Q)$, which characterise the succinctness of f-representations and d-representations of conjunctive query results, and relates these parameters to other known measures such as fractional hypertree width. The complexity of computing $s(Q)$ and $s^{\uparrow}(Q)$ for a given conjunctive query Q is unaddressed and still open. The related complexity of computing the fractional hypertree width is also open, with partial results on its approximation [Mar09].

Factorisation Beyond Conjunctive Query Results

The questions of factorisability of query results, in particular finding regular nesting structures that can be inferred from the query, and finding bounds on factorisation size, can be posed beyond conjunctive queries. The natural candidates for extension are unions of conjunctive queries and conjunctive queries with inequalities or disqualities.

For arbitrary input relations beyond query results, the computation of an optimal factorised representation is likely to be hard, similar to the hardness of minimisation of Boolean functions [BU08]. For binary relations the problem of decomposing a relation into a minimum number of products of unions is NP-complete [Ami], as is the much better studied problem of covering a binary relation by products of unions [GGM04, EHM^{+}08]: in this case the products of unions can overlap and hence yield a non-deterministic f-representation.

Determining the precise complexity of various factorisation minimisation problems, establishing lower bounds for query result factorisation size under broader classes of factorisations than those over f-trees and d-trees, as well as quantifying the succinctness gaps between f-representations and d-representations of various flavours (deterministic and nondeterministic, of bounded depth, over f-trees and d-trees, or with other regular nesting structures), is subject to future work. A robust approach to approximate instance-based factorisation would be desirable in practice (see Section 5.5 for more details).

Chapter 4

Factorisation and Readability of Provenance

Tuples in relational databases can be annotated by metadata specifying their provenance, and such provenance annotations can be propagated through queries from input to the query result [GKT07]. Tracking and managing provenance information in databases has applications in incomplete information and probabilistic databases where provenance encodes tuple probabilities, debugging and explanation if the provenance encodes tuple origin, security and access control if the provenance encodes access restrictions, as well as in query evaluation under bag semantics, view maintenance and update, annotation propagation, and others [CCT09]. Provenance information is not projected away in queries and hence tends to grow very large with the number of query operations. This chapter extends the notions of succinct representation by algebraic factorisation from relations and query results to their provenance annotations, derives analogous bounds on size of provenance factorisations, and studies a related concept of readability of provenance.

For many flavours of provenance semantics, the algebra of combining annotations of input tuples to produce the annotations of output tuples is captured by the unifying formalism of *provenance semirings* [GKT07]: tuples of relations are annotated by elements of a semiring and annotations of query result tuples are expressed as *provenance polynomials* of input annotations. Provenance polynomials yield naturally to algebraic factorisation since

Orders

	id	item
o_1	01	Printer
o_2	02	Scanner
o_3	03	Ink
o_4	04	Printer
o_5	05	Ink

Store

	location	item
s_1	Depot1	Printer
s_2	Depot1	Scanner
s_3	Depot2	Printer
s_4	StoreA	Ink

Employees

	operator	location
e_1	Joe	Depot1
e_2	Bob	Depot1
e_3	Dan	Depot2
e_4	Dan	StoreA

Figure 4.1: A simplified database of an office-supply warehouse. The tuples of each relation are annotated by distinct tuple identifiers, in the leftmost column.

multiplication distributes over addition in the provenance semiring.

Example 4.1. Consider an office-supply warehouse database, showing current orders, stocks and employee availability. In this scenario, data can be aggregated from different extraneous or uncertain source databases: Orders could come from different vendors, storage information from different stock lists at each location, and employee availability could be produced by real-time tracking. Each tuple in the input database is annotated with a distinct variable which encodes here its provenance information. When deciding on which employee should deliver what order from which location, the source of information supporting the decision, that is, its provenance, is essential, and should ideally be recorded for later reference. If results of queries evaluated on the database are used to support the decisions, the provenance of each result tuple is necessarily derived from the provenance of the input tuples that he output tuple was inferred from.

Consider a query that finds all orders with their respective items, their possible locations and workers available to access and retrieve them:

Orders $\bowtie_{\text{item}}$ Store $\bowtie_{\text{location}}$ Employees

	id	item	location	operator
$o_1 s_1 e_1$	01	Printer	Depot1	Joe
$o_1 s_1 e_2$	01	Printer	Depot1	Bob
$o_1 s_3 e_3$	01	Printer	Depot2	Dan
$o_2 s_2 e_1$	02	Scanner	Depot1	Joe
		...		

Each tuple in the result table arises as a combination of input tuples, and its provenance is the product of variables of the contributing input tuples. For example, the tuples

annotated with o_1, s_1 and e_1 contribute to the result tuple annotated with $o_1 s_1 e_1$. Consider now a query that projects the above join to the single attribute "operator", computing the list of operators that can service at least one item from at least one order.

$\pi_{\text{operator}}(\text{Orders} \bowtie_{\text{item}} \text{Store} \bowtie_{\text{location}} \text{Employees})$

	operator
$o_1 s_1 e_1 + o_2 s_2 e_1 + o_4 s_1 e_1$	Joe
$o_1 s_1 e_2 + o_2 s_2 e_2 + o_4 s_1 e_2$	Bob
$o_1 s_3 e_3 + o_3 s_4 e_4 + o_4 s_3 e_3 + o_5 s_4 e_4$	Dan

Each tuple in the projection $\pi_{\text{operator}} R$ can be derived from several tuples in the join R, and the provenance polynomial of each output tuple is the sum of the provenance polynomials of the possible source input tuples. For example, the tuple (Joe) can be derived from the tuples (01, Printer, Depot1, Joe), (02, Scanner, Depot1, Joe) or (04, Printer, Depot1, Joe) with provenance polynomials $o_1 s_1 e_1$, $o_2 s_2 e_1$ and $o_4 s_1 e_1$ respectively, so its provenance polynomial is their sum $o_1 s_1 e_1 + o_2 s_2 e_1 + o_4 s_1 e_1$.

This polynomial contains 9 occurrences of variables, but has an equivalent factorisation $((o_1 + o_4)s_1 + o_2 s_2)e_1$ that only contains 6 occurrences. □

The abstract product and sum operations on variables in the free semiring of provenance polynomials can be interpreted by multiplication and summation in different commutative semirings to model different interpretations of provenance information [GKT07]. In the Boolean semiring $(\mathbb{B}, \vee, \wedge)$, we interpret the variables as Booleans and the product and sum as logical "and" ($\wedge$) and "or" ($\vee$) respectively. The Boolean semiring captures the semantics of positive relational algebra queries under set semantics: In the previous example, if the variables o_1, s_1, and e_1 are set to true, then the input tuples they annotate are in the database and the tuple annotated with $o_1 s_1 e_1$ is in the query result. The semiring over natural numbers $(\mathbb{N}, +, \cdot)$ captures the semantics of positive queries under bag semantics: the variables are interpreted as input tuple multiplicities and the values of provenance polynomials of the output tuples then indicate their multiplicities in the query result. The semiring of all positive Boolean expressions is used in the context of incomplete and probabilistic databases, and various purpose-built semirings are used for tracing lineage and explaining

query results.

While the provenance polynomial of each tuple of an equi-join only contains one monomial, the provenance polynomial of the result of the corresponding Boolean query is the sum of all these monomials, and can be potentially very large. We exploit a correspondence between provenance polynomials and query results to obtain succinct factorisations of provenance polynomials. Factorised provenance polynomials can be exponentially smaller than their flat counterparts, while under any concrete semiring their values under a given valuation of variables can be computed in linear time without expanding to the flat sum of products. Along with being of smaller size, factorisations of provenance polynomials contain less occurrences of each individual annotation identifier than their flat counterparts, which is crucial in some applications such as probabilistic databases. In addition to size we study the readability of provenance polynomials: the minimum number of occurrences of a variable in any factorisation.

In this chapter we present the following results on the factorisability and readability of provenance polynomials.

- We express provenance polynomials of conjunctive queries using relational algebra over annotated relations, and extend the notions on factorisations of query results over f-trees to factorisations of provenance polynomials over f-trees.

- Analogous to the asymptotic bounds on factorisation size from Section 3.4, we derive asymptotic bounds on the number of occurrences of variables in provenance polynomials, and deduce upper bounds on the readability of their provenance. The asymptotic bounds are characterised by a new query parameter $r(Q)$ called the *readability width.*

- Among non-repeating conjunctive queries, we identify the class of queries with provenance of bounded readability as the hierarchical queries, which also arise in other contexts. We establish a gap between the hierarchical and non-hierarchical queries in readability width ($r(Q) = 0$ vs. $r(Q) \geq 1$) and in readability of their provenance (bounded vs. $\Omega(\sqrt{|\mathbf{D}|})$).

4.1 Provenance Polynomials and Annotated Databases

Provenance polynomials are defined formally using K-relations; relations where each tuple is annotated by a value from a semiring K. All operators of positive relational algebra can have their semantics extended to K-relations [GKT07], we will only use a limited and direct definition for conjunctive queries [Gre09]. Intuitively, a derivation of an output tuple of a conjunctive query is a set of input tuples that produce that output tuple, and each output tuple is annotated by a sum of products of annotations of input tuples, one product corresponding to each derivation of that output tuple.

Definition 4.1. Let K be a commutative semiring. A *K-relation* is a relation $\mathbf{R}$ together with a mapping $\mathrm{val}_{\mathbf{R}} : \mathbf{R} \mapsto K$. Let $Q = \pi_{\mathcal{P}}(\sigma_{\psi}(R_1 \times \ldots \times R_n))$ be a conjunctive query and let $\mathbf{D}$ be a database of K-relations. A *derivation* of a tuple t in the relational result $Q(\mathbf{D})$ is a tuple $(t_1, \ldots, t_n) \in \mathbf{R}_1 \times \cdots \times \mathbf{R}_n$ such that $t_1 \times \cdots \times t_n$ satisfies the condition ψ and its restriction to $\mathcal{P}$ equals t. The K-relation $Q(\mathbf{D})$ is defined as the relation $Q(\mathbf{D})$ together with the mapping

$$\mathrm{val}_{Q(\mathbf{D})} : \quad t \mapsto \textstyle\sum_{(t_1, \ldots, t_n) \text{ is a derivation of } t} (\mathrm{val}_{\mathbf{R}_1}(t_1) \cdot \ldots \cdot \mathrm{val}_{\mathbf{R}_n}(t_n)).$$

□

Let each input tuple be annotated by a distinct tuple identifier from a set X. The *provenance polynomial* of an output tuple in a result of a query is its annotation under the K-relation semantics with $K = \mathbb{N}[X]$, the free commutative semiring over the set of identifiers [GKT07].

For the purpose of this dissertation, in order to re-use results from Chapter 3, we will define provenance polynomials as relations, by using separate relational attributes to carry the provenance annotations. An annotated relation R is a relation with a special *annotation attribute*, denoted by $\underline{R}$, which carries annotations in the form of distinct tuple identifiers.

Example 4.2. Figure 4.2 shows an example database with relations R, S and T over schemas $\{A_R, B_R\}$, $\{B_S, C_S\}$ and $\{C_T, D_T\}$ respectively, and an annotated version of that database, in which schemas are extended with the annotation attributes $\underline{R}$, $\underline{S}$ and $\underline{T}$ respectively. The result of the query $Q_1 = \sigma_{\psi}(R \times S \times T)$ with $\psi = (B_R = B_S \wedge C_S = C_T)$ on the annotated

R	
A_R	B_R
1	2
1	3
2	3

S	
B_S	C_S
2	1
3	1
3	2

T	
C_T	D_T
1	2
1	3
2	3

R		
$\underline{R}$	A_R	B_R
r_{12}	1	2
r_{13}	1	3
r_{23}	2	3

S		
$\underline{S}$	B_S	C_S
s_{21}	2	1
s_{31}	3	1
s_{32}	3	2

T		
$\underline{T}$	C_T	D_T
t_{12}	1	2
t_{13}	1	3
t_{23}	2	3

Figure 4.2: An example database with relations R, S and T (top). An annotated version of the same database (bottom). The leftmost (underlined) column in each relation is the annotation attribute whose values are tuple identifiers.

database is

$\underline{R}$	$\underline{S}$	$\underline{T}$	A_R	B_R	B_S	C_S	C_T	D_T
r_{12}	s_{21}	t_{12}	1	2	2	1	1	2
r_{12}	s_{21}	t_{13}	1	2	2	1	1	3
r_{13}	s_{31}	t_{12}	1	3	3	1	1	2
	...				...			

□

When considering conjunctive queries on annotated databases, we never project out the annotation attributes, these are always implicitly present in the projection list. A conjunctive query $Q = \pi_{\mathcal{P}}(\sigma_{\psi}(R_1 \times \ldots \times R_n))$ on an unannotated database becomes $Q_{\text{ann}} = \pi_{\mathcal{P}\cup\mathcal{I}}(\sigma_{\psi}(R_1 \times \ldots \times R_n))$ on the annotated database, where $\mathcal{I}$ denotes the annotation attributes of $R_1, \ldots, R_n$. The result of Q on an annotated database $\mathbf{D}$ contains one tuple for each derivation of each tuple of the unannotated version of $\mathbf{D}$. We define the provenance of an output tuple t as the relation of annotations of the derivations of t.

Definition 4.2. The *provenance* $\varphi(t)$ of a tuple t in the result of a query Q on an annotated database $\mathbf{D}$ is the relation $\pi_{\mathcal{I}}\sigma_{\mathcal{P}=t}(Q(\mathbf{D}))$. □

The provenance $\varphi(t)$ of any tuple t exactly corresponds to the provenance polynomial of t as defined above. This correspondence allows us to use the factorisation methods for query results, which we developed in the previous chapter, to factorise provenance polynomials.

Proposition 4.1. *If we interpret relational product ($\times$) as multiplication and relational union ($\cup$) as addition, then $\varphi(t)$ becomes a provenance polynomial over tuple identifier singletons.*

Example 4.3. The provenance of the tuple $\langle 1, 2, 2, 1, 1, 2 \rangle$ in the result of Q_1 given in Example 4.2 is the relation

$\underline{R}$	$\underline{S}$	$\underline{T}$
r_{12}	s_{21}	t_{12}

and its provenance polynomial is $r_{12}s_{21}t_{12}$. The provenance $\varphi(\langle\rangle)$ of the tuple $\langle\rangle$ in the result of the Boolean query $\pi_{\emptyset}Q_1$ is the projection of the result of Q_1 to the annotation attributes $\{\underline{R}, \underline{S}, \underline{T}\}$

$\underline{R}$	$\underline{S}$	$\underline{T}$
r_{12}	s_{21}	t_{12}
r_{12}	s_{21}	t_{13}
r_{13}	s_{31}	t_{12}
	$\dots$	

and its provenance polynomial is

$$r_{12}s_{21}t_{12} + r_{12}s_{21}t_{13} + r_{13}s_{31}t_{12} + r_{13}s_{31}t_{13} +$$
$$r_{13}s_{32}t_{23} + r_{23}s_{31}t_{12} + r_{23}s_{31}t_{13} + r_{23}s_{32}t_{23}.$$

□

Remark 4.1. In the previous chapter we characterised queries by the factorisability of their results, and we did not need to specify whether we consider queries as (syntactical) query expressions or their (semantical) equivalence classes under set semantics (cf. Remark 3.2). However, different query expressions equivalent under relational set semantics can produce different provenance polynomials for their result tuples, since the notion of query equivalence is weaker under the Boolean semiring than under the semiring $\mathbb{N}[X]$ of provenance polynomials [Gre09]. Therefore, in this chapter, by a *query* we will always mean a query expression as defined in Chapter 2. (Two conjunctive query expressions are equivalent under the $\mathbb{N}[X]$-relation semantics iff their labelled hypergraphs are isomorphic [Gre09], therefore

we could also define a query by its labelled hypergraph. Our definition of f-trees and other parameters is formulated in terms of query hypergraphs anyway, so it would stay the same.)

4.2 Factorisation of Provenance Using F-Trees

In this section we consider the problem of factorising provenance polynomials of tuples in conjunctive query results, or equivalently the provenance of the tuples as defined in Definition 4.2. A first observation is that the f-trees of Q cannot be used to factorise effectively the provenance of Q, unless the query is a product of relations. To see this, consider a Boolean query Q joining n annotated relations.[1] Any f-tree of Q has one node for each annotation attribute, and any pair of annotation attributes is Q-dependent. The f-trees are thus paths of length n and lead to poorly factorisable representations. Our factorisation approach for provenance of a query Q does not use f-trees of Q but of a query that can be derived from Q, as discussed next.

First consider a Boolean query Q on an annotated database. In the annotated result of its equi-join $\hat{Q}(\mathbf{D})$, each tuple is multiplied by a provenance monomial giving its derivation. We can obtain the provenance of $\langle\rangle$ in $Q(\mathbf{D})$ from the equi-join result by dropping all data singletons and keeping the annotations only. Any f-tree $\hat{\mathcal{T}}$ of $\hat{Q}$ defines an f-representation $\hat{\mathcal{T}}(\hat{Q}(\mathbf{D}))$ of the equi-join result; we can obtain an f-representation of the provenance $\varphi(\langle\rangle)$ from the f-representation of the equi-join result by dropping all data singletons.

Example 4.4. Recall the Boolean query $\pi_{\emptyset}Q_1$ and the database $\mathbf{D}$ from Example 4.2, and the provenance $\varphi(\langle\rangle)$ whose polynomial is shown in Example 4.3. Using the f-tree $\mathcal{T}_1$ shown in Figure 4.3 left, and denoting the non-annotation f-tree nodes $\mathcal{A}, \mathcal{B}, \mathcal{C}$ and $\mathcal{D}$, we have

$$\begin{aligned}\mathcal{T}_1(Q(\mathbf{D})) = \langle\mathcal{B}:2\rangle\big(\langle\mathcal{A}:1\rangle\langle\underline{R}:r_{12}\rangle\big)\big(\langle\mathcal{C}:1\rangle\langle\underline{S}:s_{21}\rangle(\langle\mathcal{D}:2\rangle\langle\underline{T}:t_{12}\rangle \cup \langle\mathcal{D}:3\rangle\langle\underline{T}:t_{13}\rangle)\big) \cup \\ \langle\mathcal{B}:3\rangle\big(\langle\mathcal{A}:1\rangle\langle\underline{R}:r_{13}\rangle \cup \langle\mathcal{A}:2\rangle\langle\underline{R}:r_{23}\rangle\big)\times \\ \big(\langle\mathcal{C}:1\rangle\langle\underline{S}:t_{31}\rangle(\langle\mathcal{D}:2\rangle\langle\underline{T}:t_{12}\rangle \cup \langle\mathcal{D}:3\rangle\langle\underline{T}:t_{13}\rangle) \cup \\ \langle\mathcal{C}:2\rangle\langle\underline{S}:t_{32}\rangle\langle\mathcal{D}:3\rangle\langle\underline{T}:t_{23}\rangle\big).\end{aligned}$$

[1] By a Boolean query on an annotated database we mean a query that has no data attributes in the projection list. The annotation attributes are *always* in the projection list.

By dropping the data singletons and keeping the annotation singletons, we obtain an f-representation of the provenance $\varphi(\langle\rangle)$,

$$\langle r_{12}\rangle\langle s_{21}\rangle(\langle t_{12}\rangle \cup \langle t_{13}\rangle) \;\cup\; \big(\langle r_{13}\rangle \cup \langle r_{23}\rangle\big)\big(\langle s_{31}\rangle(\langle t_{12}\rangle \cup \langle t_{13}\rangle) \cup \langle s_{32}\rangle\langle t_{23}\rangle\big).$$

The corresponding factorisation of the provenance polynomial of $\langle\rangle$ is

$$r_{12}s_{21}(t_{12}+t_{13}) + (r_{13}+r_{23})(s_{31}(t_{12}+t_{13}) + s_{32}t_{23}).$$

$\square$

For non-Boolean queries, only the derivations that yield the tuple t contribute to the provenance of t, and the provenance of t is therefore obtained by dropping all data singletons from the result $\sigma_{\mathcal{P}=t}\hat{Q}$.

Since our machinery of f-trees is defined for conjunctive queries and not for selections with constants, we use an equivalent formulation that pushes the selection into the database. The result $\sigma_{\mathcal{P}=t}\hat{Q}$ is the same as that of the query

$$\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*} = \sigma_{\psi_{\mathcal{S}\setminus\mathcal{P}^*}}(R'_1 \times \cdots \times R'_n),$$

a restriction of $\hat{Q}$ to the attributes non-equivalent to $\mathcal{P}$, on the database

$$\mathbf{D}_t = \{\, \pi_{\mathcal{S}\setminus\mathcal{P}^*}(\sigma_{\mathcal{P}^*=t}(\mathbf{R})) \mid \mathbf{R} \in \mathbf{D}\},$$

where each relation from $\mathbf{D}$ is restricted to tuples agreeing with t on all attributes equivalent to $\mathcal{P}$ and projected to the attributes non-equivalent to $\mathcal{P}$.

The provenance of a tuple t in $Q(\mathbf{D})$ therefore equals

$$\varphi(t) = \pi_{\mathcal{I}}\sigma_{\mathcal{P}=t}\hat{Q}(\mathbf{D}) = \pi_{\mathcal{I}}\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t),$$

and f-representations $\varphi(t)$ can be obtained by dropping all data singletons from f-representations of $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t)$. Note that different tuples in the result $Q(\mathbf{D})$ lead to the same Boolean query $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$ and hence the same f-trees, only the database $\mathbf{D}_t$ is different.

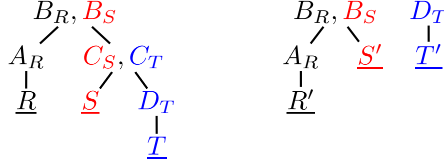


Figure 4.3: Left to right: f-tree $\mathcal{T}_1$ for the query Q from Example 4.2, and a provenance f-tree $\mathcal{T}_2$ for $Q_2 = \pi_{C_S} Q$, which is a forest of two trees.

Example 4.5. Consider the query $Q_2 = \pi_{C_S} Q_1$ and the provenance $\varphi(\langle C_S\!:\!2\rangle)$ of the result tuple $t = \langle C_S\!:\!2\rangle$. We first rewrite the query Q_2 to by dropping the head attributes and their equivalence classes: $(\hat{Q}_2)_{\mathcal{S}\setminus\mathcal{P}} = \sigma_{B_R=B_S}(R' \times S' \times T')$, where R' has schema $\{A_R, B_R\}$, S' has schema $\{B_S\}$ and T' has schema $\{D_T\}$. Then we construct the database $\mathbf{D}_{\langle C_S:2\rangle}$ by requiring $\sigma_{C_S=2}$ and projecting away C_S, obtaining relations $\mathbf{R}' = \mathbf{R}$, $\mathbf{S}' = \{\langle s_{32}, 3\rangle\}$ and $\mathbf{T}' = \{\langle t_{23}, 3\rangle\}$. The provenance $\varphi(\langle C_S\!:\!2\rangle)$ in $Q_2(\mathbf{D})$ is the projection $\pi_{\mathcal{I}}(\hat{Q}_2)_{\mathcal{S}\setminus\mathcal{P}}(\mathbf{D}_{\langle C_S:2\rangle})$,

$$\varphi(\langle C_S\!:\!2\rangle) = \langle \underline{R}\!:\!r_{13}\rangle\langle \underline{S}\!:\!s_{32}\rangle\langle \underline{T}\!:\!t_{23}\rangle \cup \langle \underline{R}\!:\!r_{23}\rangle\langle \underline{S}\!:\!s_{32}\rangle\langle \underline{T}\!:\!t_{23}\rangle.$$

An f-tree $\mathcal{T}_2$ of $(\hat{Q}_2)_{\mathcal{S}\setminus\mathcal{P}}$ is shown right in Figure 4.3. The f-representation of the annotated result $(\hat{Q}_2)_{\mathcal{S}\setminus\mathcal{P}}(\mathbf{D}_{\langle C_S:2\rangle})$ over $\mathcal{T}_2$ is

$$\langle \mathcal{B}\!:\!3\rangle(\langle \mathcal{A}\!:\!1\rangle\langle \underline{R}\!:\!r_{13}\rangle \cup \langle \mathcal{A}\!:\!2\rangle\langle \underline{R}\!:\!r_{23}\rangle)\langle \underline{S}\!:\!s_{32}\rangle\langle \mathcal{D}\!:\!3\rangle\langle \underline{T}\!:\!t_{23}\rangle$$

and by dropping the data values we get the f-representation of $\varphi(\langle C_S\!:\!2\rangle)$ over $\mathcal{T}_2$,

$$\mathcal{T}_2(\varphi(\langle C_S\!:\!2\rangle)) = (\langle \underline{R}\!:\!r_{13}\rangle \cup \langle \underline{R}\!:\!r_{23}\rangle)\langle \underline{S}\!:\!s_{32}\rangle\langle \underline{T}\!:\!t_{23}\rangle,$$

or the corresponding provenance polynomial $(r_{13} + r_{23})s_{32}t_{23}$. □

The above two cases form a complete approach for factorising provenance of tuples in the results of an arbitrary conjunctive query Q. The f-trees that can be used for factorisation are the f-trees valid for the equi-join $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$. We call this query the *provenance query of* Q and its f-trees the *provenance f-trees of* Q. Given a provenance f-tree $\mathcal{T}$ of Q, the f-representation obtained by dropping all data singletons from $\mathcal{T}(\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t))$ is denoted by

$\mathcal{T}(\varphi(t))$. The following result establishes the soundness of our approach.

Proposition 4.2. *For any conjunctive query Q, any tuple t in its result $Q(\mathbf{D})$, and any provenance f-tree $\mathcal{T}$ of Q, the f-representation $\mathcal{T}(\varphi(t))$ as defined above represents the provenance $\varphi(t)$.*

Similar to f-representations of query results, the f-representation $\mathcal{T}(\varphi(t))$ is unique up to commutativity of product and union and can be computed using the algorithm in Section 3.6. Asymptotic bounds on the numbers of singletons of each type in an f-representation of a relation over an f-tree, derived in Section 3.4, extend to f-representations of provenance over a provenance f-tree, since the latter is obtained from the former only by dropping data (non-annotation) singletons. We derive such bounds in the following section as a by-product of bounding another factorisation measure; readability.

4.3 Readability of Provenance

Besides minimum factorisation size, a further measure of interest for provenance is readability, which captures the minimum number of times that some identifier must occur in any factorisation. This measure has been previously defined for Boolean functions [GPR06] and our motivation draws on the close connection between f-representations of annotated relations and algebraic factorisations of Boolean functions. Functions of low readability are preferred over functions of high readability, since the former can be implemented with smaller logical circuits. As discussed in the next section, our characterisation of queries by the readability of their results sheds light on a key structural property of the queries that goes beyond compactness of query results. We next give a definition of readability tailored to f-representations.

Definition 4.3. An f-representation E is *read-k* if any value occurs in at most k singletons of E. The *readability* of a relation R is the smallest k such that R has a read-k f-representation. □

Example 4.6. The flat f-representation of provenance $\varphi(\langle\rangle)$, which is given in Example 4.3 as a polynomial, is read-4 since the singleton $\langle \underline{S} : s_{31} \rangle$ (and hence the value s_{31}) occurs four

times and no value occurs more than four times. Its f-representation over the f-tree $\mathcal{T}_1$ from Figure 4.3 is given in Example 4.4; it is read-2, since the values t_{12} and t_{13} occur twice and all other values only occur once. The readability of $\varphi(\langle\rangle)$ is 2 because it admits no read-1 f-representation. □

The main result of this section is the following characterisation of conjunctive queries based on the readability of provenance of their result tuples. For any query Q, let M be the maximum number of repeating relation symbols in Q.

Theorem 4.1. *For any conjunctive query Q, there is a rational number $r(Q)$ such that:*

- *For any database $\mathbf{D}$ and tuple $t \in Q(\mathbf{D})$, the readability of the provenance $\varphi(t)$ is at most $M \cdot |\mathbf{D}|^{r(Q)}$.*
- *For any provenance f-tree $\mathcal{T}$ of Q there exist arbitrarily large databases $\mathbf{D}$ and tuples $t \in Q(\mathbf{D})$ for which the f-representation $\mathcal{T}(\varphi(t))$ is at least read-$(|\mathbf{D}|/|Q|)^{r(Q)}$.*

Proof. The theorem is a restatement of Corollaries 4.1 and 4.2 proved later in this section. □

The parameter $r(Q)$ is called the *readability width* of Q. The case $r(Q) = 0$ precisely captures the *hierarchical queries* studied in other contexts (cf. Section 4.4), for other queries the readability width $r(Q)$ captures "how far" Q is from a hierarchical query similarly as various hypertree widths capture how far a query is from an acyclic query.

Our study of readability bounds follows the one on size bounds, with the difference that we now need only consider singletons for annotation attributes. The number of occurrences of any given singleton in any f-representation over an f-tree is quantified by Corollary 3.2. In particular, for a query result $Q(\mathbf{D})$ and an f-tree $\mathcal{T}$ of Q, an annotation attribute $\underline{R}$ and an identifier r of a tuple t, the number of occurrences of the singleton $\langle \underline{R} : r \rangle$ in $\mathcal{T}(Q(\mathbf{D}))$ is $|\pi_{\text{anc}(\underline{R})}\sigma_{\underline{R}=r}\mathcal{T}(Q(\mathbf{D}))|$. However, since the attribute $\underline{R}$ is a key for R, this is equal to $|\pi_{\text{anc}(\underline{R})}\sigma_{\mathcal{S}_R=t}\mathcal{T}(Q(\mathbf{D}))|$, where the condition $\mathcal{S}_R = t$ means that we assign to each attribute of R its value in tuple t. Just as the bounds for the number of singletons of type A were derived using the query $Q_{\text{path}(A)}$, which is Q restricted to path(A), the number of individual annotation singletons $\langle \underline{R} : r \rangle$ are derived using a query $Q_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}$: the query Q restricted to path$(\underline{R})$ but *without* the attribute classes with attributes of R, since the attributes of R

are already fixed by values in the tuple t. This is a key difference between our analysis of readability in case of annotated relations vs. size in case of standard relations.

Lemma 4.1. *For any annotated query result $Q(\mathbf{D})$ and an f-tree $\mathcal{T}$ of Q, the number of occurrences of any $\underline{R}$-singleton in the f-representation $\mathcal{T}(Q(\mathbf{D}))$ is at most $|\mathbf{D}|^{\rho^*(Q_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R})}$.*

Recall that the provenance $\varphi(t)$ of a tuple t in a query result $Q(\mathbf{D})$ can be factorised over any f-tree $\mathcal{T}$ of the provenance query $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$ of Q. The f-representation $\mathcal{T}(\varphi(t))$ of $\varphi(t)$ is obtained by dropping all data singletons from the f-representation $\mathcal{T}(\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t))$, where $\mathbf{D}_t$ is constructed by restricting to tuples agreeing with t on all equivalent attributes and then projecting them away, as described in Section 4.2. The number of occurrences of any annotation identifier in $\mathcal{T}(\varphi(t))$ is thus the same as in $\mathcal{T}(\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t))$, for which we can use Lemma 4.1.

Similarly to $s(\mathcal{T})$ and $s(Q)$, we define parameters $r(\mathcal{T})$ and $r(Q)$ that control the exponent of the number of occurrences of any identifier in a similar way as $s(\mathcal{T})$ and $s(Q)$ do for the overall size.

Definition 4.4. Let Q be a conjunctive query and let $Q' = \hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$ be its provenance query. For any provenance f-tree $\mathcal{T}$ of Q, define

$$r(\mathcal{T}) = \max\{\rho^*(Q'_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}) \mid R \in Q\}$$

to be the maximum possible $\rho^*(Q'_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R})$ over all head attributes A of Q, and

$$r(Q) = \min\{r(\mathcal{T}) \mid \mathcal{T} \text{ is a provenance f-tree of } Q\}$$

to be the minimum possible $r(\mathcal{T})$ over all provenance f-trees $\mathcal{T}$ of Q. □

Next we prove the analogous bounds for readability of provenance. An additional factor M arises because a given value can appear in the singletons of up to M different types.

