

Updates in Factorised Databases



Laura Draghici

St Hugh's College

Dr. Dan Olteanu

Supervisor

Dissertation submitted in partial fulfilment of the degree of Master of Science in
Computer Science, Department of Computer Science, University of Oxford

September 2013

Abstract

Factorised databases are relational databases that use compact factorised representations at the physical layer to reduce data redundancy and boost query performance. Succinctness is achieved by algebraic factorisation using distributivity of product over union and commutativity of product and union. Factorised databases use the so-called factorisation trees to define the structure of their representations. This thesis studies the problem of updates (insertion/value modification/deletion) in the context of such databases.

We study two types of insertions. We first discuss an algorithm that inserts a tuple into a factorised representation. We also propose and experimentally compare three algorithms for bulk insertion which, in the context of factorised databases, is defined as the construction of a factorised representation for the relation consisting of the tuples to be inserted.

We provide two approaches for value modifications and deletions on factorised databases. The first approach is supported without restructuring for a given update statement by a rather restricted class of factorisation trees and generates as a result a single representation. The second approach described is supported without restructuring by a wider class of factorisation trees, but generates a result that consists of a union of f -representations over the same f -tree.

Acknowledgements

I would like to express my most sincere gratitude to my supervisor Dr. Dan Olteanu for his constant guidance and for the many hours we spent discussing about the project. I would also like to thank Jakub Závodný for participating in our discussions and helping me with experiments.

I thank my parents for their unconditional love and support, especially in the difficult moments of writing this thesis.

Finally, many thanks to Paul Baltescu and Liana Baltescu for their unending support and encouragement throughout the year.

Contents

1	Introduction	1
1.1	Motivation and Challenges	2
1.2	Contributions	5
1.3	Outline	6
2	Preliminaries	7
2.1	Factorised Representations	7
2.2	Factorisation Trees	8
2.2.1	Factorisation Trees for Relations	8
2.2.2	Factorisation Trees for Queries	9
2.3	Size Bounds For Factorised Representations	11
2.4	Restructuring Operators	12
2.4.1	The Push-up Operator	12
2.4.2	The Swap Operator	12
2.4.3	Selection Operators	13
3	Insertions	15
3.1	Efficient Insertions on Factorised Representations	15
3.1.1	Nesting Structures Supporting Efficient Insertions	15
3.1.2	Incremental Factorisation Maintenance	17
3.2	Bulk Insertions	20
3.2.1	TF Algorithm: Factorisation over Tree Nesting Structures	20
3.2.2	PF Algorithm: Factorisation over Path Nesting Structures	21
3.2.3	RCF Algorithm: Factorisation using Rectangle Coverings	26
3.2.4	Comparison of Bulk Insertion Algorithms	32
3.2.5	Experimental Evaluation	36

4	Value Modifications	43
4.1	Nesting Structures Supporting Efficient Value Modifications	44
4.1.1	The Case of a Result Consisting of One Factorised Representation	44
4.1.2	The Case of a Result Consisting of a Union of Factorised Representations over the Same Factorisation Tree	48
4.2	Supporting Value Modifications by Restructuring	60
4.3	Experimental Evaluation	61
5	Deletions	65
5.1	Nesting Structures Supporting Efficient Deletions	66
5.1.1	The Case of a Result Consisting of One Factorised Representation	66
5.1.2	The Case of a Result Consisting of a Union of Factorised Representations over the Same Factorisation Tree	68
5.2	Supporting Deletions by Restructuring	71
5.3	Experimental Evaluation	72
6	Implementation	74
6.1	Bulk Insertions	74
6.2	Value Modifications and Deletions	77
7	Related Work	79
8	Conclusions and Future Work	81
8.1	Conclusions	81
8.2	Future Work	82
	Bibliography	84

Chapter 1

Introduction

Relational databases are present at the core of many systems that need to store, process and retrieve information. Often, these systems need to handle large amounts of information. The database system they use should not only be able to store this information, but it also needs to offer fast techniques to process the data it stores in order to meet the requirements of the systems using it.

Factorised representations of relational data offer a solution to such requirements. A factorised representation is a more compact representation of a relation that can be obtained by algebraic factorisation using distributivity of product over union and commutativity of product and union. The main idea behind a factorised representation is that the relation it represents has some hidden structure. By exploiting this structure, we can construct factorised representations that are, in some cases, exponentially more succinct than the relations they represent. Such hidden structures are present, for example, in relations that satisfy join dependencies. Results of conjunctive queries also present such structures that are independent of the database and can always be inferred from the queries only. The structures of factorised representations are defined by the so-called factorisation trees – unordered rooted forests of trees.

Factorised representations are relational algebra expressions that use the union and Cartesian product symbols and unary relations with a single value called singletons. These three types can also be used to represent any relation as a union of products of singletons, where each product of singletons represents a single tuple of the relation.

Example 1.1. Figure 1.1 shows relation R_0 and its representation as a union of products of singletons, denoted by P_0 . The same figure also shows a factorised representation E_0 of R_0 and the factorisation tree \mathcal{T}_0 defining the structure of E_0 . Notice that E_0 can be obtained from P_0 by factoring out either singletons, or unions of singletons. For the first two products of singletons we factor out $\langle A : 1 \rangle \times \langle B : 2 \rangle \times \langle C : 3 \rangle$. For the last four singletons, the

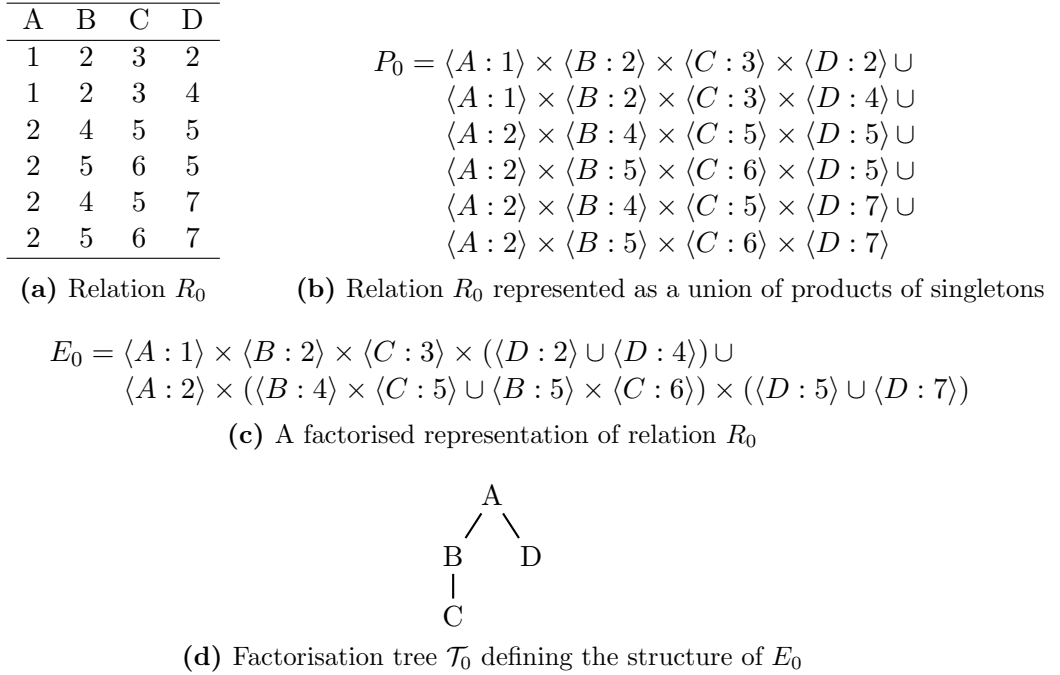


Figure 1.1: Relation R_0 , its representation as a product of singletons P_0 , a factorised representation E_0 of R_0 and the f-tree \mathcal{T}_0 defining the structure of E_0

factorisation is performed in several steps: we factor out $\langle A : 2 \rangle \times \langle D : 3 \rangle$ for the first two products in this group, then we factor out $\langle A : 2 \rangle \times \langle D : 7 \rangle$ for the last two products in the group. Finally, we can factor out $\langle A : 2 \rangle \times (\langle B : 4 \rangle \times \langle C : 5 \rangle \cup \langle B : 5 \rangle \times \langle C : 6 \rangle)$.

Notice also how E_0 follows the structure of \mathcal{T}_0 . The root of \mathcal{T}_0 is labelled by attribute A . Similarly, E_0 has a union of A -singletons as its top-most union. The root of \mathcal{T}_0 is the parent of two subtrees, imposing in any factorised representation following its structure a product between representations over these subtrees. E_0 satisfies this constraint and each of its A -singletons is in a product with a representation over the path (B, C) and another representation over a tree consisting only of node D . \square

1.1 Motivation and Challenges

Factorised representations lie at the foundation of a new kind of relational database system that represents relations more succinctly than traditional relational database systems. Previous work [3] introduced FDB, an in-memory query engine for factorised databases and showed that for select-project-join queries this engine can outperform other relational engines such as SQLite and Postgre-SQL by orders of magnitude. More recently, the framework has been extended to support queries with order-by clauses and aggregates [5] and results have shown that a similar performance gap between FDB and other relational engines holds in these cases too. In addition to evaluation techniques, the engine was also

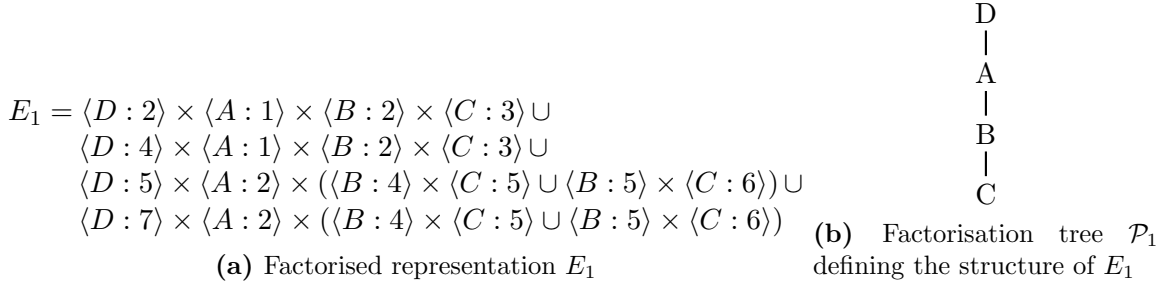


Figure 1.2: Factorised representation E_1 representing relation R_0 from figure 1.1a over factorisation tree \mathcal{P}_1

extended with optimisation techniques that take into consideration not only the time to compute the result representation, but also the size of the result representation.

There is no previous work, however, on updates on factorised representations, without which we could not have a fully usable relational database system. This thesis aims to cover this missing part of the current framework by giving a detailed theoretical insight into the problems that insertion, value modification and deletion statements raise in the context of factorised representations, as well as propose algorithms that perform these types of updates.

The main problem that the task of updates on factorised representations raises is that some update statements cannot be performed directly on the existing factorised representation. We will use the factorisation tree that defines the structure of the factorised representation to be updated to decide whether the representation supports the given statement or not. If the statement is not supported by the existing representation, we will first need to restructure it such that the new structure of the representation supports the given statement.

Example 1.2. Consider the following insertion statement on factorised representation E_0 shown in Figure 1.1c:

$$I_1 = \text{Insert into } E_0 \text{ values (A:1, B:2, C:3, D:5)}$$

This insertion statement is supported by E_0 without restructuring. The result representation is

$$\langle A : 1 \rangle \times \langle B : 2 \rangle \times \langle C : 3 \rangle \times (\langle D : 2 \rangle \cup \langle D : 4 \rangle \cup \langle D : 5 \rangle) \cup$$

$$\langle A : 2 \rangle \times (\langle B : 4 \rangle \times \langle C : 5 \rangle \cup \langle B : 5 \rangle \times \langle C : 6 \rangle) \times (\langle D : 5 \rangle \cup \langle D : 7 \rangle)$$

Consider also the following insertion statement:

$$I_2 = \text{Insert into } E_0 \text{ values (A:2, B:3, C:5, D:8)}$$

This statement cannot be performed without restructuring representation E_0 . To preserve the current structure of E_0 defined by \mathcal{T}_0 , each A -singleton should be in a product with a representation over path (B, C) and a union of D -singletons. After inserting the product of singletons $\langle A : 2 \rangle \times \langle B : 3 \rangle \times \langle C : 5 \rangle \times \langle D : 8 \rangle$, such a product cannot be built for singleton $\langle A : 2 \rangle$.

Figure 1.2a shows factorised representation E_1 obtained from E_0 by restructuring. The factorisation tree \mathcal{P}_1 that defines the structure of E_1 is also shown in Figure 1.2b. Unlike E_0 , E_1 supports the insertion of tuple $(A:2, B:3, C:5, D:8)$ without any other additional restructuring step. To perform the insertion, we simply need to add the product of singletons $\langle D : 8 \rangle \times \langle A : 2 \rangle \times \langle B : 3 \rangle \times \langle C : 5 \rangle$ to the top-most union of E_1 . Notice though that in order to be able to perform I_2 , we had to restructure E_0 to a representation that has a larger size.

Consider now the following value modification statement on representation E_0 :

$$U_1 = \text{Update } E_0 \text{ set } D = D+1 \text{ where } D = A+3$$

This value modification statement can be performed directly on E_0 . Notice that each D -singleton is in a product with exactly one A -singleton, so the condition of U_1 can be checked for each D -singleton before updating its value. The result representation is

$$\begin{aligned} &\langle A : 1 \rangle \times \langle B : 2 \rangle \times \langle C : 3 \rangle \times (\langle D : 2 \rangle \cup \langle D : 5 \rangle) \cup \\ &\langle A : 2 \rangle \times (\langle B : 4 \rangle \times \langle C : 5 \rangle \cup \langle B : 5 \rangle \times \langle C : 6 \rangle) \times (\langle D : 6 \rangle \cup \langle D : 7 \rangle) \end{aligned}$$

Consider also the value modification statement shown below:

$$U_2 = \text{Update } E_0 \text{ set } A = 3 \text{ where } D = 7$$

This is another example of an update statement that cannot be performed without restructuring. The problem here is that singleton $\langle A : 2 \rangle$ is in a product with a union of D -singletons. One of the singletons in this union satisfies the condition, while the other does not satisfy it. The update of singleton $\langle A : 2 \rangle$ would produce an incorrect representation, since this would correspond to an update of attribute A for both singletons $\langle D : 5 \rangle$ and $\langle D : 7 \rangle$. We can perform this update statement if we restructure representation E_0 to E_1 .

Finally, let's also look at a deletion statement:

$$D_1 = \text{Delete from } E_0 \text{ where } C = 6 \text{ and } D = 7$$

Again, this statement cannot be performed without a prior restructuring step. Notice that singletons $\langle C : 6 \rangle$ and $\langle D : 7 \rangle$ are placed inside different unions that are found in a product. Removing the product $\langle B : 5 \rangle \times \langle C : 6 \rangle$ and the singleton $\langle D : 7 \rangle$ would again lead to an incorrect result representation. This delete statement can be performed if we restructure E_0 to E_1 . In fact, E_1 supports, without restructuring, the insertion of any tuple and any delete statement. \square

1.2 Contributions

The contributions of this work are as follows:

- We characterise all factorisation trees that allow the insertion of *any* tuple without prior restructuring in *any* representation following their structure. We also characterise all factorised representations that allow the insertion of a given tuple without prior restructuring and give an algorithm that performs the insertion of a tuple in a factorised representation.
- We present three algorithms for bulk insertion in factorised databases. In the context of factorised databases, bulk insertion is defined as the construction of a factorised representation for the relation consisting of all tuples we want to insert. The first algorithm builds the factorised representation of a given relation over a given factorisation tree. The second algorithm is given a path instead of a factorisation tree and can discover nesting structures hidden in the data that are consistent with the path taken as input. The last algorithm uses rectangle coverings of Boolean matrices representing the occurrence of various partial factorised representations to compute even more succinct factorised representations. We will show how rectangle coverings of matrices can be used as encodings of factorised representations and present a greedy heuristic that searches rectangle coverings that generate the most succinct factorised representations of a relation.
- We compare the three bulk insertion algorithms from a theoretical point of view and experimentally. We prove that the second algorithm computes a representation that has a smaller or equal size than the representation computed by the first bulk insertion algorithm, whenever the path considered by the second algorithm is consistent with the structure of the factorisation tree considered by the first algorithm. We also prove that the third algorithm builds a representation that is at least as good, in terms of size, as the second algorithm. Experimental evaluation has also been conducted and confirms our theoretical claims.
- We propose two techniques of performing value modification statements on factorised representations. The first technique generates a result consisting of a single factorised representation, while the second produces a result consisting of a union of factorised representations over the same factorisation tree. For both cases, we characterise all factorisation trees that allow a given value modification statement without restructuring on any representation following their structure.

- We approach the deletion task similarly to the value modification task. The two techniques developed for value modifications can be applied to deletions also, with only minor changes. For both techniques, we precisely characterise all factorisation trees that allow a given deletion statement without restructuring on any representation following their structure.
- We evaluate the performance of one of our value modification and deletion techniques against SQLite updates. For statements that do not require restructuring of the existing representation, results have shown that our algorithms can be up to four orders of magnitude faster than traditional update algorithms implemented by SQLite. For statements that require restructuring, our algorithms still perform better than SQLite updates, by one order of magnitude for all cases considered.
- The bulk insertion algorithms and the first technique proposed for value modifications and deletions have been implemented on top of an existing implementation of a factorised database system.

1.3 Outline

The remainder of this thesis is organized as follows:

- Chapter 2 introduces the theory of factorised representations and factorisation trees.
- Chapter 3 discusses insertions in factorised representations. The first section of this chapter focuses on the insertion of one tuple at a time, while the second section presents and compares three algorithms for bulk insertion in factorised databases.
- Chapter 4 describes techniques to change the values of a factorised representation.
- Chapter 5 presents techniques to delete values from a factorised representation.
- Chapter 6 gives an overview of the module implementing updates on factorised representations and discusses key implementation details.
- Chapter 7 discusses work related to factorised representations.
- Finally, chapter 8 concludes this work and discusses some areas for future research.

Chapter 2

Preliminaries

This chapter describes previous work [18, 3] on factorised databases needed to understand the thesis.

2.1 Factorised Representations

We consider relational databases with named attributes. A schema \mathcal{S} is a set of attributes. A tuple t of schema \mathcal{S} is a mapping from \mathcal{S} to a domain \mathcal{D} . A relation \mathbf{R} over \mathcal{S} is a set of tuples of schema \mathcal{S} . A database \mathbf{D} is a collection of relations.

We will use the notation $t(A)$ to refer to the value to which an attribute A in the schema of the tuple t is mapped.

A factorised representation of a relation \mathbf{R} is an algebraic expression that uses unions, Cartesian products and singleton relations to represent the relation \mathbf{R} .

Definition 2.1. [18] A factorised representation, or f-representation for short, over a schema \mathcal{S} , is a relational algebra expression of the form:

- \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} ,
- $(E_1 \cup \dots \cup E_n)$, representing the union of the relations represented by E_i , where each E_i is an f-representation over \mathcal{S} ,
- $(E_1 \times \dots \times E_n)$, 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 .

An expression $\langle A : a \rangle$ is called an *A-singleton*. The size of an f-representation $|E|$ is the number of singletons in E .

Proposition 2.1. [18] *Factorised representations form a complete representation system for relational data.*

This representation system is complete because any relation \mathbf{R} has at least one representation, called flat representation. In this representation, each tuple is represented by a product of singletons and the relation is the union of all these products. Using algebraic factorisation, we can compute nested f-representations that can be exponentially more succinct than their equivalent flat representation. Although much more compact than the flat representation, they still allow efficient enumeration of all products of singletons. The enumeration requires $O(|E|)$ space and precomputation time and $O(|\mathcal{S}|)$ delay between two consecutive products of singletons.

The system mentioned above is not injective because one relation can have several f-representations. The space of possible f-representations of a relation is defined by the distributivity of product over union.

2.2 Factorisation Trees

2.2.1 Factorisation Trees for Relations

The nesting structures of f-representations are defined by factorisation trees.

Definition 2.2. [18] A factorisation tree, or f-tree for short, over a schema \mathcal{S} is an unordered 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.

Definition 2.3. [18] An f-representation E over a given f-tree \mathcal{T} is recursively defined as follows:

- 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 $E = \bigcup_a \langle A_1 : a \rangle \times \dots \times \langle A_k : a \rangle$.
- 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 $E = \bigcup_a \langle A_1 : a \rangle \times \dots \times \langle A_k : a \rangle \times E_a$, 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 $E = E_1 \times \dots \times E_k$, where each E_i is an f-representation over \mathcal{T}_i .

R_1	A	B	C
1	2	5	
1	2	6	
1	3	5	
1	3	6	
2	4	7	
2	4	8	

S_1	A	B	C
1	2	5	
1	2	6	
1	3	5	
1	3	7	

Figure 2.1: Relations R_1 and S_1

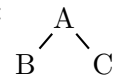
In an f-representation over f-tree \mathcal{T} , the attributes labelling a node in \mathcal{T} have equal values in the represented relation. Throughout the thesis, we will also consider that the distinct values a in a union \bigcup_a are sorted. This sorting is important for query evaluation and the insertion, value modification and deletion algorithms presented in subsequent chapters will preserve it.

An f-tree \mathcal{T} defines the nesting structure of the f-representation. In case of a rooted tree, we group the tuples by the values of the attributes labelling the root and then recursively apply the procedure to lower subtrees. A forest of trees denotes that attributes in different subtrees are independent and we can create a product of f-representations over those subtrees.

Not all relations over a schema \mathcal{S} can be represented over a particular f-tree. The relations that can be represented over an f-tree are only those that allow products of f-representations over subtrees that are siblings in the f-tree.

Any f-representation has a parse tree. Its internal nodes are labelled by \cup or \times operations, while the leaves are labelled by singletons or empty relations.

Example 2.1. Consider the schema $\mathcal{S} = \{A, B, C\}$, f-tree \mathcal{T}_1 over \mathcal{S} :



and relations R_1 and S_1 shown in Figure 2.1.

The f-representation of R_1 over f-tree \mathcal{T}_1 is

$$\langle A : 1 \rangle \times (\langle B : 2 \rangle \cup \langle B : 3 \rangle) \times (\langle C : 5 \rangle \cup \langle C : 6 \rangle) \cup \langle A : 2 \rangle \times \langle B : 4 \rangle \times (\langle C : 7 \rangle \cup \langle C : 8 \rangle)$$

Relation S_1 cannot be represented over f-tree \mathcal{T}_1 . The f-representation should have the form $\langle A : 1 \rangle \times E_B \times E_C$, where E_B is a union of B -singletons and E_C is a union of C -singletons, but for relation S_1 we cannot put in a product the union of B -singletons and the union of C -singletons. \square

2.2.2 Factorisation Trees for Queries

We consider conjunctive queries having the form $Q = \pi_{\mathcal{P}}(\sigma_{\varphi}(R_1 \times \dots \times R_n))$, where R_1, \dots, R_n are distinct relation symbols over disjoint schemas $\mathcal{S}_1, \dots, \mathcal{S}_n$, \mathcal{P} is the pro-

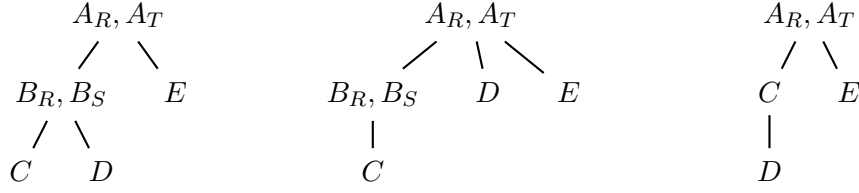


Figure 2.2: F-trees \mathcal{T}_2 , \mathcal{T}_3 and \mathcal{T}_4 .

jection list, $\mathcal{P} \subseteq \bigcup_i \mathcal{S}_i$ and φ is a list of equalities between attributes. The size of Q is $|Q| = n$.

We can use f-representations over f-trees to represent in a succinct form the results of conjunctive queries. Such representations use less memory and can be computed faster than their equivalent flat representations, but still allow fast enumeration of all tuples. F-trees that define f-representations of the query result $Q(\mathbf{D})$ for any input database \mathbf{D} can be inferred directly from the query Q . The nodes of these f-trees are labelled by classes of attributes. Each such class consists of attributes that are transitively equal in the selection condition of Q .

Proposition 2.2. [18] *Given a conjunctive query Q , $Q(\mathbf{D})$ has an f-representation over an f-tree \mathcal{T} for any database \mathbf{D} iff any two dependent nodes lie along the same root-to-leaf path in \mathcal{T} .*

The condition in the proposition given above is called the path constraint. The attributes of a relation are dependent: we have seen in Example 2.1 that we cannot make any assumptions about the structure of a relation. These are the only dependent attributes in an equi-join query (a conjunctive query that does not project away any attribute). For such a query, we can always put non-join attributes from different relations in different subtrees of the factorisation tree. For general conjunctive queries, however, we can also have dependent attributes in different relations. Such dependencies can arise if some relations are joined on a set of attributes and all these attributes are then removed. In such a case, non-join attributes which were previously independent can become dependent.

Example 2.2. Let's consider the relations R , S and T over schemas $\{A_R, B_R, C\}$, $\{B_S, D\}$ and $\{A_T, E\}$ and the query $Q_1 = \sigma_\varphi(R \times S \times T)$ with $\varphi = (A_R = A_T, B_R = B_S)$. The left f-tree in Figure 2.2 is a valid f-tree for Q_1 , while the f-tree in the middle is not because B_S and D are not on the same root-to-leaf path. A valid f-tree for the query $Q_2 = \pi_{\{A_R, A_T, C, D, E\}}(Q_1)$ is f-tree \mathcal{T}_4 , shown in the right column of Figure 2.2. Notice that attributes C and D are on the same path of \mathcal{T}_4 . Projecting away both B_R and B_S introduced their dependency. \square

2.3 Size Bounds For Factorised Representations

Previous work [18] has derived tight size bounds for f-representations of results of conjunctive queries. The following theorem describes these size bounds.

Theorem 2.4. [18] *For any non-Boolean query $Q = \pi_{\mathcal{P}}(\sigma_{\varphi}(R_1 \times \dots \times R_n))$, there is a rational number $s(Q)$ such that:*

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

To explain the significance of the rational number $s(Q)$ we should consider the hypergraph of Q . This hypergraph has one node for each attribute class of Q and one hyperedge for each relation symbol R . Each hyperedge includes all nodes with attributes in the corresponding relation. For an attribute symbol A labelling a node in an f-tree \mathcal{T} , let $\text{path}(A)$ represent the set of attributes labelling the nodes in \mathcal{T} from root to the node labelled by A (including the node labelled by A). Let's also consider the restriction of query Q to an attribute A : $Q_A = (\sigma_{\varphi_A}(R_1^A \times \dots \times R_n^A))$, where the selection condition and the relations in the query are restricted to the attributes in $\text{path}(A)$ and the database \mathbf{D} is projected onto the attributes in $\text{path}(A)$. The idea is to cover all attributes in Q_A with a number of their relations as small as possible. This corresponds to the fractional edge cover number of the query hypergraph $\rho^*(Q_A)$, which can be computed as the cost of an optimal solution to the following linear program:

$$\begin{aligned} & \text{minimise } \sum_i x_{R_i} \\ & \text{subject to } \sum_{i: R_i \in \text{rel}(\mathcal{A})} x_{R_i} \geq 1, \text{ for each attribute class } \mathcal{A}, \\ & x_{R_i} \geq 0, \text{ for all } i \end{aligned}$$

where $\text{rel}(\mathcal{A})$ is the set of all relations including attributes from the attribute class \mathcal{A} and x_{R_i} is the weight associated to relation R_i (and the corresponding hyperedge in the hypergraph).

We can now define: $s(\mathcal{T}) = \max_{A \in \mathcal{P}} \rho^*(Q_A)$ and $s(Q) = \min_{\mathcal{T}} s(\mathcal{T})$.

The number $s(\mathcal{T})$ can be computed in polynomial time. Computing $s(Q)$, however, might take exponential time because we might need to enumerate and compute the fractional edge cover for an exponential number of f-trees.

Example 2.3. Consider f-tree \mathcal{T}_2 in Figure 2.2. To compute $s(\mathcal{T}_2)$, we first need to compute the fractional edge cover for each root-to-leaf path. We have $\rho^*(Q_C) = 1$ and $\rho^*(Q_E) = 1$ because the paths of these restricted queries can be covered by a single relation. We also have $\rho^*(Q_D) = 2$. In this case, we need to assign $x_S = 1$ to be able to cover the node labelled by D ; the associated hyperedge will also cover the node labelled by (B_R, B_S) . Finally, to cover the node labelled by (A_R, A_T) , we need $x_R + x_T = 1$. $s(\mathcal{T})$ is the maximal possible $\rho^*(Q_A)$ over all attributes A , so $s(\mathcal{T}_2) = 2$. \square

2.4 Restructuring Operators

Previous work [3] describes query evaluation techniques on f-representations. F-representations support selection, projection and Cartesian product, but in some cases we cannot apply these operations directly on the given f-representation and we need to restructure it first. Such restructuring steps will be necessary in some cases before value modification or deletion statements also. More precisely, we sometimes need to apply the swap operator described in a subsequent section.

All restructuring operators need to preserve the path constraint, the increasing order of values a in any union expression \bigcup_a and normalisation. We say that an f-tree is normalised if there are no nodes that could be brought at higher levels in the f-tree without violating the path constraint.

2.4.1 The Push-up Operator

The push-up operator factors out common expressions. It is shown in Figure 2.3a. We can apply this operator whenever we have a parent node A and one of its children B together with all its descendants are independent of A . In such a case, B can be brought up in the f-tree and become a sibling of A . This restructuring operation will decrease the size of the f-representation. Before the restructuring, an f-representation over the f-tree rooted in node B may occur several times in the f-representation, once for each A -singleton. Because it occurs once for each A -singleton, it can be factored out and after the restructuring, it will occur only once in a product with the f-representation over the f-tree rooted in A .

We can normalise an f-tree by performing a bottom-up traversal and applying the push-up operator on each node of the f-tree as many times as possible.

2.4.2 The Swap Operator

The swap operator exchanges a node B with its parent node A . It is shown in Figure 2.3b. Before the restructuring, the values are first grouped by A and then by B , while in the restructured f-representation the values are first grouped by B and then by A . The

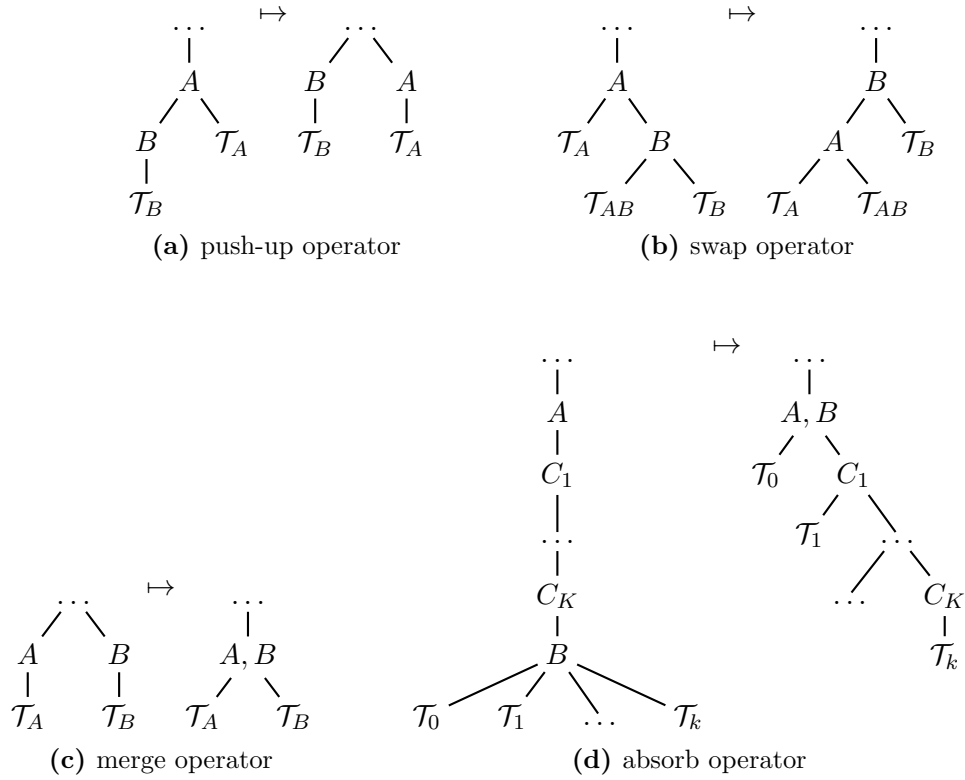


Figure 2.3: Restructuring operators

descendants of B are split into two categories: those that depend on both A and B and those that depend only on B . The descendants that depend only on B (represented as \mathcal{T}_B in the figure) can go up in the f-tree together with B such that, after the restructuring, the f-representations over \mathcal{T}_B will be in a product with the f-representations over the subtree rooted in A ; this preserves normalisation. The descendants that depend on A and B will become children of A ; this is necessary to preserve the path constraint.

2.4.3 Selection Operators

The next two restructuring operators are useful for selection queries with equality conditions of the form $A = B$. The merge operator can be applied when A and B are siblings, while the absorb operator can be applied when A is an ancestor of B . In any other case, we first need to restructure the f-tree using the swap operator until we reach one of the two cases mentioned above.

The merge operator merges the sibling nodes A and B into a node labelled by the attributes of A and B . The children of both A and B will become the children of the new node. The merge operator is shown in Figure 2.3c. The new f-representation will keep only A and B singletons (and associated f-representations) whose values occur in both unions

$\bigcup_a \langle A : a \rangle \times E_a$ and $\bigcup_b \langle B : b \rangle \times F_b$, where E_a is an f-representation over \mathcal{T}_A and F_b is an f-representation over \mathcal{T}_B . The procedure is implemented as a sort-merge join to preserve the value order constraint.

The absorb operator absorbs a node B into its ancestor A . It is shown in Figure 2.3d. Node B will be removed from the f-tree and all attributes labelling it will be added to the set of attributes labelling node A . Similar to the case of the merge operator, the new f-representation will keep only A and B singletons (and associated f-representations) whose values occur in both unions $\bigcup_a \langle A : a \rangle \times E_a$ and $\bigcup_b \langle B : b \rangle \times F_b$, where E_a is an f-representation over the f-tree rooted in A and F_b is an f-representation over the f-tree rooted in B . Since each union of B -singletons is found inside a union of A -singletons, for each A -singleton we keep at most one B -singleton (together with its associated f-representation). In case a descendant of B was dependent on B , but not on some ancestors of B , that descendant can go up in the f-tree without violating the path constraint. For this reason, after absorbing node B in node A , we also need to normalise the f-tree.

The last two operators can be performed in one pass over the f-representation. However, such efficient algorithms can be applied only under an additional assumption: that the values a in an union $\bigcup_a \langle A : a \rangle \times E_a$ are distinct. This assumption might not hold for an f-representation on which a value modification statement was performed. As we will see in Chapter 4, the value modification algorithm will need an additional step following the update of the f-representation which restructures the resulting f-representation such that we have distinct values a in all unions $\bigcup_a \langle A : a \rangle \times E_a$.

Chapter 3

Insertions

This chapter presents techniques of insertion in an f -representation. The first section of the chapter studies the insertion of one tuple in an f -representation over a given f -tree. We will first characterise the f -trees that allow the insertion of any tuple in any f -representation following their structure without prior restructuring. Then, we will discuss the case where an f -representation is defined by an f -tree that does not allow the insertion of any tuple without prior restructuring, but the structure and the data in the f -representation allows the insertion of particular tuples. We will state the condition under which such insertions are possible and present an algorithm that performs the insertion of a tuple if it is possible.

We will see that the class of f -trees that allow insertions is very restrictive. This was one of the reasons why we turned our attention to bulk insertion, which is presented in the second section. The bulk insertion procedure builds a factorised representation of the relation representing the tuples to be inserted. We discuss three algorithms for bulk insertion. The first algorithm builds the f -representation of the relation over a given f -tree. The second one takes as input a path and derives from the path an f -tree over which the relation is representable; at the same time, it builds the f -representation over that f -tree. The last one is similar to the second one, the difference being that it tries to find more common patterns in the data in an attempt to reduce the size of the f -representation as much as possible.

3.1 Efficient Insertions on Factorised Representations

3.1.1 Nesting Structures Supporting Efficient Insertions

The following theorem characterises all f -trees that allow the insertion of any tuple in any f -representation following their structure:

Theorem 3.1. *Given an f -tree \mathcal{T} over schema \mathcal{S} , we can insert any tuple t of schema \mathcal{S} on any f -representation over \mathcal{T} without prior restructuring iff the f -tree \mathcal{T} is a single path.*

Before proving the theorem, let's have a closer look at f-trees that are paths and f-representations over such f-trees. An f-representation over a path will first group its values by the first attribute, then by the second one and so on until it reaches the leaf. The only products present in such an f-representation are products between a singleton and a partial f-representation over a subpath. The quality of such an f-representation is low, because it was precisely the branching into several subtrees that saved the most in terms of size.