Corollary 4.1 (Lemma 4.1)**.** *For any database $\mathbf{D}$, any tuple $t \in Q(\mathbf{D})$, and any provenance f-tree $\mathcal{T}$ of Q, the factorisation $\mathcal{T}(\varphi(t))$ is at most read-$M \cdot |\mathbf{D}|^{r(\mathcal{T})}$. The readability of $\varphi(t)$ is at most $M \cdot |\mathbf{D}|^{r(Q)}$.*

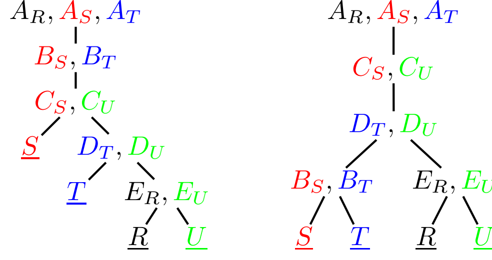


Figure 4.4: F-trees from Figure 3.2 extended with annotated attributes at leaves.

Example 4.7. Consider the Boolean query $\pi_\emptyset Q$ for Q from Example 4.2. The f-tree $\mathcal{T}_1$ shown left in Figure 4.3 is an f-tree of Q and hence a provenance f-tree of $\pi_\emptyset Q$. For relation R, all ancestor nodes of $\underline{R}$ contain attributes of R, so the query $Q_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}$ is empty and $\rho^*(Q_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}) = 0$. This is also the case for the relation S. For relation T we have $Q_{\text{path}(\underline{T})\setminus\mathcal{S}^*_T} = \sigma_{B_R=B_S}(\pi_{B_R}R \times \pi_{B_S}S)$. The hypergraph of $Q_{\text{path}(\underline{T})\setminus\mathcal{S}^*_T}$ has a single node and two edges of size 1, so $\rho^*(Q_{\text{path}(\underline{T})\setminus\mathcal{S}^*_T}) = 1$. It follows that each annotation singleton of type $\underline{T}$ appears at most $|\mathbf{D}|$ times in any f-representation over $\mathcal{T}_1$, while all other annotation singletons appear at most once. The provenance of $\langle\rangle$ in $\pi_\emptyset Q(\mathbf{D})$ is therefore at most read-$|\mathbf{D}|$ for any database $\mathbf{D}$. □

Example 4.8. We now repeat Example 3.15, for readability bounds instead of size bounds.

Consider the Boolean query $\pi_\emptyset Q_2$ for Q_2 in Example 3.15, and the f-tree $\mathcal{T}_3$ of Q_2 extended with annotation attributes at leaves (shown left in Figure 4.4). The query Q_R has attributes $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$, its hypergraph is a triangle and hence $\rho^*(Q_R) = 3/2$. For the other relations, the fractional edge cover number of their queries is less and hence $r(\mathcal{T}_3) = 3/2$.

For the right f-tree $\mathcal{T}_4$ in Figure 4.4, each of Q_R, Q_S, Q_T and Q_U can be covered by a single relation and hence $r(\mathcal{T}_4) = 1$. This is the smallest possible value for an f-tree of Q_2, or equivalently, for a provenance f-tree of $\pi_\emptyset Q_2$, so $r(\pi_\emptyset Q_2) = 1$. We deduce that the provenance of $\langle\rangle$ in $\pi_\emptyset Q_2(\mathbf{D})$ is at most read-$|\mathbf{D}|$ for any database $\mathbf{D}$.

Compare these bounds for the readability of provenance of Boolean query results bounds on the representation sizes of the corresponding equi-joins obtained in Example 3.15. The f-representation $\mathcal{T}_4(\mathbf{D})$ has size at most $|\mathbf{D}|^{5/3}$, while a naive argument for readability, stating that each of the $|\mathbf{D}|$ identifiers can appear at most $|\mathbf{D}|$ times, gives the suboptimal

bound $|\mathbf{D}|^2$. □

The lower bounds can also be adapted correspondingly. First we prove a lower bound on the number of occurrences of some singleton of a given type, and then we deduce lower bounds on readability of f-representations with respect to f-trees.

Lemma 4.2. *For any query Q, f-tree $\mathcal{T}$ of Q, and annotation attribute $\underline{R}$ in Q, there exist arbitrarily large annotated databases $\mathbf{D}$ such that the number of occurrences of some $\underline{R}$-singleton in the f-representation $\mathcal{T}(Q(\mathbf{D}))$ of the annotated query result $Q(\mathbf{D})$ is at least $(|\mathbf{D}|/|Q|)^{\rho^*(Q_{\text{path}(\underline{R})} \setminus \mathcal{S}^*_{\underline{R}})}$.*

Corollary 4.2 (Lemma 4.2)**.** *For any provenance f-tree $\mathcal{T}$ of Q, there exist arbitrarily large databases $\mathbf{D}$ and tuple $t \in Q(\mathbf{D})$ such that the f-representation $\mathcal{T}(\varphi(t))$ is at least read-$(|\mathbf{D}|/|Q|)^{r(\mathcal{T})}$, which is at least read-$(|\mathbf{D}|/|Q|)^{r(Q)}$.*

4.4 Dichotomy for Readability

In this section we present a dichotomy of conjunctive queries: we syntactically divide the queries for which the result tuple provenance has bounded readability and those for which the readability can be unbounded. Remarkably, the queries whose result tuples have provenance of bounded readability are precisely the *hierarchical queries* that stand out in other diverse settings.

Definition 4.5. [DS07] A conjunctive query is *hierarchical*, if for any two attributes A and B that are not equivalent to any head attribute, we have either $\text{rel}(A) \subseteq \text{rel}(B)$, or $\text{rel}(A) \supset \text{rel}(B)$, or $\text{rel}(A) \cap \text{rel}(B) = \emptyset$. □

We next present this property in more detail. At the outset is the observation that there are queries for which the readability width $r(Q)$ is zero and hence the upper bound on readability is constant. Unlike the size parameter $s(Q)$, which is at least 1 for non-empty queries, $r(Q) = 0$ for a wide class of queries.

Proposition 4.3. *A conjunctive query Q is hierarchical iff its readability width $r(Q)$ is zero.*

Proof. Since $r(\mathcal{T})$ is always nonnegative, $r(Q) = 0$ iff $r(\mathcal{T}) = 0$ for some f-tree $\mathcal{T}$. This happens when $\rho^*(Q_{\text{path}(\underline{R})\setminus\mathcal{S}_R^*}) = 0$ for all relations R, that is, $Q_{\text{path}(\underline{R})\setminus\mathcal{S}_R^*}$ is empty for all R. In other words, for each relation R, each node in $\text{path}(\underline{R})$ contains an attribute of R. We show that Q has an f-tree with this property if and only if it is hierarchical.

Let Q be a hierarchical query and consider the set of equivalence classes containing no head attributes. They are exactly the equivalence classes of the provenance query $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$ of Q; call them nodes. For any two such nodes $\mathcal{A}$ and $\mathcal{B}$, $\text{rel}(\mathcal{A})$ and $\text{rel}(\mathcal{B})$ are either disjoint or one is contained in the other. For a set of nodes, define $\text{tree}(S)$ recursively as follows. Find a set of nodes $\mathcal{A}_1, \ldots, \mathcal{A}_k \in S$ that have disjoint $\text{rel}(\mathcal{A}_i)$ and all other nodes $\mathcal{B}$ are partitioned into sets $S_i = \{\mathcal{B} : \text{rel}(\mathcal{B}) \subseteq \text{rel}(\mathcal{A}_i)\}$. Then construct $\text{tree}(S)$ as a forest consisting of the trees with root $\mathcal{A}_i$ and a forest of subtrees $\text{tree}(S_i)$. Finally, let $\mathcal{T}$ be the tree $\text{tree}(S)$ where S is the set of all nodes of $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$.

In this tree, if $\mathcal{A}$ and $\mathcal{B}$ are in sibling subtrees, then $\text{rel}(\mathcal{A})$ and $\text{rel}(\mathcal{B})$ are disjoint, so the path constraint is satisfied in $\mathcal{T}$ and $\mathcal{T}$ is an f-tree of the provenance query $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$ Moreover, for any relation, the annotation attribute $\underline{R}$ is a node by itself, and there are no empty nodes, so the node $\{\underline{R}\}$ is a leaf of $\mathcal{T}$. Also, if $\mathcal{A}$ is an ancestor of $\mathcal{B}$, then $\text{rel}(\mathcal{A}) \supseteq \text{rel}(\mathcal{B})$. Therefore, for all nodes $\mathcal{A}$ in $\text{path}(\{\underline{R}\})$ we have $\text{rel}(\mathcal{A}) \supseteq \text{rel}(\{\underline{R}\}) = \{R\}$.

Conversely, suppose that $\mathcal{T}$ is an f-tree for Q such that each node in $\text{path}(\underline{R})$ contains an attribute of R. For any two attribute classes $\mathcal{A}$ and $\mathcal{B}$ of Q, either one is an ancestor of the other, or they appear in sibling subtrees. In the latter case, $\text{rel}(\mathcal{A})$ and $\text{rel}(\mathcal{B})$ are disjoint. In the former case, suppose wlog that $\mathcal{A}$ is an ancestor of $\mathcal{B}$. Any relation $R \in \text{rel}(\mathcal{B})$ must have its annotation attribute $\underline{R}$ in a leaf under $\mathcal{B}$, and thus also $\mathcal{A}$ is on the path from $\underline{R}$ to the root, i.e. $R \in \text{rel}(\mathcal{A})$. This shows that $\text{rel}(\mathcal{B}) \subseteq \text{rel}(\mathcal{A})$ and completes the proof. □

For any non-hierarchical query Q, we have $r(Q) > 0$. However, $r(Q) = r(\mathcal{T}) = \rho^*(Q_{P_R})$ for some f-tree $\mathcal{T}$ and equi-join Q_{P_R}, so $r(Q) > 0$ implies $r(Q) \geq 1$. This gap in readability width creates a gap in the possible readability bounds for queries.

Theorem 4.2. *Let Q be a conjunctive query.*

1. *If Q is hierarchical, then the readability of the provenance $\varphi(t)$ for any tuple $t \in Q(\mathbf{D})$ and database $\mathbf{D}$ is bounded by a constant.*

2. If Q is non-hierarchical, for any provenance f-tree $\mathcal{T}$ of Q there exist arbitrarily large databases $\mathbf{D}$ and tuple $t \in Q(\mathbf{D})$ such that $\mathcal{T}(\varphi(t))$ is read-$\Omega(|\mathbf{D}|)$.

The lower bound states that no f-tree defines factorisations of sublinear readability for all databases. For non-repeating queries, we can strengthen the above dichotomy to readability *irrespective* of f-trees. We first state the readability for the simplest non-hierarchical query

$$Q_{nh} = \pi_{\emptyset}\sigma_{A_R=A_S \wedge B_S=B_T}(R \times S \times T)$$

where the relations R, S, T are over schemas $\{A_R\}$, $\{A_S, B_S\}$, and $\{B_T\}$ respectively. Consider the relation instances $\mathbf{R} = [N]$, $\mathbf{T} = [N]$ and $\mathbf{S} = [N] \times [N]$ annotated by identifiers r_i, t_j and s_{ij} respectively, with $1 \leq i, j \leq N$. The flat f-representation of the provenance of $\langle\rangle$ in Q_{nh}'s result is then

$$\Phi_N = \bigcup_{i,j=1}^{N} \langle r_i \rangle \langle s_{ij} \rangle \langle t_j \rangle.$$

Lemma 4.3. *The relation Φ_N has readability $\frac{N}{2} + O(1)$.*

Proof. We first show that Φ_N has readability at least $\frac{N}{2}$. Let E be any f-representation equivalent to Φ_N and consider its parse tree, where adjacent union nodes are aggregated into a single node, and any expressions representing an empty relation have been removed. If we expand E into tuples by the distributivity of product over sum, we must obtain the expression $\Phi_N = \bigcup_{i,j=1}^{N} \langle r_i \rangle \langle s_{ij} \rangle \langle t_j \rangle$. Therefore, there must be exactly one occurrence of $\langle s_{ij} \rangle$ in the parse tree, and it can have at most two products on its path to the root. If there are two products, E is of the form

$$(((\langle s_{ij} \rangle \cup E_1)(\langle r_i \rangle \cup E_2) \cup E_3)(\langle t_j \rangle \cup E_4) \cup E_5 \qquad \text{or}$$
$$(((\langle s_{ij} \rangle \cup E_1)(\langle t_j \rangle \cup E_2) \cup E_3)(\langle r_i \rangle \cup E_4) \cup E_5,$$

but then necessarily, E_1, E_2 and E_4 are empty (because if any two of $\langle r_i \rangle$, $\langle s_{ij} \rangle$, or $\langle t_j \rangle$ appear in a tuple of E, that tuple must be $\langle r_i \rangle \langle s_{ij} \rangle \langle t_j \rangle$). Similarly, if there is one multiplication, E is of the form

$$(\langle s_{ij} \rangle \cup E_1)(((\langle r_i \rangle \cup E_2)(\langle t_j \rangle \cup E_3) \cup E_4) \cup E_5,$$

but then necessarily all of E_1, E_2, E_3 and E_4 are empty. In any case, $\langle s_{ij}\rangle$ appears in one of the forms

$$(\langle s_{ij}\rangle\langle r_i\rangle \cup \dots)\langle t_j\rangle \cup \dots \quad \text{or} \quad (\langle s_{ij}\rangle\langle t_j\rangle \cup \dots)\langle r_i\rangle \cup \dots .$$

Therefore, each $\langle s_{ij}\rangle$ appears directly in a product with an r- or t-identifier but no other s-identifiers. Since there are N^2 of the s-identifiers, and $2N$ different r- and t-identifiers, at least one of the latter occurs at least $\frac{N}{2}$ times in the expression E.

To complete the proof, it is enough to exhibit a read-$(\frac{N}{2} + O(1))$ factorisation of R_N. Defining A_N and B_N as

$$\begin{aligned}
A_N &= \bigcup_{i=1}^{N} \bigcup_{j=0}^{\lfloor N/2\rfloor-1} \langle r_i\rangle\langle s_{i(i+j)}\rangle\langle t_{i+j}\rangle \\
&= \bigcup_{i=1}^{N} \langle r_i\rangle \times \left(\sum_{j=0}^{\lfloor N/2\rfloor-1} \langle s_{i(i+j)}\rangle\langle t_{i+j}\rangle\right), && \text{(A)} \\
B_N &= \bigcup_{i=1}^{N} \bigcup_{j=\lfloor N/2\rfloor}^{N-1} \langle r_i\rangle\langle s_{i(i+j)}\rangle\langle t_{i+j}\rangle \\
&= \bigcup_{i=1}^{N} \sum_{j=\lfloor N/2\rfloor}^{N-1} \langle r_{i-j}\rangle\langle s_{(i-j)i}\rangle\langle t_i\rangle \\
&= \bigcup_{i=1}^{N} \left(\bigcup_{j=\lfloor N/2\rfloor}^{N-1} \langle r_{i-j}\rangle\langle s_{(i-j)i}\rangle\right) \times \langle t_i\rangle, && \text{(B)}
\end{aligned}$$

where all indices are considered modulo N, we get $R_N = A_N \cup B_N$. Each $\langle s_{ij}\rangle$ occurs once either in expression A or expression B. Each $\langle r_i\rangle$ occurs once in A and $\lceil \frac{N}{2}\rceil$ times in B, and each $\langle t_j\rangle$ occurs $\lfloor \frac{N}{2}\rfloor$ times in A and once in B. Both A and B are valid f-representations for A_N and B_N respectively, so writing R_N as the union of expressions A and B, we get a read-$(\lceil \frac{N}{2}\rceil + 1)$ factorisation. □

Lemma 4.3 can be generalised to a non-symmetric case where relation T has size M, which may be different from N. The f-representation becomes $\bigcup_{i=1}^{N} \bigcup_{j=1}^{M} \langle r_i\rangle\langle s_{ij}\rangle\langle t_j\rangle$ and readability $\frac{NM}{N+M} + O(1)$.

The main dichotomy result that establishes a gap for the readability of non-repeating conjunctive queries, irrespective of f-trees, is given next.

Theorem 4.3. *Let Q be a non-repeating conjunctive query.*

1. *If Q is hierarchical, the readability of the provenance $\varphi(t)$ for any tuple $t \in Q(\mathbf{D})$ and database $\mathbf{D}$ is 1.*

2. *If Q is non-hierarchical, there exist arbitrarily large databases $\mathbf{D}$ and tuple $t \in Q(\mathbf{D})$ such that the readability of the provenance $\varphi(t)$ is $\Omega(\sqrt{|\mathbf{D}|})$.*

Proof. If Q is hierarchical then $r(Q) = 0$ by Proposition 4.3, and by Corollary 4.1, the readability of $Q(\mathbf{D})$ is at most $M \cdot |\mathbf{D}|^0 = M = 1$, since Q is non-repeating.

If Q is not hierarchical, there exist attribute classes $\mathcal{A}$ and $\mathcal{B}$ such that $\text{rel}(\mathcal{A}) \not\subseteq \text{rel}(\mathcal{B})$, $\text{rel}(\mathcal{B}) \not\subseteq \text{rel}(\mathcal{A})$ and $\text{rel}(\mathcal{A}) \cap \text{rel}(\mathcal{B}) \neq \emptyset$. Thus there must exist a relation S with attributes from $\mathcal{A}$ and $\mathcal{B}$, a relation R with attributes from $\mathcal{A}$ but not $\mathcal{B}$, and a relation T with attributes from $\mathcal{B}$ but not $\mathcal{A}$.

Fix any positive integer N. Consider a database instance $\mathbf{D}$ in which the domains of attributes in $\mathcal{A}$ and $\mathcal{B}$ are $\{1, \ldots, N\}$ and the domains of all other attributes are $\{1\}$. For each relation of Q, let its interpretation in $\mathbf{D}$ be the set of all possible tuples with the above domains, which respect the equivalence classes of attributes. We annotate the tuple in R with $\mathcal{A}$-value i by $\langle r_i \rangle$, tuple in T with $\mathcal{A}$-value j by $\langle t_j \rangle$, and tuple in S with $\mathcal{A}$-value i and $\mathcal{B}$-value j by $\langle s_{ij} \rangle$. All relations contain N^2, N, or only one tuple, depending on whether they contain attributes from $\mathcal{A}$ or $\mathcal{B}$, both or none. Thus, $|\mathbf{D}| = \Theta(N^2)$.

Consider any f-representation E of $Q(\mathbf{D})$, and construct the f-representation E' by restricting E to the identifiers from R, S and T. (That is, from each product remove the factors whose schema does not contain any of $\underline{R}$, $\underline{S}$ or $\underline{T}$.) The f-representation E' now represents the relation R_N as defined in Lemma 4.3. By Lemma 4.3, this R_N has readability $\Omega(N)$, so E' contains at $\Omega(N)$ occurrences of some identifier, and hence also E contains $\Omega(N)$ occurrences of that identifier. Since this holds for any f-representation E of $Q(\mathbf{D})$, $Q(\mathbf{D})$ has readability $\Omega(N) = \Omega(\sqrt{|\mathbf{D}|})$. □

4.5 Related Work

Annotations for relational databases were first introduced to capture incomplete information [IL84]. A variety of annotation semantics have since been proposed [CCT09], many of them can be expressed in the unifying formalism of provenance polynomials that capture the algebra of provenance propagation through positive relational algebra queries [GKT07].

Provenance Compression and Minimisation

The problem of large provenance data has been observed and approached in broader context than annotated relational databases and several systems for efficient storage and querying of provenance have been proposed [BCC06, CJR08]. These systems do not focus on managing the provenance of large conjunctive queries; their compression exploits the fact that different data items may have shared (parts of) provenance.

In the context of provenance polynomials, recent work [ADMT12] investigates questions of finding an equivalent query with smallest possible provenance. Different query expressions equivalent under relational set semantics can produce different provenance polynomials for their tuples, since the notion of query equivalence is weaker under the Boolean semiring than under the semiring $\mathbb{N}[X]$ of provenance polynomials [Gre09] (cf. Remark 4.1). This direction is orthogonal to our work: we consider finding the smallest representation of a given fixed provenance polynomial that arises from a given query expression. There is potential for combining these approaches and minimising provenance with respect to both dimensions.

Another possible approach to minimising provenance is a non-exact one, where we require a small (or smallest) provenance factorisation that is in some sense approximately equivalent to the target provenance. Approximations of provenance polynomials has been considered over the semiring of Boolean expressions in the context of probabilistic databases, where the lower and upper bound approximations with readability one are sought [FO11]. The key contributions are equivalences of syntactic and model-theoretic characterisations of optimal bounds for provenance polynomials of results to positive relational algebra queries, as well as algorithms to enumerate such bounds with polynomial delay.

Hierarchical Queries

Our characterisation of query readability revolves around how far the query is from a hierarchical query. This is quantified by the readability width $r(Q)$ of the query, and is similar in spirit to the parameters $s(Q)$ and $s^{\uparrow}(Q)$ introduced in Chapter 3, and existing width measures that capture the complexity of conjunctive queries, such as (generalised) hypertree

with and fractional hypertree width [GLS01, GM06].

The hierarchical property plays a central role in studies with seemingly disparate focus, including the present one, probabilistic databases, parallel query evaluation, and streamed query evaluation. Theorem 4.3 draws on earlier work on probabilistic databases [OH08] that uses provenance polynomials over the Booleans (under the semiring $\mathbb{B}[X]$ instead of $\mathbb{N}[X]$) to describe probabilistic events. Read-once events are useful since their exact probability can be computed in polynomial time [SORK11], while for read-m events with $m > 3$, probability computation is #P-hard [Vad01]. In our case, however, a readability that is polynomial in the sizes of the input database and query is acceptable, since it means that the size of the f-representation of the query result is polynomial, too.

The hierarchical property also divides queries that can be evaluated in one step from those that cannot in the finite cursor machine model of computation [GGL+09]. In this model, queries are evaluated by first sorting each relation, followed by one pass over each relation. Furthermore, in the Massively Parallel computation model, any conjunctive query that can be evaluated (under bag semantics) with one synchronisation step is hierarchical [KS11].

4.6 Conclusion and Directions for Future Work

This chapter extends the idea of succinct factorised representation from relational data to its provenance, using the language of provenance polynomials. Regular nesting structures for factorisations of query results, as inferred from the query syntax and as defined by f-trees, can also be extended to factorisations of provenance polynomials, and asymptotic bounds on factorisation sizes carry over as well. We also derive asymptotic bounds for the readability of provenance polynomials, a related measure that is of interest in the context of probabilistic databases, where a special status is enjoyed by queries whose result provenance has readability equal to 1. Among non-repeating conjunctive queries we identify that queries with read-1 provenance are exactly the hierarchical queries as used in other contexts, and that non-hierarchical queries have provenance of unbounded readability. A natural direction for future research is to extend this readability characterisation to repeating conjunctive

queries, and to establish further lower bounds for readability.

As outlined in the previous section, the problem of provenance minimisation by factorisation can be generalised by enlarging the search space in two ways: (1) considering provenance polynomials of all queries equivalent under set semantics, or (2) considering provenance polynomials of all conjunctive queries but searching for a good approximation of the original polynomial under some metric.

Beyond Conjunctive Queries

A natural direction for further research is extending the work on readability of query results beyond conjunctive queries. The readability results become more involved even for a simple extension such as a simple disequality join. As an example, consider a Boolean projection of a disequality join $\pi_{\emptyset}\sigma_{A \neq B}(R \times S)$, where A and B attributes of R and S respectively. If the relations R and S are of size N, the provenance of the only tuple in the result of this query is $\varphi(\langle\rangle) = C_N := \bigcup_{i,j=1; i \neq j}^{N} \langle r_i \rangle \langle s_j \rangle$.

Proposition 4.4. *The readability of C_N is $\Omega(\frac{\log N}{\log \log N})$ and $O(\log N)$.*

If $i \neq j$ is replaced by $i \leq j$, the lower and upper bounds on readability still hold and we obtain an inequality query. In the case of Boolean factorisation a lower bound of $\sqrt{\frac{\log N}{\log \log N}}$ on readability is already known [GPR06].

Chapter 5

FDB: A Factorised Database Engine

This chapter introduces FDB, an in-memory relational query engine that presents relations to the user at the logical layer but uses factorisations to represent them at the physical layer. At the outset of this work lies the observation that results of conjunctive queries admit factorised representations asymptotically smaller than the results themselves (Section 3.4), and can be computed in asymptotically less time (Section 3.6). However, FDB can not only compute factorised query results from flat relational input, but can evaluate further queries directly on factorised input. This chapter develops techniques for evaluation of queries with selection, projection, joins, aggregation and group-by, order-by and limit clauses on factorised relations.

Due to the succinctness of factorised representations, FDB can achieve a reduction of storage requirements by several orders of magnitude against relational systems. Additionally, novel techniques for evaluation of queries directly on factorised representations, without first expanding to the represented flat relations, allow a similar gap in time performance. New optimisation techniques are also needed to ensure that intermediate and final results are factorised as succinctly as possible. We illustrate the basic principles and challenges in query evaluation on factorised relations by examples.

Example 5.1. Consider a database of a grocery retailer with delivery orders, stock avail-

ability at different locations, availability of dispatcher units for each location, and external suppliers with items and locations they supply to (Figure 5.1). The join query $Q_1 = \text{Order} \bowtie_{\text{item}} \text{Store} \bowtie_{\text{location}} \text{Disp}$ returns all orders with their respective items, possible locations to retrieve them from, and dispatchers available to deliver them:

$Q_1 = \text{Order} \bowtie_{\text{item}} \text{Store} \bowtie_{\text{location}} \text{Disp}$

oid	item	location	dispatcher
01	Milk	Istanbul	Adnan
01	Milk	Istanbul	Yasemin
01	Milk	Izmir	Adnan
01	Milk	Antalya	Volkan
...			

This query result can be factorised over the f-tree $\mathcal{T}_1$, shown in Figure 5.2 top left, which reflects the join dependencies induced by the query. For each item, we construct a union of its orders and a union of its possible locations with dispatchers, and output a product of the two unions and a singleton with that item. While the flat result has 48 singletons (4 for each of the 12 tuples), the factorisation over $\mathcal{T}_1$ has only 23:

$$\begin{aligned}
&\langle \text{Milk} \rangle \times \langle 01 \rangle \times (\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \cup \\
&\qquad \langle \text{Izmir} \rangle \times \langle \text{Adnan} \rangle \cup \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle) \cup \\
&\langle \text{Cheese} \rangle \times (\langle 01 \rangle \cup \langle 03 \rangle) \times (\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \cup \\
&\qquad \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle) \cup \\
&\langle \text{Melon} \rangle \times (\langle 02 \rangle \cup \langle 03 \rangle) \times \langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle).
\end{aligned}$$

Similarly, an f-representation of the result of $Q_2 = \text{Produce} \bowtie_{\text{supplier}} \text{Serve}$ over the f-tree $\mathcal{T}_2$ (Figure 5.2 top centre) is:

$$\begin{aligned}
&\langle \text{Guney} \rangle \times (\langle \text{Milk} \rangle \cup \langle \text{Cheese} \rangle) \times \langle \text{Antalya} \rangle \cup \\
&\langle \text{Dikici} \rangle \times \langle \text{Milk} \rangle \times (\langle \text{Istanbul} \rangle \cup \langle \text{Izmir} \rangle \cup \langle \text{Antalya} \rangle) \cup \\
&\langle \text{Byzantium} \rangle \times \langle \text{Melon} \rangle \times \langle \text{Istanbul} \rangle.
\end{aligned}$$

Orders	
oid	item
01	Milk
01	Cheese
02	Melon
03	Cheese
03	Melon

Store	
location	item
Istanbul	Milk
Istanbul	Cheese
Istanbul	Melon
Izmir	Milk
Antalya	Milk
Antalya	Cheese

Disp	
dispatcher	location
Adnan	Istanbul
Adnan	Izmir
Yasemin	Istanbul
Volkan	Antalya

Produce	
supplier	item
Guney	Milk
Guney	Cheese
Dikici	Milk
Byzantium	Melon

Serve	
supplier	location
Guney	Antalya
Dikici	Istanbul
Dikici	Izmir
Dikici	Antalya
Byzantium	Istanbul

Figure 5.1: An example database for a grocery retailer.

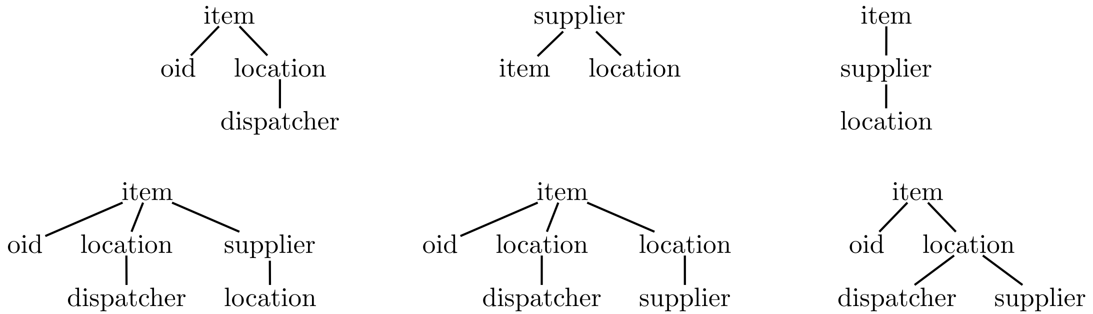


Figure 5.2: Examples of factorisation trees. Top row from left to right: $\mathcal{T}_1$ for the result of query Q_1; $\mathcal{T}_2$ and $\mathcal{T}_3$ for the result of Q_2. Bottom row from left to right: $\mathcal{T}_4$ is obtained after joining $\mathcal{T}_1$ and $\mathcal{T}_3$ on item, $\mathcal{T}_5$ is $\mathcal{T}_4$ after swapping location and supplier, and $\mathcal{T}_6$ is $\mathcal{T}_5$ after joining on location.

Consider now the query $Q_1 \bowtie_{\text{location,item}} Q_2$, finding possible suppliers of ordered items, in a scenario where the results of Q_1 and Q_2 are already computed in factorised form over the f-trees $\mathcal{T}_1$ and $\mathcal{T}_2$. Joining these factorisations on the attributes location and item is not immediate, since tuples with given values for location and item are not represented separately in the factorisation over $\mathcal{T}_2$. If we restructure the factorisation of Q_2's result to follow the f-tree $\mathcal{T}_3$ (Figure 5.2 top right) so that tuples are grouped by item first, we obtain

$$\begin{aligned}
&\langle \text{Milk} \rangle \times (\langle \text{Guney} \rangle \times \langle \text{Antalya} \rangle \cup \\
&\qquad\qquad \langle \text{Dikici} \rangle \times (\langle \text{Istanbul} \rangle \cup \langle \text{Izmir} \rangle \cup \langle \text{Antalya} \rangle)) \\
&\langle \text{Cheese} \rangle \times \langle \text{Guney} \rangle \times \langle \text{Antalya} \rangle \cup \\
&\langle \text{Melon} \rangle \times \langle \text{Byzantium} \rangle \times \langle \text{Istanbul} \rangle,
\end{aligned}$$

which can be readily joined with the factorisation over $\mathcal{T}_1$ on the attribute item, since both factorisations have items as topmost values, and tuples with a given value for item are

represented by separate subexpressions of the factorisations. It suffices to find all pairs of subexpressions with equal items from both factorisations, and return a union of their products. The resulting factorisation of the join on item follows the f-tree $\mathcal{T}_4$ (Figure 5.2 bottom left), where we simply merged the roots of the two f-trees. An excerpt of this factorisation is:

$$
\begin{aligned}
&\langle \text{Milk} \rangle \times \langle 01 \rangle \times (\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \cup \\
&\qquad\qquad\qquad \langle \text{Izmir} \rangle \times \langle \text{Adnan} \rangle \cup \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle) \\
&\qquad \times (\langle \text{Guney} \rangle \times \langle \text{Antalya} \rangle \cup \\
&\qquad\quad \langle \text{Dikici} \rangle \times (\langle \text{Istanbul} \rangle \cup \langle \text{Izmir} \rangle \cup \langle \text{Antalya} \rangle)) \cup \\
&\langle \text{Cheese} \rangle \times (\dots) \times (\dots) \times (\dots) \cup \dots
\end{aligned}
$$

To perform the second join condition on location, we first rearrange for each item the subexpression for suppliers and locations, so that it is grouped by locations as opposed to suppliers. The subexpression for dispatchers and locations stays intact, there is no need to expand the product. In the f-tree of the factorisation, this amounts to swapping supplier and location, resulting in the f-tree $\mathcal{T}_5$ (Figure 5.2 bottom centre). The join on location can now be performed between the possible locations of each item. The obtained factorisation follows the schema $\mathcal{T}_6$ (Figure 5.2 bottom right).

$$
\begin{aligned}
&\langle \text{Milk} \rangle \times \langle 01 \rangle \times (\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \times \langle \text{Dikici} \rangle \cup \\
&\qquad\qquad\qquad \langle \text{Izmir} \rangle \times \langle \text{Adnan} \rangle \times \langle \text{Dikici} \rangle \cup \\
&\qquad\qquad\qquad \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle \times (\langle \text{Dikici} \rangle \cup \langle \text{Guney} \rangle)) \cup \\
&\langle \text{Cheese} \rangle \times (\dots) \times (\dots) \cup \dots
\end{aligned}
$$

□

Example 5.1 highlights the challenges involved in the use of factorised representations for evaluating queries with joins. Firstly, a query result may have different (albeit equivalent) factorised representations whose sizes can differ substantially (by an exponential factor, as we have shown in Chapter 3). Both in the case of a flat or factorised input, we seek f-trees

that define succinct representations of the output: such f-trees can be statically derived from the query and the input schema, independently of the database content.