If Proof. In this part of the proof, we assume that we are given an f-tree \mathcal{T} that is a path and a tuple t over its schema and we want to prove that the tuple can be inserted in any f-representation over \mathcal{T} .

Let f-tree \mathcal{T} be the path (A_0, A_1, \dots, A_n) and the tuple $t = (A_0 : \alpha_0, A_1 : \alpha_1, \dots, A_n : \alpha_n)$. Any f-representation over \mathcal{T} will have the following form:

$$\bigcup_{a_0} \langle A_0 : a_0 \rangle \times \left(\bigcup_{a_1} \langle A_1 : a_1 \rangle \times \left(\dots \times \bigcup_{a_n} \langle A_n : a_n \rangle \right) \dots \right)$$

To insert the tuple t in this f-representation, we first search α_0 in the list of values a_0 in $\bigcup_{a_0} \langle A_0 : a_0 \rangle$. If we do not find it, we build the f-representation of tuple t over \mathcal{T} and add it to the top-most union of the f-representation such that the increasing order of the values a_0 is preserved. If we find it, we apply this technique recursively for the f-representation found in a product with singleton $\langle A_0 : \alpha_0 \rangle$ and the value α_1 . If α_1 is also present in the list of values a_1 in $\bigcup_{a_1} \langle A_1 : a_1 \rangle$, we continue to go at lower levels of the f-tree until we find a value α_i that is not present in the list of value a_i of the union $\bigcup_{a_i} \langle A_i : a_i \rangle$. If we reach the leaf A_n and find α_n in the list of values a_n in $\bigcup_{a_n} \langle A_n : a_n \rangle$, it means that the tuple we wanted to insert is already present. \square

Only If Proof. In this part of the proof, we are given an arbitrary f-representation over an f-tree \mathcal{T} and an arbitrary tuple over the schema of \mathcal{T} . We know that it is possible to insert the tuple in the f-representation and we want to prove that the f-tree is a path.

We will prove this part of the theorem by contradiction. We assume that the f-tree \mathcal{T} is not a path. This means that the f-tree has at least one node that has at least two children. Let A be the parent node, B and C the children nodes and \mathcal{T}_B and \mathcal{T}_C represent all descendants of B and C , respectively.

The insertion can be performed following the procedure described in the "if" proof until we reach node A , because A is the first node that has more than one child. When reaching node A , however, the insertion procedure will have to continue on two paths. We can notice that if the insertion leads to the addition of new singletons on both f-representations over these paths, then we end up adding more than one tuple. This happens because the f-representations on which we perform the insertions are found in a product. This means that

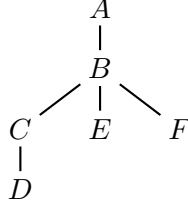


Figure 3.1: F-tree \mathcal{T}_5

the insertion of an arbitrary tuple t is not possible for any f-representation over \mathcal{T} , which contradicts the hypothesis. We can conclude that the assumption made in the beginning of the "only if" proof is wrong and the f-tree is indeed a path. \square

3.1.2 Incremental Factorisation Maintenance

There are cases where the f-tree of an f-representation is not a path, but it is still possible to insert certain tuples in the f-representation. Let's have a look at a few such cases.

Example 3.1. Consider f-tree \mathcal{T}_5 in Figure 3.1 and f-representation E_2 over \mathcal{T}_5 shown below

$$E_2 = \langle A : 1 \rangle \times \langle B : 2 \rangle \times \langle C : 3 \rangle \times \langle D : 4 \rangle \times (\langle E : 1 \rangle \cup \langle E : 2 \rangle \cup \langle E : 3 \rangle) \times \langle F : 5 \rangle$$

We can insert tuple $(A:3, B:5, C:1, D:3, E:7, F:2)$ in E_2 , because there is no A -singleton with value 3 in this f-representation. To perform the insert, we simply need to put the product of singletons $\langle A : 3 \rangle \times \langle B : 5 \rangle \times \langle C : 1 \rangle \times \langle D : 3 \rangle \times \langle E : 7 \rangle \times \langle F : 2 \rangle$ in a union with the existing f-representation.

We can also insert tuple $(A:1, B:5, C:1, D:3, E:7, F:2)$ in E_2 , because there is no B -singleton with value 5 under singleton $\langle A : 1 \rangle$. The new f-representation will be:

$$\langle A : 1 \rangle \times (\langle B : 2 \rangle \times \langle C : 3 \rangle \times \langle D : 4 \rangle \times (\langle E : 1 \rangle \cup \langle E : 2 \rangle \cup \langle E : 3 \rangle) \times \langle F : 5 \rangle) \cup \langle B : 5 \rangle \times \langle C : 1 \rangle \times \langle D : 3 \rangle \times \langle E : 7 \rangle \times \langle F : 2 \rangle$$

We can also insert tuple $(A:1, B:2, C:3, D:4, E:7, F:5)$ in E_2 . This insertion will add only singleton $\langle E : 7 \rangle$ to the union of E -singletons. However, we cannot insert tuple $(A:1, B:2, C:3, D:5, E:3, F:5)$. This insertion should add singleton $\langle D : 5 \rangle$ and the new f-representation would be:

$$\langle A : 1 \rangle \times \langle B : 2 \rangle \times \langle C : 3 \rangle \times (\langle D : 4 \rangle \cup \langle D : 5 \rangle) \times (\langle E : 1 \rangle \cup \langle E : 2 \rangle \cup \langle E : 3 \rangle) \times \langle F : 5 \rangle$$

The addition of singleton $\langle D : 5 \rangle$ caused in fact the addition of 3 new tuples instead of 1. The tuples introduced are $(A:1, B:2, C:3, D:5, E:1, F:5)$, $(A:1, B:2, C:3, D:5, E:2, F:5)$, $(A:1, B:2, C:3, D:5, E:3, F:5)$ \square

The following proposition states the condition that has to be satisfied in order to be able to insert a given tuple in an f-representation over an arbitrary f-tree.

Proposition 3.1. *We can insert a tuple t of schema \mathcal{S} in an f-representation E over an f-tree \mathcal{T} without prior restructuring if for each product of f-representations $E_0 \times E_1 \times \dots \times E_k$ reached by the insertion procedure, there is at most one f-representation E_i where new singletons are added and all other f-representations $E_j, j \neq i$, represent a single tuple over the schema of E_j . In addition to this, this tuple must be the same with the tuple resulted from the restriction of the tuple t to the schema of E_j .*

Figure 3.2 shows the pseudo-code of an algorithm that performs the insertion of a tuple in an f-representation. If the tuple can be inserted, the insertion is performed. Otherwise, the f-representation remains unchanged and the procedure returns an error. There are two subroutines: the *insert* subroutine performs the insertion, if it is possible, while the *check* subroutine tests if the f-representation received as an argument represents only the tuple received as an argument.

The *insert* subroutine will first check the structure of the f-tree. If the f-tree is a forest of trees, we first need to check if the insertion is possible. We call the *check* subroutine for each tree. If all subroutines return true, it means that the f-representation E given as an argument represents exactly the tuple we want to insert. In this case, no other steps are required. If more than one *check* subroutine returns false, then the insertion is not possible and the algorithm returns an error. If exactly one *check* subroutine returns false, then the insertion is possible so far and we recursively call the *insert* procedure for the f-representation for which the *check* subroutine returned false. We do not need to call the *insert* subroutine for any other f-representation, because all others represent exactly the data we wanted to insert over their associated f-trees. If the f-tree is a rooted tree \mathcal{T} , then we need to search the tuple value for the attribute labelling the root of the f-tree in the list of singleton values in the current union. If we find it, then we recursively call the *insert* procedure for the f-representation found in a product with the singleton containing the tuple value. Otherwise, we need to build the f-representation of the tuple over \mathcal{T} and add it to the current union.

The time complexity of this algorithm is $O(|E|)$. In general, the *insertion* procedure does not scan the entire f-representation, so the computation time can be even sublinear. For example, given an f-representation over a rooted f-tree, we first scan the singletons in the top-most union. If we do not find the tuple value corresponding to the attribute labelling the root of the f-tree in this list, then we do not need to scan any other part of the f-representation. If this tuple value is found, we need to scan only the f-representation

```

procedure INSERT(f-tree  $\mathcal{T}$ , f-representation  $E$ , tuple  $t$ )
  if  $\mathcal{T}$  is empty then return done
  if  $\mathcal{T}$  is a forest  $\mathcal{T}_1, \dots, \mathcal{T}_k$  then
    Let  $E = E_1 \times \dots \times E_k$ 
    for  $1 \leq i \leq k$  do
      check( $\mathcal{T}_i, E_i, t$ )
    if all check subroutines returned true then
      return done
    if more than one check subroutine returned false then
      return error
    if exactly one check subroutine returned false then
      Let  $\mathcal{T}_i$  be the subtree for which the check subroutine returned false
      return insert( $\mathcal{T}_i, E_i, t$ )
  if  $\mathcal{T}$  is a single rooted tree  $A(\mathcal{U})$  then
    Let  $E = \langle A : a_1 \rangle \times E_{a_1} \cup \dots \cup \langle A : a_n \rangle \times E_{a_n}$ 
    if  $t(A) \in \{a_1, \dots, a_n\}$  then
      return insert( $\mathcal{U}, E_{t(A)}, t$ )
    else
      Let  $t^*$  be the restriction of  $t$  to the attributes in the schema of  $\mathcal{T}$ 
      Build the f-representation  $F$  of tuple  $t^*$  over  $\mathcal{T}$ 
       $E = E \cup F$ 
      return done

procedure CHECK(f-tree  $\mathcal{T}$ , f-representation  $E$ , tuple  $t$ )
  if  $\mathcal{T}$  is empty then return true
  if  $\mathcal{T}$  is a forest  $\mathcal{T}_1, \dots, \mathcal{T}_k$  then
    Let  $E = E_1 \times \dots \times E_k$ 
    for  $1 \leq i \leq k$  do
      check( $\mathcal{T}_i, E_i, t$ )
    if all check subroutines returned true then
      return true
    else
      return false
  if  $\mathcal{T}$  is a single rooted tree  $A(\mathcal{U})$  then
    Let  $E = \langle A : a_1 \rangle \times E_{a_1} \cup \dots \cup \langle A : a_n \rangle \times E_{a_n}$ 
    if  $n = 1$  and  $a_1 = t(A)$  then
      return check( $\mathcal{U}, E_{a_1}, t$ )
    else
      return false

```

Figure 3.2: Insertion of a tuple in an f-representation

found in a product with the singleton containing the value. All other f-representations will not be scanned.

The insertions performed by the *insert* procedure are very efficient because they can be applied directly on the f-representation. If the f-tree is not a path and the *insert* procedure cannot perform the insertion, then the f-representation will first need to be restructured by repeatedly applying the swap operator until the f-tree of the resulting f-representation is a path.

3.2 Bulk Insertions

This section presents three techniques of bulk insertion in an f-representation. All three procedures take as input a set of tuples, but do not attempt to insert these tuples in an existing f-representation, because this can require, in many cases, the restructuring of the existing f-representation. Instead, they build a new f-representation for the input relation. The resulting f-representation will be a union of two f-representations that can have different nesting structures: the initial f-representation on which we wanted to perform the insertion and the f-representation built by the bulk insertion procedure. The advantage of this technique is that the size of the two f-representations can be much smaller than the size of a single f-representation representing the same relation. The disadvantage is that whenever we perform queries or apply restructuring operators on the new f-representation, we will have to work on two separate f-representations.

3.2.1 TF Algorithm: Factorisation over Tree Nesting Structures

The Tree Factorisation algorithm, or TF algorithm for short, takes as input an f-tree \mathcal{T} and a relation \mathbf{R} and computes the f-representation of \mathbf{R} over \mathcal{T} . Previous work [18] has already studied the factorisation of a relation over an f-tree. The following lemma is stated in [18]:

Lemma 3.1. [18] *Let \mathbf{R} be a relation over schema \mathcal{S} and \mathcal{T} be an f-tree over \mathcal{S} . If there is an f-representation of \mathbf{R} over \mathcal{T} , then it is unique up to commutativity of union and product.*

The construction of an f-representation of a relation over a given f-tree is also presented in [18]. The procedure shown in Figure 3.3 follows closely this technique. The notation $\pi_{\mathcal{T}}(\mathbf{R})$ used in the procedure denotes the projection of relation \mathbf{R} on the attributes of \mathcal{T} .

We notice that the f-representation computed by the TF algorithm does not represent exactly the relation if the relation is not representable over the f-tree given. In this case, the

```

procedure TF(f-tree  $\mathcal{T}$ , relation  $\mathbf{R}$ )
  if  $\mathcal{T}$  is empty then
    if  $\mathbf{R}$  is empty then  $E = \emptyset$ 
    if  $\mathbf{R}$  consists of the nullary tuple then  $E = \langle \rangle$ 
  if  $\mathcal{T}$  is a single node  $A$  then
    Let  $\mathcal{A} = \pi_A(\mathbf{R})$ 
     $E = \bigcup_{a \in \mathcal{A}} \langle A : a \rangle$ 
  if  $\mathcal{T}$  is a single rooted tree  $A(\mathcal{U})$  then
    Let  $\mathcal{A} = \pi_A(\mathbf{R})$ 
     $E = \bigcup_{a \in \mathcal{A}} \langle A : a \rangle \times \text{tf}(\mathcal{U}, \pi_{\mathcal{U}}(\sigma_{A=a}(\mathbf{R})))$ 
  if  $\mathcal{T}$  is a forest of f-trees  $\mathcal{T}_1, \dots, \mathcal{T}_k$  then
     $E = \text{tf}(\mathcal{T}_1, \pi_{\mathcal{T}_1}(\mathbf{R})) \times \dots \times \text{tf}(\mathcal{T}_k, \pi_{\mathcal{T}_k}(\mathbf{R}))$ 
  return  $E$ 

```

Figure 3.3: TF algorithm: Factorisation of a relation over an f-tree

relation represented by the f-representation built is an upper bound of the initial relation in the sense that the new relation contains all tuples in the initial relation, but can also contain some extra tuples. The following example illustrates this case.

Example 3.2. Recall from Example 2.1 f-tree \mathcal{T}_1 and relation S_1 , which is not representable over \mathcal{T}_1 . The output of the TF algorithm for \mathcal{T}_1 and S_1 will be the following f-representation:

$$\langle A : 1 \rangle \times (\langle B : 2 \rangle \cup \langle B : 3 \rangle) \times (\langle C : 5 \rangle \cup \langle C : 6 \rangle \cup \langle C : 7 \rangle)$$

This factorisation represents a relation that includes all the tuples of S_1 and has two additional tuples: $(A:1, B:2, C:7)$ and $(A:1, B:3, C:6)$. \square

3.2.2 PF Algorithm: Factorisation over Path Nesting Structures

Sometimes we do not know the f-tree over which the relation is factorisable. In this case we can use the Path Factorisation algorithm, or PF algorithm for short, which takes as input a path and a relation. The key feature of this algorithm is that it can infer from the path and the relation given, f-trees over which the relation or a subset of its tuple can be represented. Unlike the factorisation produced by the TF algorithm in which all partial f-representations over a certain schema have the same structure, the factorisation produced by the PF algorithm can contain partial f-representations that have the same schema, but different nesting structures. By allowing more than one nesting structure, the PF algorithm can reduce even more the size of the f-representation.

```

procedure PF(path  $\mathcal{P}$ , relation  $\mathbf{R}$ )
  if  $\mathcal{P}$  is empty then
    if  $\mathbf{R}$  is empty then return  $\emptyset$ 
    if  $\mathbf{R}$  consists of the nullary tuple then return  $\{\langle \rangle\}$ 
  Let  $\mathcal{A} = \pi_A(\mathbf{R})$ 
  if  $\mathcal{P}$  is a single node  $A$  then return  $\{\bigcup_{a \in \mathcal{A}} \langle A : a \rangle\}$ 
  if  $\mathcal{P}$  is a path tree  $A(\mathcal{P}_A)$  with root  $A$  and  $\mathcal{P}_A$  the path child of  $A$  then
    for each  $a \in \mathcal{A}$  do
       $\mathcal{L}_a = \text{pf}(\mathcal{P}_A, \pi_{\mathcal{P}_A}(\sigma_{A=a}(\mathbf{R})))$ 
    Compute  $\mathcal{I} = \bigcap_{a \in \mathcal{A}} \mathcal{L}_a$ 
     $E = \emptyset$ 
    for each  $a \in \mathcal{A}$  do
       $E_a = \langle A : a \rangle$ 
      for each  $F \in \mathcal{L}_a$  and  $F \notin \mathcal{I}$  do
         $E_a = E_a \times F$ 
       $E = E \cup E_a$ 
    return  $\mathcal{I} \cup \{E\}$ 

```

Figure 3.4: PF algorithm: Factorisation of a relation over a path

Definition 3.2. Let \mathbf{R} be a relation over schema \mathcal{S} and E an f-representation of \mathbf{R} . A factor of \mathbf{R} is an f-representation E_1 over a schema $\mathcal{S}_1 \subseteq \mathcal{S}$ such that there is another f-representation E_2 over schema $\mathcal{S}_2 = \mathcal{S} \setminus \mathcal{S}_1$ with $E = E_1 \times E_2$.

The pseudo-code of the PF algorithm is shown in Figure 3.4. The *pf* procedure returns the factors of the relation taken as input. Whenever the output contains more than one factor, the f-representation of the relation can be computed by putting the factors in a product. For a path consisting of a single node A , the *pf* procedure returns a single factor representing a union of A -singletons. For a path with more than one node and root A , we first compute the list of distinct A values in the relation. For each value a in this list, we also compute the list of factors for the relation containing all tuples that have a as A value by recursively applying procedure *pf*. The procedure then computes the intersection of all factor lists. Since the factors in the intersection occur in each factor list, they can appear in the f-representation only once, in a product with the f-representation containing all other factors. The procedure returns a set containing all factors in the intersection and an f-representation containing all other factors.

Each time the *pf* procedure returns a list of factors of size greater than 1, the f-tree of the f-representation is changed from a path to a forest with several trees. Each factor is an f-representation over a tree in this forest. If all intersections computed by the PF algorithm

are empty, then the list of factors returned by the PF algorithm will have only one factor that is exactly the f-representation of the relation over the input path.

Example 3.3. We exemplify the PF algorithm with relation S shown below and the path (A, B, C, D, E) . In the following figures, we considered

$$S_{v_1, \dots, v_k} = \pi_{S \setminus \{Attr(v_1), \dots, Attr(v_k)\}}(\sigma_{Attr(v_1)=v_1, \dots, Attr(v_k)=v_k}(S)),$$

where S is the schema of S , v_1, \dots, v_k are values of tuples of relation S and $Attr(v)$ represents the attribute to which a value v of a tuple of S is associated.

S	A	B	C	D	E
	a_1	b_1	c_1	d_1	e_1
	a_1	b_1	c_1	d_1	e_2
	a_1	b_1	c_1	d_2	e_1
	a_1	b_1	c_1	d_2	e_2
	a_1	b_1	c_2	d_1	e_1
	a_1	b_1	c_2	d_1	e_2
	a_1	b_1	c_2	d_2	e_1
	a_1	b_1	c_2	d_2	e_2
	a_1	b_2	c_1	d_3	e_3
	a_1	b_2	c_2	d_3	e_3
	a_2	b_2	c_1	d_1	e_1
	a_2	b_2	c_3	d_1	e_1

S_{a_1}	B	C	D	E
	b_1	c_1	d_1	e_1
	b_1	c_1	d_1	e_2
	b_1	c_1	d_2	e_1
	b_1	c_1	d_2	e_2
	b_1	c_2	d_1	e_1
	b_1	c_2	d_1	e_2
	b_1	c_2	d_2	e_1
	b_1	c_2	d_2	e_2
	b_2	c_1	d_3	e_3
	b_2	c_2	d_3	e_3

S_{a_2}	B	C	D	E
	b_2	c_1	d_1	e_1
	b_2	c_3	d_1	e_1

During the first pf procedure call, we find two distinct values for root A : a_1 and a_2 . We proceed to the next recursion level with S_{a_1} first. Here, we have again two distinct values for B : b_1 and b_2 . We proceed to depth 2 with S_{a_1, b_1} first.

S_{a_1, b_1}	C	D	E
	c_1	d_1	e_1
	c_1	d_1	e_2
	c_1	d_2	e_1
	c_1	d_2	e_2
	c_2	d_1	e_1
	c_2	d_1	e_2
	c_2	d_2	e_1
	c_2	d_2	e_2

S_{a_1, b_2}	C	D	E
	c_1	d_3	e_3
	c_2	d_3	e_3

At depth 3, we have to compute the factors of two relations that are equal $S_{a_1, b_1, c_1} = S_{a_1, b_1, c_2}$.

S_{a_1, b_1, c_1}	D	E
	d_1	e_1
	d_1	e_2
	d_2	e_1
	d_2	e_2

S_{a_1, b_1, c_2}	D	E
	d_1	e_1
	d_1	e_2
	d_2	e_1
	d_2	e_2

S_{a_1, b_1, c_1, d_1}	E
	e_1
	e_2

S_{a_1, b_1, c_1, d_2}	E
	e_1
	e_2

At depth 4, we reach the leaf E of the path. Both relation S_{a_1, b_1, c_1, d_1} and relation S_{a_1, b_1, c_1, d_2} return a list of factors containing a single factor: $\langle E : e_1 \rangle \cup \langle E : e_2 \rangle$.

Returning at depth 3, we have $\mathcal{L}_{d_1} = \mathcal{L}_{d_2} = \{\langle E : e_1 \rangle \cup \langle E : e_2 \rangle\}$. Their intersection is $\mathcal{I} = \{\langle E : e_1 \rangle \cup \langle E : e_2 \rangle\}$ and the list of factors returned is

$$\{\langle D : d_1 \rangle \cup \langle D : d_2 \rangle, \langle E : e_1 \rangle \cup \langle E : e_2 \rangle\}.$$

At depth 2, we have $\mathcal{L}_{c_1} = \mathcal{L}_{c_2} = \{\langle D : d_1 \rangle \cup \langle D : d_2 \rangle, \langle E : e_1 \rangle \cup \langle E : e_2 \rangle\}$. Their intersection contains both factors and the list of factors returned is

$$\{\langle C : c_1 \rangle \cup \langle C : c_2 \rangle, \langle D : d_1 \rangle \cup \langle D : d_2 \rangle, \langle E : e_1 \rangle \cup \langle E : e_2 \rangle\}.$$

At depth 1, we have $\mathcal{L}_{b_1} = \{\langle C : c_1 \rangle \cup \langle C : c_2 \rangle, \langle D : d_1 \rangle \cup \langle D : d_2 \rangle, \langle E : e_1 \rangle \cup \langle E : e_2 \rangle\}$ and $\mathcal{L}_{b_2} = \{\langle C : c_1 \rangle \cup \langle C : c_2 \rangle, \langle D : d_3 \rangle, \langle E : e_3 \rangle\}$. We compute $\mathcal{I} = \{\langle C : c_1 \rangle \cup \langle C : c_2 \rangle\}$ and the list of factors returned is

$$\{\langle C : c_1 \rangle \cup \langle C : c_2 \rangle,$$

$$\langle B : b_1 \rangle \times (\langle D : d_1 \rangle \cup \langle D : d_2 \rangle) \times (\langle E : e_1 \rangle \cup \langle E : e_2 \rangle) \cup \langle B : b_2 \rangle \times \langle D : d_3 \rangle \times \langle E : e_3 \rangle\}.$$

At depth 0, we have $\mathcal{L}_{a_1} = \{\langle C : c_1 \rangle \cup \langle C : c_2 \rangle, \langle B : b_1 \rangle \times (\langle D : d_1 \rangle \cup \langle D : d_2 \rangle) \times (\langle E : e_1 \rangle \cup \langle E : e_2 \rangle) \cup \langle B : b_2 \rangle \times \langle D : d_3 \rangle \times \langle E : e_3 \rangle\}$ and $\mathcal{L}_{a_2} = \{\langle C : c_1 \rangle \cup \langle C : c_3 \rangle, \langle B : b_2 \rangle, \langle D : d_1 \rangle, \langle E : e_1 \rangle\}$. We should notice here that relations S_{a_1} and S_{a_2} were factorised over different nesting structures: relation S_{a_1} was factorised over a forest with two trees, one over schema $\{B, D, E\}$ and the other consisting of node C , while relation S_{a_2} was factorised over a forest with four trees, each consisting of only one node. The intersection $\mathcal{L}_{a_1} \cap \mathcal{L}_{a_2}$ is empty, so the result is a list with a single factor:

$$\langle A : a_1 \rangle \times (\langle C : c_1 \rangle \cup \langle C : c_2 \rangle) \times$$

$$(\langle B : b_1 \rangle \times (\langle D : d_1 \rangle \cup \langle D : d_2 \rangle) \times (\langle E : e_1 \rangle \cup \langle E : e_2 \rangle) \cup \langle B : b_2 \rangle \times \langle D : d_3 \rangle \times \langle E : e_3 \rangle) \cup$$

$$\langle A : a_2 \rangle \times (\langle C : c_1 \rangle \cup \langle C : c_3 \rangle) \times \langle B : b_2 \rangle \times \langle D : d_1 \rangle \times \langle E : e_1 \rangle \quad \square$$

The order of the attributes in the path considered by the PF algorithm has a high impact on the nesting structures inferred by the algorithm. The following proposition characterises all f-trees that can be inferred from a path \mathcal{P} .

Proposition 3.2. *Consider a path \mathcal{P} over schema \mathcal{S} and an f-tree \mathcal{T} over the same schema. The PF algorithm can infer \mathcal{T} given \mathcal{P} if \mathcal{P} follows a topological sorting of \mathcal{T} .*

The condition stated in the proposition above is necessary, but not sufficient. The relation given to the PF algorithm (or a subset of its tuples) should be factorisable over the f-tree \mathcal{T} in order for the PF algorithm to infer this f-tree.

The mechanism of f-tree inference from a path is based on the fact that the PF algorithm is able to promote an f-representation one level up at each recursion depth. All f-representations in the intersection list are promoted and instead of remaining under the parent attribute in the f-representation parse tree, they become siblings with an f-representation

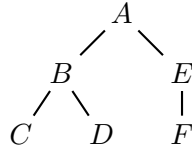


Figure 3.5: F-tree \mathcal{T}_6

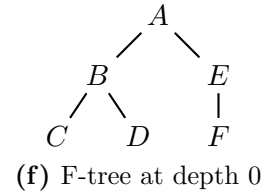
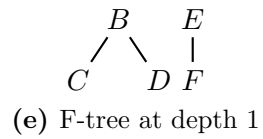
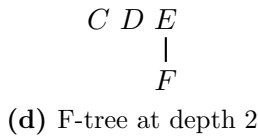
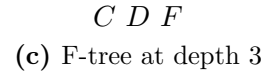
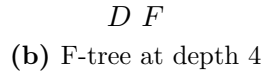
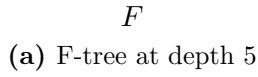


Figure 3.6: F-trees inferred at various levels of the PF algorithm from path (A, B, E, C, D, F) which follows the topological sorting of f-tree \mathcal{T}_6 .

whose schema includes the parent attribute. The promotion of an f-representation is possible if all attributes of its schema are independent of the parent attribute. If the given path \mathcal{P} does not follow the topological sorting of the f-tree \mathcal{T} , it means that there are at least two nodes A and B such that B is a descendent of A in \mathcal{T} and an ascendant of A in \mathcal{P} . When the PF algorithm reaches the level of B , the f-representation whose schema includes attribute A cannot be promoted, since node A and node B are dependent. This means that node A will remain under node B in all f-trees inferred by the PF algorithm.

Example 3.4. We will exemplify the promotions of partial f-representations during the PF algorithm for path (A, B, E, C, D, F) and a relation factorisable over f-tree \mathcal{T}_6 shown in Figure 3.5. We show in Figure 3.6 only the f-trees of the f-representations at each recursion level. At depth 5, the f-representation over node F is built. At depth 4, this f-representation is pushed up, since D and F are independent nodes in \mathcal{T}_6 . At depth 3, the f-representation over node D and the f-representation over node F are promoted because node C is independent of D and F . At depth 2, we can promote only the f-representations over node C and node D . The f-representation over F remains under E . At depth 1, we can promote the f-representation over the path (E, F) since both nodes are independent of B . At depth 0, none of the f-representations is promoted.

Let's also have a look at the f-trees inferred by the PF algorithm if the path does not follow a topological sorting of f-tree \mathcal{T}_6 . We consider the path (C, D, A, B, E, F) . Figure 3.7 shows the f-trees inferred at each recursion level. We should notice in this example that the last 4 nodes in the path considered follow a topological sorting of f-tree \mathcal{T}_6 . Sometimes, the PF algorithm is able to infer the independence of certain nodes or f-trees for subpaths

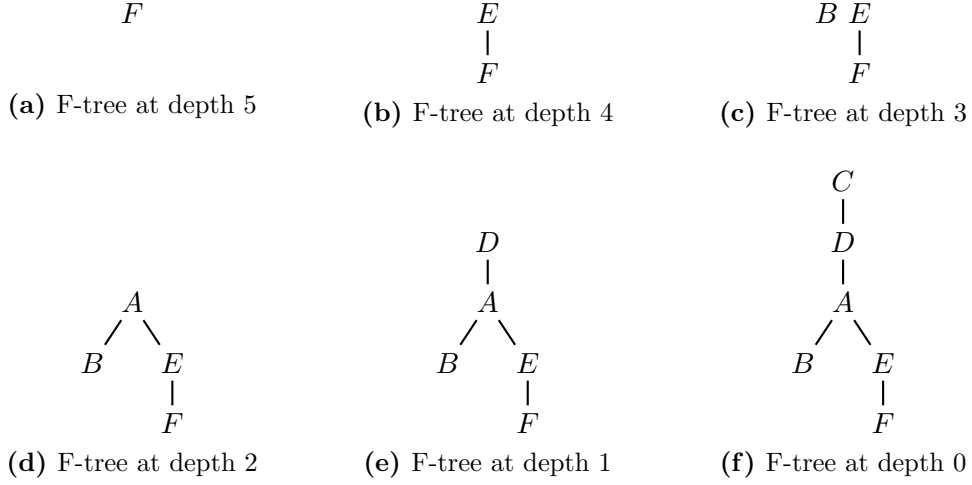


Figure 3.7: F-trees inferred at various levels of the PF algorithm from path (C, D, A, B, E, F) , which does not follow the topological sorting of f-tree \mathcal{T}_6 .

that follow a topological sorting of the f-tree, although the entire path does not follow a topological sorting of the f-tree. In this example, the PF algorithm was able to infer products between f-representations over B and f-representations over the path (E, F) .

At depth 1, the topological sorting of \mathcal{T}_6 is not satisfied any more as node D occurs before its ascendant A . Nodes E and F are independent of node D , but the partial f-representation over path (E, F) cannot be promoted since it occurs inside a larger f-representation whose f-tree includes node A . A similar situation is encountered at depth 0. \square

3.2.3 RCF Algorithm: Factorisation using Rectangle Coverings

Similar to the PF algorithm, the RCF algorithm has as input a relation and a path over the schema of the relation. It can be considered an extension of the PF algorithm and inherits some of its properties, such as the ability to infer nesting structures over which the relation is representable. We have seen in the previous section that the PF algorithm factors out only f-representations that are common to all tuple groups. The RCF algorithm aims to build a more compressed f-representation by factoring out, in addition to factors common to all tuple groups, f-representations that are common to only a subset of the tuple groups.

Definition 3.3. Let \mathbf{R} be a relation over schema \mathcal{S} , $A \in \mathcal{S}$ and $\mathcal{A} = \pi_A(\mathbf{R})$. For $a \in \mathcal{A}$, let's also define $\mathbf{R}_a = \pi_{\mathcal{S} \setminus \{A\}}(\sigma_{A=a}(\mathbf{R}))$, \mathcal{L}_a as the set of factors of relation \mathbf{R}_a and $\mathcal{L} = \bigcup_{a \in \mathcal{A}} \mathcal{L}_a$. The A -factor matrix M of relation \mathbf{R} is a Boolean matrix of size $|\mathcal{A}| \times |\mathcal{L}|$ defined as follows:

$$M_{i,j} = \begin{cases} 1, & \text{if } F_j \in \mathcal{L}_{a_i} \\ 0, & \text{otherwise} \end{cases}$$

We considered that the sets \mathcal{A} and \mathcal{L} are ordered. In this context, a_i represents the i -th value in \mathcal{A} and F_i represents the i -th factor in the list of factors \mathcal{L} .

Example 3.5. Let R_2 be a relation over a schema \mathcal{S} , $A \in \mathcal{S}$ and $\pi_A(R_2) = \{a_0, a_1, a_2, a_3\}$. Let the lists of factors be:

$$\mathcal{L}_{a_0} = \{F_0, F_1, F_2, F_3\}$$

$$\mathcal{L}_{a_1} = \{F_0, F_1, F_2, F_4\}$$

$$\mathcal{L}_{a_2} = \{F_1, F_2, F_3, F_5\}$$

$$\mathcal{L}_{a_3} = \{F_0, F_1, F_4, F_6\}$$

The union of these factor lists is $\mathcal{L} = \{F_0, F_1, F_2, F_3, F_4, F_5, F_6\}$. Each row of the A -factor matrix corresponds to a value $a_i, i \in \{0, 1, 2, 3\}$ and each column corresponds to a factor contained by \mathcal{L} . The A -factor matrix of R_2 is

	F_0	F_1	F_2	F_3	F_4	F_5	F_6
a_0	1	1	1	1	0	0	0
a_1	1	1	1	0	1	0	0
a_2	0	1	1	1	0	1	0
a_3	1	1	0	0	1	0	1

□

Definition 3.4. A rectangle of a Boolean matrix M is a pair of sets (R, C) such that for each pair (i, j) , $i \in R$ and $j \in C$, we have $M_{i,j} = 1$. R represents the set of rows and C represents the set of columns.

Definition 3.5. A rectangle covering of a Boolean matrix M is a set of rectangles $\mathcal{RC} = \{(R_k, C_k)\}$, such that for each pair (i, j) with $M_{i,j} = 1$ there is at least one rectangle $(R, C) \in \mathcal{RC}$ with $i \in R$ and $j \in C$.

There are certain rectangle coverings of an A -factor matrix of a relation that can encode f-representations of the relation. The following proposition states the conditions that a rectangle covering must satisfy in order to encode an f-representation. We will then show how an f-representation is built from a rectangle covering of an A -factor matrix.

Proposition 3.3. *A rectangle covering \mathcal{RC} of an A -factor matrix M of a relation \mathbf{R} is an encoding of an f-representation if it satisfies the following two conditions:*

- *The rectangles in the covering do not overlap, i.e. for any two rectangles $(R_i, C_i) \in \mathcal{RC}$, $(R_j, C_j) \in \mathcal{RC}$, there is no pair (k, l) such that $k \in R_i, l \in C_i$ and $k \in R_j, l \in C_j$*
- *For any two rectangles $(R_i, C_i) \in \mathcal{RC}$, $(R_j, C_j) \in \mathcal{RC}$, if $R_i \cap R_j \neq \emptyset$, then either $R_i \subsetneq R_j$, or $R_j \subsetneq R_i$.*

Definition 3.6. We say that a rectangle (R_i, C_i) is row-included in another rectangle (R_j, C_j) if $R_i \subset R_j$.

Definition 3.7. Consider a rectangle covering \mathcal{RC} of an A -factor matrix M of a relation \mathbf{R} . We say that a rectangle $(R_i, C_i) \in \mathcal{RC}$ is directly row-included in rectangle $(R_j, C_j) \in \mathcal{RC}$ if (R_i, C_i) is row-included in (R_j, C_j) and there is no other rectangle $(R_k, C_k) \in \mathcal{RC}$ row-included in (R_j, C_j) such that (R_i, C_i) is row-included in (R_k, C_k) .

We show below how an f-representation is built from a rectangle covering \mathcal{RC} of an A -factor matrix of a relation \mathbf{R} . Let \mathcal{A} be the ordered list of distinct A values of relation \mathbf{R} and \mathcal{L} be the ordered list of factors that generated the factor matrix. Also let $\mathcal{P} = \{P_1, \dots, P_k\}$ be the set of rectangles in \mathcal{RC} that are not row-included in any other rectangles. The f-representation $E(P)$ over a rectangle $P \in \mathcal{P}$ can be built as follows:

- If $P = (\{r_1, \dots, r_m\}, \{c_1, \dots, c_n\})$ is a rectangle that does not row-include other rectangles, the f-representation encoded by P is

$$E(P) = F_{c_1} \times \dots \times F_{c_n} \times (\langle A : a_{r_1} \rangle \cup \dots \cup \langle A : a_{r_m} \rangle),$$

$$a_i \in \mathcal{A}, \forall i \in \{r_1, \dots, r_m\}, F_j \in \mathcal{L}, \forall j \in \{c_1, \dots, c_n\}$$

- If $P = (\{r_1, \dots, r_m\}, \{c_1, \dots, c_n\})$ is a rectangle that includes other rectangles, let I_1, \dots, I_p be the rectangles that are directly row-included by rectangle P . The f-representation encoded by P is

$$E(P) = F_{c_1} \times \dots \times F_{c_n} \times (E(I_1) \cup \dots \cup E(I_p)), F_j \in \mathcal{L}, \forall j \in \{c_1, \dots, c_n\}$$

The f-representation encoded by the rectangle covering \mathcal{RC} is $E(\mathcal{RC}) = \bigcup_{P \in \mathcal{P}} E(P)$, where \mathcal{P} is the set of rectangles in \mathcal{RC} that are not row-included in any other rectangles.