Secondly, we would like to compute the factorised result as efficiently as possible. This means that we must avoid the computation of intermediate results in flat form. The FDB query engine has operators for selection, projection and product that operate directly on factorisations without expanding the product, and use time at most quasilinear in the sizes of their input and output. However, some of the operator are only applicable to factorisations with selected nesting structures in order to achieve such efficiency. Therefore, in addition to the standard query operators, the search space for a good query and factorisation plan, or f-plan for short, needs to consider additional operators for restructuring schemas and factorisations. We propose two such operators: a swap operator, which exchanges a given child with its parent in an f-tree, and a push-up operator, which moves an entire sub-tree up in the f-tree. For instance, the swap operator is used to transform the f-tree $\mathcal{T}_2$ into $\mathcal{T}_3$ in Figure 5.2, enabling an efficient selection operator that subsequently merges the item nodes in the f-trees $\mathcal{T}_1$ and $\mathcal{T}_3$ and creates the f-tree $\mathcal{T}_4$. The transformation of $\mathcal{T}_4$ into $\mathcal{T}_5$, which corresponds to a join on location, needs a swap of supplier and location and then a merge of the two location nodes yields the final f-tree $\mathcal{T}_6$.

With the additional choice of different equivalent factorised representations for the same relation, query optimisation has to consider two objectives: minimising the cost of computing a factorised query result and minimising the size of the output factorisation. These objectives may be in conflict with each other as achieving a smaller factorisation may require more restructuring.

Aggregation and Ordering Queries

In addition to queries with joins, selections and projections, factorisation can benefit aggregate computation. For instance, counting tuples of a relation factorised as a union of products of relations can be expressed as a sum of multiplications of cardinalities of those latter relations. Further speedup is achieved by evaluating aggregation functions as sequences of repeated partial aggregations on factorised data, possibly intertwined with restructuring of the factorisation. We illustrate these techniques and the arising challenges in the following

Orders		
customer	date	pizza
Mario	Monday	Capricciosa
Mario	Tuesday	Margherita
Pietro	Friday	Hawaii
Lucia	Friday	Hawaii
Mario	Friday	Capricciosa

Pizzas	
pizza	item
Margherita	base
Capricciosa	base
Capricciosa	ham
Capricciosa	mushrooms
Hawaii	base
Hawaii	ham
Hawaii	pineapple

Items	
item	price
base	6
ham	1
mushrooms	1
pineapple	2

Figure 5.3: An example pizzeria database.

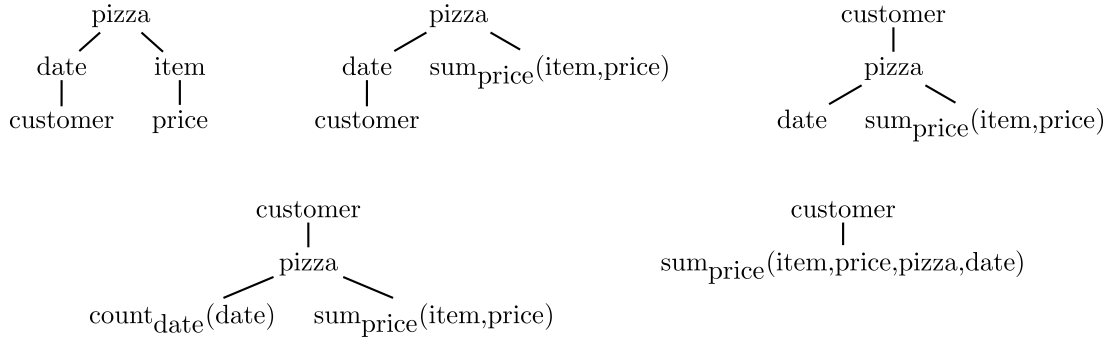


Figure 5.4: Factorisation trees used Example 5.2. Top row from left to right: f-tree $\mathcal{T}_1$ used to factorise the join query Sales, f-tree $\mathcal{T}_2$ obtained after summing price over the subtree with nodes item and price, and f-tree $\mathcal{T}_3$ after swapping customer twice. Bottom row from left to right: $\mathcal{T}_4$ which is $\mathcal{T}_3$ after aggregation count over the attribute date, and $\mathcal{T}_5$ obtained from $\mathcal{T}_4$ by evaluating $\text{sum}_{\text{price}}$ over the subtree rooted at pizza.

example.

Example 5.2. Figure 5.3 shows a database with pizzas on offer, prices of toppings, and pizza orders by date. The join query Sales = Orders $\bowtie$ Pizzas $\bowtie$ Items that computes the itemised list of all sales for all customers can be factorised using the f-tree $\mathcal{T}_1$ (Figure 5.4 top left), the resulting factorisation groups by pizza and represents the orders separately from items and prices:

$$
\begin{aligned}
&\langle \text{Capricciosa} \rangle \times (\langle \text{Monday} \rangle \times \langle \text{Mario} \rangle \cup \\
&\qquad \langle \text{Friday} \rangle \times \langle \text{Mario} \rangle) \\
&\qquad \times (\langle \text{base} \rangle \times \langle 6 \rangle \cup \\
&\qquad \langle \text{ham} \rangle \times \langle 1 \rangle \cup \\
&\qquad \langle \text{mushrooms} \rangle \times \langle 1 \rangle) \cup \\
&\langle \text{Hawaii} \rangle \times \langle \text{Friday} \rangle \times (\langle \text{Lucia} \rangle \cup \langle \text{Pietro} \rangle) \\
&\qquad \times (\langle \text{base} \rangle \times \langle 6 \rangle \cup \\
&\qquad \langle \text{ham} \rangle \times \langle 1 \rangle \cup \\
&\qquad \langle \text{pineapple} \rangle \times \langle 2 \rangle) \cup \\
&\langle \text{Margherita} \rangle \times \langle \text{Tuesday} \rangle \times \langle \text{Mario} \rangle \times \langle \text{base} \rangle \times \langle 6 \rangle.
\end{aligned}
$$

We present three scenarios of increasing complexity where factorisation can benefit aggregate computation. All three scenarios assume the factorisation of the materialised view Sales given, and consider the evaluation of an aggregation operator of the form $\varpi_{G;agg}$, which groups by attributes G and applies the aggregation function agg within each group.

1. We first consider the case when the aggregation only applies locally to a fragment of the factorisation and there is no need to restructure the factorisation or expand to a flat representation. An example query would find the price of each ordered pizza:

$$S = \varpi_{\text{customer, date, pizza; sum(price)}}(\text{Sales}).$$

We can evaluate this query directly on the factorisation of Sales over $\mathcal{T}_1$, where for each pizza we replace the expressions over items and price by the sum of the prices of all its items, obtaining a factorisation over $\mathcal{T}_2$ (Figure 5.4 top middle):

$$
\begin{aligned}
&\langle \text{Capricciosa} \rangle \times (\langle \text{Monday} \rangle \times \langle \text{Mario} \rangle \ \cup \langle \text{Friday} \rangle \times \langle \text{Mario} \rangle) \times \langle 8 \rangle \cup \\
&\langle \text{Hawaii} \rangle \times \langle \text{Friday} \rangle \times (\langle \text{Lucia} \rangle \cup \langle \text{Pietro} \rangle) \times \langle 9 \rangle \cup \\
&\langle \text{Margherita} \rangle \times \langle \text{Tuesday} \rangle \times \langle \text{Mario} \rangle \times \langle 6 \rangle.
\end{aligned}
$$

2. If the aggregation attributes are distributed over the f-tree, we may need to restructure the factorisation to be able to aggregate locally as in the previous example. Addition-

ally, we can decompose the aggregation operation into several partial aggregation steps and intertwine them with restructuring operations. An example query in this category finds the revenue per customer:

$$P = \varpi_{\text{customer; sum(price)}}(\text{Sales}).$$

To evaluate this query, we first partially aggregate prices per pizza, see S above. The factorisation of S is then restructured from the f-tree $\mathcal{T}_2$ to $\mathcal{T}_3$ (Figure 5.4 top right), so that we first group by customers:

$$\begin{aligned}
&\langle\text{Lucia}\rangle \times \langle\text{Hawaii}\rangle \times \langle\text{Friday}\rangle \times \langle 9\rangle \cup \\
&\langle\text{Mario}\rangle \times (\langle\text{Capricciosa}\rangle \times (\langle\text{Monday}\rangle \cup \langle\text{Friday}\rangle) \times \langle 8\rangle \cup \\
&\qquad\qquad \langle\text{Margherita}\rangle \times \langle\text{Tuesday}\rangle \times \langle 6\rangle) \cup \\
&\langle\text{Pietro}\rangle \times \langle\text{Hawaii}\rangle \times \langle\text{Friday}\rangle \times \langle 9\rangle.
\end{aligned}$$

Next, we next count the number of order dates for each pizza ordered by a customer, and obtain a factorisation over the f-tree $\mathcal{T}_4$ (Figure 5.4 bottom left):

$$\begin{aligned}
&\langle\text{Lucia}\rangle \times \langle\text{Hawaii}\rangle \times \langle 1\rangle \times \langle 9\rangle \cup \\
&\langle\text{Mario}\rangle \times (\langle\text{Capricciosa}\rangle \times \langle 2\rangle \times \langle 8\rangle \cup \\
&\qquad\qquad \langle\text{Margherita}\rangle \times \langle 1\rangle \times \langle 6\rangle) \cup \\
&\langle\text{Pietro}\rangle \times \langle\text{Hawaii}\rangle \times \langle 1\rangle \times \langle 9\rangle.
\end{aligned}$$

Finally, we compute the revenue per customer by aggregating the whole subtree under customer. This is accomplished by first computing the revenue per pizza, which is obtained by multiplying the partial count and sum aggregates, and then summing over all pizzas for each customer. The final result over the f-tree $\mathcal{T}_5$ (Figure 5.4 bottom right) is:

$$\langle\text{Lucia}\rangle \times \langle 9\rangle \;\cup\; \langle\text{Mario}\rangle \times \langle 22\rangle \;\cup\; \langle\text{Pietro}\rangle \times \langle 9\rangle.$$

Note that the last aggregation operator re-uses the results of previous aggregation op-

erators. This is essential to partial aggregation, which further helps reduce the size of factorisations in intermediate results.

3. If we are interested in enumerating the tuples in query results with constant delay, as opposed to materialising their factorisations, we can avoid several restructuring steps and thus save computation. For instance, if we would like to compute the revenue per customer and pizza, we could readily use the factorisation over $\mathcal{T}_4$, since for each customer we could multiply the partial aggregates for the number of order dates and the price per pizza for each pair of customer and pizza on the fly. In general, if all group-by attributes are above the other attributes in the f-tree of a factorised relation, then we can enumerate its tuples while executing partial aggregates on the other attributes on the fly. □

Finally we turn to queries with order-by clauses. In Section 3.1 we showed that the tuples of an f-representations can be enumerated in *some* order with the same time complexity (delay between tuples linear in tuple size) as listing them from a flat representation. We characterise all orders that a factorisation over a given f-tree can enumerate with constant delay; the factorisation can always be restructured to support constant-delay enumeration in other orders.

Example 5.3. Factorisations over $\mathcal{T}_1$ can support constant-delay enumeration in different orders: (pizza); (pizza, date); (pizza, item); (pizza, item, date); (pizza, date, item); and so on. The order (date, pizza, item) is not supported for $\mathcal{T}_1$, but becomes possible after swapping the node date past pizza. This, however, need not change the factorisation for pizza, items, and price representing the right branch in $\mathcal{T}_1$. □

The restructuring to support enumeration in a given order is in most cases partial and builds on the intuition that, even in the relational case, sorting can partially use an existing order instead of starting from scratch. For instance, if a relation is sorted by A,B,C, re-sorting by B,A,C need not re-sort the C-values for any pair of values for A and B.

Experimental Findings

Our main experimental observation is that the succinctness of factorised representations of query results, and the evaluation techniques illustrated above, allow the FDB query engine

to clearly outperform relational engines in several scenarios. An orders of magnitude gap in representation size and computation time is observed if query output is allowed in factorised form or if only a limited number of tuples is enumerated. A similar performance gap is achieved when input data is supplied factorised: this case fits a read-optimised scenario with views materialised as factorisations and on which subsequent processing is conducted.

Summary of Contributions

The contribution of this chapter lies in addressing the evaluation and optimisation problems for queries with selection, projection, join, aggregation and ordering on factorised databases. In particular:

- We propose a set of operators for factorised representations that evaluate selections, projections, products and aggregates, as well as operators for restructuring the factorisation, and describe algebraic rules by which they compose and interact. Any query with selections, projections, joins, aggregation and order-by can be compiled into a sequence of such operators, called an *f-plan*.

- We describe efficient algorithms for the evaluation of each f-plan operator on factorised data. The worst-case time complexity of each operator is at most quasilinear in the size of its input and output.

- We characterise those f-trees that support efficient enumeration of result tuples in queries with group-by and order-by clauses. For all other f-trees, factorisations can be partially restructured so as to enable efficient enumeration.

- We define the search space for optimal f-plans on factorisations and introduce one exhaustive and one heuristic optimisation strategy for finding f-plans that compute a given query. As cost metric, we use the compile-time parameter $s(\mathcal{T})$ from Section 3.4 that defines tight bounds on the sizes of factorisations over an f-tree $\mathcal{T}$. Cardinality and selectivity estimates can also be used as a cost metric.

- The optimisation and evaluation algorithms have been implemented in the FDB in-memory query engine.

- We report on an extensive experimental evaluation showing that FDB can outperform a home-bred in-memory and two open-source (SQLite and PostgreSQL) relational query engines by orders of magnitude. Our experiments confirm that the performance of these engines follows the succinctness gap for input data representations.

- We analyse the experimental results and identify the scenarios and workloads in which query evaluation can benefit from factorised representations and the presented evaluation techniques. We describe evaluation techniques for flat relational data analogous to evaluation on factorisations, and discuss related existing work.

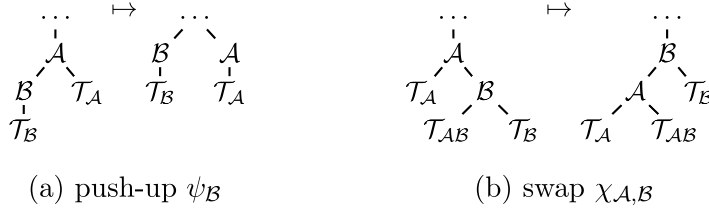
5.1 Query Evaluation

In this section we present a query evaluation technique on f-representations. We propose a set of operators that map between f-representations over f-trees. In addition to the relational operators select, project, Cartesian product and aggregation, we introduce new operators that can restructure f-representations. Restructuring is sometimes needed to enable the execution of a selection, aggregation, or enumeration in a given order, as exemplified in the introduction. Any query with selections, projections, joins, aggregation and ordering can be evaluated by a sequential composition of operators called an *f-plan*.

We consider f-representations over f-trees as defined in Section 3.2. F-trees conveniently represent the structure of factorisations as well as attributes and equality conditions on the attributes. An f-tree uniquely determines the f-representation of a given relation (up to commutativity of $\cup$ and $\times$). Therefore, the semantics of each of our operators may be described solely by the transformation of f-trees $\mathcal{T} \mapsto \mathcal{T}'$ and the transformation $\mathbf{R} \mapsto \mathbf{R}'$ of the underlying relation. Since the f-tree also carries information about the schema and attribute equalities, the latter transformation can be deduced from the former.

We also introduce the notion of normalised f-trees, whose f-representations cannot be further compacted by factoring out subexpressions. We define an operator for normalising f-trees, and all other operators expect normalised input f-trees and preserve normalisation.

We present efficient algorithms to carry out the transformations on f-representations. These algorithms are optimal or almost optimal in the sense that they need at most linear,



(a) push-up $\psi_{\mathcal{B}}$ (b) swap $\chi_{\mathcal{A},\mathcal{B}}$

Figure 5.5: Transformations performed by restructuring operators depicted on f-trees. The normalisation operator is composed of push-up operators and depends on the input f-tree.

and one of them quasilinear, time in the sizes of both input and output f-representations in the worst case. In practice, most operators only transform parts of the f-representation and run in sublinear time.

Theorem 5.1. *The time complexity of each f-plan operator is at most $O(|\mathcal{T}|^2 \cdot N \log N)$, where N is the sum of sizes of the input and output f-representations and $\mathcal{T}$ is the input f-tree.*

In all input f-representations we assume that for any union expression $\bigcup_a \langle A\!:\!a\rangle \times E_a$, the values a occur in increasing order, and that the path constraint (Section 3.3) holds for the input f-tree. Our algorithms preserve these two constraints.

5.1.1 Restructuring Operators

Normalisation Operator and Push-up Operator

The normalisation operator factors out expressions common to all terms of a union. We first present a simple one-step normalisation captured by the push-up operator $\psi_{\mathcal{B}}$, and then a normalisation procedure for f-trees that repeatedly applies the push-up operator bottom-up to each node in the f-tree.

Consider an f-tree $\mathcal{T}$, a node $\mathcal{A}$ and its child $\mathcal{B}$ in $\mathcal{T}$. If no node in the subtree rooted at $\mathcal{B}$ is dependent on $\mathcal{A}$, the entire subtree rooted at $\mathcal{B}$ can be brought one level up (so that $\mathcal{B}$ becomes sibling of $\mathcal{A}$) without violating the path constraint. Proposition 3.7 guarantees that there is an f-representation over the new f-tree. Lifting up a node can only reduce the length of root-to-leaf paths in $\mathcal{T}$ and thus decrease the parameter $s(\mathcal{T})$ and the size of the f-representation, cf. Section 3.4. The transformation only alters the structure of

the factorisation (dictated by the change in the f-tree), the represented relation remains unchanged.

Figure 5.5(a) shows the transformation of the relevant fragment of $\mathcal{T}$, where $\mathcal{T}_\mathcal{A}$ and $\mathcal{T}_\mathcal{B}$ denote the subtrees under $\mathcal{A}$ and $\mathcal{B}$. F-representations over this fragment have the form

$$\Phi_1 = \bigcup_a \left(\langle \mathcal{A}:a \rangle \times (\bigcup_b \langle \mathcal{B}:b \rangle \times F_b) \times E_a\right)$$

and change into

$$\Phi_2 = (\bigcup_b \langle \mathcal{B}:b \rangle \times F_b) \times (\bigcup_a \langle \mathcal{A}:a \rangle \times E_a),$$

where each E_a is over $\mathcal{T}_\mathcal{A}$ and each F_b is over $\mathcal{T}_\mathcal{B}$. Since neither $\mathcal{B}$ nor any node in $\mathcal{T}_\mathcal{B}$ depend on $\mathcal{A}$, all copies of $(\bigcup_b \langle \mathcal{B}:b \rangle \times F_b)$ in Φ_1 are equal, so the transformation amounts to factoring out subexpressions over the subtree rooted at $\mathcal{B}$. In any f-representation over $\mathcal{T}$, the change shown above occurs for all unions over $\mathcal{A}$, and can be executed in linear time in one pass over the f-representation.

Definition 5.1. An f-tree $\mathcal{T}$ is *normalised* if no node in $\mathcal{T}$ can be pushed up without violating the path constraint.

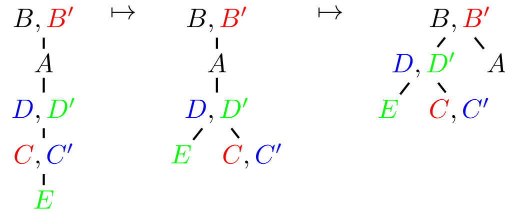
Any f-tree $\mathcal{T}$ can be turned into a normalised one as follows. We traverse $\mathcal{T}$ bottom up and push each node $\mathcal{B}$ and its subtree upwards as far as possible using the operator $\eta_\mathcal{B}$. In case a node $\mathcal{A}$ is pushed up, we mark it so that we do not consider it again. This procedure defines the normalisation operator η.

Proposition 5.1. *The normalisation operator as defined above terminates with a normalised f-tree and its time complexity on an f-representation E over $\mathcal{T}$ is $O(|\mathcal{T}|^2 \cdot |E|)$.*

Proof. If a node is marked, so are all the nodes in its subtree, and at least one of them is dependent on the parent of $\mathcal{A}$ (or $\mathcal{A}$ is a root). The parent of $\mathcal{A}$ and the subtree of $\mathcal{A}$ do not change anymore after $\mathcal{A}$ is marked, so $\mathcal{A}$ cannot be brought upwards again. All nodes are marked after at most $|\mathcal{T}|^2$ applications of the push-up operator, and since the size of the f-representation decreases with each push-up, the time complexity of normalising an f-representation E is $O(|\mathcal{T}|^2 \cdot |E|)$. Since all nodes are marked at the end of the procedure, the resulting f-tree is normalised. ☐

The normalisation condition serves as a counterpart to the path constraint: the path constraint prevents the f-trees from splitting dependent nodes into different branches, while normalisation forces nodes into different branches when they are conditionally independent on their ancestors. In the remainder we only consider normalised f-trees and operators that preserve normalisation.

Example 5.4. Let us normalise the left f-tree below with relations over schemas $\{A, B\}$, $\{B', C\}$, $\{C', D\}$ and $\{D', E\}$.



The transformation depicted above is obtained by ψ_E followed by $\psi_{\{D,D'\}}$. We can bring up E since it is not dependent on its parent in the left f-tree. We then mark E. We also mark $\{C, C'\}$, since it cannot be brought upwards. The lowest unmarked node is now $\{D, D'\}$. It can be brought upwards next to its parent A since A is not dependent on it nor on any of its descendants. The resulting f-tree is normalised. □

Swap Operator

The swap operator $\chi_{\mathcal{A},\mathcal{B}}$ exchanges a node $\mathcal{B}$ with its parent node $\mathcal{A}$ in $\mathcal{T}$ while preserving the path constraint and normalisation of $\mathcal{T}$. We promote $\mathcal{B}$ to be the parent of $\mathcal{A}$, and also move up its children that do not depend on $\mathcal{A}$. The effect of the swapping operator $\chi_{A,B}$ on the relevant fragment of $\mathcal{T}$ is shown in Figure 5.5(b), where $\mathcal{T}_\mathcal{B}$ and $\mathcal{T}_{\mathcal{AB}}$ denote the collections of subtrees under $\mathcal{B}$ that do not depend, and respectively depend, on $\mathcal{A}$, and $\mathcal{T}_A$ denotes the subtree under $\mathcal{A}$. Separate treatment of the subtrees $\mathcal{T}_\mathcal{B}$ and $\mathcal{T}_{\mathcal{AB}}$ is required so as to preserve the path constraint and normalisation. The resulting f-tree has the same nodes as $\mathcal{T}$ and the represented relation remains unchanged.

Any f-representation over the relevant part of the input f-tree $\mathcal{T}$ in Figure 5.5(b) has

ALGORITHM 2: Algorithm for the swap operator $\chi_{\mathcal{A},\mathcal{B}}$.

Data: F-representation E over an f-tree $\mathcal{T}$, node $\mathcal{A}$ and its child $\mathcal{B}$.
Result: F-representation E' over an f-tree $\chi_{\mathcal{A},\mathcal{B}}(\mathcal{T})$ after the swap $\chi_{\mathcal{A},\mathcal{B}}$.

```
identify the subtrees T_A, T_B, T_AB for each expression S_in over the part of T in Figure 5.5(b) do
    S_out ⟵ a new empty union;
    Q ⟵ a new min-priority queue;
    for each ⟨A:a⟩ × E_a × ⋃_b (⟨B:b⟩ × F_b × G_ab) in S_in do
        U_a ⟵ ⋃_b (⟨B:b⟩ × F_b × G_ab);
        p_a ⟵ the first value b in the union U_a;
        Q.InsertValueWithKey(a, p_a);
    end
    while Q is not empty do
        b_min ⟵ MinKey(Q);
        V_{b_min} ⟵ a new empty union;
        while MinKey(Q) = b_min do
            a ⟵ RemoveMinElement(Q);
            append expression ⟨A:a⟩ × E_a × G_ab to V_{b_min};
            if p_a is not the last value in U_a then
                p_a ⟵ the next value b in the union U_a;
                Q.InsertValueWithKey(a, p_a);
            end
        end
        append expression ⟨B:b_min⟩ × F_{b_min} × V_{b_min} to S_out;
    end
    replace S_in by S_out in E;
end
return E;
```

the form

$$\bigcup_a \left(\langle\mathcal{A}\!:\!a\rangle \times E_a \times \bigcup_b \left(\langle\mathcal{B}\!:\!b\rangle \times F_b \times G_{ab}\right)\right),$$

while the corresponding restructured f-representation is

$$\bigcup_b \left(\langle\mathcal{B}\!:\!b\rangle \times F_b \times \bigcup_a \left(\langle\mathcal{A}\!:\!a\rangle \times E_a \times G_{ab}\right)\right).$$

The expressions E_a, F_b and G_{ab} denote the f-representations over the subtrees $\mathcal{T}_\mathcal{A}$, $\mathcal{T}_\mathcal{B}$ and respectively $\mathcal{T}_{\mathcal{AB}}$.

The swap operator $\chi_{\mathcal{A},\mathcal{B}}$ thus takes an f-representation where data is grouped first by $\mathcal{A}$ then $\mathcal{B}$, and produces an f-representation grouped by $\mathcal{B}$ then $\mathcal{A}$. An efficient algorithm to execute $\chi_{\mathcal{A},\mathcal{B}}$ is given in Algorithm 2. We use a priority queue Q to keep for each value a of attributes in $\mathcal{A}$ the minimal values b of attributes in $\mathcal{B}$. This minimal value occurs first in the union U_a due to the order constraint of f-representations. We then extract the values

b from the priority queue Q in increasing order to construct the union over them, and for each of them we obtain the pairing values a. When a value a is removed from Q, we insert it back into Q with the next value b in its union U_a.

Proposition 5.2. *Algorithm 2 executes the swap operator* $\chi_{\mathcal{A},\mathcal{B}}$ *in time* $O(|\mathcal{T}|^2 + N \log N)$*, where* N *is the sum of the sizes of the input and output f-representations.*

Proof. The algorithm must first identify the subtrees $\mathcal{T}_{\mathcal{B}}$ and $\mathcal{T}_{\mathcal{AB}}$ of $\mathcal{B}$ that are independent and dependent on $\mathcal{A}$ respectively. With dependency information this can be done in time quadratic in the number of nodes of $\mathcal{T}$. In the main loop of the algorithm, except for the operations on the priority queue, the total time taken by the algorithm in any given iteration of the outermost loop is linear in the size of the input S_{in} plus the size of the output S_{out}. For each a in S_{in} and b in U_a, the value a is inserted into the queue with key b once and removed once. There are at most $|S_{in}|$ such pairs (a, b) and each of the priority queue operations runs in time $O(\log |S_{in}|)$. The time complexity result follows. □

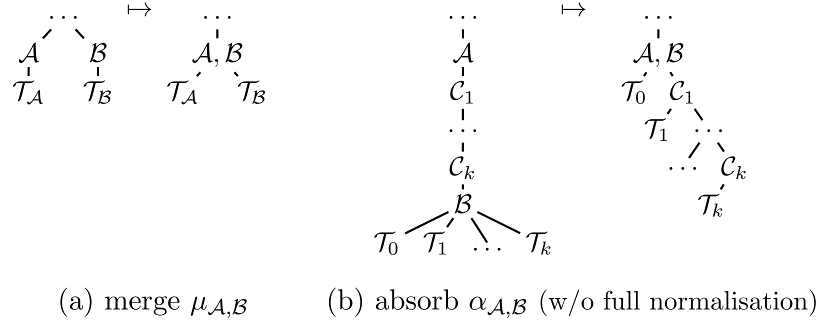
Example 5.5. The tree $\mathcal{T}_2$ in Figure 5.2 is transformed into $\mathcal{T}_3$ by the operator $\chi_{\text{item,location}}$. The effect of the operator on the f-representation amounts to regrouping it primarily by location instead of item, as illustrated in Example 5.1. □

5.1.2 Cartesian Product Operator

Given two f-representations E_1 and E_2 over disjoint sets of attributes, the product of the represented relations can be computed simply as $E_1 \times E_2$. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are the input f-trees, then the resulting f-tree is the forest of $\mathcal{T}_1$ and $\mathcal{T}_2$. The operator can be executed in linear time. It is easy to check that the relation represented by E is indeed the product of the relations of E_1 and E_2, and that this operator preserves the constraints on order of values, path constraint, and normalisation.

5.1.3 Selection Operators

We next present operators for selections with equality conditions of the form $A = B$. Since equality joins are equivalent to equality selections on top of products, and the product of f-representations is just their concatenation, we can evaluate equality joins in the same way



(a) merge $\mu_{\mathcal{A},\mathcal{B}}$ (b) absorb $\alpha_{\mathcal{A},\mathcal{B}}$ (w/o full normalisation)

Figure 5.6: Transformations performed by selection operators depicted on f-trees.

as equality conditions on attributes of the same relation, and do not distinguish between these two cases in the sequel.

If both attributes A and B label the same node in $\mathcal{T}$, then by construction of $\mathcal{T}$ the two attributes are in the same equivalence class, and hence the condition $A = B$ already holds. If $\mathcal{A}$ and $\mathcal{B}$ are two distinct nodes labelled by A and B respectively in an f-tree $\mathcal{T}$, the condition $A = B$ implies that $\mathcal{A}$ and $\mathcal{B}$ should be merged into a single node labelled by the union of the equivalence classes of A and B.

We propose two selection operators: the *merge* operator $\mu_{\mathcal{A},\mathcal{B}}$, which can only be applied in case $\mathcal{A}$ and $\mathcal{B}$ are sibling nodes in $\mathcal{T}$, and the *absorb* operator $\alpha_{\mathcal{A},\mathcal{B}}$, which can only be applied in case $\mathcal{A}$ is an ancestor of $\mathcal{B}$ in $\mathcal{T}$. For all other cases of $\mathcal{A}$ and $\mathcal{B}$ in $\mathcal{T}$, we first need to apply the swap operator until we transform $\mathcal{T}$ in one of the above two cases. The reason for supporting these selection operators only is that they are simple, atomic, can be implemented very efficiently, and any selection can be expressed by a sequence of swaps and selection operators. We next discuss them in depth.