Any A -factor matrix has at least one rectangle covering that encodes an f-representation, a so-called flat rectangle covering. Each rectangle of such a covering has in its set of rows a single row r and the set of columns contains all non-zero columns of row r , i.e.

$$\mathcal{RC} = \{(\{r\}, \{j; M_{r,j} = 1\}); 0 \leq r < n\}, \text{ where } n \text{ is the number of rows in the } A\text{-factor matrix } M$$

Coverings containing rectangles that span over several rows can produce smaller f-representations, because the factors corresponding to the columns in such a rectangle appear only once in the f-representation instead of once for each row in the rectangle. Using the distributivity of product over union, we can expand an f-representation obtained from a non-flat covering and obtain the f-representation that the flat covering produces.

Example 3.6. Let's consider the A -factor matrix in Example 3.5. A rectangle covering of this matrix that encodes an f-representation is $C = \{R_0, R_1, R_2, R_3, R_4, R_5, R_6\}$, where

$$\begin{aligned} R_0 &= (\{0, 1, 2, 3\}, \{1\}) & R_4 &= (\{1\}, \{4\}) \\ R_1 &= (\{0, 1, 2\}, \{2\}) & R_6 &= (\{2\}, \{3, 5\}) \\ R_2 &= (\{0, 1\}, \{0\}) & R_7 &= (\{3\}, \{0, 4, 6\}) \\ R_3 &= (\{0\}, \{3\}) \end{aligned}$$

The f-representation for this rectangle is

$$\begin{aligned} F_1 \times (F_2 \times (F_0 \times (F_3 \times \langle A : a_0 \rangle \cup F_4 \times \langle A : a_1 \rangle) \cup F_3 \times F_5 \times \langle A : a_2 \rangle) \cup \\ F_0 \times F_4 \times F_6 \times \langle A : a_3 \rangle) \end{aligned} \quad \square$$

The RCF algorithm searches the rectangle covering that encodes the smallest f-representation for a given relation. One way to find the best rectangle covering for an A -factor matrix of a relation would be to enumerate all rectangle coverings for that matrix and choose the best one, but this procedure can be computationally too expensive. The RCF algorithm uses an heuristic to find a good rectangle covering. At each step in this heuristic, the RCF algorithm has to choose between several rectangles. In order to do this, we need to associate weights to rectangles. We would like the RCF algorithm to choose the rectangle cover that generates the smallest f-representation, so a good idea would be to choose first the rectangles that save the highest number of singletons. This motivates the following weight for a rectangle (R, C) :

$$w(R, C) = (|R| - 1) \sum_{c \in C} |F_c|$$

This weight can be interpreted as the number of singletons that the rectangle (R, C) saves. Each factor in the rectangle occurs only once in the f-representation, instead of $|R|$ times, so we are able to save $|R| - 1$ occurrences of each factor in the rectangle. We should notice that it assigns weight 0 to rectangles that have a single row because such rectangles do not factor out anything and the number of singletons saved is indeed 0.

Figure 3.8 shows the pseudo-code of the RCF algorithm. The *rcf* procedure takes as arguments a path and a relation and returns the factors of that relation. It computes the set of distinct A values, where A is the root of the path and for each group of tuples sharing the same A value, it also computes the list of factors by recursively calling the *rcf* procedure. It builds the A -factor matrix of the relation and the set Q representing all rows of the matrix. It then repeatedly calls the *extractRectangles* procedure until the set of rows is empty.

The *extractRectangles* procedure takes as arguments a set of rows Q and an A -factor matrix and returns a set of rectangles over the matrix. This set of rectangles cover entirely a subset of rows $Q_e \subseteq Q$; all these rows will also be removed from Q . We start with

```

procedure RCF(path  $\mathcal{P}$ , relation  $\mathbf{R}$ )
  if  $\mathcal{P}$  is empty then
    if  $\mathbf{R}$  is empty then return  $\emptyset$ 
    if  $\mathbf{R}$  consists of the nullary tuple then return  $\{\langle \rangle\}$ 
  Let  $\mathcal{A} = \pi_{\mathcal{A}}(\mathbf{R})$ 
  if  $\mathcal{P}$  is a single node  $A$  then return  $\{\bigcup_{a \in \mathcal{A}} \langle A : a \rangle\}$ 
  if  $\mathcal{P}$  is a path tree  $A(\mathcal{P}_A)$  with root  $A$  and  $\mathcal{P}_A$  the path child of  $A$  then
    for each  $a \in \mathcal{A}$  do
       $\mathcal{L}_a = \text{pf}(\mathcal{P}_A, \pi_{\mathcal{P}_A}(\sigma_{A=a}(\mathbf{R})))$ 
       $\mathcal{L} = \bigcup_{a \in \mathcal{A}} \mathcal{L}_a$ 
      Build the  $A$ -factor matrix  $M$  using  $\mathcal{L}$  and  $\mathcal{A}$ 
       $Q = \{0, 1, \dots, |\mathcal{A}| - 1\}$ 
       $Cov = \emptyset$ 
      while  $Q \neq \emptyset$  do
         $Cov = Cov \cup \text{extractRectangles}(Q, M)$ 
      Compute  $E(Cov)$ 
      Let  $E_F$  be the set of factors of f-representation  $E(Cov)$ 
      return  $E_F$ 

procedure EXTRACTRECTANGLES(set of rows  $Q$ ,  $A$ -factor matrix  $M$ )
  Let  $R = \{r\}, r \in Q$ 
  Let  $C = \{i; M_{r,i} = 1\}$ 
  Let  $P = (R, C)$ 
   $Q = Q \setminus \{r\}$ 
   $\mathcal{R} = \emptyset$ 
  while  $Q \neq \emptyset$  do
    Compute  $P_i = (R_i, C_i) = (R \cup \{i\}, \{j; j \in C \text{ and } M_{i,j} = 1\}), \forall i \in Q$ 
    if  $|C_i| = 0, \forall i \in Q$  then
      break
    else
      Find row  $r \in Q$  such that  $w(P_r) = \max(w(P_i)), \forall i \in Q$ 
      Let  $P_r = (R_r, C_r)$ 
       $Q = Q \setminus \{r\}$ 
      Create new rectangle  $S = (R_S, C_S) = (R, \{j; j \in C \text{ and } j \notin C_r\})$ 
      if  $C_S \neq \emptyset$  then
         $\mathcal{R} = \mathcal{R} \cup \{S\}$ 
      Create new rectangle  $T = (R_T, C_T) = (\{r\}, \{j; M_{r,j} = 1 \text{ and } j \notin C_r\})$ 
      if  $C_T \neq \emptyset$  then
         $\mathcal{R} = \mathcal{R} \cup \{T\}$ 
       $P = P_r$ 
     $\mathcal{R} = \mathcal{R} \cup \{P\}$ 
  return  $\mathcal{R}$ 

```

Figure 3.8: RCF algorithm: Factorisation of a relation over a path using rectangle coverings

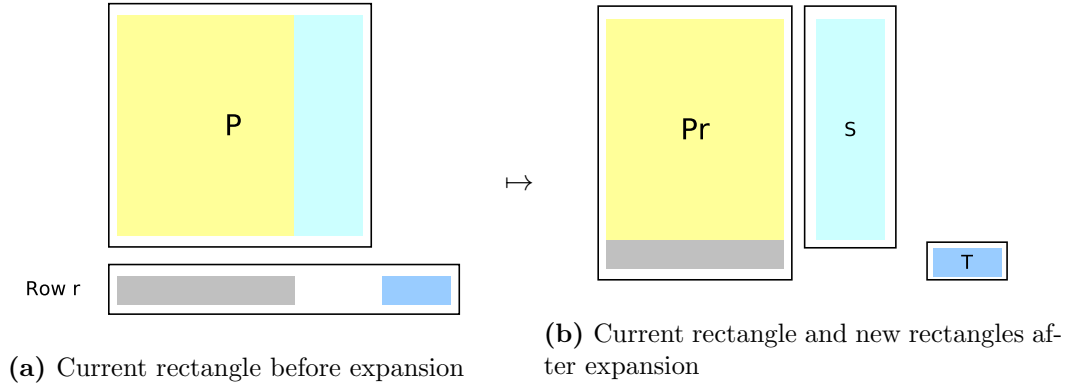


Figure 3.9: RCF Algorithm: the rectangles built during an expansion round

a single rectangle whose set of rows consists of only one row r and the set of columns contains all its non-zero columns. Then, we repeatedly try to expand this rectangle to form a rectangle that spans over several rows. The expansion has several rounds, each round adding one new row to the current rectangle. During an expansion round, we compute the weights of all rectangles resulted from the expansion of the current rectangle with a row from Q and we choose a row r that leads to the rectangle with the highest weight. The current rectangle changes during the expansion and two new rectangles are built. Figure 3.9a shows the current rectangle P before expansion coloured in yellow and blue. The yellow part corresponds to factors that can be found in row r also, while the light blue part corresponds to factors that are not present in row r . Row r is also divided into two parts: the grey part represents the factors found in the current rectangle also, while the dark blue part represents the factors not included by the current rectangle. Figure 3.9b displays the rectangles after expansion: the common factors of row r and rectangle P are kept in the current rectangle, while the non-common factors form new rectangles.

The expansion of the rectangle ends when there isn't any row that can be added to the current rectangle. This happens in two situations:

- The set of rows Q is empty.
- There is no other row $r \in Q$ such that the current rectangle has common factors with r .

When the set of rows Q is empty, the rectangle covering built by the *rcf* procedure covers all rows of the A -factor matrix. The *rcf* procedure computes the f-representation E encoded by this rectangle covering and returns a set containing the factors of E .

3.2.4 Comparison of Bulk Insertion Algorithms

This section compares the bulk insertion algorithms described in previous sections. We first show that the f-representation built by the PF algorithm is, in terms of size, at least as good as the f-representation built by the TF algorithm if the PF algorithm is given a path that follows the topological sorting of the f-tree considered by the TF algorithm. Then, we will show that the f-representation built by the RCF algorithm is, in terms of size, at least as good as the f-representation built by the PF algorithm.

Theorem 3.8. *Consider an f-tree \mathcal{T} over schema \mathcal{S} and a relation \mathbf{R} representable over \mathcal{T} . If the path used by the PF algorithm for representation construction follows a topological sorting of \mathcal{T} , then the PF algorithm builds an f-representation of \mathbf{R} that is, in terms of size, at least as good as the f-representation of \mathbf{R} over \mathcal{T} .*

Proof. We will prove this theorem by induction over the number of attributes of schema \mathcal{S} .

Base case: Let's consider a path with a single node A . In this case, the algorithms return the same f-representation: $\langle A : a_1 \rangle \cup \langle A : a_2 \rangle \cup \dots \cup \langle A : a_n \rangle$ where a_1, a_2, \dots, a_n are distinct A values of relation \mathbf{R} .

Induction step: We assume that the hypothesis holds for all relations over schemas with k or less nodes, $k \geq 1$. We also assume given an f-tree \mathcal{T} over schema \mathcal{S} , $|\mathcal{S}| = k + 1$, a path \mathcal{P} over the same schema and a relation \mathbf{R} factorisable over \mathcal{T} . We assume that the path \mathcal{P} follows a topological sorting of the f-tree \mathcal{T} and we want to prove that the f-representation built by the PF algorithm using path \mathcal{P} for representation construction is at least as good as the f-representation of \mathbf{R} over \mathcal{T} .

Let A be the root of the path \mathcal{P} and \mathcal{P}_A be the path child of \mathcal{P} . Path \mathcal{P} follows a topological sorting of f-tree \mathcal{T} , which means that all nodes in \mathcal{P}_A are either descendants of A in \mathcal{T} , or belong to subtrees of \mathcal{T} that are independent of A . Let $\mathcal{U}_A = \{U_1, \dots, U_m\}$ be the forest representing all descendants of node A in \mathcal{T} and $\mathcal{T}_A = \{T_1, \dots, T_n\}$ be the forest of trees independent of node A in \mathcal{T} .

The PF algorithm computes $\mathcal{A} = \pi_A(\mathbf{R})$ and the list of factors \mathcal{L}_a for each relation $\mathbf{R}_a = \pi_{\mathcal{S} \setminus \{A\}}(\sigma_{A=a}(\mathbf{R}))$, $a \in \mathcal{A}$ by recursively applying the PF algorithm for each such relation and path \mathcal{P}_A . Since \mathbf{R} is representable over \mathcal{T} , each relation \mathbf{R}_a is representable over the f-tree $\mathcal{T}^* = \mathcal{U}_A \cup \mathcal{T}_A$. Path \mathcal{P} follows a topological sorting of f-tree \mathcal{T} , so path \mathcal{P}_A will also follow a topological sorting of f-tree \mathcal{T}^* . We can apply the induction step for each pair $(\mathbf{R}_a, \mathcal{P}_A)$, $a \in \mathcal{A}$ as both conditions are satisfied. Applying the induction hypothesis, we can assume that the f-representation built by the PF algorithm for a pair $(\mathbf{R}_a, \mathcal{P}_A)$, $a \in \mathcal{A}$ is as good as the f-representation of \mathbf{R}_a over \mathcal{T}^* .

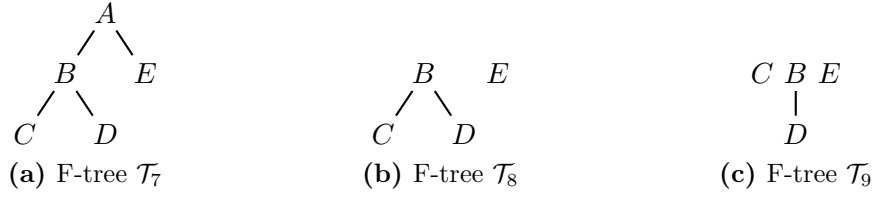


Figure 3.10: F-trees $\mathcal{T}_7, \mathcal{T}_8, \mathcal{T}_9$

F-tree \mathcal{T}^* is a forest with $m + n$ trees, so each relation $\mathbf{R}_a, a \in \mathcal{A}$ can be represented as a product of $m + n$ f-representations. This means that the output \mathcal{L}_a of the PF algorithm for a pair $(\mathbf{R}_a, \mathcal{P}_A), a \in \mathcal{A}$ must be a set with at least $m + n$ factors:

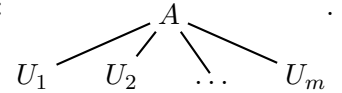
$$\mathcal{L}_a = \{E_{a,U_1}, \dots, E_{a,U_m}\} \cup \{E_{a,T_1}, \dots, E_{a,T_n}\}$$

In the equation above, we used the notation $E_{a,T}$ to denote the factor of relation $\mathbf{R}_a, a \in \mathcal{A}$ over the schema of the f-tree T .

Node A is independent of all f-trees in \mathcal{T}_A , which means that the f-representation over the schema of the f-tree containing A can be put in a product with the f-representations over the schemas of the f-trees in \mathcal{T}_A . As a result, we have

$$E_{a_1, T_i} = E_{a_2, T_i} = \dots = E_{a_l, T_i}, \forall i \in \{1, 2, \dots, n\} \text{ and } \mathcal{A} = \{a_1, a_2, \dots, a_l\}$$

The intersection list computed by the PF algorithm contains a factor over the schema of each tree in \mathcal{T}_A . The f-representations over schemas of f-trees in \mathcal{U}_A do not appear in general in the intersection list because the f-trees of \mathcal{U}_A contain nodes that are dependent on node A . The PF algorithm builds a single f-representation over f-tree T_0 :



The f-representation of \mathbf{R} over \mathcal{T} is a product of f-representations over f-trees T_0, T_1, \dots, T_n . The PF algorithm returns a set of factors over the schemas of T_0, T_1, \dots, T_n that are at least as good as the f-representations over these f-trees. By putting the factors in a product, we can obtain an f-representation of relation \mathbf{R} that is at least as good as its f-representation over f-tree \mathcal{T} .

There are cases when the intersection list computed by the PF algorithm contains f-representations over the schemas of f-trees in \mathcal{U}_A . The f-representation produced by the PF algorithm includes these f-representations only once, while the f-representation over f-tree \mathcal{T} contains these f-representations once for each group of tuples sharing the same A value. In such a case, the size of the f-representation produced by the PF algorithm is smaller than the size of the f-representation over \mathcal{T} . \square

Example 3.7. Consider relation R shown below, representable over f-tree \mathcal{T}_7 shown in Figure 3.10a.