Merge Operator

The merge operator $\mu_{\mathcal{A},\mathcal{B}}$ for equality selection merges the sibling nodes $\mathcal{A}$ and $\mathcal{B}$ of $\mathcal{T}$ into one node labelled by the attributes of $\mathcal{A}$ and $\mathcal{B}$ and whose children are those of $\mathcal{A}$ and $\mathcal{B}$, see Figure 5.6(a). This operator preserves the path constraint, since the root-to-leaf paths in $\mathcal{T}$ are preserved in the resulting f-tree. Also, normalisation is preserved: merging two nodes of a normalised f-tree produces a normalised f-tree. To preserve the value order constraint, node merging is implemented as a sort-merge join. Any f-representation over the relevant

part of $\mathcal{T}$ has the form

$$\Phi_1 = (\bigcup_a \langle \mathcal{A}\!:\!a \rangle \times E_a) \times (\bigcup_b \langle \mathcal{B}\!:\!b \rangle \times F_b),$$

and change into

$$\Phi_2 = \bigcup_{a:a=b} \langle \mathcal{A}\!:\!a \rangle \times \langle \mathcal{B}\!:\!b \rangle \times E_a \times F_b,$$

where the union in Φ_2 is over the equal values a and b of the unions in Φ_1. An algorithm for $\mu_{\mathcal{A},\mathcal{B}}$ needs one pass over the input f-representation to identify expressions like Φ_1, and for each such expression it computes a standard sort-merge join on the sorted lists of values of these unions.

Proposition 5.3. *The time complexity of the merge operator is linear in the input size.*

Example 5.6. Consider an f-tree that is the forest of $\mathcal{T}_1$ and $\mathcal{T}_3$ from Figure 5.2. The two attributes with the same name item are siblings (at the topmost level). By merging them, we obtain the f-tree $\mathcal{T}_4$. Similarly, merging the two sibling nodes named location in $\mathcal{T}_5$ results in the f-tree $\mathcal{T}_6$. Example 5.1 shows f-representations over the input and output f-trees of these merge operations. □

Absorb Operator

The absorb operator $\alpha_{\mathcal{A},\mathcal{B}}$ for equality selection absorbs a node $\mathcal{B}$ into its ancestor $\mathcal{A}$ in an f-tree $\mathcal{T}$, and then normalises the resulting f-tree. The attributes labelling $\mathcal{B}$ are added to the label of $\mathcal{A}$. The absorption of $\mathcal{B}$ into $\mathcal{A}$ preserves the path constraint since all attributes in $\mathcal{B}$ remain on the same root-to-leaf paths. By definition, the absorb operator finishes with a normalisation step, thus it preserves the normalisation constraint. It is also immediate from the algorithm that the operator preserves the ordering property of subexpressions in each union.

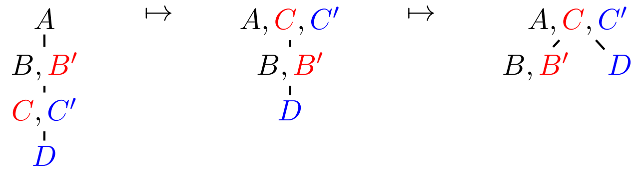
In any f-representation, each union over $\mathcal{B}$ is inside a union over its ancestor $\mathcal{A}$, and hence inside a product with a particular value a of $\mathcal{A}$. Enforcing the constraint $A = B$ amounts to restricting each such union over $\mathcal{B}$ by $B = a$, by which it remains with only one or zero subexpression. This can be executed in one pass over the f-representation, needs

linear time in the input size, and does not increase the size of the f-representation. The subsequent normalisation then takes time $O(|\mathcal{T}|^2 \cdot |E|)$ by Proposition 5.1.

Proposition 5.4. *The complexity of the absorb operator on an f-representation* $|E|$ *over an f-tree* $\mathcal{T}$ *is* $O(|\mathcal{T}|^2 \cdot |E|)$.

To normalise the f-tree after absorbing $\mathcal{B}$ into $\mathcal{A}$, we use the normalisation operator η. If the original tree was normalised, it is sufficient to push up the subtrees of $\mathcal{B}$ as shown in Figure 5.6(b), and possibly also push up some of the nodes $\mathcal{C}_1, \dots, \mathcal{C}_k$ on the path between $\mathcal{A}$ and $\mathcal{B}$.

Example 5.7. Consider the selection $A = C$ on the leftmost f-tree below with relations over schemas $\{A, B\}$, $\{B', C\}$ and $\{C', D\}$. Since A and C correspond to ancestor and respectively descendant nodes, we can use the absorb operator to enforce the selection. When absorbing $\{C, C'\}$ into A (middle f-tree), the nodes $\{B, B'\}$ and D become independent and D can be pushed upwards (right f-tree):



□

Selection with Constant Operator

The operator $\sigma_{A\theta c}$ for a selection with a constant can be evaluated in one pass over the input f-representation E. Whenever we encounter a union $\bigcup_a(\langle A\!:\!a\rangle \times E_a)$ in E, we remove all expressions $\langle A\!:\!a\rangle \times E_a$ for which $a\neg\theta c$. If the union becomes empty and appears in a product with another expression, we then remove that expression too and continue until no more expressions can be removed. In case θ is an equality comparison, then all remaining A-values are equal to c and we can factor out the singleton $\langle A\!:\!c\rangle$.

For a comparison θ different from equality, the f-tree remains unchanged. In case of equality, we can infer that all A-values in the f-representation are equal to c and thus the node $\mathcal{A}$ labelled by A is independent of the other nodes in the f-tree and can be pushed

up as the new root. When computing the parameter $s(\mathcal{T})$, we can ignore $\mathcal{A}$ since the only f-representation over it is the singleton $\langle A\!:\!c\rangle$.

Theta Conditions

Currently, FDB can only consider theta-joins via an indirection. We can evaluate an arbitrary condition $A\theta B$ on a relation R (which may be the result of subsequent joins) by joining R with a binary relation $\Theta_{A,B} = \sigma_{A\theta B}(\pi_{A,B}(R))$ on the attributes A and B. Any such join can be incorporated in our system by introducing the extra relation $\Theta_{A,B}$, and hence an extra dependency set $\{A, B\}$. The relation $\Theta_{A,B}$ need not be materialised in the database, it is sufficient to decide the membership $A\theta B$ on the fly.

5.1.4 Projection Operator

We define a projection operator π_{-A}, which projects away the attribute A, i.e., it acts as a projection $\pi_{\mathcal{S}\setminus\{A\}}$ on the represented relation of schema $\mathcal{S}$. A relational projection $\pi_{\mathcal{P}}$ can be evaluated by a sequence of projection operators, one for each attribute in $\mathcal{S} \setminus \mathcal{P}$.

In an f-representation E, π_{-A} replaces all A-singletons $\langle A\!:\!a\rangle$ with the nullary singleton $\langle\rangle$. Subsequently, any union of nullary singletons is replaced by one nullary singleton and any nullary singleton in a product with other singletons is removed from E. This procedure can be performed in one scan over the input f-representation E and trivially preserves the order constraint. In the corresponding f-tree $\mathcal{T}$, π_{-A} removes the attribute A. If there is no further attribute labelling the node of A, we also remove that node.

We only permit the operator π_{-A} when there are further attributes labelling the node of A or when A labels a leaf. By removing this node, any two attributes B and C previously dependent on A now become dependent on each other (cf. Section 3.3). If A is a leaf, such attributes lie on a path from A to root, and the path constraint is thus preserved. If A is at an inner node, then B and C may be on different branches under A and the path constraint is violated. In this latter case, π_{-A} is therefore not permitted and the f-representation must be first restructured using the swap operator.

5.1.5 Aggregation Operator

In this section we propose a new aggregation operator on factorised data: we describe its syntax and action on f-trees, its semantics and action on f-representations, we describe how a query aggregate can be decomposed into one or more aggregation operators, and finally we give linear-time algorithms for the operator.

The syntax of the operator is $\gamma_{F(\mathcal{U})}$, where F is the aggregation function, which in our case can be any of sum, count, max, or min (avg is recovered as a pair of sum and count), and $\mathcal{U}$ is a subtree in the f-tree $\mathcal{T}$ of the input factorisation. In case of an aggregation function sum_A, min_A or max_A, the subtree $\mathcal{U}$ must contain the attribute A.

Given a factorisation over $\mathcal{T}$, the operator evaluates the aggregation function F over all attributes in $\mathcal{U}$ and stores the result in a new attribute $F(\mathcal{U})$. Expressed as a transformation of the relation $[\![E]\!]$ represented by the factorisation E, this operator maps $[\![E]\!]$ to a relation $R = \varpi_{\mathcal{T}\setminus\mathcal{U};F(\mathcal{U})\leftarrow F}[\![E]\!]$ over the schema[1] $(\mathcal{T} \setminus \mathcal{U}) \cup \{F(\mathcal{U})\}$.

In the resulting f-tree $\mathcal{T}'$, the subtree $\mathcal{U}$ is replaced by a new node $F(\mathcal{U})$. The resulting factorisation is uniquely characterised by its underlying relation R and f-tree $\mathcal{T}'$. Later in this section we give algorithms for the aggregation operator that compute such factorisations.

Example 5.8. Figure 5.4 shows f-trees before and after the execution of aggregate operators. For $F = \text{sum}_{\text{price}}$ and the subtree $\mathcal{U}$ rooted at node item in $\mathcal{T}_1$, the resulting f-tree after the execution of the operator $\gamma_{F(\mathcal{U})}$ is $\mathcal{T}_2$. For $F = \text{count}_{\text{date}}$ and the input f-tree $\mathcal{T}_3$, the resulting f-tree after the execution of the operator $\gamma_{F(\text{date})}$ is $\mathcal{T}_4$. □

Similarly to the case of conjunctive queries, we can characterise precisely all f-trees $\mathcal{T}'$ that define the nesting structures of factorisations for possible results of the aggregation operator $\gamma_{F(\mathcal{U})}$. The characterisation via the path constraint in Proposition 3.7 also holds for aggregation, with the addition that the aggregation operator introduces new dependencies among the attributes in the f-tree $\mathcal{T}'$, similarly as for the projection operator. By projecting away the attributes in $\mathcal{U}$, all attributes dependent on attributes in $\mathcal{U}$ now become dependent on each other (exactly as for the projection operator). In addition, the new attribute $F(\mathcal{U})$

[1] To avoid clutter, we slightly abuse notation and use $\mathcal{T}$ to also denote the set of attributes in the f-tree $\mathcal{T}$.

depends on each of these attributes. The path constraint then stipulates that any two dependent attributes must lie along a same root-to-leaf path in the new f-tree $\mathcal{T}'$. Thus, the f-tree $\mathcal{T}'$ resulting from the aggregation operator $\gamma_{F(\mathcal{U})}$ satisfies the path constraint and hence the resulting factorisation exists and is uniquely defined.

Example 5.9. Consider the aggregation operators described in Example 5.2. In $\mathcal{T}_2$, the only new dependency introduced by the aggregate operator is between the new attribute $\text{sum}_{\text{price}}(\text{item}, \text{price})$ and the attribute pizza, since we projected away the attributes item and price that depended on the attribute pizza.

In $\mathcal{T}_4$, the new attribute $\text{count}_{\text{date}}$ depends on all attributes that the attribute data depended on, namely pizza and customer. □

Composing Aggregation Operators

As exemplified in the introduction, for reasons of efficiency we would often like to execute a query by implementing aggregates via several aggregation operators and by possibly interleaving them with restructuring operators. This requires an approach that can compose aggregation operators so as to implement a larger aggregate. We next describe such an approach.

We give special status to the attributes that hold results of previous aggregate operators, and interpret them as pre-computed values of an aggregate instead of arbitrary data values. We refer to such attributes as *aggregate attributes*; all other attributes are *atomic*. An aggregate attribute $G(\mathcal{X})$ carries along the aggregation function G and the original attributes $\mathcal{X}$ to which G was applied. The aggregation operator then interprets factorisations $\langle G(\mathcal{X}):v\rangle$ over the f-tree consisting of the node $G(\mathcal{X})$ as a relation over schema $\mathcal{X}$ and where the aggregate value for G is v. This special interpretation of aggregate attributes helps us distribute the evaluation of a query aggregate $\varpi_{G;\alpha\leftarrow F}$ over several aggregation operators. After the last operator is executed, we execute a renaming operator that changes the name of the last aggregation function application to the attribute α, as specified by the query aggregate.

Example 5.10. After applying the operator $\gamma_{\text{count(item)}}$ to the relation Pizzas in Figure 5.3,

we get the factorisation

$$
\begin{aligned}
&\langle \text{pizza}:\text{Margherita}\rangle \times \langle \text{count(item)}:1\rangle \cup\\
&\langle \text{pizza}:\text{Capricciosa}\rangle \times \langle \text{count(item)}:3\rangle \cup\\
&\langle \text{pizza}:\text{Hawaii}\rangle \times \langle \text{count(item)}:3\rangle .
\end{aligned}
$$

A subsequent count(pizza, item) aggregation must interpret the singleton $\langle \text{count(item)}:3\rangle$ as a relation with three items to obtain the correct result $\langle \text{count(pizza, item)}:7\rangle$ and not $\langle \text{count(pizza, item)}:3\rangle$. □

The composition rules for these operators are specified next using the standard notation with a binary operator $\circ$: $B \circ A$ means that we first evaluate A and then B.

Proposition 5.5. *For any (sum, count, min, max) aggregation functions F and G and f-trees $\mathcal{U}$ and $\mathcal{V}$, it holds that:*

- *If $\mathcal{U} \supseteq \mathcal{V}$, then $\gamma_{F(\mathcal{U})} \circ \gamma_{F(\mathcal{V})} = \gamma_{F(\mathcal{U})}$.*
- *If $\mathcal{U} \supseteq \mathcal{V}$ and $A \notin \mathcal{V}$, then*

$$\gamma_{\text{sum}_A(\mathcal{U})} \circ \gamma_{\text{count}(\mathcal{V})} = \gamma_{\text{sum}_A(\mathcal{U})}.$$

- *If $\mathcal{U} \cap \mathcal{V} = \emptyset$, then $\gamma_{F(\mathcal{U})} \circ \gamma_{G(\mathcal{V})} = \gamma_{G(\mathcal{V})} \circ \gamma_{F(\mathcal{U})}$.*

Using Proposition 5.5, we can deduce that

$$
\begin{aligned}
\gamma_{F(\mathcal{U}_n)} \circ \cdots \circ \gamma_{F(\mathcal{U}_1)} &= \gamma_{F(\mathcal{U}_n)}\\
\gamma_{F_n(\mathcal{U}_n)} \circ \cdots \circ \gamma_{F_1(U_1)} &= \gamma_{F_n(\mathcal{U}_n)}
\end{aligned}
$$

for any sequence of composable aggregation operators such that for all $i = 1, \ldots, n$ we have $\mathcal{U}_i \subseteq \mathcal{U}_n$, F can be any (sum, count, min, max) aggregation function, and F_i is sum_A whenever $A \in \mathcal{U}_i$, and the count aggregation function otherwise. In other words, if the last operator aggregates over an attribute set $\mathcal{U}$, we can do pre-aggregations on subsets of $\mathcal{U}$.

The query aggregates can then decompose as follows: count aggregates can decompose into several count operators; sum_A aggregates can decompose into a mix of sum_A and count operators, min aggregates can decompose into several min operators, and max aggregates into max operators.

Example 5.11. Consider the f-tree $\mathcal{T}_4$ in Figure 5.4 and the factorisation in Example 5.2 that was obtained by executing the operators $\gamma_{\text{sum}_{\text{price}}(\text{item,price})}$, followed by restructuring and then $\gamma_{\text{count(date)}}$ on relation R. A subsequent operator $\gamma_{\text{sum}_{\text{price}}(\mathcal{U})}$, with $\mathcal{U}$ the subtree rooted at node pizza, uses the results of the first two operators to compute the result of the query aggregate $\varpi_{\text{customer;sum(price)}}(R)$, as detailed in Example 5.2. An alternative evaluation would only execute $\gamma_{\text{sum}_{\text{price}}(\mathcal{U})}$ without executing the previous two aggregation operators: this would possibly cause larger intermediate results, but the same final result. We can capture this equivalence as

$$\gamma_{\text{sum}_{\text{price}}(\mathcal{U})} \circ \gamma_{\text{count(date)}} \circ \gamma_{\text{sum}_{\text{price}}(\text{item,price})} = \gamma_{\text{sum}_{\text{price}}(\mathcal{U})}.$$

Algorithms for the Aggregation Operator

In a factorisation over an f-tree $\mathcal{T}$, the expressions over a subtree $\mathcal{U}$ of $\mathcal{T}$ represent the values of the attributes of $\mathcal{U}$ grouped by the remaining attributes $\mathcal{T} \setminus \mathcal{U}$. The aggregation operator $\gamma_{F(\mathcal{U})}$ then only replaces each such expression over $\mathcal{U}$ by a singleton $\langle F(\mathcal{U}) : v \rangle$ where v is the value of the aggregation function F on the relation represented by that expression. The value of $F([\![E]\!])$ for a given factorisation E over an f-tree $\mathcal{U}$ can be computed recursively on the structure of E in time linear in the size of E, even though E can be much smaller than the relation $[\![E]\!]$ it represents.

We next give algorithms for each aggregation function.

Algorithm for count

We first give a recursive counting algorithm. The input is a factorisation E over an f-tree and the output is the cardinality of the relation $[\![E]\!]$ represented by E.

count(E):

- **If** $E = \langle A\!:\!a\rangle$ for atomic attribute A and value a, **then return** 1.
- **If** $E = \langle \text{count}(\mathcal{X})\!:\!c\rangle$ for any set of atomic attributes $\mathcal{X}$ and number c, **then return** c.
- **If** $E = \bigcup_i E_i$, **then return** $\sum_i \text{count}(E_i)$,

  since the relations $[\![E_i]\!]$ represented by the subexpressions E_i are disjoint.
- **If** $E = \times_i E_i$, **then return** $\prod_i \text{count}(E_i)$.

Algorithm for sum_A

The case of a sum aggregate is similar to count. The following algorithm sum_A takes as input a factorisation E over an f-tree that contains the attribute A and outputs the sum of all A-values in the relation $[\![E]\!]$ represented by E.

$\text{sum}_A(E)$:

- **If** $E = \langle A\!:\!a\rangle$, **then return** a.
- **If** $E = \langle \text{sum}_A(\mathcal{X})\!:\!s\rangle$ for any set of attributes $\mathcal{X}$ and number s, **then return** s.
- **If** $E = \bigcup_i E_i$, **then return** $\sum_i \text{sum}_A(E_i)$,

  since the relations $[\![E_i]\!]$ represented by the subexpressions E_i are disjoint.
- **If** $E = \times_i E_i$, then exactly one of the expressions E_i has the attribute A in its schema; let it be E_j.

  Then return $\text{sum}_A(E_j) * \prod_{i \neq j} count(E_i)$.

Example 5.12. Consider the factorisation

$$
\begin{aligned}
&\langle \text{customer}\,{:}\,\text{Lucia}\rangle \times \langle \text{pizza}\,{:}\,\text{Hawaii}\rangle \times \langle \text{count}_{\text{date}}(\text{date})\,{:}\,1\rangle \\
&\qquad \times \langle \text{sum}_{\text{price}}(\text{item}, \text{price})\,{:}\,9\rangle \cup \\
&\langle \text{customer}\,{:}\,\text{Mario}\rangle \times (\langle \text{pizza}\,{:}\,\text{Capricciosa}\rangle \times \langle \text{count}_{\text{date}}(\text{date})\,{:}\,2\rangle \\
&\qquad \times \langle \text{sum}_{\text{price}}(\text{item}, \text{price})\,{:}\,8\rangle \cup \\
&\qquad\quad \langle \text{pizza}\,{:}\,\text{Margherita}\rangle \times \langle \text{count}_{\text{date}}(\text{date})\,{:}\,1\rangle \\
&\qquad \times \langle \text{sum}_{\text{price}}(\text{item}, \text{price})\,{:}\,6\rangle) \cup \\
&\langle \text{customer}\,{:}\,\text{Pietro}\rangle \times \langle \text{pizza}\,{:}\,\text{Hawaii}\rangle \times \langle \text{count}_{\text{date}}(\text{date})\,{:}\,1\rangle \\
&\qquad \times \langle \text{sum}_{\text{price}}(\text{item}, \text{price})\,{:}\,9\rangle
\end{aligned}
$$

over the f-tree $\mathcal{T}_4$ from Example 5.2. The operator $\gamma_{\text{sum}_{\text{price}}(\mathcal{U})}$, where $\mathcal{U}$ is the subtree of $\mathcal{T}_4$ rooted at node pizza, replaces each expression over $\mathcal{U}$ with the aggregate value $\text{sum}_{\text{price}}$ of its represented relation. That is, we must calculate $v = \text{sum}_{\text{price}}[\![E]\!]$, where

$$
\begin{aligned}
E = &\langle \text{pizza}\,{:}\,\text{Hawaii}\rangle \times \langle \text{count}_{\text{date}}(\text{date})\,{:}\,1\rangle \times \\
&\langle \text{sum}_{\text{price}}(\text{item}, \text{price})\,{:}\,9\rangle),
\end{aligned}
$$

and replace E in the factorisation by v:

$$
\langle \text{customer}\,{:}\,\text{Lucia}\rangle \times \langle \text{sum}_{\text{price}}(\text{item}, \text{price}, \text{pizza}, \text{date})\,{:}\,9\rangle).
$$

Similarly, we obtain the following for Mario and Pietro:

$$
\begin{aligned}
&\langle \text{customer}\,{:}\,\text{Mario}\rangle \times \langle \text{sum}_{\text{price}}(\text{item}, \text{price}, \text{pizza}, \text{date})\,{:}\,22\rangle) \\
&\langle \text{customer}\,{:}\,\text{Pietro}\rangle \times \langle \text{sum}_{\text{price}}(\text{item}, \text{price}, \text{pizza}, \text{date})\,{:}\,9\rangle).
\end{aligned}
$$

Using the algorithm for sum, the value $v = \text{sum}_{\text{price}}[\![E]\!]$ can be computed as

$$[\![\langle \text{pizza}:\text{Hawaii}\rangle \times \langle \text{count}_{\text{date}}(\text{date}):1\rangle \times \langle \text{sum}_{\text{price}}(\text{item}, \text{price}):9\rangle)]\!]$$
$$= 1 \cdot [\![\langle \text{count}_{\text{date}}(\text{date}):1\rangle]\!] \cdot [\![\langle \text{sum}_{\text{price}}(\text{item}, \text{price}):9\rangle]\!] = 1 \cdot 1 \cdot 9 = 9.$$

Similarly, $v = 1 \cdot (1 \cdot 2 \cdot 8 + 1 \cdot 1 \cdot 6) = 16 + 6 = 22$ for Mario and $v = 1 \cdot 1 \cdot 9 = 9$ for Pietro, yielding the final result as shown in Example 5.2. □

Algorithm for $\min_A$ and $\max_A$

We next give an algorithm for the aggregation function min_A; the case for max_A is analogous.

$min_A(E)$:

- **If** $E = \langle A:a\rangle$, **then return** a.
- **If** $E = \langle \min_A(\mathcal{X}):c\rangle$ for any set of attributes $\mathcal{X}$ and value c, **then return** c.
- **If** $E = \bigcup_i E_i$, **then return** $\min_i \min_A(E_i)$.
- **If** $E = \times_i E_i$, where E_j is the expression that has the attribute A in its schema, **then return** $\min_A(E_j)$.

Composite Aggregation Functions

For a composite aggregation function (F, G), such as $\text{avg}_A = (\text{sum}_A, \text{count})$, we apply the algorithms for the constituent aggregation functions F and G separately. For an input factorisation E, we then obtain $(F(E), G(E))$, e.g., $(\text{sum}_A(E), \text{count}(E))$ in case of avg_A.

Query aggregates with more than one aggregation function also call for composite aggregate functions. For instance, the query aggregate $\varpi_{G;\alpha_1 \leftarrow F_1,\ldots,\alpha_k \leftarrow F_k}$ require the evaluation of a k-ary aggregation function $F = (F_1, \ldots, F_k)$. As for unary aggregates, the evaluation of composite aggregates can be distributed over several aggregation operators. Since the grouping attributes G are the same for all aggregation functions in F, each of these operators aggregates over the same f-tree for all aggregation functions in F. The resulting singletons

in the factorisation would have the form $\langle(F_1,\dots,F_k):(v_1,\dots,v_k)\rangle$, where v_i would be the result of applying the aggregation F_i on the input.

If the same aggregation function has to be applied several times, we calculate its result value only once. This situation arises e.g. in case of the avg_A aggregate or more generally for query aggregates with count and sum_A functions, since sum_A is decomposed into sum_A and count, and the two count computations can be shared.

Projection Revisited

The projection operator $\pi_{\mathcal{P}}$ that projects away entire equivalence classes of attributes (nodes) is equivalent to an aggregation $\varpi_{\mathcal{P}}$ with no aggregation functions. Aggregation with no aggregation function can be simulated by an aggregation function ε defined by $\varepsilon(E) = \langle\rangle$ for any f-representation E. Expressing projections using projection operators π_{-A} as defined in Section 5.1.4 is equivalent to expressing them using the aggregation operators $\gamma_{\varepsilon(\mathcal{A})}$: both require that the projected out node $\mathcal{A}$ is a leaf, and both remove it from the f-tree and its singletons from the f-representation.

5.1.6 Group-by and Order-by Clauses

We next address the problem of evaluating group-by and order-by clauses on factorised data. While these query constructs are relevant to enumeration only and do not change the represented data, they may require restructuring to enable fast enumeration.

On relational data, grouping by a set G of attributes partitions the input tuples into groups that agree on the value for G. In this work, grouping is solely used in connection with aggregates, where a set of aggregates are applied on the tuples within each group. One approach to implementing grouping is to sort the input relation on the attributes of G using some order of these attributes; this is similar in spirit to the approach taken by FDB.

Ordering an input relation by a list O of attributes sorts the input relation lexicographically on the attributes in the order given by O; for each attribute in O, we can specify whether the sorting is in ascending or descending order. The limit query operator λ_k limits the result to the first k tuples from the sorted relation.

The tuples of a sorted relation can be enumerated in order with *constant delay*, i.e., the

time between listing a tuple and listing its next tuple in the desired order is linear in tuple size and thus constant in data complexity (independent of the number of tuples). In the following, we characterise those factorisations that support constant-delay enumeration in a given order. Other factorisations must be restructured using our restructuring operators to meet the constraint.

F-tree Characterisation by Constant-Delay Enumeration in Given Orders

For any factorisation E over an f-tree $\mathcal{T}$, it is possible to enumerate the tuples in the represented relation $[\![E]\!]$ in *no particular order* with constant delay; more precisely, the delay is linear in the size of the schema, which is fixed (see Section 3.1). The goal of this section is to characterise those f-trees $\mathcal{T}$ defining factorisations for which constant-delay enumeration also exists for some given orders.

For any attribute, its singletons within each union are kept sorted in ascending order and all operators preserve this ordering constraint. This sorting is used for efficient implementation of equality selections as intersection of sorted lists, as well as for efficient restructuring. It also serves well our enumeration purpose. In particular, any factorisation already supports constant-delay enumeration in certain orders, for example those representing prefixes of paths in the f-tree of the factorisation.

Example 5.13. The f-tree $\mathcal{T}_1$ in Figure 5.4 supports constant-delay enumeration in any of the orders (pizza); (pizza, date); (pizza, date, customer); (pizza, item); or (pizza, item, price); (pizza, date, item); but not in the orders (pizza, customer, date); (customer, pizza). This can be verified on the factorisation over $\mathcal{T}_1$ given in Figure 5.3. The order on each of these attributes need not be ascending: if for instance the order on the pizza attribute is descending, we iterate on the sorted list of pizzas from the end of the list to the front. □

In contrast to ordering, grouping is less restrictive since the order of the attributes in the group is not relevant. An f-tree then readily supports constant-delay enumeration of tuples by groups in a larger number of orders.

Example 5.14. The f-tree $\mathcal{T}_1$ in Figure 5.4 supports constant-delay enumeration for grouping over all orders mentioned in the previous example as well as all their permutations. □

We next make this intuition more precise. For the following statements, we assume without loss of generality that no two attributes in the attribute group G or order list O are within the same equivalence class; if they are, their values are the same for each tuple and we can ignore one of these attributes in G and the last of the two in the ordered list O.

Theorem 5.2. *For an f-tree $\mathcal{T}$ and a set G of group-by attributes, the tuples of $[\![E]\!]$ for any factorisation E over $\mathcal{T}$ can be enumerated with constant delay in groups with equal values of G if and only if each attribute of G is either a root in $\mathcal{T}$ or a child of another attribute of G.*

The case of order-by clauses is more restrictive.

Theorem 5.3. *For an f-tree $\mathcal{T}$ and a list O of order-by attributes, the tuples of $[\![E]\!]$ for any factorisation E over $\mathcal{T}$ can be enumerated with constant delay in sorted lexicographic order by O if and only if each attribute X of O is either a root in $\mathcal{T}$ or a child of an attribute appearing before X in O.*

5.2 Query Optimisation

FDB evaluates queries on factorised representations by a sequence of operators introduced in Section 5.1, called an f-plan. In this section we discuss the problem of query optimisation: finding an f-plan that implements a given query. Typically there exist many f-plans for a given query that differ in the choice of selection operators to execute each join, the sequence of aggregation operators that together evaluate the query aggregate, the order of selection and aggregation operators, and the factorisation restructuring in between.

Among the many possible f-plans, the task of a query optimiser is to choose one that can be evaluated in minimal possible computation time. In addition to the standard objective of minimising computation time, present in the standard (flat) relational case, the nature of factorised data calls for a new objective: from the space of equivalent f-representations for the query result, we would like to find a small, ideally minimal, f-representation. These objectives can work together (processing smaller factorisations requires less computation), but can also conflict (extra restructuring may be needed to achieve a smaller factorisation).

The aim of FDB's optimiser is to find an f-plan for the given query such that the cost of the sequence of operators is low *and* the query result is well-factorised.

We introduce two different numerical cost measures for f-plans: one based on the asymptotic size measure from Section 3.4, and one based on cardinality estimates of restricted relations. We define the space of all possible f-plans and give a local characterisation of the space, so that a search algorithm can be used to find the cheapest f-plan under a given cost metric. However, the space can be exponential in the size of the query, so we also propose a greedy heuristic algorithm that selects an f-plan in polynomial time.

5.2.1 Cost Metrics for F-Plans

We next define two cost measures for f-plans. One measure is based on the parameter $s(\mathcal{T})$ that defines size bounds on factorisations over f-trees for any input database. The second measure is based on cardinality estimates inferred from the intermediate f-trees and the database catalogue with information on relation sizes and selectivity estimates. Both measures can be used by the exhaustive search procedure and the greedy heuristic for query optimisation presented later in this section.

Cost Based on Asymptotic Bounds

As discussed in Section 3.4, the size of any f-representation over an f-tree $\mathcal{T}$ depends exponentially on the parameter $s(\mathcal{T})$, i.e., its size is in $O(|\mathbf{D}|^{s(\mathcal{T})})$. Since the cost of each operator is quasilinear in the sum of sizes of its input and output, it is dictated by the parameter $s(\mathcal{T})$. For an f-plan $f = \omega_1, \ldots, \omega_k$ that performs the following sequence of transformations:

$$\mathcal{T}_{\text{initial}} = \mathcal{T}_0 \overset{\omega_1}{\mapsto} \mathcal{T}_1 \overset{\omega_2}{\mapsto} \ \ldots \ \overset{\omega_k}{\mapsto} \mathcal{T}_k = \mathcal{T}_{\text{final}},$$

the evaluation time is $O(|\mathbf{D}|^{s(f)} \cdot \log |\mathbf{D}|)$, where

$$s(f) = \max(s(\mathcal{T}_0), s(\mathcal{T}_1), \ldots, s(\mathcal{T}_k)).$$

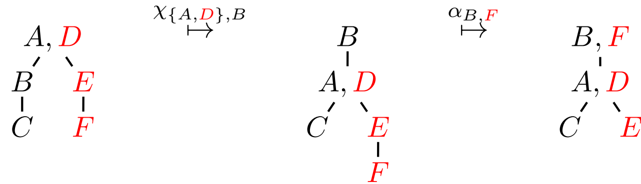
The sizes of the intermediate f-representations thus dominate the execution time. Using

this cost measure, a good f-plan is one whose intermediate f-trees $\mathcal{T}_i$ have small $s(\mathcal{T}_i)$.