R	A	B	C	D	E
	a_1	b_1	c_1	d_1	e_1
	a_1	b_1	c_1	d_2	e_1
	a_1	b_1	c_2	d_1	e_1
	a_1	b_1	c_2	d_2	e_1
	a_1	b_2	c_1	d_1	e_1
	a_1	b_2	c_1	d_3	e_1
	a_1	b_2	c_2	d_1	e_1
	a_1	b_2	c_2	d_3	e_1
	a_2	b_3	c_3	d_3	e_3
	a_2	b_3	c_3	d_4	e_3
	a_3	b_1	c_1	d_1	e_4
	a_3	b_1	c_2	d_1	e_4

The f-representation of R over \mathcal{T}_7 is

$$\begin{aligned}
E_3 = & \langle A : a_1 \rangle \times (\langle B : b_1 \rangle \times (\langle C : c_1 \rangle \cup \langle C : c_2 \rangle) \times (\langle D : d_1 \rangle \cup \langle D : d_2 \rangle) \cup \\
& \langle B : b_2 \rangle \times (\langle C : c_1 \rangle \cup \langle C : c_2 \rangle) \times (\langle D : d_1 \rangle \cup \langle D : d_3 \rangle) \times \langle E : e_1 \rangle) \cup \\
& \langle A : a_2 \rangle \times \langle B : b_3 \rangle \times \langle C : c_3 \rangle \times (\langle D : d_3 \rangle \cup \langle D : d_4 \rangle) \times \langle E : e_3 \rangle \cup \\
& \langle A : a_3 \rangle \times \langle B : b_1 \rangle \times (\langle C : c_1 \rangle \cup \langle C : c_2 \rangle) \times \langle D : d_1 \rangle \times \langle E : e_4 \rangle
\end{aligned}$$

The f-representation produced by the PF algorithm using any path that follows a topological sorting of f-tree \mathcal{T}_7 is

$$\begin{aligned}
E_4 = & \langle A : a_1 \rangle \times (\langle B : b_1 \rangle \times (\langle D : d_1 \rangle \cup \langle D : d_2 \rangle) \cup \\
& \langle B : b_2 \rangle \times (\langle D : d_1 \rangle \cup \langle D : d_3 \rangle)) \times (\langle C : c_1 \rangle \cup \langle C : c_2 \rangle) \times \langle E : e_1 \rangle) \cup \\
& \langle A : a_2 \rangle \times \langle B : b_3 \rangle \times \langle C : c_3 \rangle \times (\langle D : d_3 \rangle \cup \langle D : d_4 \rangle) \times \langle E : e_3 \rangle \cup \\
& \langle A : a_3 \rangle \times \langle B : b_1 \rangle \times (\langle C : c_1 \rangle \cup \langle C : c_2 \rangle) \times \langle D : d_1 \rangle \times \langle E : e_4 \rangle
\end{aligned}$$

Notice that f-representation E_3 contains the factor $\langle C : c_1 \rangle \cup \langle C : c_2 \rangle$ three times, while E_4 contains this factor only two times. The TF algorithm builds the f-representation of relation $R_{a_1} = \pi_{B,C,D,E}(\sigma_{A=a_1}(R))$ over the forest shown in Figure 3.10b, since it follows entirely the structure of the f-tree given. However, relation R_{a_1} is also representable over the forest shown in Figure 3.10c and the factorisation over this forest is smaller. The PF algorithm detects that relation R_{a_1} is representable over \mathcal{T}_9 and builds the f-representation over this f-tree. \square

Theorem 3.9. *The RCF algorithm builds an f-representation that is, in terms of size, at least as good as the f-representation produced by the PF algorithm for a given relation \mathbf{R} and the same path \mathcal{P} used for representation construction.*

Proof. We will prove this theorem by induction over the number of attributes of schema \mathcal{S} of relation \mathbf{R} .

Base case: Let's consider a path with a single node A . In this case, the algorithms return the same f-representation: $\langle A : a_1 \rangle \cup \langle A : a_2 \rangle \cup \dots \cup \langle A : a_n \rangle$ where a_1, a_2, \dots, a_n are distinct A values of relation \mathbf{R} .

Induction step: We assume that the hypothesis holds for all relations with k or less attributes, $k \geq 1$ and we want to prove that the RCF algorithm produces an f-representation that is at least as good as the f-representation produced by the PF algorithm for a relation \mathbf{R} over a schema $\mathcal{S}, |\mathcal{S}| = k + 1$ and a path \mathcal{P} over the same schema.

Let A be the root of path \mathcal{P} and \mathcal{P}_A be the path child of \mathcal{P} . Both algorithms compute $\mathcal{A} = \pi_A(\mathbf{R})$ and the list of factors for each relation $\mathbf{R}_a = \pi_{\mathcal{S} \setminus \{A\}}(\sigma_{A=a}(\mathbf{R})), a \in \mathcal{A}$ by recursively applying the algorithm for each such relation and path \mathcal{P}_A . Let $\mathcal{L}_{a,pf}$ be the factor list of relation $\mathbf{R}_a, a \in \mathcal{A}$ computed by the PF algorithm. Also, let $\mathcal{L}_{a,rcf}$ be the factor list of relation $\mathbf{R}_a, a \in \mathcal{A}$ computed by the RCF algorithm. We can apply the induction step for each relation $\mathbf{R}_a, a \in \mathcal{A}$ and assume that the factors in $\mathcal{L}_{a,rcf}$ are at least as good as the factors in $\mathcal{L}_{a,pf}, \forall a \in \mathcal{A}$.

The f-representation produced by the PF algorithm can be encoded by the following rectangle covering

$$\begin{aligned} \mathcal{RC} = & \{(\{0, 1, \dots, |\mathcal{A}| - 1, \}, \{j; F_j \in \mathcal{I}_{pf}\})\} \cup \\ & \{(\{i\}, \{j; F_j \notin \mathcal{I}_{pf} \text{ and } M_{i,j} = 1\}); i \in \{0, 1, \dots, |\mathcal{A}| - 1\}\} \\ & \text{where } \mathcal{I}_{pf} = \bigcap_{a \in \mathcal{A}} \mathcal{L}_{a,pf} \text{ and } F_j \text{ is the } j\text{-th factor in the ordered set } \mathcal{L} = \bigcup_{a \in \mathcal{A}} \mathcal{L}_{a,pf} \end{aligned}$$

This covering contains a rectangle that represents the intersection of all factor lists, if this intersection is not empty. If there are factors that are not present in the intersection list, it also contains exactly one rectangle for each row i of the matrix. Each such rectangle has as columns all non-zero columns of the A -factor matrix whose corresponding factors are not present in the intersection.

The RCF algorithm will produce the same rectangle covering if the factors in the intersection list are the only factors that occur several times in the lists of factors. Since the factors in each list $\mathcal{L}_{a,rcf}$ are at least as good as the factors in $\mathcal{L}_{a,pf}, \forall a \in \mathcal{A}$, the f-representation built by the RCF algorithm is at least as good as the f-representation built by the PF algorithm.

If there are factors which do not occur in the intersection list, but occur in the factor list of at least two relations \mathbf{R}_{a_i} and \mathbf{R}_{a_j} , $a_i \in \mathcal{A}$, $a_j \in \mathcal{A}$, $i \neq j$, the rectangle covering computed by the RCF algorithm includes rectangles which are not found by the PF algorithm and which span over several rows. The factors in such rectangles are represented fewer times in the f-representation produced by the RCF algorithm than in the f-representation computed by the PF algorithm. \square

Example 3.8. Consider relation R from Example 3.7 and f-tree \mathcal{T}_7 shown in Figure 3.10a. The f-representation produced by the RCF algorithm for relation R using any path that follows a topological sorting of f-tree \mathcal{T}_7 is

$$\begin{aligned} E_5 = & (\langle C : c_1 \rangle \cup \langle C : c_2 \rangle) \times (\langle A : a_1 \rangle \times (\langle B : b_1 \rangle \times (\langle D : d_1 \rangle \cup \langle D : d_2 \rangle) \cup \\ & \langle B : b_2 \rangle \times (\langle D : d_1 \rangle \cup \langle D : d_3 \rangle))) \times \langle E : e_1 \rangle \cup \\ & \langle A : a_3 \rangle \times \langle B : b_1 \rangle \times \langle D : d_1 \rangle \times \langle E : e_4 \rangle \cup \\ & \langle A : a_2 \rangle \times \langle B : b_3 \rangle \times \langle C : c_3 \rangle \times (\langle D : d_3 \rangle \cup \langle D : d_4 \rangle) \times \langle E : e_3 \rangle \end{aligned}$$

The f-representation E_5 produced by the RCF algorithm contains the factor $\langle C : c_1 \rangle \cup \langle C : c_2 \rangle$ only once, while f-representation E_4 produced by the PF algorithm for the same relation using the same path (discussed in Example 3.7), contains this factor twice. The PF algorithm does not factor out the f-representation $\langle C : c_1 \rangle \cup \langle C : c_2 \rangle$ since it occurs only for values a_1 and a_2 . The RCF algorithm, on the other hand, will factor out this union of singletons. \square

3.2.5 Experimental Evaluation

This section describes two experiments we performed for bulk insertions and their results. The first experiment considers relations representable over a given f-tree and reports execution time and representation sizes for all three bulk insertion algorithms presented in previous sections. The second experiment involves only the PF and RCF algorithms since it considers relations that are not necessarily representable over f-trees that present branching into several children, but the data they represent has some common patterns that can be identified by the PF and RCF algorithms and lead to compressed f-representations.

Experimental Setup All experiments were performed on an Intel(R) Xeon(R) X5650 2.67GHz/64bit/59GB running VMWare VM with Linux 3.5.0. All results (wall-clock time for the first experiment and representation size for the second experiment) were averaged over 10 runs.

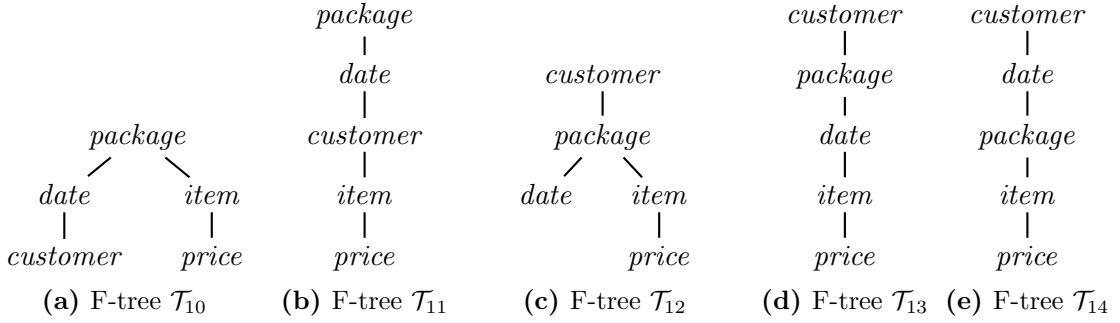
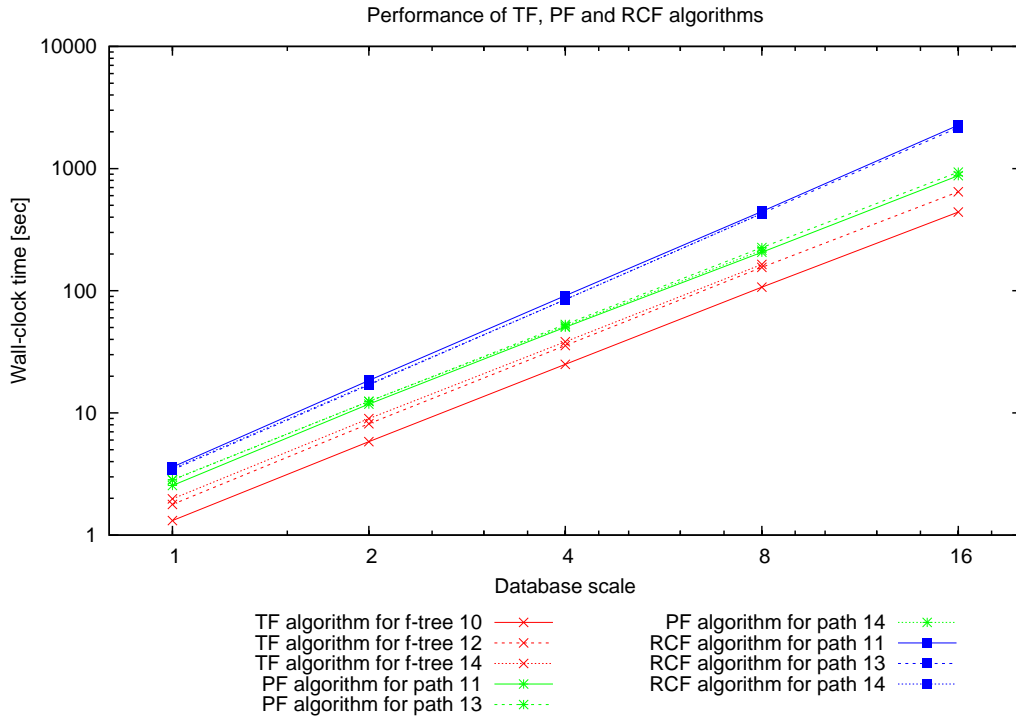


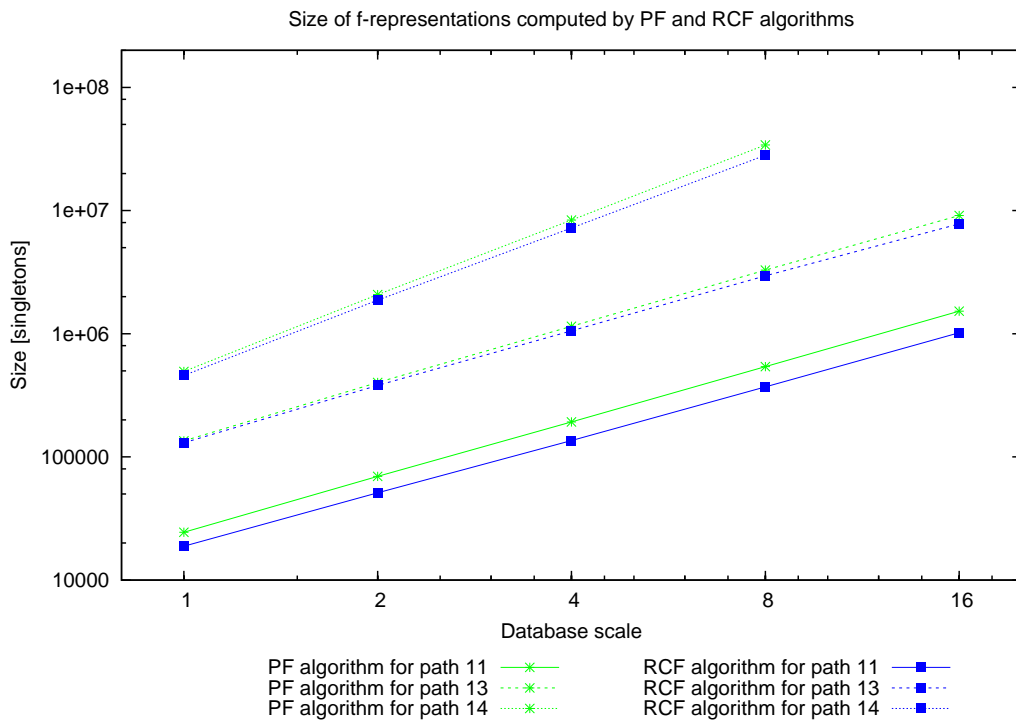
Figure 3.11: F-trees \mathcal{T}_{10} , \mathcal{T}_{11} , \mathcal{T}_{12} , \mathcal{T}_{13} and \mathcal{T}_{14}

Experimental Design We use the same synthetic dataset used in [5] for experimental evaluation. It consists of three relations: Orders(package, date, customer), Items(item, price) and Packages(package, item). We control its size using the scale parameter s : there are $100 \cdot s$ items, $40 \cdot s$ packages and $12800 \cdot s^2$ orders. We model a realistic scenario by scaling the data statistics as follows. The number of different dates scales linearly with s , the number of customers scales with \sqrt{s} . The number of dates each customer places an order follows a binomial distribution with mean proportional to \sqrt{s} . The average number of placed orders on such days is 2 and the ordered packages are chosen uniformly at random. We considered 5 database scales. For the largest scale, the natural join R of all relations has 66M tuples which makes 330M singletons. The size of the factorisation of this join relation over f-tree \mathcal{T}_{10} is 1.5M singletons. By scaling the database, the factorisation size grows as s^3 while the join size grows as s^4 .

Experiment 1 For this experiment, we ran the TF algorithm for relation R and f-trees \mathcal{T}_{10} , \mathcal{T}_{12} and \mathcal{T}_{14} shown in Figure 3.11. For the PF and RCF algorithms, we considered three different paths: \mathcal{T}_{11} , \mathcal{T}_{13} and \mathcal{T}_{14} shown by the same figure. Path \mathcal{T}_{11} follows a topological sorting of f-tree \mathcal{T}_{10} , so the PF algorithm using this path builds the f-representation of relation R over \mathcal{T}_{10} . Path \mathcal{T}_{13} does not follow a topological sorting of f-tree \mathcal{T}_{10} , but the PF algorithm still infers some products between partial f-representations. The nesting structure built by the PF algorithm using this path is represented by f-tree \mathcal{T}_{12} . The last path considered, \mathcal{T}_{14} , is also not following a topological sorting of f-tree \mathcal{T}_{10} , but unlike the previous case, the PF algorithm can no longer infer any products and the f-representation built is exactly the f-representation of R over path \mathcal{T}_{14} . We ran the bulk insertion algorithms for all 5 database scales, except for the case of path \mathcal{T}_{14} . The size of the f-representations built by all three algorithms using this path were too large to fit into main memory for scale 5.



(a) Performance of TF, PF and RCF algorithms for various f-trees and database scales.



(b) Size of representations computed by the PF and RCF algorithms for various paths and database scales. The size of the representation computed by the TF algorithm is equal to the size of the f-representation produced by the PF algorithm for all relation instances considered.

Figure 3.12: Bulk insertions experiment 1: Time and representation sizes for TF, PF and RCF algorithms.

Figure 3.12a shows the times for all three algorithms. As expected, the TF algorithm is the most efficient. The PF algorithm is slower than the TF algorithm, because at each recursive level, except for the last one, it has to compare several f-representations in order to compute the intersection list. To fasten this step, the f-representations are hashed to a positive number and only if the hash values are equal, the f-representations are compared. Even so, the PF algorithm has to compare partial f-representations whenever the intersection list is not empty, which explains the constant gap we have between the TF and PF algorithms. The RCF algorithm is the slowest. Similar to the PF algorithm, it has to compare at each recursion level, except for the last one, several partial f-representations in order to compute the union of all factor lists which is needed to build the factor matrix. In addition to this, it has to compute a rectangle covering for the factor matrix which takes polynomial time with respect to the size of the matrix. The gap between the PF and the RCF algorithms is increasing with the database scale because the factor matrices are larger for larger database scales.

Figure 3.12b shows the size of f-representations built by the PF and RCF algorithms. Representations sizes for the TF algorithm are not displayed since they are equal to the sizes of the representations produced by the PF algorithm in all cases considered. We notice that the f-representations built by the PF and RCF algorithms considering path \mathcal{T}_{11} are almost one order of magnitude smaller than the f-representations built using path \mathcal{T}_{13} . A large gap can also be observed between f-representations over path \mathcal{T}_{13} and f-representations over path \mathcal{T}_{14} . We also notice that the RCF algorithm produces an f-representation smaller than the f-representation computed by the PF algorithm for all cases considered. For path \mathcal{T}_{11} , the PF algorithm factors out partial f-representations over path $(item, price)$. A closer look at the rectangle coverings found by the RCF algorithm showed that, in addition to this, the RCF algorithm also factors out some unions of *customer* singletons or *price* singletons. For path \mathcal{T}_{13} , both PF and RCF algorithms factor out f-representations over path $(item, price)$, but the RCF algorithm also factors out some *price* singletons. In the case of path \mathcal{T}_{14} , the PF algorithm does not factor out anything, while the RCF algorithm factors out for certain tuple groups *price* singletons and partial f-representations over the path $(package, item, price)$.