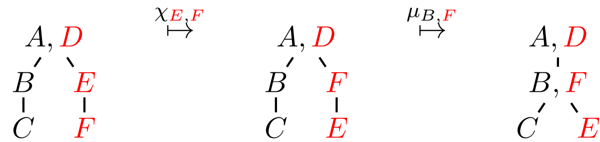
In defining a notion of optimality for f-plans, we would like to optimise for two objectives, namely minimise the f-plan cost $s(f)$ and the cost of the final result $s(\mathcal{T}_{\text{final}})$. However, it might not be possible to optimise for both objectives $<_{s(f)}$ and $<_{s(\mathcal{T})}$ at the same time. Instead, we prioritise $s(f)$ over $s(\mathcal{T}_{\text{final}})$ and use the lexicographic order $<_{s(f)} \times <_{s(\mathcal{T})}$ on f-plans: given f-plans f_1 and f_2, we consider f_1 better than f_2 and write $f_1 <_{s(f)} \times <_{s(\mathcal{T})} f_2$ if either (1) the most expensive operator in f_1 is less expensive than the most expensive operator in f_2, or (2) their most expensive operators have the same cost but the cost of the result is smaller for f_1. An f-plan f_1 for a query Q is *optimal* if there is no other f-plan f_2 for Q such that $f_2 <_{s(f)} \times <_{s(\mathcal{T})} f_1$.

This notion of optimality is over f-plans consisting of operators defined in Section 5.1. Since these operators preserve f-tree normalisation, this also means that we consider optimality only over the space of possible normalised f-trees.

Example 5.15. Consider the following f-plan evaluating the selection $B = F$ on the leftmost f-tree, with dependencies $\{A, B, C\}$ and $\{D, E, F\}$.



The input f-tree and the output f-tree have both cost 1, as each root-to-leaf path is covered by a single relation. However, the intermediate f-tree has cost 2 (as on the path from B to F each of B and F must be covered by a separate relation), so the cost of the f-plan is 2. An alternative f-plan starts by swapping F with its parent to obtain an intermediate f-tree with cost 1, and then merges F with B.



Although both f-plans result in an f-tree with cost 1, the latter f-plan has cost 1 while the

former has cost 2. $\square$

Cost Based on Estimates

We can also estimate the cost of an f-plan using cardinalities and join selectivities for the input f-representation E. One approach is to keep cardinalities and selectivities for the relation $\mathbf{R}_E$ represented by E, in which case we fall back to relational catalogues. A different approach would take advantage of the conditional independence of attributes, as specified by the input f-tree, to more accurately record join selectivities and cardinalities (e.g., how many A-singletons are in average under a B-singleton). This latter approach is subject to future work.

Given a query Q on the input f-representation E, we can estimate the size of the f-representation F of the query result over an f-tree $\mathcal{T}$ as follows. Let $\mathbf{R}_F$ be the relation represented by F, i.e., $\mathbf{R}_F = Q(\mathbf{R}_E)$. The number of A-singletons in F is equal to the size of the relation $\pi_{\text{path}(A)}\mathbf{R}_F$, where path$(A)$ is the set of attributes along the path from the root to the node of A in $\mathcal{T}$ (cf. Section 3.4). The cardinality of $\pi_{\text{path}(A)}\mathbf{R}_F = \pi_{\text{path}(A)}(Q(\mathbf{R}_E))$ can now be computed using known techniques for relational databases, or alternatively using factorisation-aware cardinalities and selectivities as mentioned above. The size of the f-representation of $Q(\mathbf{R}_E)$ over $\mathcal{T}$ is then estimated as $\sum_{A\in\mathcal{P}} |\pi_{\text{path}(A)}\mathbf{R}_F|$, where $\mathcal{P}$ is the projection list of Q, and the cost $e(f)$ of an f-plan f can be estimated as the sum of the size estimates for the intermediate and final f-trees created by f [Wyl12].

5.2.2 Space of F-plans for a Given Query

We consider a general [2] query with ordering, aggregates and selections of the form

$$Q = o_L\big(\varpi_{G;\alpha\leftarrow F}(\sigma_{A_1=B_1,\ldots,A_m=B_m,\varphi}(R_1 \times \cdots \times R_n))\big)$$

Since product operators are the cheapest operators to execute on factorisations and are only symbolic, we always execute them first: a product of n relations can be represented as a factorisation that is a product relational expression whose children are the n relations.

[2] This discussion can be extended to composed aggregation functions following our remarks from Section 5.1.5. Projections can be expressed using the aggregation operator.

Selections with constants expressed by the condition φ can also be evaluated in one traversal of this factorisation. The remaining query constructs can be implemented using further f-plan operators.

Let Q be a query without products or selections with constants, and let E be a factorisation over an f-tree $\mathcal{T}$. A sequence of operators S correctly implements the query Q on E if and only if it satisfies the following three conditions:

- For each selection condition $A_i = B_i$ in Q, S contains a selection operator that merges the equivalence classes of A_i and B_i as well as their nodes in its input f-tree. No selection operator in S merges nodes with attributes that are not equivalent in the selection of Q or in the original f-tree $\mathcal{T}$.
- The sequence S contains the aggregation operator $\gamma_{F(\mathcal{U})}$ for some subtree $\mathcal{U}$ whose set of attributes is $\mathcal{T} \setminus G$. It may be preceded by any number of aggregates $\gamma_{F(\mathcal{V})}$ with $\mathcal{V} \subseteq (\mathcal{T} \setminus G)$ to implement partial aggregation, as allowed by the composition rules of Proposition 5.5. There are no other aggregation operators. A renaming operator occurs after $\gamma_{F(\mathcal{U})}$ in S to rename $F(\mathcal{U})$ to α.
- The output f-tree of the last operator must satisfy the condition of Theorem 5.3 for the order-by list L.

These conditions present global requirements on an f-plan: they characterise which sequences of operators correctly execute a given query Q. Next we turn them into local requirements: at any point in the f-plan we characterise which single operator can be evaluated next so that we can still arrive at the result of Q. This is possible since at any stage of the f-plan, the f-tree encodes information about the previous operators in the f-plan as well as about the structure of the factorisation. The nodes of the f-tree encode information about the underlying relation (equalities and aggregates already performed), and the shape of the f-tree encodes information about how the relation is factorised.

Consider the scenario of executing a query Q on an input factorisation over the f-tree $\mathcal{T}$, and suppose we already executed a sequence S' of operators. Call an operator *permissible* if it is one of the following:

- A selection operator for any of the remaining selections $A_i = B_i$ to be executed. The merge operator is permissible if the attributes A_i and B_i are siblings in the f-tree, the

absorb operator is permissible if one of the attributes is a descendant of the other in the f-tree.

- Any aggregate operator $\gamma_{F(U)}$ with $U \subseteq (\mathcal{T} \setminus G)$ is permissible unless U contains an attribute A_i that is still to be equated with B_i. (Otherwise the equality $A_i = B_i$ could not be performed afterwards.)
- Any restructuring (swap) operator is permissible.

Proposition 5.6. *The sequence S' followed by the operator x can be extended to an f-plan of Q if and only if x is permissible.*

We can represent the space of all f-plans as a graph whose nodes are f-trees and whose edges are operators between them. An f-plan then corresponds to a path in the graph, and an f-plan for the query Q is a path to any f-tree satisfying the selection, aggregation, and order conditions. In the presence of a cost metric for individual operators, such as the one based on size bounds for factorisations of operator outputs, we can utilise Dijkstra's algorithm to find the minimum-cost f-plan executing the query Q. Proposition 5.6 characterises the outgoing edges for each node and allows us to construct the graph incrementally as it is explored.

5.2.3 Greedy Heuristic

The size of the space of all f-plans is exponential in the size of the query, and searching for the optimal f-plan becomes impractical even for moderately-sized queries. We propose a polynomial-time greedy heuristic algorithm for finding an f-plan for a given query Q and input f-tree $\mathcal{T}_0$:

Repeat

1. If there are any permissible selection operators, choose one involving a highest-placed node in the f-tree and execute it.
2. Else if there are any permissible aggregate operators $\gamma_{F(U)}$, choose one with maximal $\mathcal{U}$ and execute it.
3. Else if there still exists a condition $A_i = B_i$ such that A_i and B_i are not in the same node, calculate the cost for repeatedly pushing up (a) A_i, or (b) B_i, or (c) both A_i and

B_i, until A_i and B_i are siblings or one is an ancestor of the other. Pick the cheapest option and execute it.

4. Else if there is an attribute $A \in G$ with parent $B \notin G$, swap A with B.
5. Else if there is an attribute $A \in L$ with parent B such that B is not before A in L, swap A with B,
6. Else break.

After this algorithm terminates, all selection conditions have been evaluated in (1) possibly using the restructuring from (3), all attributes not in G have been aggregated in (2) possibly using the restructuring in (4), and the order condition is met because of (5). There may still be several aggregate attributes in the f-tree, the value of the final aggregate is the product (or min or max, depending on the aggregation function F) of these values. This can be calculated during enumeration.

If we require the result of the aggregate in a single attribute, we need to arrange all nodes dependent on the aggregation attributes into a single path. This can be achieved by repeatedly swapping them up:

7. Let P be the least common ancestor in $\mathcal{T}$ of all attributes of $\mathcal{T} \setminus G$. While P has a child with an atomic attribute R, swap P and R.

5.3 Experimental Evaluation

The query evaluation and optimisation techniques presented in Sections 5.1 and 5.2 are implemented in a main-memory query engine called FDB (Factorised DataBase). In this section we present an experimental evaluation of FDB against three relational engines: one home-bred in-memory (RDB) and two open-source engines (SQLite and PostgreSQL). Our main finding is that FDB clearly outperforms relational engines for data sets with many-to-many relationships, especially in settings where the input is factorised or the result can be output in factorised form. In particular, in our experiments we found that:

- The size of factorised query results is typically at most quadratic in the input size for random queries of up to eight relations and nine join conditions (Figure 5.7 right).

- Finding optimal f-trees for results of queries of up to eight relations and six join conditions takes under 0.1 seconds (Figure 5.7 left). Finding optimal f-plans for queries on factorised data is about an order of magnitude slower. In contrast, the greedy optimiser takes under 5 ms (Figure 5.9) without any significant loss in the succinctness of factorisation (Figure 5.8).

- For conjunctive queries on synthetically generated flat data, factorised query results are up to six orders of magnitude smaller than their flat equivalents and FDB outperforms RDB by up to four orders of magnitude (Figure 5.10 right). The gap increases with increasing data size following a power law (Figure 5.10 left). For the same workload SQLite performed about three times slower than RDB, and PostgreSQL performed three times slower than SQLite; both systems have additional overhead of fully functioning engines. The gap between RDB and SQLite increases with the number of joins (Figure 5.11 right) as RDB implements an optimised multi-way merge-sort join.

  This holds for uniform and Zipf data distributions.

- The evaluation of subsequent queries on input data representing query results has the same time performance gap, since the new input is more succinct as factorised representation than as flat relation (Figure 5.12).

- For data expressing one-to-many relationships, the performance gap is smaller, since even the flat result sizes for one-to-many joins only depend linearly on the input size and the possible gain brought by factorisation is less dramatic. For instance, in the TPC-H benchmark all joins are on keys and therefore the sizes of the join results do not exceed that of the relation with foreign keys. Factorised query results are still more succinct than their relational representations, but only by a constant factor bounded by the query size (Figure 5.11 left).

- The evaluation of aggregation queries on results of joins can be up to two orders of magnitude faster with FDB if the joins are presented as factorisations, compared to relational engines on flat relations (Figure 5.15). The performance gap increases with

increasing data size, following a power law (Figure 5.14). Another speedup of up to two orders of magnitude is achieved if FDB's output is factorised or if the query output is limited to first 10 tuples.

- FDB can outperform the relational engines SQLite and PostgreSQL by up to three orders of magnitude in evaluating aggregation queries on flat relations with flat output. This gap disappears if SQLite and PostgreSQL are instructed to use manually crafted query plans that push aggregates past joins (Figure 5.16).
- FDB can enumerate data in a variety of orderings with no or little restructuring, reusing the existing order present in the factorisation. Even simple queries with order-by clauses (such as ones that sort the input relation in a given order) can benefit from these techniques and speed up processing by a small factor. In cases where the output is limited to first 10 tuples, the more efficient restructuring becomes more prominent and the performance gap rises to up to two orders of magnitude (Figure 5.18).

Next we give some technical details of FDB's implementation, the competing engines, and the experimental setup, and then we report in on two sets of experiments, one using conjunctive queries and one extending them with aggregation, ordering and limit.

Implementation Details

FDB is implemented in C++ for execution in main memory. The parameter $s(\mathcal{T})$ is computed using the GLPK package v4.45 for solving linear programs. F-representations are stored as parse trees and the elements of each union are always in a sorted order. FDB implements the evaluation and optimisation algorithms described in previous sections with the following exceptions. For evaluation on flat data it computes the result f-representation using a multi-way merge-sort join algorithm [OZ12] similar to the one from Section 3.6, with the order of joins given by an optimal f-tree. FDB sorts its input relations before executing the merge-sort join; the sorting time is included in the reported evaluation time. The FDB optimiser defers projections until the end (we only focus on the impact of joins). We consider two modes of operation for FDB, by default it produces flat relational output

using the enumeration algorithm from Section 3.1, whereas **FDB f/o** produces a factorised output (e.g., for later use as a materialised view).

We also implement in C++ a lightweight in-memory engine **RDB** using flat relations, for the purpose of a fair comparison without the overhead of fully-functional database engines. RDB uses the same multi-way merge-sort join as FDB, though producing a flat result this time. Selections are performed by a simple linear scan of the relations, group-by is achieved by sorting (using a C++ STL sort) and aggregates by a linear scan on the computed groups.

Competing Engines

For comparison with practical real-world database engines we also used two popular open-source engines, the lightweight in-memory engine SQLite and the full-blown PostgreSQL database system with a client-server architecture.

The lightweight query engine **SQLite** 3.6.22 (3.7.7 in the second set of experiments) was tuned for main memory operation by turning off the journal mode and synchronisations and by instructing it to use in-memory temporary store. Similarly, we run PostgreSQL 9.1.8 (**PSQL**) with the following parameters: fsync, synchronous commit, full page writes and background writer are off, shared buffers, working memory and effective cache size increased to 12 GB. For PSQL we run each query three times and time the last repetition, for which internal tables are cached and queries are optimised for main memory. For all engines we report wall-clock times to execute the query plans; these times exclude importing the data from files on disk and writing the result to disk. Our measurements indicate that the disk I/O for SQLite is zero and zero or negligible for PSQL (always smaller than when reading input and writing output to disk, which increases execution time by at most 10%.)

All experiments were performed on an Intel® Xeon® X5650 Quad 2.67GHz 64bit processor with 32GB RAM running VMWare virtual machine with Linux 2.6.32/gcc4.4.5 (Linux 3.0.0/gcc4.6.1 in the second set of experiments).

5.3.1 Experiments with Join Queries

The flow of our experiments on join queries is as follows. We generate random data and queries, then repeat a number of times four optimisation and evaluation experiments and

report averages of optimisation time, execution time, representation sizes, and quality of produced f-plans.

We generate a schema of R relations and distribute uniformly A attributes over them. Each relation has a given number of tuples, each value is generated independently from $\{1, \ldots, M\}$ using a uniform or Zipf distribution. The queries are equality joins over all of these relations. We first generate an initial query Q with K non-redundant equality conditions and then a further query Q' with L additional non-redundant equality conditions.

For each generated initial query Q and database $\mathbf{D}$, we do the following. In Experiment 1, we run the FDB optimiser to find an optimal f-tree $\mathcal{T}$ for the query result and report the optimisation time and the value of the parameter $s(Q)$ that controls the size of the f-representation of $Q(\mathbf{D})$ over $\mathcal{T}$. In Experiment 3, we compute the result $Q(\mathbf{D})$ using RDB, SQLite, and PostgreSQL, and the factorised query result using FDB. We then report on both the evaluation time and size of the result as the number of its singletons; a singleton holds an 8 byte integer. We also unfold the factorised result of FDB into a flat relation for correctness check, and report the unfolding (or tuple enumeration) time.

In Experiments 2 and 4, we evaluate the second set of queries on top of the query results from Experiment 3. For each new query Q', we run the FDB optimiser to find an optimal f-plan to evaluate the query and the resulting f-tree of the query result. In Experiment 2, we report the optimisation time and cost of the found f-plans for both the exhaustive and greedy optimisation algorithms. Here, we consider the cost of the f-plan defined by the parameter $s(\cdot)$ of the intermediate and final f-trees; in our experiments, the alternative cost estimate $e(\cdot)$ would lead to very similar choices of optimal f-plans since our data generator treats attributes independently and introduces no dependencies or irregularities. In Experiment 4, we execute f-plans with FDB on the f-representations of the query results (and with RDB on the flat query results) computed in Experiment 3.

We also repeat Experiments 1 and 3 with TPC-H data and queries. We considered the conjunctive parts of TPC-H queries 2, 5, 7, 10 and compare the evaluation times of FDB, RDB, and SQLite on flat TPC-H databases of varying scale.

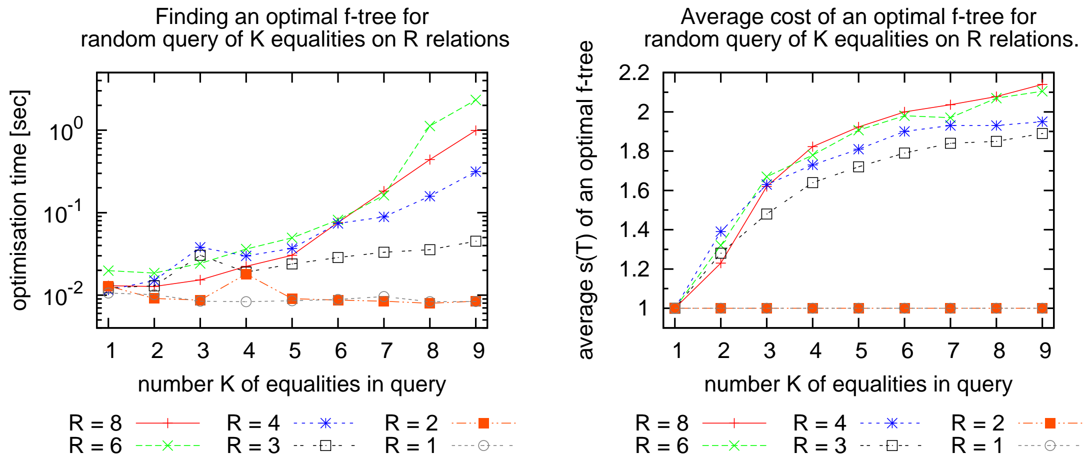


Figure 5.7: Experiment 1: Query optimisation on flat data, K equalities on R relations with $A = 40$ attributes.

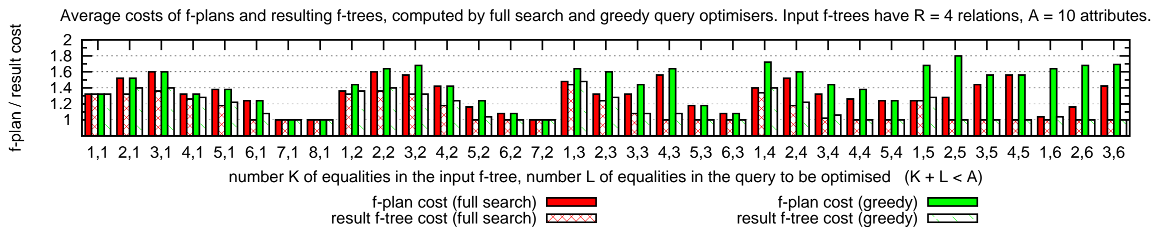


Figure 5.8: Experiment 2: Comparison of full-search and greedy query optimisers by the quality of found f-plans.

Experiment 1: Optimisation of Joins on Flat Data

The left plot in Figure 5.7 shows average times for optimising a query on flat data, and the right plot in Figure 5.7 shows average costs $s(\mathcal{T})$ for the chosen optimal f-tree $\mathcal{T}$ of the query result. For schemas with $A = 40$ attributes over $R = 1, \ldots, 8$ relations, we optimised queries of $K = 1, \ldots, 9$ equality selections. The cost $s(\mathcal{T})$ of an optimal f-tree $\mathcal{T}$ is always 1 for queries of up to two relations. For $R \geq 3$ and a sufficient number of joins we often get queries with optimal $s(\mathcal{T}) = 2$ and in very rare cases $s(\mathcal{T}) > 2$. This means that the sizes of f-representations for the query results are in most cases quadratic in the size of the input database even in the case of 9 equality selections on 8 relations. The optimiser searches a potentially exponentially large space of f-trees to find an optimal one, but runs under 1 second for queries with less than 8 joins on up to 8 relations.

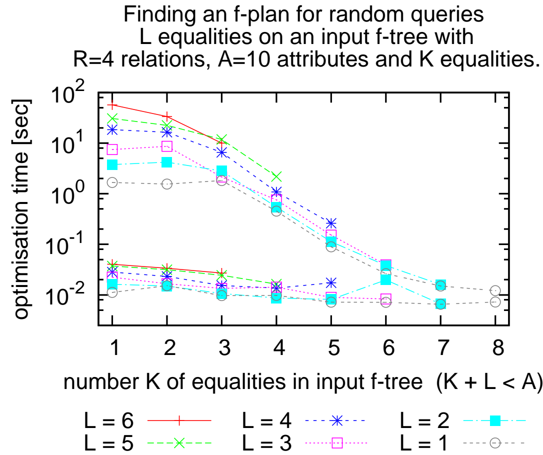


Figure 5.9: Experiment 2: Time performance of query optimisation on factorised data, L equalities on f-tree with K equalities. Full search (slower, top series) and greedy optimiser (faster, bottom series).

Experiment 2: Optimisation of Joins on Factorised Data

Figure 5.8 shows the behaviour of query optimisation for factorised data. It shows the costs of the computed f-plans as well as the costs of the f-tree of the result computed by the f-plans for our full-search and greedy optimisation algorithms.

The queries under consideration have $L \leq 6$ equality conditions on f-representations resulting after $K \leq 8$ equality conditions on $R = 4$ relations with $A = 10$ attributes. The greedy optimiser gives optimal or nearly optimal results in most cases (by comparison with the optimal outcome of full search). The exceptions are queries joining most attributes of an f-representation produced by a query with few joins (small K, large L). In all cases the average f-plan cost is between 1 and 2, which means that the f-plans take at most quadratic time for processing a join of up to 4 relations. The results show that for small queries (small L) the cost of the optimal f-plan is dominated by the cost of the final f-tree. As the number of equalities L grows, the result f-tree has less attribute classes and its cost is smaller than the cost of the f-plan, which is the maximum cost of intermediate f-trees needed to evaluate the query.

Figure 5.9 shows the execution times for both exhaustive (full search) and greedy optimisers. The search space of possible f-trees grows exponentially with the number L of equalities and also with the size of the input f-tree (i.e., with decreasing K). The perfor-

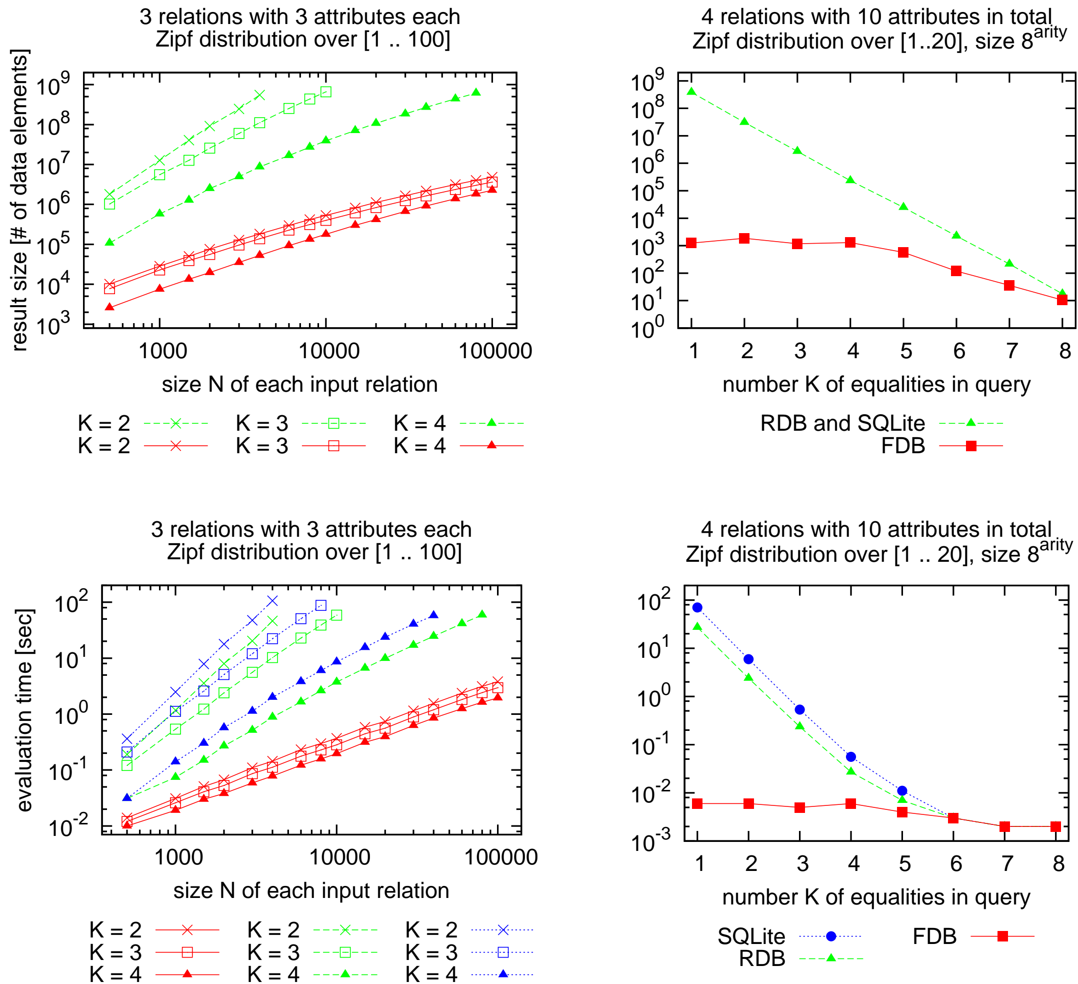


Figure 5.10: Experiment 3: Query evaluation on flat relational data: FDB in solid red, RDB in dashed green, SQLite in dotted blue. PostgreSQL is ca. 3 times slower than SQLite and not shown.

mance of the full-search algorithm is proportional to the size of the search space; we process about 1k f-trees/second. The greedy heuristic has running time polynomial in both K and L, and in our scenario is 2-3 orders of magnitude faster than full search.

Experiment 3: Evaluation of Joins on Flat Data

We compared the performance of FDB, RDB, SQLite, and PostgreSQL for query evaluation on flat input data. The left column of Figure 5.10 shows the result sizes and evaluation times for queries with up to four equality conditions on three ternary relations of increasing sizes, generated using a Zipf distribution over the range $[1, 100]$.

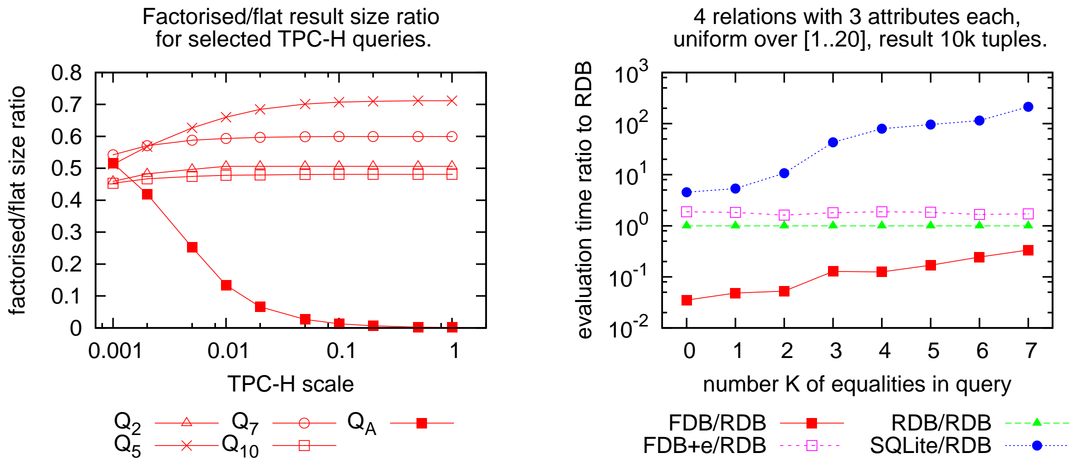


Figure 5.11: Experiment 3: Query evaluation on flat relational data. Ratio of factorised vs. flat result size (left). Ratio of query evaluation time vs. RDB: FDB in solid red, RDB in dashed green, SQLite in dotted blue, FDB with tuple enumeration in dashed pink. PostgreSQL is ca. 3 times slower than SQLite and not shown.

The size gap between factorised and relational results is largest for queries with fewer equality conditions, since the results are larger yet factorisable. The plots support the claim that the sizes are bounded by a power law, with a smaller exponent for FDB than for the relational engines.

The right column in Figure 5.10 considers queries with increasing number of equality conditions over two binary relations of size $8^2 = 64$ and two ternary relations of size $512 = 8^3$, whose values are drawn from a Zipf distribution over $[1, 20]$. Each equality condition in the query decreases the number of result tuples approximately by a factor of 20, which is exhibited in the flat result size produced by RDB. FDB factorises the up to 500M data values into less than 4k singletons for all considered queries. The execution time for all engines is approximately proportional to their result sizes except for the millisecond region, where constant overhead dominates. For all plots, results for uniformly distributed data exhibit the same trends as those for Zipf-distributed data and are not shown due to lack of space.

The right plot in Figure 5.11 considers a similar scenario, but the input size is increased with the number of equality conditions to keep the output size constant at about 10k tuples. We show the evaluation time ratio with respect to RDB. FDB outperforms RDB

by one to two orders of magnitude. As the number of equality conditions increases, $s(Q)$ increases (cf. Figure 5.7 left), the query result becomes less factorisable and the size and time performance gaps decrease. Evaluation using FDB followed by enumerating all tuples of the factorised result is only by a small constant factor slower than direct evaluation using RDB. In both cases, most of the evaluation time is spent on writing the result to memory. SQLite performs significantly worse than RDB (and FDB) for queries with many equality conditions since it only implements in-memory nested-loops joins while RDB implements a multi-way merge-sort join.

For TPC-H data and queries, the compression factor for factorised output (versus flat output) remains roughly constant for each query even with increasing database size (left plot in Figure 5.11). Since all joins are key-foreign key joins, the query result also has a primary key, and each distinct value of this key must appear in the factorisation: hence the factorisation cannot be asymptotically smaller than the result itself. This behaviour is contrasted by the query $Q_A = \text{Suppliers} \bowtie_{\text{Nation}} \text{Customers}$ whose join is many-to-many and the factorised/flat size ratio decreases inversely with database scale (reaches 1/800 for scale 1).

In summary, a combinatorial explosion of the number of tuples in the query result benefits FDB: the tuples are processed one-by-one by RDB (and the other relational engines), while FDB can represent them very succinctly in factorised form.

Experiment 4: Evaluation of Joins on Factorised Data

Figure 5.12 compares the performance of FDB and RDB for query evaluation on query results computed in Experiment 3 and with f-plans computed in Experiment 2. The behaviour of SQLite and PostgreSQL closely follows that of RDB.

FDB evaluates queries consisting of L selections on factorised representations. FDB uses the optimal f-plan found by the full-search optimiser. Additional experiments (not reported) reveal that the f-plans found by the greedy optimiser are up to 50% slower than the optimal f-plans.