Experiment 2 In this experiment we consider a single relation R representing the natural join of relations Orders, Packages and Items, with 237693 tuples and 40 distinct *package* values. Initially, this relation is representable over f-tree \mathcal{T}_{10} . We then randomly choose a number K of tuples and change the value of one of their attributes to a new value different from all values in the relation. The effect of this update is that all products present

in the f-representation of R over \mathcal{T}_{10} that contain updated values are broken and the relation is no longer representable over \mathcal{T}_{10} . It may contain tuple groups that are still representable over this f-tree since there may still be products that do not contain any updated values. This experiment examines the size of representations built by the PF and RCF algorithms when such updates are performed on various attributes of the relation taken as input. For all cases considered, both algorithms use path \mathcal{T}_{11} for representation construction.

Figure 3.13 shows the size of the representations when we update *package*, *date* and *customer* values. For all three attributes, the plots follow the same trend. The left column shows the evolution of the representation size for the RCF algorithm as the number of modified tuples increases from 0 to 100. The middle column compares the size of f-representations computed by the PF and RCF algorithms for the same interval of modified tuples. The size of the f-representation produced by the PF algorithm grows much faster than the size of the f-representation produced by the RCF algorithm such that for 100 tuples changed, the size for the PF algorithm is more than one order of magnitude larger than the size for the RCF algorithm. The representation sizes for the PF algorithm have a very high standard deviation for cases that change less than 100 tuples because for such cases the number of products broken for the same number of tuples changed varies highly and each product covers around 6000 tuples.

The right column displays representation sizes for both algorithms and larger numbers of updated values. We notice that the size of the f-representation computed by the PF algorithm continues to grow very fast until $K = 10000$, but then the increase is slower. For $K = 10000$, all tuples groups that have the same *date* and *package* value, but more than one *customer* value have at least one modified value, which means that from this point on, the PF algorithm is not able to factor out any f-representations. After this point, the increase of the representation size is linear since we continue to introduce distinct values for the updated attribute. The RCF algorithm follows a similar trend. We notice that the f-representation size for the RCF algorithm is closest to the f-representation size for the PF algorithm for $K = 50000$ when all products are broken. As the number of modified tuples increases, the RCF algorithm can factor out more f-representations that occur several times in the group of updated tuples, which explains the increasing gap we notice between the PF algorithm and RCF algorithm for $K > 50000$.

Overall, the main results discovered are the following:

- The PF and TF algorithms compute f-representations that have the same size if the f-tree considered by the TF algorithm can be inferred from the path used by the PF algorithm for representation construction. In general, the f-representation computed

by the PF algorithm can have a smaller size, but for all cases considered in the experiments performed, the algorithms built f-representations of the same size.

- The RCF algorithm can produce f-representations that are more than one order of magnitude smaller than the f-representations produced by the PF algorithm if both algorithms use the same path for representation construction.
- The path used by the PF and RCF algorithms for representation construction influences highly the size of the f-representations produced. The f-representations produced by the algorithms using paths that follow a topological sorting of an f-tree over which the relation is representable can be up to one order of magnitude smaller than the f-representations produced using paths that do not follow such a topological sorting.

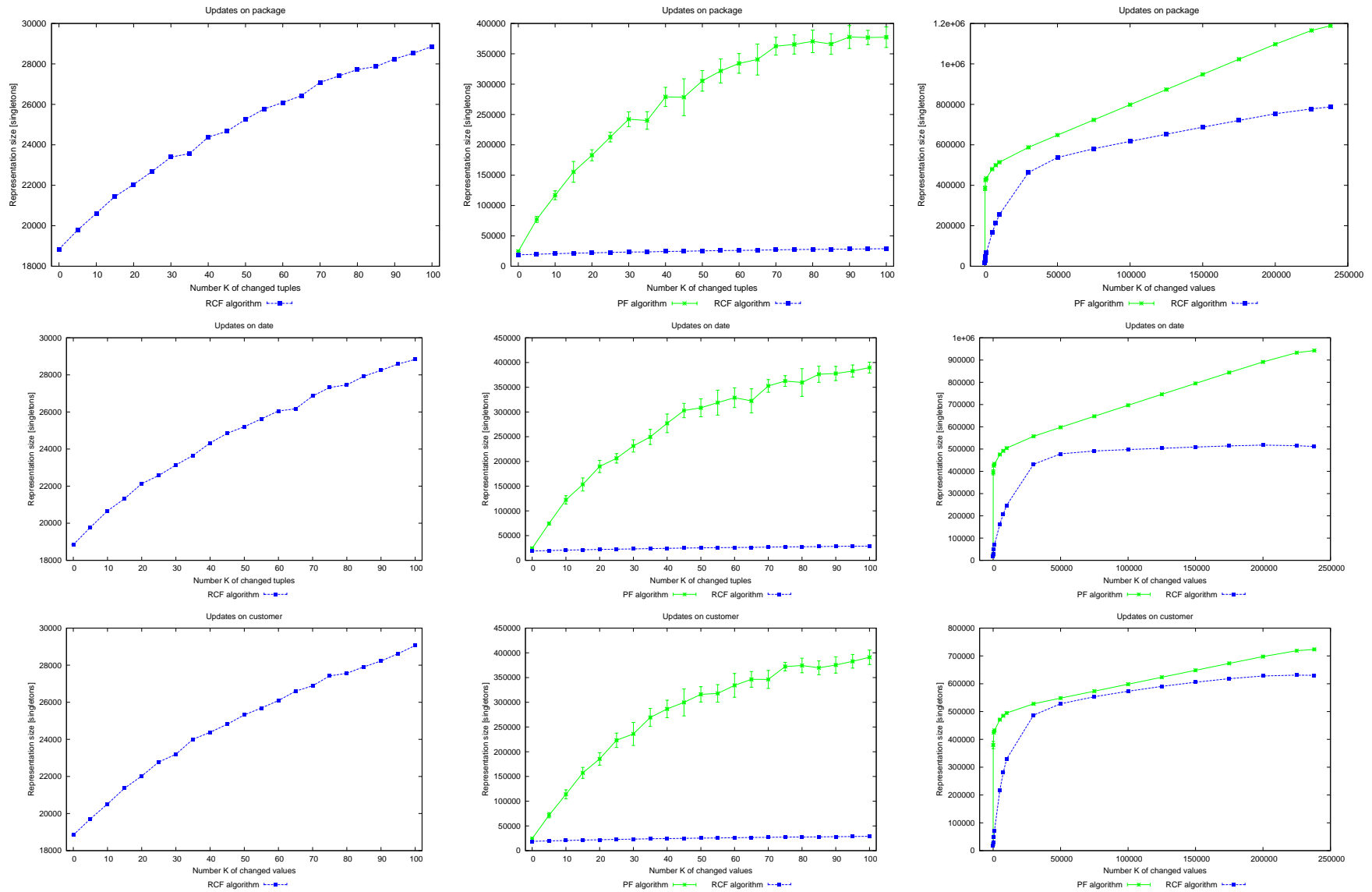


Figure 3.13: Bulk insertion experiment 2: updates on package, date and customer. The left column shows the size of the representations computed by RCF algorithm for $0 \leq K \leq 100$. The middle column shows the size of the representations computed by PF and RCF algorithms for $0 \leq K \leq 100$. The right column shows the size of the representations computed by PF and RCF algorithms for $0 \leq K \leq |R|$.

Chapter 4

Value Modifications

This chapter presents two techniques to change the values of an f-representation. Both techniques approach the value modification task from a static analysis perspective as the reasoning behind them considers only the shape of the f-tree that defines the f-representation to be updated and not the data in the f-representation. This approach is the norm in database research as it is performed on the small update statements, instead of the large data.

The first value modification technique described in this chapter produces a single result f-representation. Under this constraint, we characterise all f-trees that allow a given value modification statement without a prior restructuring step on any f-representation following their structure. We also present an algorithm that performs a given value modification statement using this approach. The algorithm was implemented on top of an existing implementation of a factorised database and experimentally evaluated against the SQLite open-source relational engine.

The conditions that the f-trees must satisfy in order to support a statement using the first value modification technique without restructuring can be relaxed by allowing the result of the query to be a union of f-representations over the same f-tree, instead of a single f-representation. This is the approach used by the second value modification technique described in this chapter. We characterise all f-trees that support a given statement without restructuring using this value modification technique and discuss the algorithm that builds the union of result f-representations from the f-representation to be updated.

We consider value modification statements of the following form:

$$\begin{aligned} U = \text{Update E set } & \bigwedge_{1 \leq i \leq m} A_i = f_i(A_{i_1}, \dots, A_{i_{s_i}}, c_{i_1}, \dots, c_{i_{t_i}}) \\ \text{where } & \bigwedge_{1 \leq j \leq n} B_j \theta_j g_j(B_{j_1}, \dots, B_{j_{s_j}}, c_{j_1}, \dots, c_{j_{t_j}}) \end{aligned}$$

where

- E is an f-representation of a relation \mathbf{R} over schema \mathcal{S}
- $A_i, A_{i_k}, 1 \leq k \leq s_i, 1 \leq i \leq m$ are attributes of schema \mathcal{S} of the same type
- $c_{i_k}, 1 \leq k \leq t_i$ are constant values of the same type as attribute $A_i, 1 \leq i \leq m$
- $B_j, B_{j_l}, 1 \leq l \leq s_j, 1 \leq j \leq n$ are attributes of \mathcal{S} of the same type
- $c_{j_k}, 1 \leq k \leq t_j$ are constant values of the same type as attribute $B_j, 1 \leq j \leq n$
- $f_i, 1 \leq i \leq m$ and $g_j, 1 \leq j \leq n$ are functions that take as arguments either values of attributes of \mathcal{S} , or constants
- $\theta_j, 1 \leq j \leq n$ are comparison operators

Throughout this chapter, we will refer to attributes $A_i, 1 \leq i \leq m$ in the assignment clause of the statement as target attributes and attributes $A_{i_k}, 1 \leq k \leq s_i$ as attributes that target attribute A_i depends on. For a given statement of the form shown above and an f-tree \mathcal{T} , we define the head attribute of condition $B_j \theta_j g_j(B_{j_1}, \dots, B_{j_{s_j}}, c_{j_1}, \dots, c_{j_{t_j}})$ as the attribute B_k occurring in this condition and labelling the node found at the lowest level in f-tree \mathcal{T} . If there are several such attributes, any of them can be used as a head attribute. All other attributes in a condition will be referred to as non-head attributes. For simplicity, we will assume that an attribute of \mathcal{S} is the head attribute of at most one condition, but all algorithms described in this chapter can easily be extended to admit several conditions with the same head attribute. Throughout this chapter and the following, we will use the notation $\mathcal{C}_{U,A}$ to denote the condition of a statement U with head attribute A . When the context is clear, we will drop the statement annotation.

4.1 Nesting Structures Supporting Efficient Value Modifications

This section studies value modification statements that can be performed on the f-representation given without changing its nesting structure. Since the f-representation does not require restructuring, these statements can be performed very efficiently.

4.1.1 The Case of a Result Consisting of One Factorised Representation

The following proposition characterises all f-trees that support a given value modification statement whose result consists of a single f-representation.

Proposition 4.1. *Consider an f-tree \mathcal{T} and a value modification statement U . We can perform statement U on any f-representation over \mathcal{T} without restructuring and obtain a single result f-representation if the following conditions are satisfied for each target attribute A of the statement:*

- 1 *The node labelled by A is on the same root-to-leaf path in \mathcal{T} with all nodes labelled by attributes that A depends on.*
- 2 *The level of the node labelled by attribute A in \mathcal{T} is greater or equal than the levels of all nodes labelled by attributes that A depends on.*
- 3 *The node labelled by attribute A is on the same root-to-leaf path in \mathcal{T} with all nodes labelled by attributes present in conditions of statement U .*
- 4 *The level of the node labelled by attribute A in \mathcal{T} is greater or equal than the levels of all nodes labelled by head attributes of conditions of statement U .*
- 5 *The subtree of \mathcal{T} rooted in A is a path.*

This proposition enforces that the attributes in all conditions of a statement are on the same root-to-leaf path. The target attributes can be spread over several paths, as long as they are found on the same path with the attributes they depend on and the attributes present in conditions and have lower levels than all these attributes.

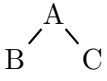
Let's see what could happen if conditions 1 or 2 in Proposition 4.1 are not satisfied. Let B be an attribute that a target attribute A depends on found in \mathcal{T} either on another path than A , or at a lower level. In both cases, an f-representation over \mathcal{T} may contain a product between an A -singleton and a union of B -singletons. Since this A -singleton is "shared" by all B -singletons in the union, the update cannot be performed.

A similar situation is encountered whenever conditions 3 or 4 are not satisfied. Let C be the head attribute of a condition of the value modification statement found either on another path than a target attribute A , or at a lower level. In such cases, an f-representation over \mathcal{T} may contain a product between an A -singleton and a union of C -singletons. The update cannot be performed because for some of the C -singletons in this union the condition may be satisfied, while for others it may not be satisfied.

The update can be performed when condition 5 is not satisfied, but the result f-representation might violate two constraints imposed on factorised representations in previous work [18]:

- For any union expression $\bigcup_a \langle A : a \rangle$ or $\bigcup_a \langle A : a \rangle \times E_a$ in the f-representation, all values a are distinct.
- The products of singletons obtained by expanding the f-representation using the distributivity of product over union are distinct.

If we expect a result f-representation that satisfies the conditions mentioned above, then an additional step will need to be performed after the value modification statement is performed. During this step, we merge all singletons inside a union that have the same value. If the singletons merged are found in products with other f-representations, then we will have to merge those f-representations too in a single f-representation representing the data in all the f-representations merged. Merging such f-representations is not always possible unless the f-tree defining their structure is a single path.

Example 4.1. Recall f-tree \mathcal{T}_1 :  from Example 2.1 and consider the following f-representation over it:

$$E_6 = \langle A : 1 \rangle \times (\langle B : 1 \rangle \cup \langle B : 2 \rangle) \times (\langle C : 2 \rangle \cup \langle C : 3 \rangle) \cup \\ \langle A : 2 \rangle \times (\langle B : 2 \rangle \cup \langle B : 4 \rangle) \times (\langle C : 2 \rangle \cup \langle C : 5 \rangle)$$

Consider also the value modification statement

$$U_3 = \text{Update } E_6 \text{ set } A = 3$$

This statement satisfies conditions 1-4 of Proposition 4.1, so the update can be performed. The result f-representation is

$$E_7 = \langle A : 3 \rangle \times (\langle B : 1 \rangle \cup \langle B : 2 \rangle) \times (\langle C : 2 \rangle \cup \langle C : 3 \rangle) \cup \\ \langle A : 3 \rangle \times (\langle B : 2 \rangle \cup \langle B : 4 \rangle) \times (\langle C : 2 \rangle \cup \langle C : 5 \rangle)$$

Notice that singleton $\langle A : 3 \rangle$ occurs twice in the top-most union of E_7 and the product of singletons $\langle A : 3 \rangle \times \langle B : 2 \rangle \times \langle C : 2 \rangle$ occurs twice if we expand E_7 to a single union of products of singletons.

The merging step following the value modification should change E_7 such that the new f-representation has the form $\langle A : 3 \rangle \times E_B \times E_C$, where E_B is a union of B -singletons and E_C is a union of C -singletons. Such an f-representation cannot be built because we cannot build a single product between a union of B -singletons and a union of C singletons to represent the following union of products:

$$(\langle B : 1 \rangle \cup \langle B : 2 \rangle) \times (\langle C : 2 \rangle \cup \langle C : 3 \rangle) \cup (\langle B : 2 \rangle \cup \langle B : 4 \rangle) \times (\langle C : 2 \rangle \cup \langle C : 5 \rangle) \quad \square$$

If the subtree \mathcal{T}_A rooted in a target attribute A is a single path, the merging procedure can always be performed because the only products present in an f-representations over \mathcal{T}_A are between a singleton and a union of other f-representations. The procedure should be applied to each union of target singletons after the f-representation has been updated and should follow the steps described below:

```

procedure VALUEMODIFICATION(f-tree  $\mathcal{T}$ , f-representation  $E$ , statement  $U$ )
  if  $\mathcal{T}$  is empty then return
  if  $\mathcal{T}$  is a forest  $\mathcal{T}_1, \dots, \mathcal{T}_k$  then
    Let  $E = E_1 \times \dots \times E_k$ 
    for each  $\mathcal{T}_i$  whose schema contains a target attribute of statement  $U$  do
      valueModification( $\mathcal{T}_i, E_i, U$ )
  if  $\mathcal{T}$  is a single rooted tree  $A(\mathcal{U})$  then
    Let  $E = \bigcup_a \langle A : a \rangle \times E_a$ 
    for each singleton  $\langle A : a \rangle$  in  $E$  do
      if  $A$  is not the head attribute of any condition of  $U$  or
       $A$  is head attribute of a condition  $\mathcal{C}_A$  of  $U$  and value  $a$  satisfies  $\mathcal{C}_A$  then
        if  $A$  is a target attribute of statement  $U$  then
          Update value  $a$  of singleton  $\langle A : a \rangle$ 
        if the schema of  $\mathcal{U}$  contains target attributes of statement  $U$  then
          valueModification( $\mathcal{U}, E_a, U$ )
    Sort items of union  $\bigcup_a \langle A : a \rangle \times E_a$  using values  $a$  as sorting keys

```

Figure 4.1: Value modification procedure that generates a result consisting of a single f-representation

- If the target attribute A labels the leaf of \mathcal{T}_A , then the union has the form $E = \bigcup_{a \in \mathcal{A}} \langle A : a \rangle$, where \mathcal{A} is a multiset. The new union will keep a single instance of each distinct singleton in the initial union E .
- If the target attribute A does not label the leaf of \mathcal{T}_A , let B be the attribute labelling its child. Depending on whether B labels the leaf of \mathcal{T}_A or not, the union will have one of the following forms

$$E = \bigcup_{a \in \mathcal{A}} \langle A : a \rangle \times \left(\bigcup_{b \in \mathcal{B}} \langle B : b \rangle \right) \quad \text{or} \quad E = \bigcup_{a \in \mathcal{A}} \langle A : a \rangle \times \left(\bigcup_{b \in \mathcal{B}} \langle B : b \rangle \times E_b \right),$$

where \mathcal{A} and \mathcal{B} are multisets

The main idea is to factor out A -singletons that occur several times in union E . We keep a single instance of each distinct singleton $\langle A : a \rangle$ of E and put it in a product with the union of the B -singletons occurring in all unions that were found in the initial f-representation in a product with singleton $\langle A : a \rangle$. The newly formed unions of B -singletons may contain a B -singleton several times, since the singletons come from different unions of B -singletons of E . We will have to recursively apply the same procedure to each newly formed union of B -singletons. The procedure will, in fact, be recursively applied until we reach unions over the attribute labelling the leaf of \mathcal{T}_A .