RDB just evaluates a selection with a conjunction of L equality conditions on the attributes of the input relation. This can be done in one scan over the input relation. For

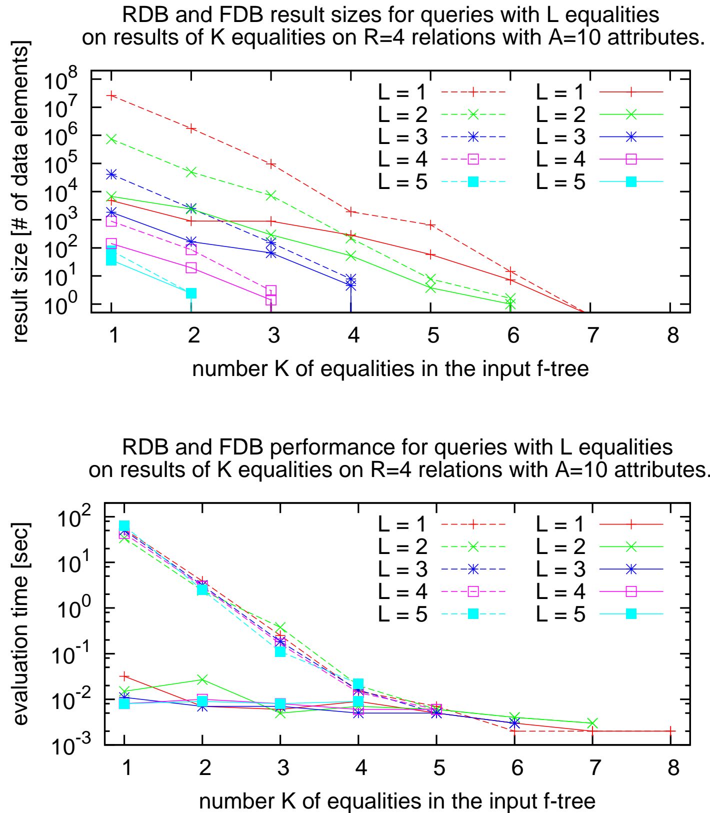


Figure 5.12: Experiment 4: Performance of FDB on factorised data (solid lines, bottom series in the right plot) and RDB on equivalent flat data (dashed lines, top series in the right plot). RDB needs one scan over the input, whereas FDB needs to restructure the factorised input. SQLite has similar performance to RDB.

FDB, the cost of the f-plan may be non-trivial: the more the f-plan needs to unfold the f-representation, the more expensive the evaluation becomes. Figure 5.12 suggests that FDB only unfolds the f-representations to a small extent. Similar to query evaluation on flat data, FDB shows up to 4 orders of magnitude improvement over RDB for both evaluation time and result size. The gap closes once the size of the input data decreases to about 1000 tuples and both FDB and RDB perform in under 0.1 seconds.

Experiments 2 and 4 show that using f-representations for data processing is sustainable in the sense that the quality of factorisations, in particular their compactness and sizes, does not decay with the number of operations on the data.

$$\underbrace{\begin{array}{l}R_1 = \text{Orders} \bowtie \text{Items} \bowtie \text{Packages}\\ Q_1 = \varpi_{\text{package, date, customer; sum(price)}}(R_1)\\ Q_2 = \varpi_{\text{customer; revenue} \leftarrow \text{sum(price)}}(R_1)\\ Q_3 = \varpi_{\text{date, package; sum(price)}}(R_1)\\ Q_4 = \varpi_{\text{package; sum(price)}}(R_1)\\ Q_5 = \varpi_{\text{sum(price)}}(R_1)\end{array}}_{\text{AGG}}$$

$$\underbrace{\begin{array}{l}Q_6 = o_{\text{customer}}(Q_2)\\ Q_7 = o_{\text{revenue}}(Q_2)\\ Q_8 = o_{\text{date, package}}(Q_3)\\ Q_9 = o_{\text{package, date}}(Q_3)\end{array}}_{\text{AGG+ORD}}$$

$$\underbrace{\begin{array}{l}R_2 = o_{\text{package, date, item}}(R_1)\\ R_3 = o_{\text{date, customer, package}}(\text{Orders})\\ Q_{10} = R_2\\ Q_{11} = o_{\text{package, item, date}}(R_2)\\ Q_{12} = o_{\text{date, package, item}}(R_2)\\ Q_{13} = o_{\text{customer, date, package, item}}(R_3)\end{array}}_{\text{ORD}}$$

Figure 5.13: Three sets of queries used in the experiments. Relations R_1, R_2, and R_3 are materialised.

5.3.2 Experimens with Aggregation and Ordering Queries

We use a synthetic dataset that consists of three relations: Orders, Items and Packages, modelling sales data of a business that offers different products (packages) composed or constructed from items, generalising the pizzeria database from Example 5.2. We control the sizes of the relations, and therefore the succinctness gap between factorised and flat results of queries on this dataset, using a scale parameter s.

In particular, the number of dates on which orders are placed is $800s$, the average number of order dates per customer is $80s$ and the average number of orders per order date is 2, both with a binomial distribution. There are $100\sqrt{s}$ different items and $40\sqrt{s}$ packages of $20\sqrt{s}$ items in average. Scaling generates database instances for which the size of the natural join of all three relations grows as s^4 while its factorisation over the following f-tree $\mathcal{T}$ grows as s^3. The values of constant multipliers were chosen to yield realistic sizes and ranges of all relations for scale factors ranging from 1 to 32. For the scale factor 32, the

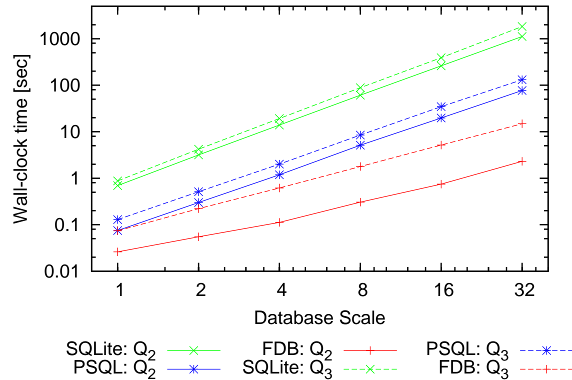
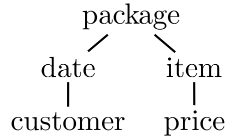


Figure 5.14: Experiment 5: The effect of dataset scale on performance of aggregates on materialised joins.

join has 280M tuples (1.4G singletons), while the factorisation has 4.2M singletons.



We use three sets of queries, cf. Figure 5.13. The set AGG consists of five queries with aggregates and group-by clauses. The set AGG+ORD consists of four queries with order-by clauses and aggregates. The set ORD consists of four order-by queries on top of sorted relations R_2 and R_3.

We next present five experiments whose focus is on performance of query evaluation. For a comprehensive experimental evaluation of query optimisation techniques for aggregation and ordering see [Koč12].

Experiment 5: Evaluating Aggregates on Materialised Joins

Figure 5.14 shows that FDB clearly outperforms SQLite and PSQL in our experiments on the factorised materialised view with AGG queries Q_2 and Q_3. The evaluation of both queries in FDB is done using partial aggregation and restructuring, similar to the query P in the introduction. The relational engines only perform grouping and aggregation on the materialised view; PSQL uses hashing while SQLite uses sorting to implement grouping. The performance gap widens as we increase the scale factor and raises from one order of

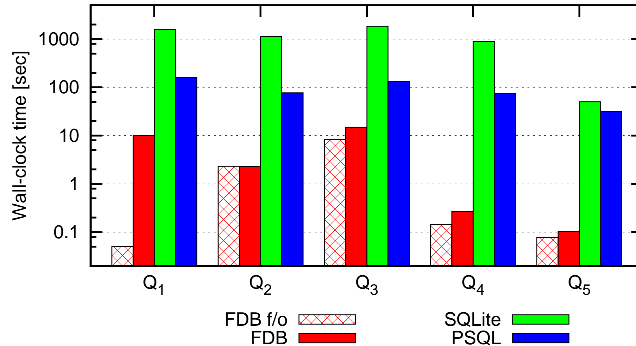


Figure 5.15: Experiment 5: Performance of AGG queries on the (factorised) materialised view R_1 at scale 32.

magnitude for scale 1 to two orders of magnitude for scale 32 when compared to PSQL; SQLite shows one additional order of magnitude gap. Notably, the reported timing for FDB includes the enumeration of result tuples, i.e., its output is flat as for relational engines.

Figure 5.15 looks closer at this scenario for scale 32 and AGG queries. When computing the result as factorised data, the performance gap further widens by two orders of magnitude for Q_1. This is the time needed to enumerate the tuples in the query result and is directly impacted by the cardinality of the result. The result of Q_1 is large as it consists of all joinable triples of packages, dates, and customers. The results of the other queries have less attributes and smaller sizes. For them, the enumeration time takes comparable or much less time as computing the factorised result.

In both figures, PSQL consistently outperforms SQLite with one exception: Q_5 in Figure 5.15. This query aggregates over the whole relation and SQLite's table scanning procedure is faster in this case.

Experiment 6: Evaluating Queries with Aggregates on Flat Data

Figure 5.16 presents FDB's performance for evaluating aggregate queries on flat, relational data (no materialised view this time) and producing flat output. Surprisingly, FDB outperforms SQLite and PSQL in their domain. A closer look revealed that both relational engines do not use partial aggregation and hence only consider sub-optimal query plans. With handcrafted plans that make use of partial and eager aggregation [YL95], all engines perform similarly. If we set for factorised output, then FDB f/o outperforms FDB in case

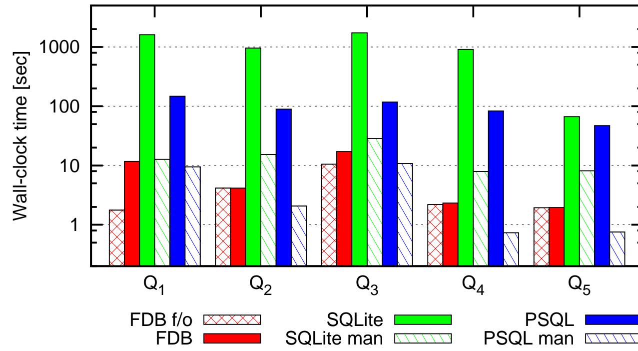


Figure 5.16: Experiment 6: Performance of AGG queries on flat input at scale 32. SQLite and PSQL using their own plans and also manually optimised plans (man).

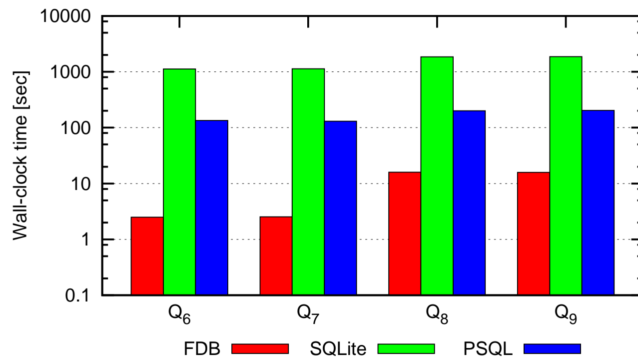


Figure 5.17: Experiment 7: Performance of AGG+ORD queries on the (factorised) materialised view R_1 at scale 32.

of large factorisable results (Q_1, Q_3); for small results, there is no difference to FDB as expected.

Experiment 7: Evaluating Aggregates with Ordering on Materialised Joins

Figure 5.17 shows that ordering only adds a small overhead to queries with aggregates. For FDB, the result of Q_2 is already ordered by customer, and thus the additional order-by clause in Q_6 is simply ignored by FDB. Re-ordering by the result of aggregation, as done in Q_7, does only add a marginal overhead, not visible in the plot due to the log scale on the y axis. This is explained by the relatively small result of Q_2. A similar situation is witnessed for the pair of queries Q_8 and Q_9 that apply different orders on the result to Q_3. Overall, it takes longer since Q_3 has a larger result than Q_2 (there are more pairs of date and package than customers). Following the pattern for queries Q_2 and Q_3 discussed in

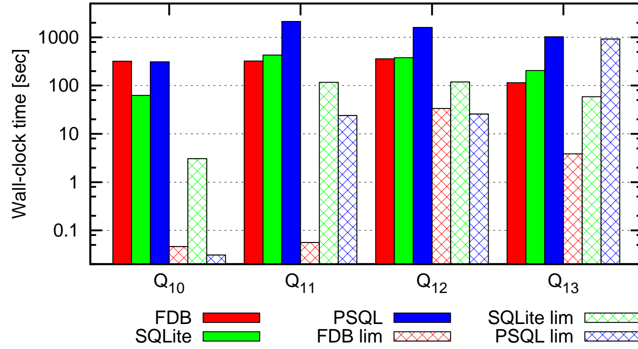


Figure 5.18: Experiment 8: Performance of ORD queries with and without a LIMIT 10 (lim) on the (factorised) materialised view R_1 at scale 16.

Experiment 1 and the lack of impact of ordering in this experiment, FDB outperforms the relational engines in this experiment, too.

Experiment 8: Partial Sorting via Restructuring of Factorisations

In this experiment we investigate the performance of the class ORD of order-by queries, and their versions asking for the first 10 tuples only, see Figure 5.18. FDB restructures the factorisation whenever necessary before enumeration in the required order. The time required for the restructuring is essentially captured by the execution time of the *limit* variant, since enumerating the first 10 tuples only adds a small constant overhead.

Query Q_{10} asks for a specific order on the materialised view R_2 that is already sorted in that order. FDB thus needs no restructuring of the view and enumerates the results. The relational engines need no additional sorting and only scan the relation R_1. This allows us to see how relation scanning compares against FDB enumeration: The latter is about the same speed as the PSQL scan. SQLite is faster than both; this is possibly because FDB is not optimised for string manipulation (by, e.g., hashing all strings in a relation). Returning the first 10 tuples only takes negligible time for both FDB and PSQL, SQLite takes longer.

Query Q_{11} asks for a slightly different order than the one existing in the relational input R_2. FDB need not do any work, since its factorisation of R_2 does already support this new order, as well as the original order. Simultaneous support for several orders is a key feature of FDB. The f-tree of the factorised materialised view R_1 is $\mathcal{T}_1$ from the introduction,

where we have package instead of pizza. It can therefore support both orders (package, date, item) and (package, item, date). In contrast, both relational engines need to sort the relation from scratch. Enumeration with FDB takes the same time as for Q_{10}, yet now the relational engines need more time to sort, PSQL about one order of magnitude more. Even to return the first 10 tuples requires major work for the relational engines, while FDB returns each of these tuples with constant delay and no precomputation.

Query Q_{12} asks for an order that is not already supported by the f-tree of the factorised materialised view. In this case, FDB needs restructuring before the enumeration: one swap between date and its parent node package is enough, and is still faster than sorting from scratch using either of the relational engines. Returning the first 10 tuples in the required order using PSQL takes the same time as the swap.

Query Q_{13} just sorts the relation R_3. Remarkably, even for sorting a (non-factorised) relation, FDB outperforms the relational engines since it only needs to partially re-sort the input. This is achieved by swapping the attributes date and customer. The factorisation constructed by FDB groups the relation by the first attribute in the sorting order, then by the second, and so on. The swap of date and customer re-groups by customer instead of date, yet the list of packages for each date and customer remains sorted.

Experiment 9: Measuring Overhead of Relational Engines.

PSQL and SQLite are full-fledged engines while FDB is not. To understand their overhead, we also ran all aggregation and ordering queries in the previous experiments also with our basic main-memory engine RDB. Where grouping is required, RDB first sorts the records (using C++ STL sort) and then performs aggregation in one scan. We found that RDB's performance is very close to SQLite's (which implements grouping by sorting using B-trees) and we therefore not explicitly show it in the plots. (This is in contrast to the experiments with join queries, where SQLite and PSQL perform significantly worse than RDB for queries with many equality conditions, since they implements in-memory nested-loop joins while RDB implements a multi-way merge-sort join.)

5.4 Related Work

There is a wealth of related work on succinct data representation, storage layout and schema design for database systems, and on aggregate processing techniques that are partly analogous to the treatment of aggregates in FDB.

Query Evaluation on Factorised and Nested Relations

There are several representation systems equivalent to factorised representations over f-trees, we review them in Section 3.7. Among these, the problem of query evaluation has been studied for compacted relations over compaction formats [BRS82], but only that of evaluating selection conditions with disjunctions in one sequential pass over the compacted relation.

The Verso project [AB86, Bid87] defines a complete set of operations on nested relations, including union, intersection, difference, join, projection and selections. Since the data model is not relational and a nested relation represents a collection of flat relations over the format skeleton as opposed to a single relation over the entire schema, the semantics of the operators is different than what one would define for f-representations. Verso selection benefits from the hierarchical data model by allowing selection predicates including "exists" clauses and disjunctions to be evaluated in a single pass over the nested relations. However, selection conditions equating two different attributes are not considered, and the Verso model only benefits joins that are implicitly present in the hierarchical schema.

Prior work on nested relations introduces nesting an unnesting operators [JS82], studies their interaction [FG85] and uses them to build nested relations from flat ones. However, the schema of the underlying data already contains an implicit hierarchical structure, and the nest and unnest operators are used to gradually add or remove the nesting. Our set of restructuring operators for f-representations f-trees can change the nesting order arbitrarily (with branching as allowed by the join dependencies in the represented relation).

Query Evaluation on Related Succinct Representation Systems

Evaluating relational algebra queries on world-set decompositions, which serve as succinct representations of sets of possible worlds and are equivalent to products of unions of single-

tons, has been studied in the context of incomplete databases [OKA08].

In a broader context, factorisation of sparse feature matrices were recently used to scale up machine learning algorithms [Ren13], performing operations on Boolean functions encoded by circuits or binary decision diagrams is considered in logical synthesis and circuit design [Bry92], and succinct representations of data that can speed up data processing or data analysis are of interest in various fields of computer science [Sim13].

Storage Layout: Vertical Partitioning and Column Stores

Columnar stores such as MonetDB [BMK99] and C-Store [BMK99, SAB+05] employ vertical partitioning of relational data and a number of optimisations similar in spirit to the techniques for factorised databases, targeting read-optimised scenarios similar to FDB [AMH08]. They partition relations into individual columns (or projections to small sets of attributes) and store the values of each column separately. Storing data values of the same attribute sequentially allows for easier bit-packing and more effective data compression, especially if the values are sorted. Increasing data locality within a single column improves cache performance, increases access speed, and decreases the amount of data that needs to be read for workloads heavy on single columns, such as aggregation.

Furthermore, columnar stores benefit from aggressively pushing selections and aggregates before joins, thus decreasing the size of the join once it is finally materialised. A per-column data layout allows optimisations such as referring to values by positions (row number) only, hash lookups to efficiently perform the actual join on filtered relations, or special optimisations for selection predicates that specify a contiguous range of values. Past research on columnar stores targets scenarios with star schemas and optimises for queries with selections and aggregation on the star join. The execution of FDB on such queries is in principle equivalent to the operation of columnar stores, and many optimisation techniques of column stores (hash join instead of a merge join, run-length encoding compression, or implementing range predicates) can be adapted for FDB. Unlike columnar stores, FDB replicates this behaviour on each level of branching in the f-tree, benefiting larger queries with multiple join attributes. A unique feature of FDB is the flexible on-the-fly modification of the schema hierarchy (f-tree) and corresponding restructuring of the data (f-representation)

to enable joins on previously unrelated attributes.

Storage Layout: Horizontal Partitioning

Horizontal partitioning is used for data distribution between multiple computers and can increase parallelisation of query processing. Partitioning-based automated physical database design [ANY04, GKP+10] has been proposed for maximising the performance of a particular workload. RodentStore is an adaptive and declarative storage system providing a high-level interface for describing the physical representation of data [CMWM09].

Distributed database systems such as Google's F1 [Sea13] and Microsoft's Cloud SQL Server [Bea11] achieve scalability by factorising databases to increase data locality for common access patterns: the tables are pre-joined and clustered following an f-tree (called tree schema) predefined by existing key-foreign key constraints. The data is then partitioned across servers into factorisation fragments rooted at different tuples of the root table. In these systems, tables are only pre-joined along one root-to-leaf path, but their partitioning techniques can potentially be extended to full f-representation semantics of FDB.

Data Compression

Data compression shares with factorisation the goal of compact data representation, as used e.g., for column compression [SAB+05, AMF06, GKP+10] and dictionary-based value compression in Oracle [PP03]. Such data compression schemes can benefit FDB and complement the structural compression brought by factorised representations. Structure-preserving compression of trees into directed acyclic graphs has been used to boost the performance of XML query evaluation [BGK03], and is similar to the transition between f-representations and d-representations (not currently considered by FDB).

Schema Design

Factorisation trees rely on join dependencies, which form the basis of the fifth normal form [RG03]. Database schema normalisation thus addresses the same redundancy in data as factorisations, but is static by nature. FDB can restructure factorisations dynamically as join dependencies change under query operations. Join dependencies were not used

previously as a basis for a representation system for relational data that can support query processing, although the admissibility of nesting structures for nested relations in presence of join and multi-valued dependencies has been studied [Del78, OY87].

Although not considered in this thesis, factorisations can go beyond the class defined by factorisation trees and the query processing techniques developed in this chapter can be adapted to more general factorisations.

Enumeration and Ordering

There has been no previous work on enumeration in sorted order on factorised data. The closest in spirit to ours is on polynomial-delay enumeration in sorted order for results to acyclic conjunctive queries [KS06].

Aggregate Processing

Our approach to partial aggregation before restructuring is intimately related to work by Yan and Larson on partially pushing *sum* and *count* aggregation past joins [YL95]. This is called eager aggregation and contrasts with lazy aggregation, which is applied after joins. While their technique relies on query rewriting, our approach conveys information about partial aggregation in the f-trees (schemas) of temporary results, and replaces elaborate rewrite rules by simple compositional rules for aggregation operators.

In relational processing, lazy aggregation can benefit from a selective join by processing a small join result. On the other hand, eager aggregation reduces the size of relations participating in a join and prevents unnecessary computation of combinations of values that are later aggregated anyway. FDB already avoids the explicit enumeration of such combinations by means of factorisation, whose size is at most the size of the join input, and much less for selective joins. FDB thus combines the advantages of both lazy and eager aggregation.

The technique of invisible joins [AMH08] used in columnar stores also combines lazy and eager aggregation plans for relational processing, and simulates aggregation on factorisations for star schemas. The framework of selection and aggregation operators for factorisations is more general, it can deal with longer chains of joins and aggregate efficiently over several

relations.

5.5 Conclusion and Directions for Future Work

In the final chapter of this dissertation we presented a relational database FDB that brings factorised representations to practical use, employing them to represent relations internally at the physical level. FDB exploits the representability of conjunctive query results by succinct factorisations with regular nesting structures, and fast algorithms to compute them directly from the input database, as studied in Chapter 3. The succinctness of factorisations is complemented by a range novel techniques for evaluation and optimisation of queries directly on factorised relations. FDB currently supports relational selections, projections, joins, aggregation, order-by and limit statements, and extensive experiments show dramatic improvement with respect to relational engines in representation size and query evaluation time for several scenarios. The flagship scenario where FDB's techniques bring the most improvement is the context of managing and querying materialised joins of many-to-many relationships.

We see two main directions for future work in the context of query evaluation on factorised representations. Firstly, there is scope for improvement in query processing on factorisations over f-trees: adding new functionality to FDB and optimising the existing techniques. Secondly, a new world of query evaluation challenges opens for every extension of the representation system beyond f-representations over f-trees: to d-representations over d-trees, factorisations with other regular nesting structures, or factorised representations in general.

Additional Functionality

Staying in the realm of f-representations of query results over f-trees, FDB could benefit from additional functionality such as the support of transactions, or indices to support faster data access, or incremental view maintenance, which is highly relevant to the materialised view scenarios where FDB achieves the greatest speedup.

Optimising Queries with Limit Statements

Although the presented techniques for evaluating queries with order-by and limit statements already achieve substantial speedup over relational engines in some scenarios, further speedup could be achieved by intertwining restructuring and enumeration. Currently FDB performs any restructuring strictly before the subsequent constant-delay enumeration, stopping after the first k tuples in the presence of a limit-k statement. More general methods utilising enumeration with larger than constant delay while needing less restructuring, thus saving time for small k, are subject to future work.

FDB with D-representations

One direction for future work is extending FDB to work with factorised representations with definitions (d-representations) as defined in Section 3.1. D-representations over d-trees can be even more succinct than f-representations over f-trees while still supporting fast tuple enumeration and fast computation of conjunctive query result directly as a d-representation (cf. Section 3.6). All further query evaluation methods described in this chapter must be adapted.

Instance-based Factorisation

An intriguing research direction is to go beyond factorisations defined by f-trees or d-trees and consider more general representations such as decision diagrams, unions of factorisations over f-trees or d-trees, arbitrary unions of products of unions of singletons, or other f-representations and d-representations of prescribed nesting structures. The latter class is inspired by rectangle coverings for algebraic factorisation of logic functions [Bra87], and existing methods that extract rectangles directly from the relation instance [GGM04, CBRB09] instead of relying on dependence information extracted from a query to factorise a query result. Inserting, deleting or modifying tuples in query results also breaks the regular nesting structures of f-trees and d-trees and calls for instance-based methods.

Bibliography

[AB86] Serge Abiteboul and Nicole Bidoit. Non first normal form relations: An algebra allowing data restructuring. *Journal of Computer and System Sciences*, 33(3):361–393, 1986.

[ADMT12] Yael Amsterdamer, Daniel Deutch, Tova Milo, and Val Tannen. On provenance minimization. *ACM Trans. Database Syst.*, 37(4):30, 2012.

[AGM08] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *Foundations of Computer Science (FOCS)*, 2008.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[AKO07] Lyublena Antova, Christoph Koch, and Dan Olteanu. "10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information". In *ICDE*, 2007.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*, pages 671–682. ACM, 2006.

[AMH08] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[Ami] J. Amilhastre. Complexity of minimum biclique decomposition of bipartite graphs.

[ANY04] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[BCC06] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2006.

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222, 2007.

[Bea11] Philip A. Bernstein and et al. Adapting Microsoft SQL Server for cloud computing. In *ICDE*, pages 1255–1263, 2011.

[BGK03] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *VLDB*, pages 141–152, 2003.

[Bid87] N. Bidoit. The verso algebra or how to answer queries with fewer joins. *Journal of Computer and System Sciences*, 35(3):321 – 364, 1987.

[BKOZ13] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. Aggregation and ordering in factorised databases. *PVLDB*, 2013.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[BOZ12a] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. Demonstration of the FDB query engine for factorised databases. *PVLDB*, 5(12):1950–1953, 2012.

[BOZ12b] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. FDB: A query engine for factorised relational databases. *PVLDB*, 5(11):1232–1243, 2012.

[Bra87] R. K. Brayton. "Factoring logic functions". *IBM J. Res. Develop.*, **31**, 1987.

[BRS82] François Bancilhon, Philippe Richard, and Michel Scholl. On line processing of compacted relations. In *VLDB*, pages 263–269, 1982.

[Bry92] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[BU08] David Buchfuhrer and Christopher Umans. The complexity of boolean formula minimization. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part I*, ICALP '08, pages 24–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[CBRB09] Loïc Cerf, Jérémy Besson, Céline Robardet, and Jean-François Boulicaut. Closed patterns meet n-ary relations. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3(1):3, 2009.

[CCT09] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[CDF96] Kevin Cattell, Michael J. Dinneen, and Michael R. Fellows. A simple linear-time algorithm for finding path-decompositions of small width. *Inf. Process. Lett.*, 57(4):197–203, 1996.

[CG10] Hubie Chen and Martin Grohe. Constraint satisfaction with succinctly specified relations. *Journal of Computer and System Sciences*, 76(8):847 – 860, 2010.

[CJR08] Adriane P Chapman, HV Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 993–1006. ACM, 2008.

[CMWM09] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. The case for RodentStore: An adaptive, declarative storage system. In *CIDR*, 2009.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[Del78] Claude Delobel. Normalization and hierarchical dependencies in the relational data model. *TODS*, 3(3):201–222, 1978.

[DS04] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875, 2004.

[DS07] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, 16(4), 2007.

[EHM+08] Alina Ene, William Horne, Nikola Milosavljevic, Prasad Rao, Robert Schreiber, and Robert E Tarjan. Fast exact and heuristic methods for role minimization problems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 1–10. ACM, 2008.

[FG85] Patrick C. Fischer and Dirk Van Gucht. Structure of relations satisfying certain families of dependencies. In Kurt Mehlhorn, editor, *STACS*, volume 182 of *Lecture Notes in Computer Science*, pages 131–142. Springer, 1985.

[FO11] Robert Fink and Dan Olteanu. On the Optimal Approximation of Queries Using Tractable Propositional Languages. In *ICDT*, 2011.

[GGL+09] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. V. den Bussche. Database query processing using finite cursor machines. *Theory of Computing Systems*, 44(4), 2009.

[GGM04] Floris Geerts, Bart Goethals, and Taneli Mielikäinen. Tiling databases. In *Discovery science*, pages 278–289. Springer, 2004.

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[GKT07] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Principles of Database Systems (PODS)*, 2007.

[GLS99] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Principles of Database Systems (PODS)*, 1999.

[GLS00] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural csp decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.

[GLS01] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of ACM*, 48, 2001.

[GM06] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *Symposium on Discrete Algorithms (SODA)*, 2006.

[Got12] Georg Gottlob. On minimal constraint networks. *Artif. Intell.*, 191-192:42–60, 2012.

[GPR06] Martin Charles Golumbic, Uri N. Peled, and Udi Rotics. Chain graphs have unbounded readability. Technical report, University of Haifa, 2006.

[Gre09] Todd J. Green. Containment of conjunctive queries on annotated relations. In *ICDT*, Saint Petersburg, Russia, March 2009.

[IL84] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of ACM*, 31(4), 1984.

[INV91] T. Imielinski, S. Naqvi, and K. Vadaparty. "Incomplete objects — a data model for design and planning applications". In *SIGMOD*, pages 288–297, 1991.

[IW94] Balakrishna R Iyer and David Wilhite. Data compression support in databases. In *VLDB*, volume 94, pages 695–704, 1994.

[JS82] G. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In *PODS*, pages 124–138, 1982.

[Ken83] William Kent. A simple guide to five normal forms in relational database theory. *Commun. ACM*, 26(2):120–125, February 1983.

[Koč12] Tomáš Kočiský. Queries with order-by clauses and aggregates on factorised relational data. Master's thesis, Oxford, 2012. http://www.tomas.kocisky.eu/masterthesis.pdf.

[KS93] Ephraim Korach and Nir Solel. Tree-width, path-width, and cutwidth. *Discrete Applied Mathematics*, 43(1):97 – 101, 1993.

[KS06] Benny Kimelfeld and Yehoshua Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In *NGITS*, pages 141–152, 2006.

[KS11] Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *Principles of Database Systems (PODS)*, 2011.

[Mak77] Akifumi Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *VLDB*, pages 447–453, 1977.

[Mar09] Dániel Marx. Approximating fractional hypertree width. In *Symposium on Discrete Algorithms (SODA)*, 2009.

[NNRR13] Hung Q. Ngo, Dung T. Nguyen, Christopher Re, and Atri Rudra. Towards instance optimal join algorithms for data in indexes. *CoRR*, abs/1302.0914, 2013.

[NPRR12] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[OH08] Dan Olteanu and Jiewen Huang. Using OBDDs for efficient query evaluation on probabilistic databases. In *SUM*, pages 326–340, 2008.

[OKA08] Dan Olteanu, Christoph Koch, and Lyublena Antova. World-set decompositions: Expressiveness and efficient algorithms. *TCS*, 403(2-3):265–284, 2008.

[OY87] Z. Meral Ozsoyoglu and Li Yan Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, 12:111–136, 1987.

[OZ11] Dan Olteanu and Jakub Závodný. Factorised representations of query results. Technical report, Oxford, April 2011. http://arxiv.org/abs/1104.0867.

[OZ12] Dan Olteanu and Jakub Závodný. Factorised representations of query results: size bounds and readability. In *ICDT*, pages 285–298. ACM, 2012.

[OZ13] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. Submitted for publication to TODS, July 2013.

[Pea89] Judea Pearl. *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. Morgan Kaufmann, 1989.

[PP03] M. Pöss and D Potapov. Data compression in Oracle. In *VLDB*, pages 937–947, 2003.

[Ren13] Steffen Rendle. Scaling factorization machines to relational data. *PVLDB*, 2013. to appear.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 2003.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. "C-Store: A Column-oriented DBMS". In *VLDB*, pages 553–564, 2005.

[SDG10] Prithviraj Sen, Amol Deshpande, and Lise Getoor. Read-once functions and query evaluation in probabilistic databases. *PVLDB*, 3(1):1068–1079, 2010.

[Sea13] Jeff Shute and et al. F1: A distributed SQL database that scales. *PVLDB*, 6, 2013. to appear.

[Sim13] Simons Institute for the Theory of Computing, UC Berkeley. *Workshop on "Succinct Data Representations and Applications"*, September 2013. http://simons.berkeley.edu/workshop_bigdata1.html.

[SORK11] Dan Suciu, Dan Olteanu, Chris Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[SU11] Edward R Scheinerman and Daniel H Ullman. *Fractional graph theory*. DoverPublications. com, 2011.

[SWW98] Gerd Stumme, Rudolf Wille, and Uta Wille. Conceptual knowledge discovery in databases using formal concept analysis methods. In Jan M. Żytkow and Mohamed Quafafou, editors, *Principles of Data Mining and Knowledge Discovery*, volume 1510 of *Lecture Notes in Computer Science*, pages 450–458. Springer Berlin Heidelberg, 1998.

[Vad01] SP Vadhan. The Complexity of Counting in Sparse, Regular, and Planar Graphs. *SIAM Journal on Computing*, 32(2), 2001.

[Vel12] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[Wyl12] Szymon Wyleżoł. Cost-based Query Optimisation for Factorised Relational Databases. Master's thesis, Oxford, 2012.

[YL95] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *VLDB*, pages 345–357, 1995.

Appendix A

Deferred Proofs

Proof of Lemma 2.1

Let Q be a query, $\mathbf{D}$ be a database, and $(Q_S, \mathbf{D}_S)$ be the S-restriction for a subset S of the set of head attributes of Q. We prove that $|\pi_S(Q(\mathbf{D}))| \leq |Q_S(\mathbf{D}_S)|$.

Intuitively, $\pi_S(Q(\mathbf{D}))$ performs all joins of Q and projects to S, while $Q_S(\mathbf{D}_S)$ first projects to S and performs only the joins on the attributes in S. Formally,

$$\begin{aligned}
|\pi_S(Q(\mathbf{D}))| &= |\pi_S(\pi_{\mathcal{P}}(\sigma_\psi(R_1 \times \cdots \times R_n)))(\mathbf{D})| \\
&= |\pi_S(\sigma_\psi(R_1 \times \cdots \times R_n))(\mathbf{D})| && (2) \\
&= |\pi_{S^*}(\sigma_\psi(R_1 \times \cdots \times R_n))(\mathbf{D})| && (3) \\
&\leq |\pi_{S^*}(\sigma_{\psi_S}(R_1 \times \cdots \times R_n))(\mathbf{D})| \\
&= |\sigma_{\psi_S}(\pi_{S^*}(R_1 \times \cdots \times R_n))(\mathbf{D})| \\
&= |\sigma_{\psi_S}(\pi_{S^*}R_1 \times \cdots \times \pi_{S^*}R_n)(\mathbf{D})| \\
&= |Q_S(\mathbf{D}_S)|,
\end{aligned}$$

where S^* denotes the transitive closure of S, equality (2) holds because $S \subseteq \mathcal{P}$, and equality (3) holds because each attribute in $S^* \setminus S$ is equivalent to some attribute to S.

Proof of Lemma 2.2

We prove that (1) if the attributes A and B are Q-dependent for a query Q, then there exists a database $\mathbf{D}$ for which A and B are dependent in the relation $Q(\mathbf{D})$, and (2) if A and B are not Q-dependent then for all databases $\mathbf{D}$, A and B are not dependent in the relation $Q(\mathbf{D})$.

Proof of statement (1). Let A and B be Q-dependent. Then there are attributes A' equivalent to A and B' equivalent to B, and a chain of relations $R_1, \dots, R_k$ in Q, such that A' is in the schema of R_1, B' is in the schema of R_k, and each successive R_i and R_{i+1} are joined on an attribute J_i that does not belong to the projection list $\mathcal{P}$ and neither does any equivalent attribute. Let $\mathcal{J}$ be the set of attributes equivalent to some J_i.

Consider a database $\mathbf{D}$ in which all attributes only attain the value 1 except for the attributes equivalent to A or B and the attributes from $\mathcal{J}$, which attain values 2 and 3. Let each relation contain all possible tuples over these domains which do not have both values 2 and 3. The result $Q(\mathbf{D})$ then contains two tuples; one for which $A = B = 2$ and one for which $A = B = 3$. All attributes not equivalent to A nor B have the value 1, because the attributes from $\mathcal{J}$ are projected out in Q. Then $Q(\mathbf{D})$ cannot be the natural join of two relations $\mathbf{R}_A \bowtie_{\mathcal{C}} \mathbf{R}_B$ for a set $\mathcal{C}$ that does not contain A nor B. Any such join would contain the tuples where $A = 2$ and $B = 3$, and vice versa. Therefore, A and B are dependent in $Q(\mathbf{D})$.

Proof of statement (2). Call two relations R_1 and R_k in a query Q *coupled* if there exists a chain of relations $R_1, \dots, R_k$ such that each successive R_i and R_{i+1} are joined on an attribute whose class is disjoint with the projection list $\mathcal{P}$. This partitions the relations of Q into equivalence classes of relations all coupled to each other. If two attributes A and B are not Q-dependent, then no attributes A' equivalent to A and B' equivalent to B may belong to coupled relations. Therefore, the relations of Q can be divided into two groups $\{R_i\}_{i \in I}$ and $\{R_j\}_{j \in J}$ such that no relation from one group is joined on projected-out attribute class with a relation in the other group, and all attributes equivalent to A belong to some R_i with $i \in I$, and all attributes equivalent to B belong to some R_j with $j \in J$. Let $\mathcal{I}$ be the set of attributes only equivalent to attributes from R_i, $\mathcal{J}$ the set of

attributes only equivalent to attributes from R_j, and $\mathcal{K}$ the set of attributes equivalent to both an attribute from R_i and to an attribute from R_j. Let ψ_I, ψ_J, ψ_K be the fragments of the selection condition ψ that equate attributes from I, J and K respectively, and let $\mathcal{P}_I, \mathcal{P}_J, \mathcal{P}_K$ be the partition of the projection list $\mathcal{P}$ to I, J, and K respectively. Then

$$
\begin{aligned}
Q(\mathbf{D}) &= \pi_{\mathcal{P}}\sigma_{\psi_K}(\sigma_{\psi_I}(\times_{i\in I}\mathbf{R}_i) \times \sigma_{\psi_J}(\times_{j\in J}\mathbf{R}_j)) \\
&= \pi_{\mathcal{P}_K}\sigma_{\psi_K}(\pi_{\mathcal{P}_I}\sigma_{\psi_I}(\times_{i\in I}\mathbf{R}_i) \times \pi_{\mathcal{P}_J}\sigma_{\psi_J}(\times_{j\in J}\mathbf{R}_j)) \\
&= \pi_{\mathcal{P}_K}\sigma_{\psi_K}(\mathbf{R}_I \times \mathbf{R}_J),
\end{aligned}
$$

where $\mathbf{R}_I = \pi_{\mathcal{P}_I}\sigma_{\psi_I}(\times_{i\in I}\mathbf{R}_i)$ and similarly for $\mathbf{R}_J$. Let $\mathbf{R}'_I$ be $\mathbf{R}_I$ extended with attributes from K such that ψ_K holds, and similarly for $\mathbf{R}_J$. Then $\sigma_{\psi_K}(\mathbf{R}_I \times \mathbf{R}_J) = \mathbf{R}'_I \bowtie_K \mathbf{R}'_J$, and since each class of attributes equivalent in ψ_K has at least one member in $\mathcal{P}_K$, $\mathbf{R}'_I \bowtie_K \mathbf{R}'_J = \mathbf{R}'_I \bowtie_{\mathcal{P}_K} \mathbf{R}'_J$. Therefore

$$
Q(\mathbf{D}) = \pi_{\mathcal{P}_K}\sigma_{\psi_K}(\mathbf{R}_I \times \mathbf{R}_J) = \pi_{\mathcal{P}_K}(\mathbf{R}'_I \bowtie_{\mathcal{P}_K} \mathbf{R}'_J) = (\pi_{\mathcal{P}_K}\mathbf{R}'_I) \bowtie_{\mathcal{P}_K} (\pi_{\mathcal{P}_K}\mathbf{R}'_J),
$$

where A is an attribute of $\pi_{\mathcal{P}_K}\mathbf{R}'_I$ and B an attribute of $\pi_{\mathcal{P}_K}\mathbf{R}'_J$. Therefore A and B are dependent in $Q(\mathbf{D})$ conditionally on $\mathcal{P}_K$.

Proof of Proposition 3.5

We first prove a technical lemma that characterises exactly which subexpressions $E(\mathbf{R}, \mathcal{X}, t)$ comprise the f-representation $\mathcal{T}(\mathbf{R})$ and which respective fragment of $\mathbf{R}$ each $E(\mathbf{R}, \mathcal{X}, t)$ represents.

Lemma A.1. *If $\mathcal{T}$ is valid for $\mathbf{R}$, then the recursive definition of $\mathcal{T}(\mathbf{R})$ invokes $E(\mathbf{R}, \mathcal{X}, t)$ exactly once for each subtree or forest $\mathcal{X}$ and each tuple $t \in \pi_{\text{anc}(\mathcal{X})}(\mathbf{R})$, and each resulting expression $E(\mathbf{R}, \mathcal{X}, t)$ is an f-representation of $\pi_{\mathcal{X}}(\sigma_{\text{anc}(\mathcal{X})=t}(\mathbf{R}))$ over $\mathcal{X}$.*

Proof. We first prove by bottom-up induction over $\mathcal{T}$ that each $E(\mathbf{R}, \mathcal{X}, t)$ is an f-representation of $\pi_{\mathcal{X}}(\sigma_{\text{anc}(\mathcal{X})=t}(\mathbf{R}))$ over $\mathcal{X}$.

- For any leaf $\mathcal{A}$, $E(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t) = \bigcup_{a\in A}\langle \mathcal{A}\!:\!a\rangle$ is an f-representation over $\mathcal{T}_{\mathcal{A}}$. Since the union ranges over $A = \pi_{A_1}(\sigma_{\text{anc}(\mathcal{A})=t}(\mathbf{R}))$, $\bigcup_{a\in A}\langle A_1\!:\!a\rangle = \pi_{A_1}(\sigma_{\text{anc}(\mathcal{A})=t}(\mathbf{R}))$, and

since all A_i have equal values in all tuples of $\mathbf{R}$, $[\![E(\mathbf{R}, \mathcal{T}_\mathcal{A}, t)]\!] = \bigcup_{a \in A} \langle \mathcal{A}\!:\!a \rangle = \pi_\mathcal{A}(\sigma_{\text{anc}(\mathcal{A})=t}(\mathbf{R}))$.

- For any subtree $\mathcal{T}_\mathcal{A}$ with non-empty forest $\mathcal{U}$ of children subtrees, $E(\mathbf{R}, \mathcal{U}, t \times \langle \mathcal{A}\!:\!a \rangle)$ is an f-representation over $\mathcal{U}$ by the induction hypothesis and hence $E(\mathbf{R}, \mathcal{T}_\mathcal{A}, t) = \bigcup_{a \in A} \langle \mathcal{A}\!:\!a \rangle \times E(\mathbf{R}, \mathcal{U}, t \times \langle \mathcal{A}\!:\!a \rangle)$ is an f-representation over $\mathcal{T}_\mathcal{A}$. Also,

$$\begin{aligned}
[\![E(\mathbf{R}, \mathcal{T}_\mathcal{A}, t)]\!] &= \textstyle\bigcup_{a \in A} \langle \mathcal{A}\!:\!a \rangle \times [\![E(\mathbf{R}, \mathcal{U}, t \times \langle \mathcal{A}\!:\!a \rangle)]\!] \\
&= \textstyle\bigcup_{a \in A} \langle \mathcal{A}\!:\!a \rangle \times \pi_\mathcal{U}(\sigma_{\text{anc}(\mathcal{U})=t \times \langle \mathcal{A}:a \rangle}(\mathbf{R})) \\
&= \textstyle\bigcup_{a \in A} \pi_{\mathcal{T}_\mathcal{A}}(\sigma_{\text{anc}(\mathcal{U})=t \times \langle \mathcal{A}:a \rangle}(\mathbf{R})) \\
&= \pi_{\mathcal{T}_\mathcal{A}}(\textstyle\bigcup_{a \in A} \sigma_{\text{anc}(\mathcal{U})=t \times \langle \mathcal{A}:a \rangle}(\mathbf{R})) \\
&= \pi_{\mathcal{T}_\mathcal{A}}(\textstyle\bigcup_{a \in A} \sigma_{\mathcal{A}=\langle \mathcal{A}:a \rangle}(\sigma_{\text{anc}(\mathcal{T}_\mathcal{A})=t}(\mathbf{R})) \\
&= \pi_{\mathcal{T}_\mathcal{A}}(\sigma_{\text{anc}(\mathcal{T}_\mathcal{A})=t}(\mathbf{R})),
\end{aligned}$$

where the second equality is by the induction hypothesis and the last one holds because all A_i have equal values in all tuples of $\mathbf{R}$.

- Finally, for any forest $\mathcal{U}$ of subtrees $\mathcal{T}_1, \ldots, \mathcal{T}_k$, each $E(\mathbf{R}, \mathcal{T}_i, t)$ is an f-representation over $\mathcal{T}_i$ and hence their product $E(\mathbf{R}, \mathcal{U}, t)$ is an f-representation over $\mathcal{U}$. Moreover, since $[\![E(\mathbf{R}, \mathcal{T}_i, t)]\!] = \pi_{\mathcal{T}_i}(\sigma_{\text{anc}(\mathcal{T}_i)=t}(\mathbf{R})) = \pi_{\mathcal{T}_i}(\pi_\mathcal{U}(\sigma_{\text{anc}(\mathcal{U})=t}(\mathbf{R})))$, and $\pi_\mathcal{U}(\sigma_{\text{anc}(\mathcal{U})=t}(\mathbf{R}))$ is the product of its projections to $\mathcal{T}_i$ by validity of $\mathcal{T}$ for $\mathbf{R}$, it follows that $[\![E(\mathbf{R}, \mathcal{U}, t)]\!] = [\![E(\mathbf{R}, \mathcal{T}_1, t)]\!] \times \cdots \times [\![E(\mathbf{R}, \mathcal{T}_k, t)]\!] = \pi_\mathcal{U}(\sigma_{\text{anc}(\mathcal{U})=t}(\mathbf{R}))$.

Now we prove by top-down induction over $\mathcal{T}$ that $\mathcal{T}(\mathbf{R}) = E(\mathbf{R}, \mathcal{T}, \langle\rangle)$ invokes $E(\mathbf{R}, \mathcal{X}, t)$ exactly once for each $t \in \pi_{\text{anc}(\mathcal{X})}(\mathbf{R})$. For $\mathcal{X} = \mathcal{T}$, this is true by definition.

- For any subtree $\mathcal{T}_\mathcal{A}$ with a non-empty forest of children $\mathcal{U}$, assume $\mathcal{T}(\mathbf{R})$ invokes $E(\mathbf{R}, \mathcal{T}_\mathcal{A}, t)$ exactly once for each $t \in \pi_{\text{anc}(\mathcal{T}_\mathcal{A})}(\mathbf{R})$. Any $E(\mathbf{R}, \mathcal{U}, t')$ is only ever invoked from $E(\mathbf{R}, \mathcal{T}_\mathcal{A}, \pi_{\text{anc}(\mathcal{T}_\mathcal{A})} t')$ and hence at most once, and exactly once for each $t' = t \times \langle \mathcal{A}\!:\!a \rangle$ for $t \in \pi_{\text{anc}(\mathcal{T}_\mathcal{A})}(\mathbf{R})$ and $a \in \pi_{A_1}(\sigma_{\text{anc}(\mathcal{T}_\mathcal{A})=t}(\mathbf{R}))$, i.e., for each $t' \in \pi_{\text{anc}(\mathcal{U})}(\mathbf{R})$.
- For any forest $\mathcal{U}$, if $\mathcal{T}(\mathbf{R})$ invokes $E(\mathbf{R}, \mathcal{U}, t) = E(\mathbf{R}, \mathcal{T}_1, t) \times \cdots \times E(\mathbf{R}, \mathcal{T}_k, t)$ exactly once for each $t \in \pi_{\text{anc}(\mathcal{U})}(\mathbf{R})$, then it also invokes each $E(\mathbf{R}, \mathcal{T}_i, t)$ once for each $t \in \pi_{\text{anc}(\mathcal{U})}(\mathbf{R}) = \pi_{\text{anc}(\mathcal{T}_i)}(\mathbf{R})$.

This concludes the proof of Lemma A.1. □

We next proceed to prove Proposition 3.5. We show that a relation $\mathbf{R}$ has an f-representation over an f-tree $\mathcal{T}$ iff $\mathcal{T}$ is valid for $\mathbf{R}$, and that any f-representation of $\mathbf{R}$ over $\mathcal{T}$ is equal to $\mathcal{T}(\mathbf{R})$ up to commutativity of product and union.

If $\mathcal{T}$ is valid for $\mathbf{R}$, $\mathcal{T}(\mathbf{R})$ is $E(\mathbf{R}, \mathcal{T}, \langle\rangle)$ from Definition 3.8, which by Lemma A.1 is an f-representation of $\pi_{\mathcal{T}}(\sigma_{\text{true}}(\mathbf{R})) = \mathbf{R}$ over $\mathcal{T}$.

Conversely, if $\mathbf{R}$ has an f-representation over $\mathcal{T}$ we show that it is equal to $\mathcal{T}(\mathbf{R})$ up to commutativity of product and union, and that $\mathcal{T}$ is valid for $\mathbf{R}$.

To show that any f-representation of $\mathbf{R}$ over $\mathcal{T}$ is equal to $\mathcal{T}(\mathbf{R})$, we first show by bottom-up induction over $\mathcal{T}$ that for any subtree or forest $\mathcal{X}$ and any tuple $t \in \pi_{\text{anc}(\mathcal{X})}(\mathbf{R})$, any f-representation of $\pi_{\mathcal{X}}(\sigma_{\text{anc}(\mathcal{X})=t}(\mathbf{R}))$ over $\mathcal{X}$ is equal to $E(\mathbf{R}, \mathcal{X}, t)$.

- For any leaf $\mathcal{A}$, any two f-representations over $\mathcal{T}_{\mathcal{A}}$ of the same relation are equal.
- For any subtree $\mathcal{T}_{\mathcal{A}}$ with a non-empty forest of children subtrees $\mathcal{U}$, any f-representation of $\mathbf{R}' = \pi_{\mathcal{T}_{\mathcal{A}}}(\sigma_{\text{anc}(\mathcal{T}_{\mathcal{A}})=t}(\mathbf{R}))$ over $\mathcal{T}_{\mathcal{A}}$ is by definition of the form $\bigcup_a \langle \mathcal{A}{:}a \rangle \times E_a$, where each E_a is an f-representation over $\mathcal{U}$. Since no E_a contains singletons of type A_1, the union must be over the values of $\pi_{A_1}\mathbf{R}' = \pi_{A_1}(\sigma_{\text{anc}(\mathcal{X})=t}(\mathbf{R}))$. Since the union is over distinct values of a, we have $[\![E_a]\!] = \pi_{\mathcal{U}}(\sigma_{\mathcal{A}=\langle \mathcal{A}a \rangle}(\mathbf{R}')) = \pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t\times\langle \mathcal{A}a \rangle}(\mathbf{R}))$. By the induction hypothesis, E_a must be equal to $E(\mathbf{R}, \mathcal{U}, t \times \langle \mathcal{A}{:}a \rangle)$.
- For any forest $\mathcal{U}$ of subtrees $\mathcal{T}_1, \ldots, \mathcal{T}_k$, any f-representation of $\mathbf{R}' = \pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t}(\mathbf{R}))$ over $\mathcal{U}$ is a product of f-representations E_i over $\mathcal{T}_i$. Therefore we must have $[\![E_i]\!] = \pi_{\mathcal{T}_i}(\mathbf{R}') = \pi_{\mathcal{T}_i}(\sigma_{\text{anc}(\mathcal{T}_i)}(\mathbf{R}))$ for each i, and by the induction hypothesis, we must have $E_i = E(\mathbf{R}, \mathcal{T}_i, t)$.

It follows that any f-representation of $\pi_{\mathcal{T}}(\sigma_{\text{anc}(\mathcal{T})=\langle\rangle}(\mathbf{R})) = \pi_{\mathcal{T}}(\sigma_{\text{true}}(\mathbf{R})) = \mathbf{R}$ over $\mathcal{T}$ is equal to $E(\mathbf{R}, \mathcal{T}, \langle\rangle)$. Moreover, the above shows that each $\pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t}(\mathbf{R}))$ is a product of its projections to $\mathcal{T}_i$, and it is immediate that for each node $\mathcal{A}$ the attributes in $\mathcal{A}$ have equal values in all tuples of $\mathbf{R}$, so $\mathcal{T}$ is valid for $\mathbf{R}$, $\mathcal{T}(\mathbf{R})$ is defined to be $E(\mathbf{R}, \mathcal{T}, \langle\rangle)$ and hence is also equal to the f-representation of $\mathbf{R}$.

Proof of Proposition 3.6

We prove that if $\mathcal{T}^{\uparrow}$ is valid for $\mathbf{R}$, then $\mathcal{T}^{\uparrow}(\mathbf{R})$ is a d-representation and its traversal is $\mathcal{T}(\mathbf{R})$.

First note that if $\mathcal{T}^{\uparrow}$ is valid for $\mathbf{R}$, then not only $\pi_{\mathcal{T}_{\mathcal{A}}}(\sigma_{\text{anc}(\mathcal{A})=t_1}(\mathbf{R})) = \pi_{\mathcal{T}_{\mathcal{A}}}(\sigma_{\text{anc}(\mathcal{A})=t_2}(\mathbf{R}))$ whenever $\pi_{\text{key}(\mathcal{A})}(t_1) = \pi_{\text{key}(\mathcal{A})}(t_2)$ for any subtree $\mathcal{T}_{\mathcal{A}}$, but also $\pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t_1}(\mathbf{R})) = \pi_{\mathcal{U}}(\sigma_{\text{anc}(\mathcal{U})=t_2}(\mathbf{R}))$ whenever $\pi_{\text{key}(\mathcal{U})}(t_1) = \pi_{\text{key}(\mathcal{U})}(t_2)$ for any forest $\mathcal{U}$. We can reformulate this independence condition to claim that for any forest or subtree $\mathcal{X}$, we have $\pi_{\mathcal{X}}(\sigma_{\text{key}(\mathcal{X})=t'}(\mathbf{R})) = \pi_{\mathcal{X}}(\sigma_{\text{anc}(\mathcal{X})=t}(\mathbf{R}))$ for $t' = \pi_{\text{key}(\mathcal{X})}t$.

If we label each $D(\mathbf{R}, \mathcal{X}, t)$ by the schema consisting of the attributes of $\mathcal{X}$, then $\mathcal{T}^{\uparrow}(\mathbf{R})$ is a d-representation with root $D(\mathbf{R}, \mathcal{T}, \langle\rangle)$. To prove that traversal$(\mathcal{T}^{\uparrow}(\mathbf{R})) = \mathcal{T}(\mathbf{R})$, we show by bottom-up structural induction over $\mathcal{T}$ that traversal$(D(\mathbf{R}, \mathcal{X}, t')) = E(\mathbf{R}, \mathcal{X}, t)$ for any $t \in \pi_{\text{anc}(\mathcal{X})}(\mathbf{R})$ such that $t' = \pi_{\text{key}(\mathcal{X})}t$.

- For any leaf $\mathcal{A}$ of $\mathcal{T}$, the traversal of $D(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t')$ is just $\bigcup_{a \in A}\langle \mathcal{A}{:}a\rangle$ where $A = \pi_{A_1}\sigma_{\text{key}(\mathcal{A})=t'}\mathbf{R} = \pi_{A_1}\sigma_{\text{anc}(\mathcal{A})=t}\mathbf{R}$, so the traversal is equal to $E(\mathbf{R}, \mathcal{X}, t)$.
- For subtree $\mathcal{T}_{\mathcal{A}}$ with a non-empty forest of children $\mathcal{U}$, the traversal of $D(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t')$ is $\bigcup_{a \in A}\langle \mathcal{A}{:}a\rangle \times \text{traversal}(D(\mathbf{R}, \mathcal{U}, \pi_{\text{key}(\mathcal{U})}(t' \times \langle \mathcal{A}{:}a\rangle)))$ where $A = \pi_{A_1}\sigma_{\text{key}(\mathcal{A})=t'}\mathbf{R} = \pi_{A_1}\sigma_{\text{anc}(\mathcal{A})=t}\mathbf{R}$, and by the induction hypothesis we have traversal$(D(\mathbf{R}, \mathcal{U}, \pi_{\text{key}(\mathcal{U})}(t' \times \langle \mathcal{A}{:}a\rangle))) = E(\mathbf{R}, \mathcal{U}, \pi_{\text{anc}(\mathcal{U})}(t \times \langle \mathcal{A}{:}a\rangle))$. This shows that traversal of $D(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t')$ is exactly $E(\mathbf{R}, \mathcal{T}_{\mathcal{A}}, t)$.
- For any forest $\mathcal{U}$ of subtrees $\mathcal{T}_1, \ldots, \mathcal{T}_k$, if $t \in \pi_{\text{anc}(\mathcal{U})}(\mathbf{R})$ and $t' = \pi_{\text{key}(\mathcal{U})}t$, then $\pi_{\text{key}(\mathcal{T}_i)}t' = \pi_{\text{key}(\mathcal{T}_i)}(\pi_{\text{anc}(\mathcal{T}_i)}t)$ and hence traversal$(D(\mathbf{R}, \mathcal{T}_i, \pi_{\text{key}(\mathcal{T}_i)}t') = E(\mathbf{R}, \mathcal{T}_i, \pi_{\text{anc}(\mathcal{T}_i)}t) = E(\mathbf{R}, \mathcal{T}_i, t)$, so traversal$(D(\mathbf{R}, \mathcal{U}, t')) = \times_i E(\mathbf{R}, \mathcal{T}_i, t) = E(\mathbf{R}, \mathcal{U}, t)$.

Proof of Proposition 3.8

Let Q be a conjunctive query and let $\mathcal{T}^{\uparrow}$ be a d-tree whose nodes are labelled by the equivalence classes of attributes of Q. We prove that $Q(\mathbf{D})$ has a d-representation over $\mathcal{T}^{\uparrow}$ for any database $\mathbf{D}$ iff $\mathcal{T}^{\uparrow}$ is valid for Q.

We first show that if $Q(\mathbf{D})$ has a d-representation over $\mathcal{T}^{\uparrow}$ for any $\mathbf{D}$, then $\mathcal{T}^{\uparrow}$ is valid for Q. If $Q(\mathbf{D})$ has a d-representation over $\mathcal{T}^{\uparrow}$ then $\mathcal{T}^{\uparrow}$ is valid for $Q(\mathbf{D})$ and hence also

$\mathcal{T}$ is valid for $Q(\mathbf{D})$. Since this holds for any $\mathbf{D}$, $\mathcal{T}$ is valid for Q by definition. Next we need to show that there is no node $\mathcal{B}$ with an ancestor $\mathcal{A} \not\subseteq \text{key}(\mathcal{B})$ and a descendant $\mathcal{C}$ that are Q-dependent. For any $c \in \pi_{\text{key}(\mathcal{B})}$, the fragment $\pi_{\mathcal{T}_\mathcal{B}}\sigma_{\text{key}(\mathcal{B})=c}Q(\mathbf{D})$ is represented by the expression $E'(\mathbf{R}, \mathcal{T}_\mathcal{B}, c)$, so $\sigma_{\text{key}(\mathcal{B})=c}Q(\mathbf{D})$ is a product of $\pi_{\mathcal{T}_\mathcal{B}}\sigma_{\text{key}(\mathcal{B})=c}Q(\mathbf{D})$ and $\pi_{\mathcal{T}\setminus\mathcal{T}_\mathcal{B}}\sigma_{\text{key}(\mathcal{B})=c}Q(\mathbf{D})$. Therefore, $Q(\mathbf{D}) = \pi_{\mathcal{T}_\mathcal{B}\cup\text{key}(\mathcal{B})}Q(\mathbf{D}) \bowtie \pi_{\mathcal{T}\setminus\mathcal{T}_\mathcal{B}}\sigma_{\text{key}(\mathcal{B})=c}Q(\mathbf{D})$, so any ancestor $\mathcal{A} \not\subseteq \text{key}(\mathcal{B})$ and any $\mathcal{C} \subseteq \mathcal{T}_\mathcal{B}$ are independent conditioned on $\text{key}(\mathcal{B})$. This holds for any $\mathbf{D}$, so $\mathcal{A}$ and $\mathcal{C}$ cannot be Q-dependent.

Conversely, suppose that $\mathcal{T}^\uparrow$ is valid for Q. Firstly, this means that $\mathcal{T}$ is valid for Q and hence $\mathcal{T}$ is valid for $Q(\mathbf{D})$ for any database $\mathbf{D}$. We need to show that $\mathcal{T}^\uparrow$ is also valid for any $Q(\mathbf{D})$, i.e., that $\pi_{\mathcal{T}_\mathcal{A}}(\sigma_{\text{anc}(\mathcal{A})=t_1}(Q(\mathbf{D}))) = \pi_{\mathcal{T}_\mathcal{A}}(\sigma_{\text{anc}(\mathcal{A})=t_2}(Q(\mathbf{D})))$ whenever $\pi_{\text{key}(\mathcal{A})}(t_1) = \pi_{\text{key}(\mathcal{A})}(t_2)$. Denoting $t := \pi_{\text{key}(\mathcal{A})}(t_1) = \pi_{\text{key}(\mathcal{A})}(t_2)$, $t'_1 := \pi_{\text{anc}(\mathcal{A})\setminus\text{key}(\mathcal{A})}t_1$ and $t'_2 := \pi_{\text{anc}(\mathcal{A})\setminus\text{key}(\mathcal{A})}t_2$, we have $\pi_{\mathcal{T}_\mathcal{A}}(\sigma_{\text{anc}(\mathcal{A})=t_1}(Q(\mathbf{D}))) = \pi_{\mathcal{T}_\mathcal{A}}(\sigma_{\text{anc}(\mathcal{A})\setminus\text{key}(\mathcal{A})=t'_1}(\sigma_{\text{key}(\mathcal{A})=t}(Q(\mathbf{D}))))$, and similarly for t_2. Since the nodes from $\mathcal{T}_\mathcal{A}$ are only dependent on nodes in $\mathcal{T}_\mathcal{A}$ and those in $\text{key}(\mathcal{A})$, the relation $\sigma_{\text{key}(\mathcal{A})=t}(Q(\mathbf{D}))$ is a product of its projection to $\mathcal{T}_\mathcal{A}$ and to its complement. The attributes in $\text{anc}(\mathcal{A}) \setminus \text{key}(\mathcal{A})$ belong to this complement, therefore $\pi_{\mathcal{T}_\mathcal{A}}(\sigma_{\text{anc}(\mathcal{A})\setminus\text{key}(\mathcal{A})=t'_1}(\sigma_{\text{key}(\mathcal{A})=t}(Q(\mathbf{D})))) = \pi_{\mathcal{T}_\mathcal{A}}(\sigma_{\text{anc}(\mathcal{A})\setminus\text{key}(\mathcal{A})=t'_2}(\sigma_{\text{key}(\mathcal{A})=t}(Q(\mathbf{D}))))$, and the result follows.

Detailed Proof of Theorem 3.6

We prove that for a fixed query Q, there exist arbitrarily large databases $\mathbf{D}$ for which any d-representation of the result $Q(\mathbf{D})$ over any d-tree has size $\Omega(|\mathbf{D}|^{s^\uparrow(Q)})$.

To prove this theorem, we need to slightly strengthen the requirements on the sizes of database examples witnessing the lower bounds in Lemma 3.7 and Theorem 3.5.

Lemma 3.7, adapted. For any equi-join query Q without self-joins, there exist constants b_Q, c_Q such that for any sufficiently large N, there exists a database $\mathbf{D}$ of size $N \leq |\mathbf{D}| \leq b_Q \cdot N$ such that $|Q(\mathbf{D})| \geq c_Q \cdot |\mathbf{D}|^{\rho^*(Q)}$.

Proof. We adapt the proof of Lemma 3.7 (Lemma 3 in [AGM08]). Denote by $a(R)$ the set of attribute classes of Q which contain an attribute of the relation R. The linear program for fractional independent set in Q, with variables $y_\mathcal{A}$ labelled by the attribute

classes of Q,

$$\begin{array}{lll} \text{maximising} & \sum_{\mathcal{A}} y_{\mathcal{A}} & \\ \text{subject to} & \sum_{\mathcal{A}\in a(R)} y_{\mathcal{A}} \leq 1 & \text{for all relations } R\text{, and} \\ & y_{\mathcal{A}} \geq 0 & \text{for all } \mathcal{A}, \end{array}$$

is dual to the linear program for the fractional edge cover of Q. By this duality, any optimal solution $\{y_{\mathcal{A}}\}$ to this linear program has cost $\sum_{\mathcal{A}} y_{\mathcal{A}} = \rho^*(Q)$.

For any N, construct a database $\mathbf{D}$ as follows. For each attribute class $\mathcal{A}$, let $N_{\mathcal{A}} = \lceil N^{y_{\mathcal{A}}} \rceil$. For N sufficiently large, we have $N_{\mathcal{A}} = \lceil N^{y_{\mathcal{A}}} \rceil \leq 2^{1/|\mathcal{S}|} N^{y_{\mathcal{A}}}$, where $\mathcal{S}$ is the schema of Q. We will assign values from $[N_{\mathcal{A}}] = \{1, \ldots, N_{\mathcal{A}}\}$ to the attributes in $\mathcal{A}$. For each relation R of Q, let the relation instance $\mathbf{R}$ contain all tuples t for which $t(A) \in [N_{\mathcal{A}}]$ for all attributes $\mathcal{A}$, but $t(A) = t(B)$ for any attributes A and B equated in Q (i.e., such that $\mathcal{A} = \mathcal{B}$). For each attribute class $\mathcal{A}$ in $a(R)$ there are $N_{\mathcal{A}}$ possible values of the attributes in $\mathcal{A}$, so the size of $\mathbf{R}$ is

$$|\mathbf{R}| = \prod_{\mathcal{A}\in a(R)} N_{\mathcal{A}} = \prod_{\mathcal{A}\in a(R)} \lceil N^{y_{\mathcal{A}}} \rceil \leq \prod_{\mathcal{A}\in a(R)} 2^{1/|\mathcal{S}|} N^{y_{\mathcal{A}}} \leq 2N^{\sum_{\mathcal{A}\in a(R)} y_{\mathcal{A}}} \leq 2N.$$

This implies that $|\mathbf{D}| \leq 2|Q| \cdot N$. However, for at least one relation R we have $\sum_{\mathcal{A}\in a(R)} y_{\mathcal{A}} = 1$ (otherwise we could increase any $y_{\mathcal{A}}$ to produce a better solution to the linear program), so $|\mathbf{D}| \geq N$.

Any tuple t in the result $Q(\mathbf{D})$ is given by its values for each attribute class $\mathcal{A}$, for which there are $N_{\mathcal{A}}$ possibilities, and any such combination of values gives a valid tuple in the output. The size of the output is thus

$$|Q(\mathbf{D})| = \prod_{\mathcal{A}} N_{\mathcal{A}} \geq \prod_{\mathcal{A}} N^{y_{\mathcal{A}}} = N^{\sum_{\mathcal{A}} y_{\mathcal{A}}} = N^{\rho^*(Q)} \geq (|\mathbf{D}|/(2|Q|))^{\rho^*(Q)}.$$

The claim follows by setting $b_Q = 2|Q|$ and $c_Q = 1/(2|Q|)^{\rho^*(Q)}$.

Theorem 3.5, adapted. For any query Q there exist constants b_Q, c_Q such that for any sufficiently large N and for any d-tree $\mathcal{T}^{\uparrow}$ of Q, there exists a database $\mathbf{D}_{\mathcal{T}^{\uparrow}}$ of size

$N \leq |\mathbf{D}_{\mathcal{T}^\uparrow}| \leq b_Q \cdot N$ such that $|\mathcal{T}^\uparrow(Q(\mathbf{D}_{\mathcal{T}^\uparrow}))| \geq c_Q \cdot |\mathbf{D}_{\mathcal{T}^\uparrow}|^{s^\uparrow(Q)}$.

Proof. Let $\mathcal{T}^\uparrow$ be any d-tree of Q and let A be an attribute for which $\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}) = s^\uparrow(\mathcal{T}^\uparrow) \geq s^\uparrow(Q)$. Applying the adapted version of Lemma 3.7 to $Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}$, there exist b_T, c_T such that for any sufficiently large N, there exists a $\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}$ with $N \leq |\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}| \leq b_T \cdot N$ and $|Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}})| \geq c_T \cdot |\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}|^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})}$. Moreover, by construction, its largest relation has size at least N. Then, by Lemma 3.6, there exists a database $\mathbf{D}$ with $N \leq |\mathbf{D}| \leq |\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}| \leq b_T \cdot N$ such that $|\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}(Q(\mathbf{D}))| \geq |Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}})|$. By Lemma 3.3, the number of A-singletons in $\mathcal{T}^\uparrow(Q(\mathbf{D}))$ is $|\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}(Q(\mathbf{D}))|$, and by the above,

$$|\pi_{\text{key}(\mathcal{A})\cup\mathcal{A}}(Q(\mathbf{D}))| \geq |Q_{\text{key}(\mathcal{A})\cup\mathcal{A}}(\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}})| \geq c_T \cdot |\mathbf{D}_{\text{key}(\mathcal{A})\cup\mathcal{A}}|^{\rho^*(Q_{\text{key}(\mathcal{A})\cup\mathcal{A}})} \geq c_T \cdot |\mathbf{D}|^{s^\uparrow(Q)}.$$

The claim follows by taking $\mathbf{D}_{\mathcal{T}^\uparrow}$ to be $\mathbf{D}$, b_Q to be the maximum b_T over all d-trees $\mathcal{T}^\uparrow$ of Q, and c_Q to be the minimum c_T over all d-trees of Q.

Proof of Theorem. Finally we prove the original claim of Theorem 3.6.

For any N sufficiently large let $\mathbf{D}_{\mathcal{T}^\uparrow}$ be as in the adapted version of Theorem 3.5. Construct the database $\mathbf{D}$ as a disjoint union of $\mathbf{D}_{\mathcal{T}^\uparrow}$ for all d-trees $\mathcal{T}^\uparrow$ of Q. (Label each data element in $\mathbf{D}_{\mathcal{T}^\uparrow}$ by $\mathcal{T}^\uparrow$, so that the corresponding relations of $\mathbf{D}_{\mathcal{T}^\uparrow}$ are disjoint, and for each relation symbol of Q construct a relation instance in $\mathbf{D}$ by taking a union of the corresponding relation instances in all $\mathbf{D}_{\mathcal{T}^\uparrow}$.) The result $Q(\mathbf{D})$ is a disjoint union of the results $Q(\mathbf{D}_{\mathcal{T}^\uparrow})$, and for any d-tree $\mathcal{T}^\uparrow$ the d-representation $\mathcal{T}^\uparrow(Q(\mathbf{D}))$ contains the d-representation $\mathcal{T}^\uparrow(Q(\mathbf{D}_{\mathcal{T}^\uparrow}))$, so its size is at least $c_Q \cdot |\mathbf{D}_{\mathcal{T}^\uparrow}|^{s^\uparrow(Q)}$. The size of each $\mathbf{D}_{\mathcal{T}^\uparrow}$ is at most $b_Q \cdot N$, so the size of $\mathbf{D}$ is at most $d \cdot b_Q \cdot N$, where d is the number of d-trees of Q. Therefore, for any d-tree $\mathcal{T}^\uparrow$ the d-representation $\mathcal{T}^\uparrow(Q(\mathbf{D}))$ has size at least $b_Q \cdot (|\mathbf{D}|/(c \cdot d))^{s^\uparrow(Q)}$, which is $\Omega(|\mathbf{D}|^{s^\uparrow(Q)})$ for a fixed Q.

Proof of Proposition 4.2

We prove that for any conjunctive query Q, any tuple t in its result $Q(\mathbf{D})$, and any provenance f-tree $\mathcal{T}$ of Q, the f-representation $\mathcal{T}(\varphi(t))$ as defined in Section 4.2 represents the provenance $\varphi(t)$.

We first prove the result for Boolean Q. A provenance f-tree of Q is any f-tree $\hat{T}$ of the equi-join $\hat{Q}$ of Q. The f-representation $\hat{\mathcal{T}}(\varphi(\langle\rangle))$ is obtained from the f-representation $\hat{\mathcal{T}}(\hat{Q}(\mathbf{D}))$ by dropping all data singletons.

Suppose we replace all data singletons by $\langle\rangle$, obtaining an f-representation E. Replacing all data singletons by $\langle\rangle$ and expanding into a flat f-representation is equivalent to expanding first and then replacing all data singletons by $\langle\rangle$, which is equivalent to projecting $\hat{\mathcal{T}}(\hat{Q}(D))$ to the annotation attributes $\mathcal{I}$. It follows that E indeed represents the provenance $\varphi(\langle\rangle) = \pi_{\mathcal{I}} Q(\mathbf{D})$. Also, the products of singletons of E are all distinct, since the products of singletons of the original f-representation correspond to distinct tuples for which the annotation attributes constitute a key. Finally note that the data singletons are never the direct subexpressions of a union: two different data singletons in a union would produce equal tuples with equal annotations, differing only in these singletons, which is impossible. Therefore data singletons are always in a product and replacing them by $\langle\rangle$ is equivalent to dropping them altogether.

Next we extend the proof to non-Boolean queries Q. A provenance f-tree of Q is any f-tree $\mathcal{T}$ of the equi-join $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$, and the f-representation $\mathcal{T}(\varphi(t))$ is obtained from the f-representation $\mathcal{T}(\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t))$ by dropping all data singletons. By the same reasoning as above, $\mathcal{T}(\varphi(t))$ then represents the relation obtained from $\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t)$ by dropping all data singletons, which is $\pi_{\mathcal{I}}\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t)$. Note that

$$\begin{aligned}
\pi_{\mathcal{I}}\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}(\mathbf{D}_t) &= \pi_{\mathcal{I}}(\sigma_{\psi_{\mathcal{S}\setminus\mathcal{P}^*}}(\pi_{\mathcal{S}\setminus\mathcal{P}^*}(\sigma_{\mathcal{P}^*=t}(\mathbf{R}_1)) \times \cdots \times \pi_{\mathcal{S}\setminus\mathcal{P}^*}(\sigma_{\mathcal{P}^*=t}(\mathbf{R}_n)))) \\
&= \pi_{\mathcal{I}}(\pi_{\mathcal{S}\setminus\mathcal{P}^*}(\sigma_{\psi_{\mathcal{S}\setminus\mathcal{P}^*}}(\sigma_{\mathcal{P}^*=t}(\mathbf{R}_1) \times \cdots \times \sigma_{\mathcal{P}^*=t}(\mathbf{R}_n)))) \\
&= \pi_{\mathcal{I}}(\sigma_{\psi_{\mathcal{S}\setminus\mathcal{P}^*}}(\sigma_{\mathcal{P}^*=t}(\mathbf{R}_1 \times \cdots \times \mathbf{R}_n))) \\
&= \pi_{\mathcal{I}}(\sigma_{\mathcal{P}=t}(\sigma_{\psi}(\mathbf{R}_1 \times \cdots \times \mathbf{R}_n))) \\
&= \pi_{\mathcal{I}}(\sigma_{\mathcal{P}=t}(Q(\mathbf{D}))),
\end{aligned}$$

which, by definition, is exactly the provenance $\varphi(t)$ in the result $Q(\mathbf{D})$.

Proof of Lemma 4.1

We prove that for any annotated query result $Q(\mathbf{D})$ and an f-tree $\mathcal{T}$ of Q, the number of occurrences of any $\underline{R}$-singleton in the f-representation $\mathcal{T}(Q(\mathbf{D}))$ is at most $|\mathbf{D}|^{\rho^*(Q_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R})}$.

Let R be a relation and let r be a tuple identifier for a tuple t in R. By Corollary 3.2, the number of occurrences of the singleton $\langle \underline{R}{:}r\rangle$ in the $\mathcal{T}(\mathbf{D})$ is $|\pi_{\text{anc}(\underline{R})}\sigma_{\underline{R}=r}\mathcal{T}(Q(\mathbf{D}))|$. However, since the attribute $\underline{R}$ is a key for R, this is equal to $|\pi_{\text{anc}(\underline{R})}\sigma_{\mathcal{S}_R=t}\mathcal{T}(Q(\mathbf{D}))|$, where the condition $\mathcal{S}_R = t$ means that we assign to each attribute of R its value in tuple t. We have

$$
\begin{aligned}
&|\pi_{\text{anc}(\underline{R})}\sigma_{\mathcal{S}_R=t}\mathcal{T}(Q(\mathbf{D}))| \\
=&|\pi_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}(\sigma_{\mathcal{S}_R=t}(Q(\mathbf{D})))| \\
\leq&|\pi_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}(Q(\mathbf{D}))| \\
=&|\big(\pi_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}(\sigma_{\varphi}(R_1\times\cdots\times R_n))\big)(\mathbf{D})| \\
\leq&|\big(\sigma_{\varphi_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}}(\pi_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}(R_1\times\cdots\times R_n))\big)(\mathbf{D})| \\
=&|Q_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}(\mathbf{D}_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R})|,
\end{aligned}
$$

where $Q_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}$ and $\mathbf{D}_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R}$ are the restrictions of Q and $\mathbf{D}$ to $\text{path}(\underline{R})\setminus\mathcal{S}^*_R$, the attributes from $\text{path}(\underline{R})$ that are not equivalent to an attribute in R.

By Lemma 3.4, $|Q_R(\mathbf{D}_R)|$ is at most $|\mathbf{D}_R|^{\rho^*(Q_R)}$, and the result follows by $|\mathbf{D}_R|\leq|\mathbf{D}|$.

Proof of Corollary 4.1

We show that for any database $\mathbf{D}$, any tuple $t\in Q(\mathbf{D})$, and any provenance f-tree $\mathcal{T}$ of Q, the factorisation $\mathcal{T}(\varphi(t))$ is at most read-$M\cdot|\mathbf{D}|^{r(\mathcal{T})}$. The readability of $\varphi(t)$ is at most $M\cdot|\mathbf{D}|^{r(Q)}$.

Let $Q'=\hat{Q}_{\mathcal{S}\setminus\mathcal{P}^*}$ be the provenance query of Q. By definition, the factorisation $\mathcal{T}(\varphi(t))$ is $\mathcal{T}(Q'(\mathbf{D}_t))$ where all data singletons have been projected away. For any relation symbol R, the number of R-singletons in $\mathcal{T}(\varphi(t))$ is therefore equal to the number of R-singletons in $\mathcal{T}(Q'(\mathbf{D}_t))$, which is at most $|\mathbf{D}_t|^{\rho^*(Q'_{\text{path}(\underline{R})\setminus\mathcal{S}^*_R})}\leq|\mathbf{D}_t|^{r(\mathcal{T})}\leq|\mathbf{D}|^{r(\mathcal{T})}$. In a repeating query, a given tuple identifier can appear in singletons from different relation names, representing

the same relation. If M is the maximum number of relation names in whose singletons an identifier can appear, then any identifier appears at most $M \cdot |\mathbf{D}|^{r(\mathcal{T})}$ times.

By considering $\mathcal{T}(\varphi(t))$ for a provenance f-tree of Q with minimal possible $r(\mathcal{T})$, we get a read-$M \cdot |\mathbf{D}|^{r(Q)}$ factorisation and hence the readability of $\varphi(t)$ is at most $M \cdot |\mathbf{D}|^{r(Q)}$.

Proof of Lemma 4.2

We show that for any query Q, any f-tree $\mathcal{T}$ of Q, and annotation attribute $\underline{R}$ in Q, there exist arbitrarily large annotated databases $\mathbf{D}$ such that the number of occurrences of some $\underline{R}$-singleton in the f-representation $\mathcal{T}(Q(\mathbf{D}))$ of the annotated query result $Q(\mathbf{D})$ is at least $(|\mathbf{D}|/|Q|)^{\rho^*(Q_{\text{path}(\underline{R}) \setminus \mathcal{S}^*_R})}$.

For brevity denote $P_R = \text{path}(\underline{R}) \setminus \mathcal{S}^*_R$ in this proof. We can repeat the proof of Lemma 3.8, except that we use Q_{P_R} instead of $Q_{\text{key}(\mathcal{A}) \cup \mathcal{A}}$, to produce a database $\mathbf{D}$ such that $|Q_{P_R}(\mathbf{D}_{P_R})| \geq (|\mathbf{D}|/|Q|)^{\rho^*(Q_{P_R})}$. Note that no annotation attributes participate in joins, so it is possible to arrange that all annotation attributes have distinct values in $\mathbf{D}$.

Since the relation R has no attributes in P_R, by construction the only tuple in R has the value 1 in all of its attributes. Call this tuple t and let r be its tuple identifier. By Corollary 3.2, the number of occurrences of the singleton $\langle \underline{R} : r \rangle$ in the f-representation $\mathcal{T}(\mathbf{D})$ is

$$
\begin{aligned}
& |\pi_{\text{anc}(\underline{R})}(\sigma_{\underline{R}=r}(Q(\mathbf{D})))| \\
&= |\pi_{P_R}(\sigma_{R=t}(Q(\mathbf{D})))| \\
&= |\big(\pi_{P_R}(\sigma_{R=t}(\sigma_\varphi(R_1 \times \cdots \times R_n)))\big)(\mathbf{D})| \\
&= |\big(\pi_{P_R}(\sigma_\varphi(R_1 \times \cdots \times R_n))\big)(\mathbf{D})| && (1) \\
&= |\big(\sigma_{\varphi_R}(\pi_{P_R}(R_1 \times \cdots \times R_n))\big)(\mathbf{D})| && (2) \\
&= |Q_R(\mathbf{D}_R)| \geq (|\mathbf{D}|/|Q|)^{\rho^*(Q_R)}.
\end{aligned}
$$

Equality (1) holds because $t = \langle 1, \ldots, 1 \rangle$ and each tuple from $(R_1 \times \cdots \times R_n)$ satisfies $\sigma_{R=t}$. Equality (2) holds because all values outside the attributes in P_R are equal, so the condition φ is equivalent to φ_R for our database $\mathbf{D}$.

Proof of Corollary 4.2

We show that for any provenance f-tree $\mathcal{T}$ of a query Q, there exist arbitrarily large databases $\mathbf{D}$ and tuple $t \in Q(\mathbf{D})$ such that the f-representation $\mathcal{T}(\varphi(t))$ is at least read-$(|\mathbf{D}|/|Q|)^{r(\mathcal{T})}$, which is at least read-$(|\mathbf{D}|/|Q|)^{r(Q)}$.

Let $\mathcal{T}$ be an f-tree for a query Q and consider the relation R with largest $\rho^*(Q_R)$, that is, $\rho^*(Q_R) = f(\mathcal{T})$. Lemma 4.2 states that for arbitrarily large databases $\mathbf{D}$, the f-representation $\mathcal{T}(\mathbf{D})$ contains at least $(|\mathbf{D}|/|Q|)^{\rho^*(Q_R)}$ occurrences of each singleton from R, so is at least read-$(|\mathbf{D}|/|Q|)^{r(\mathcal{T})}$. Since $r(\mathcal{T}) \leq r(Q)$, it is also at least read-$(|\mathbf{D}|/|Q|)^{r(Q)}$.

Proof of Theorem 4.2

Let Q be a conjunctive query. We show that

1. if Q is hierarchical, the readability of $Q(\mathbf{D})$ for any database $\mathbf{D}$ is bounded by a constant,
2. if Q is non-hierarchical, for any f-tree $\mathcal{T}$ of Q there exist arbitrarily large databases $\mathbf{D}$ such that $\mathcal{T}(\mathbf{D})$ is read-$\Omega(|\mathbf{D}|)$.

If Q is hierarchical then $r(Q) = 0$ by Proposition 4.3, and by Corollary 4.1, the readability of $Q(\mathbf{D})$ is at most $M \cdot |\mathbf{D}|^0 = M$.

If Q is non-hierarchical then $r(Q) > 0$ by Proposition 4.3. However, $r(Q) = \rho^*(Q_R)$ for some relation R in some f-tree $\mathcal{T}$, and if this is greater than zero, then it is at least 1. By Corollary 4.2, for any f-tree $\mathcal{T}$ of Q there exist arbitrarily large databases $\mathbf{D}$ for which the f-representation $\mathcal{T}(\mathbf{D})$ is at least read-$(|\mathbf{D}|/|Q|)^{r(Q)}$, which is $\Omega(|\mathbf{D}|)$ as $r(Q) \geq 1$.

Proof of Proposition 4.4

Let $C_N = \bigcup_{i,j=1;i\neq j}^{N} \langle r_i \rangle \langle s_j \rangle$. We show that the readability of C_N is $\Omega(\frac{\log N}{\log\log N})$ and $O(\log N)$.

We first prove the lower bound. Any f-representation of the relation $C_N = \sum_{i,j=1;i\neq j}^{N} \langle r_i \rangle \langle s_j \rangle$ is of the form $\bigcup_i R_i \times S_i$, where each R_i is a union of singletons of type R and each B_i is a union of singletons of type S. Represent each tuple $\langle r_i \rangle \langle s_j \rangle$ as an edge (a_i, b_j) in the bipartite graph $K_{N,N} = \{(a_i, b_j) \mid 1 \leq i, j \leq n\}$ and a union of such tuples as a set of the

corresponding edges. Each product $R_i \times S_i$ then represents a complete bipartite subgraph (called a *biclique*) of $K_{N,N}$, and the factorisation $\bigcup_i R_i \times S_i$ represents an edge-disjoint union of such bicliques.

If $\bigcup_i R_i \times S_i = C_N$, this union of bicliques must be equal to the graph represented by C_N, which is the crown graph $G_N = \{(a_i, b_j) \mid i \neq j\} \subset K_{N,N}$. The number of occurrences of a variable in the factorisation is the number of its bicliques containing the corresponding vertex of $K_{N,N}$.

The readability of C_N, denote it ρ_N, is then the smallest k for which G_N can be written as a union of edge-disjoint bicliques in such a manner that each its vertex is included in at most k of these bicliques.

Let M_k be the largest N for which G_N can be written as a union of bicliques in such a manner that each its vertex is included in at most k of these bicliques. Then ρ_N is the smallest k for which $M_k \geq N$.

Lemma A.2. *$M_1 = 2$.*

Proof. On one hand, $G_2 = K_{1,1} + K_{1,1}$ can be written as a vertex-disjoint union of bicliques. On the other hand, G_3 clearly cannot be written as a vertex-disjoint union of bicliques. ☐

Lemma A.3. *For $k > 1$, $M_k < k^2 M_{k-1}$.*

Proof. First introduce some notation: for a set A of vertices $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$ of G_N, all coming from one partition, by $\overline{A}$ we will denote the *opposite set* $\{b_{i_1}, b_{i_2}, \ldots, b_{i_k}\}$. Similarly we define $\overline{B}$ for a set B of vertices from the other partition.

Let $k > 1$, $N = k^2 M_{k-1}$, and suppose that G_N can be written as a union of bicliques such that each vertex is contained in at most k of them.

Consider one such collection $\mathcal{C}$ of bicliques. The vertex a_1 is contained in at most k bicliques $\{A_i \times B_i\}_i \subseteq \mathcal{C}$, and the vertex b_1 is contained in at most k bicliques $\{A'_i \times B'_i\}_i \subseteq \mathcal{C}$. Since $\bigcup \mathcal{C} = G_n$, we must have

$$\bigcup_i B_i = \{b_2, \ldots, b_N\} \qquad \text{and} \qquad \bigcup_i A'_i = \{a_2, \ldots, a_N\}.$$

Since $|\bigcup_i B_i| = N - 1 = k^2 M_{k-1} - 1$, and since $k > 1$, by the pigeonhole principle there exists

some B_j such that $|B_j| \geq kM_{k-1}$. But $\bigcup_i A'_i = \{a_2, \ldots, a_N\}$ implies $\bigcup_i \overline{A'_i} = \{b_2, \ldots, b_N\} \supseteq B_j$, so there exists some A'_i such that $|\overline{A'_i} \cap B_j| \geq M_{k-1}$. Denote $A = (A'_i \cap \overline{B_j})$. This means that $A \subseteq A'_i$ and $\overline{A} \subseteq B_j$. And $|A| \geq M_{k-1}$.

Now consider the collection $\mathcal{C}$ restricted to $A \times \overline{A}$, i.e.

$$\mathcal{D} = \mathcal{C} \restriction_{A \times \overline{A}} = \{(X \times Y) \cap (A \times \overline{A}) \mid X \times Y \in \mathcal{C}\}.$$

This is still a collection of bicliques, and it covers the graph induced by $A \times \overline{A}$, which is in fact isomorphic to $G_{|A|}$. Since $|A| \geq M_{k-1}$, at least one vertex v of this subgraph is contained in at least k bicliques in $\mathcal{D}$. Since all bicliques in $\mathcal{D}$ are restrictions of bicliques in $\mathcal{C}$, v is also included in the corresponding bicliques from $\mathcal{C}$. However, v is also included in the biclique $A_j \times B_j$ or $A'_i \times B'_i$ (depending on the partition it is in). But since $A_j \cap A = \emptyset$ and $\overline{A} \cap B'_i = \emptyset$, the restrictions of these two bicliques to $A \times \overline{A}$ are empty, and thus neither of them is one of our original k bicliques containing v. Therefore, v is in fact included in at least $k+1$ bicliques from $\mathcal{C}$, which is a contradiction to our assumption. $\square$

Corollary A.1. *For $k \geq 1$, $M_k \leq 2(k!)^2$.*

Lemma A.4. *With $k = \frac{\log N}{\log \log N}$, we have $2(k!)^2 < N$ for large enough N.*

Corollary A.2. $\rho_N = \Omega(\frac{\log N}{\log \log N})$.

For the upper bound on readability, we prove the following lemma.

Lemma A.5. *For any $N > 1$, $\rho_N \leq \rho_{\lceil N/2 \rceil + 1}$.*

Proof. Write C_N as

$$\begin{aligned} C_N = & \textstyle\bigcup_{i,j=1, i\neq j}^{\lceil N/2 \rceil} \langle r_i \rangle \langle s_j \rangle \cup \\ & \cup \textstyle\bigcup_{i,j=\lceil N/2 \rceil+1, i\neq j}^{N} \langle r_i \rangle \langle s_j \rangle \cup \\ & \cup (\textstyle\bigcup_{i=1}^{\lceil N/2 \rceil} \langle r_i \rangle) \times (\textstyle\sum_{j=\lceil N/2 \rceil+1}^{N} \langle s_j \rangle) \cup \\ & \cup (\textstyle\bigcup_{i=\lceil N/2 \rceil+1}^{N} \langle r_i \rangle) \times (\textstyle\sum_{j=1}^{\lceil N/2 \rceil} \langle s_j \rangle). \end{aligned}$$

The first two unions are equivalent to $q_{\lceil N/2 \rceil}$ and $q_{\lfloor N/2 \rfloor}$ respectively, so they are both equivalent to at most read-$\rho_{\lceil N/2 \rceil}$ expressions, but they contain different singletons. In the rest of

the expression, each singleton appears at most once. After replacing the first two unions by equivalent read-$\rho_{\lceil N/2 \rceil}$ expressions, the whole expression becomes read-$(\rho_{\lceil N/2 \rceil} + 1)$. This completes the proof. □

Corollary A.3. *By induction,* $\rho_N = O(\log N)$.

Proof of Theorem 5.2

Let $\mathcal{T}$ be an f-tree and G a set of group-by attributes. We prove that it is possible to enumerate the tuples of $[\![E]\!]$ for any factorisation E over $\mathcal{T}$ in groups with equal values of G with constant delay if and only if each attribute of G is either a root in $\mathcal{T}$ or a child of another attribute of G.

For the *if* direction of the proof, we use the *if* direction of Theorem 5.3. If the condition on the f-tree $\mathcal{T}$ is satisfied, then the nodes of $\mathcal{T}$ with attributes from G form a subtree of $\mathcal{T}$. Let O be any order of the attributes G that is their topological order in this subtree. Then each attribute X in O is either a root or its parent is an attribute from G and hence appearing in O before X. By Theorem 5.3 the tuples in $[\![E]\!]$ can be enumerated with constant delay sorted by O, and hence grouped by G.

For the *only if* direction, suppose the condition on the f-tree $\mathcal{T}$ is not satisfied, and let A be an attribute in G which is not a root and its parent B is not in G. We show that no algorithm can enumerate tuples of any f-representations over $\mathcal{T}$ grouped by G in constant delay. In particular, consider any algorithm for enumeration of tuples grouped by G, and let C be any constant. We show that there exist relations with at least two tuples for which the algorithm cannot output more than one tuple in C steps.

Let n be a natural number and let $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ be a function. Consider a relation $\mathbf{R}_f$ with n tuples, where the ith tuple has value i in attribute B, value $f(i)$ in attribute A, and a fixed constant value c in all other attributes. The fragment of the f-representation of $\mathbf{R}_f$ over the subtree rooted at B is of the form

$$\langle B\!:\!1\rangle \times (\langle A\!:\!f(1)\rangle \times E_1) \cup \langle B\!:\!2\rangle \times (\langle A\!:\!f(2)\rangle \times E_2) \cup \ldots \cup \langle B\!:\!n\rangle \times (\langle A\!:\!f(n)\rangle \times E_n).$$

In C steps, any algorithm can access at most C of the singletons $\langle A\!:\!f(i)\rangle$. If $n > C$ and

all accessed values $f(i)$ are distinct (say, $f(i) = i$), then after C steps, the cases when all remaining values $f(j)$ are distinct from the accessed values $f(i)$, or are all equal to one of the accessed values, are indistinguishable. The algorithm cannot output two tuples with equal values of A (as all tuples may have distinct values of A), nor tuples with distinct values of A (as one of the yet unaccessed A-singletons may carry the same value as the first output tuple).

Proof of Theorem 5.3

Let $\mathcal{T}$ be an f-tree and O a list of order-by attributes. We prove that it is possible to enumerate the tuples of $[\![E]\!]$ for any factorisation E over $\mathcal{T}$ with constant delay in sorted lexicographic order by O if and only if each attribute X of O is either a root in $\mathcal{T}$ or a child of an attribute appearing before X in O.

For the *if* direction, suppose that the condition on the list O and f-tree $\mathcal{T}$ is satisfied. We show how to reuse the algorithm from Proposition 3.3 to enumerate the tuples of any f-representation over $\mathcal{T}$ in lexicographic order by O. A basic building block of the algorithm is a depth-first search of the given f-representation E that follows only one child of each union node, and constructs a list $\mathcal{L}$ of nodes visited in pre-order. We alter the search to explore non-singleton nodes first and then singleton nodes in order of O. (A naive implementation using a linear-time priority queue runs in $O(|\mathcal{S}|^2)$, which is still constant in data complexity.) Since O is consistent with a topological order of $\mathcal{T}$, the singletons in the list $\mathcal{L}$ are ordered by O (and all singletons of attributes in O are before the singletons of attributes not in O). Since the list $\mathcal{L}$ becomes lexicographically greater in each iteration of step (3) of the algorithm, the enumerated tuples are sorted in lexicographical order by O. The delay the enumeration remains constant in data complexity ($O(|\mathcal{S}|^2)$ in combined complexity).

For the *only if* direction we use the *only if* direction of Theorem 5.2. Suppose an attribute A in the list is not a root in $\mathcal{T}$ and its parent B is either not in O or appears in O after A. Let G be the set of attributes in O before, and including, A. Then A is an attribute in G that is not a root in $\mathcal{T}$ and its parent is not in G. By Theorem 5.2, it is not possible to enumerate the tuples of arbitrary factorisations over $\mathcal{T}$ grouped by G. Since G is a prefix O, a sorted lexicographic order by O also groups the tuples by G, so it is also

not possible to enumerate the tuples of arbitrary factorisations over $\mathcal{T}$ sorted by O.