A	B	C	
1	3	7	$E_8 = \langle A : 1 \rangle \times (\langle B : 3 \rangle \cup \langle B : 5 \rangle) \times \langle C : 7 \rangle$ $\langle A : 1 \rangle \times (\langle B : 4 \rangle \cup \langle B : 6 \rangle) \times \langle C : 6 \rangle$
1	5	7	
1	4	6	
1	6	6	

(a) Relation R_3

(b) Relation R_3 represented as a union of f-representations over f-tree \mathcal{T}_1

Figure 4.3: Representation of a relation as a union of f-representations over the same f-tree

Figure 4.1 shows the pseudo-code of a procedure that performs a value modification statement U on an f-representation E over f-tree \mathcal{T} and produces a single result representation. The procedure updates E in-place and can be applied only if conditions 1-4 of Proposition 4.1 are satisfied for statement U and f-tree \mathcal{T} . Notice that the procedure checks a condition of U when it reaches a singleton corresponding to the head attribute of the condition. Among the attributes involved in the condition, the head attribute is found at the lowest level in f-tree \mathcal{T} . We can check the condition at this point because the values of all other attributes involved in the condition are known. After performing this procedure, the merging procedure we previously described should be applied to the result representation.

The time complexity of this algorithm is $O(|E|)$. It may be even sublinear in some cases because the procedure scans only singletons in unions $\bigcup_a \langle A : a \rangle \times E_a$ or $\bigcup_a \langle A : a \rangle$ where A is either a target attribute, or an ancestor of a target attribute.

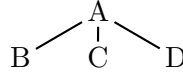
4.1.2 The Case of a Result Consisting of a Union of Factorised Representations over the Same Factorisation Tree

We can relax the conditions stated in Proposition 4.1 and enlarge the class of f-trees that support a given value modification statement without restructuring by representing the result of the statement as a union of f-representations over the same f-tree. This new formalism for relation representation is more expressive than the formalism considered so far (which represented relations only as a single f-representation over an f-tree), since there are relations that are representable as a union of f-representations over the same f-tree, but not representable as a single f-representation over the same f-tree. The following example illustrates this case.

Example 4.2. Recall f-tree \mathcal{T}_1 from Example 2.1. Consider also relation R_3 shown in Figure 4.3a. This relation does not admit an f-representation over f-tree \mathcal{T}_1 , since any such representation should have the form $\langle A : 1 \rangle \times E_B \times E_C$, where E_B is a union of B -singletons and E_C is a union of C -singletons. However, it can be represented as a union of f-representations over f-tree \mathcal{T}_1 . Figure 4.3b shows this representation. □

The main idea of the approach proposed in this section for value modification is to split the f-representation to be updated into several f-representations over the same f-tree using the distributivity of product over union such that one of these factorisations represents only tuples of the relation represented that satisfy all conditions of the statement. Since this f-representation satisfies all conditions, we can update all its target attributes. All other f-representations in the union computed will represent tuples that violate at least one of the conditions and do not need to be updated.

Example 4.3. Consider f-tree \mathcal{T}_{15} :



and the value modification statement

$$U_4 = \text{Update } E_9 \text{ set } A = 5 \text{ where } B < 3 \text{ and } C = 5 \text{ and } D > 7$$

where E_9 is an f-representation over \mathcal{T}_{15} . This statement cannot be performed on any f-representation over \mathcal{T}_{15} and produce a single result representation because the attributes in its conditions are spread over three different paths. E_9 has the following form:

$$\begin{aligned} E_9 &= \bigcup_a \langle A : a \rangle \times \left(\bigcup_b \langle B : b \rangle \right) \times \left(\bigcup_c \langle C : c \rangle \right) \times \left(\bigcup_d \langle D : d \rangle \right) \\ &= \bigcup_a \langle A : a \rangle \times (E_B^+ \cup E_B^-) \times (E_C^+ \cup E_C^-) \times (E_D^+ \cup E_D^-) \end{aligned}$$

where E_B^+ represents the union of B -singletons that satisfy the condition of statement U_4 on attribute B and E_B^- represents the union of B -singletons that do not satisfy this condition. Similar notations have been used for attributes C and D . We can split E_9 into four f-representations using the distributivity of product over union:

$$\begin{aligned} E_9 &= \bigcup_a \langle A : a \rangle \times (E_B^+ \cup E_B^-) \times (E_C^+ \cup E_C^-) \times (E_D^+ \cup E_D^-) \\ &= \bigcup_a \langle A : a \rangle \times E_B^+ \times E_C^+ \times E_D^+ \cup \\ &\quad \bigcup_a \langle A : a \rangle \times E_B^+ \times E_C^+ \times E_D^- \cup \\ &\quad \bigcup_a \langle A : a \rangle \times E_B^+ \times E_C^- \times (E_D^+ \cup E_D^-) \cup \\ &\quad \bigcup_a \langle A : a \rangle \times E_B^- \times (E_C^+ \cup E_C^-) \times (E_D^+ \cup E_D^-) \end{aligned}$$

Notice that all four representations follow the structure of \mathcal{T}_{15} . Notice also that the first f-representation has only B , C and D values that satisfy the conditions of statement U_4 and all A -singletons in this f-representations can be updated. The second f-representation has only D -singletons that do not satisfy condition $\mathcal{C}_{U_4,D}$, the third f-representation contains only C -singletons that violate condition $\mathcal{C}_{U_4,C}$ and the last representation has only singletons

that do not satisfy condition $\mathcal{C}_{U_4, B}$, so none of the A -singletons in these f-representations needs to be updated.

Notice also that after we perform the value modification, the relation represented might not necessarily be representable over f-tree \mathcal{T}_{15} , but it is still representable as a union of f-representations over this f-tree. \square

The following proposition characterises all f-trees that support a given value modification statement whose result consists of a union of f-representations over the same f-tree.

Proposition 4.2. *Consider an f-tree \mathcal{T} and a value modification statement U . We can perform statement U on any f-representation over \mathcal{T} without restructuring and obtain a result consisting of a union of f-representations over the same f-tree if*

- *For each target attribute A , the following conditions are satisfied:*
 - 1 *The node labelled by A is on the same root-to-leaf path in \mathcal{T} with all nodes labelled by attributes that A depends on.*
 - 2 *The level of the node labelled by attribute A in \mathcal{T} is greater or equal to the levels of all nodes labelled by attributes that A depends on.*
 - 3 *The subtree of \mathcal{T} rooted in A is a path.*
- *For each condition \mathcal{C} of statement U , all nodes of \mathcal{T} labelled by attributes involved in condition \mathcal{C} are on the same root-to-leaf path of \mathcal{T} .*

The conditions that an f-tree \mathcal{T} has to satisfy in order to support a statement that generates a union of result representations are less tight than the conditions for the f-tree to support a statement that generates a single result representation from two points of view:

- We no longer need to have each target attribute on the same root-to-leaf path of \mathcal{T} with all attributes in conditions of the given statement.
- We no longer need to have the target attributes at lower levels in \mathcal{T} than all attributes in conditions of the given statement.

The constraints presented above can be discarded for a value modification statement that produces a union of result f-representations, because in this case we no longer need to check the conditions before updating a target value. We will update a single f-representation that contains only target values for which all conditions of the statement are satisfied.

In order to be able to describe formally the union of result representations for a given statement, we need to define several concepts.

Definition 4.1. Let \mathcal{T} be an f-tree and U be a value modification statement on an f-representation over \mathcal{T} . A *metacondition* M of statement U and f-tree \mathcal{T} is a subset of the set of conditions of U such that the attributes of all conditions in M are on the same root-to-leaf path in \mathcal{T} .

Let the schema of \mathcal{T} be $\mathcal{S} = \{A_1, \dots, A_n\}$. Consider also the product of singletons $P = \langle A_1 : a_1 \rangle \times \dots \times \langle A_n : a_n \rangle$. We say that P satisfies a metacondition M of U and \mathcal{T} if it satisfies all conditions of M . We say that P does not satisfy metacondition M if there is at least one condition in M that the product does not satisfy. We will use the same terminology for a tuple $t = (A_1 : a_1, A_2 : a_2, \dots, A_n : a_n)$.

Definition 4.2. Consider a metacondition M of an f-tree \mathcal{T} and a value modification statement U on an f-representation over \mathcal{T} . Let H_M be the set of head attributes of all conditions of M . The *head attribute of metacondition* M is the attribute in H_M found at the lowest level in \mathcal{T} .

All other condition head attributes in H_M will be referred to as non-head attributes of metacondition M .

Definition 4.3. Let \mathcal{T} be an f-tree and U be a value modification statement on an f-representation over \mathcal{T} . A *metacondition partition* of statement U and f-tree \mathcal{T} is a partition of the set of conditions of U such that each set of the partition is a metacondition of statement U and f-tree \mathcal{T} .

Definition 4.4. Consider an f-tree \mathcal{T} , a value modification statement U on an f-representation over \mathcal{T} and a metacondition partition \mathcal{P} of U and \mathcal{T} . We say that \mathcal{P} is a *minimal metacondition partition* if there is no other metacondition partition of U and \mathcal{T} of a smaller size.

Example 4.4. Consider f-tree \mathcal{T}_6 in Figure 3.5 and the value modification statement

$$U_5 = \text{Update } E_{10} \text{ set } E = 7 \text{ where } A = 3 \text{ and } B = 4 \text{ and } C = 5 \text{ and } D = 6 \text{ and } F = 8$$

where E_{10} is an f-representation over \mathcal{T}_6 .

An example of a metacondition partition of U_5 and \mathcal{T}_6 is $\{\{\mathcal{C}_A, \mathcal{C}_B\}, \{\mathcal{C}_C\}, \{\mathcal{C}_D\}, \{\mathcal{C}_F\}\}$. This is not a minimal metacondition partition because we can find other partitions with a smaller size. Below are a few examples of minimal metacondition partitions of U_5 and \mathcal{T}_6 :

$$\begin{aligned} & \{\{\mathcal{C}_A, \mathcal{C}_B, \mathcal{C}_C\}, \{\mathcal{C}_D\}, \{\mathcal{C}_F\}\}, & \{\{\mathcal{C}_A, \mathcal{C}_B, \mathcal{C}_D\}, \{\mathcal{C}_C\}, \{\mathcal{C}_F\}\} \\ & \{\{\mathcal{C}_A, \mathcal{C}_F\}, \{\mathcal{C}_B, \mathcal{C}_C\}, \{\mathcal{C}_D\}\}, & \{\{\mathcal{C}_A, \mathcal{C}_F\}, \{\mathcal{C}_B, \mathcal{C}_D\}, \{\mathcal{C}_C\}\} \end{aligned} \quad \square$$

The following proposition characterises the size of a minimal metacondition partition of a value modification statement and an f-tree.

Proposition 4.3. Consider any minimal metacondition partition \mathcal{P} of an f-tree \mathcal{T} and a value modification statement U on an f-representation over \mathcal{T} . The size of \mathcal{P} is equal to the the number of paths in \mathcal{T} that contain attributes involved in conditions of U .

The intuition behind this proposition is evident. Each metacondition of partition \mathcal{P} should include only conditions whose attributes lie on the same root-to-leaf path in \mathcal{T} , so we cannot have less metaconditions in a partition of U and \mathcal{T} than the number of paths in \mathcal{T} that contain attributes present in conditions of statement U .

In any minimal metacondition partition, we have exactly one metacondition for each path in \mathcal{T} that contains attributes involved in conditions of statement U . These paths are not necessary root-to-leaf paths, they can be partial paths of root-to-leaf paths. If the head attribute of a condition is part of several such paths, we can place the condition in the metacondition corresponding to any of these paths. The fact that some conditions can be placed in several metaconditions is precisely the reason we can have several minimal metacondition partitions.

Definition 4.5. Consider an f-tree \mathcal{T} , an f-representation E of a relation \mathbf{R} over \mathcal{T} and a value modification statement U on E . Consider also a set \mathcal{M} of metaconditions of U and \mathcal{T} , where each metacondition can be either positively annotated, or negatively annotated. A *fragment f-representation* of E and \mathcal{M} is an f-representation over \mathcal{T} that represents only a subset of the tuples of relation \mathbf{R} . Each tuple represented by the fragment f-representation satisfies all positively annotated conditions in \mathcal{M} and does not satisfy any negatively annotated condition in \mathcal{M} .

The fragment f-representation of an f-representation E and a set of annotated metaconditions \mathcal{M} will be denoted throughout this chapter and the following by $F(E, \mathcal{M})$. A fragment f-representation $F(E, \mathcal{M})$ can always be represented over f-tree \mathcal{T} defining the structure of E because each metacondition $M \in \mathcal{M}$ consists of conditions whose attributes are on the same root-to-leaf path of f-tree \mathcal{T} .

Example 4.5. Consider f-representation E_{11} over f-tree \mathcal{T}_{16} , both shown in Figure 4.5. Also consider the following value modification statement:

$$U_6 = \text{Update } E_{11} \text{ set } A = 12 \text{ where } A \leq B \text{ and } B \leq D \text{ and } C \leq E$$

A fragment f-representation of E_{11} is shown below:

$$\begin{aligned} F(E_{11}, \{\{C_{U_6, E}\}^+, \{C_{U_6, B}, C_{U_6, D}\}^-\}) = & \langle A : 4 \rangle \times (\langle B : 2 \rangle \times (\langle D : 1 \rangle \cup \langle D : 2 \rangle \cup \langle D : 3 \rangle \cup \langle D : 4 \rangle)) \cup \\ & \langle B : 5 \rangle \times (\langle D : 2 \rangle \cup \langle D : 3 \rangle) \cup \\ & \langle B : 7 \rangle \times (\langle D : 4 \rangle \cup \langle D : 5 \rangle) \cup \\ & \times (\langle C : 7 \rangle \times \langle E : 8 \rangle \cup \langle C : 11 \rangle \times \langle E : 12 \rangle) \end{aligned}$$

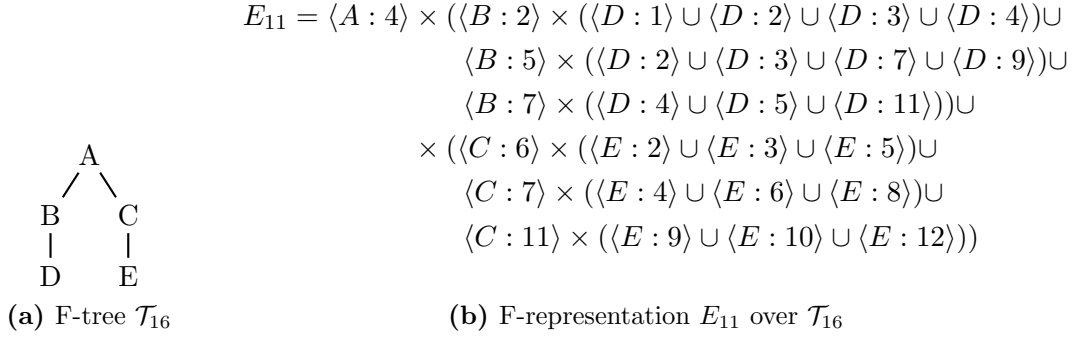


Figure 4.5: F-tree \mathcal{T}_{16} and f-representation E_{11}

Notice that all tuples represented by $F(E_{11}, \{\{\mathcal{C}_{U_6,B}, \mathcal{C}_{U_6,D}\}^-, \{\mathcal{C}_{U_6,E}\}^+\})$ satisfy metacondition $\{\mathcal{C}_{U_6,E}\}$. None of them satisfies metacondition $\{\mathcal{C}_{U_6,B}, \mathcal{C}_{U_6,D}\}$; for each tuple represented by the fragment f-representation shown above either condition $\mathcal{C}_{U_6,B}$, or condition $\mathcal{C}_{U_6,D}$, or both are not satisfied. \square

The fragment f-representation $F(E, \mathcal{M})$ can be computed directly from E , without first flattening E , finding all products of singletons that correspond to tuples that $F(E, \mathcal{M})$ should represent and refactorising them over the structure of the f-tree defining E . Figure 4.6 shows the pseudo-code of the procedure that builds the fragment f-representation $F(E, \mathcal{M})$ directly from E . The fragment representation is built while scanning E . Whenever the procedure reaches a union of A -singletons, it keeps all singletons in the union scanned if attribute A is not head attribute of any condition contained by a metacondition of \mathcal{M} . If A is a condition head attribute, it selects only a part of the singletons in the union scanned, depending on whether the values of the singletons satisfy or not the condition on A and whether the metacondition containing the condition is positively annotated or negatively annotated:

- If the metacondition containing the condition on A is positively annotated, we keep only the singletons whose values satisfy the condition.
- If A is a non-head attribute of a negatively annotated metacondition, we keep all A -singletons in the current union. For a singleton $\langle A : a \rangle$, let E_a be the f-representation found in a product with it. We have the following difference between singletons that satisfy the condition on A and singletons that do not satisfy it:
 - If value a of singleton $\langle A : a \rangle$ does not satisfy the condition, then the fragment f-representation will contain the product $\langle A : a \rangle \times E_a$. The idea is that the fragment f-representation we are building should represent all tuples that do not satisfy the

```

procedure BUILDFRAGMENT( $E, \mathcal{T}$ , set of annotated metaconditions  $\mathcal{M}$ )
  if  $\mathcal{T}$  is empty then return  $E$ 
  if  $\mathcal{T}$  is a forest  $\mathcal{T}_1, \dots, \mathcal{T}_k$  then
    Let  $E = E_1 \times \dots \times E_k$ 
    return buildFragment( $E_1, \mathcal{T}_1, \mathcal{M}$ )  $\times \dots \times$  buildFragment( $E_k, \mathcal{T}_k, \mathcal{M}$ )
  if  $\mathcal{T}$  is a single node  $A$  then
    if  $A$  is not head attribute of any condition in metaconditions of  $\mathcal{M}$  then
      return  $E$ 
    if  $A$  is head attribute of a condition in a metacondition  $M \in \mathcal{M}$  then
      Let  $\mathcal{C}_A$  be the condition whose head attribute is  $A$ 
      Let  $E = \bigcup_{a \in \mathcal{A}} \langle A : a \rangle$ 
       $\mathcal{R} = \emptyset$ 
      for each singleton  $\langle A : a \rangle$  in  $E$  do
        if  $M$  is positively annotated and value  $a$  satisfies condition  $\mathcal{C}_A$  or
           $M$  is negatively annotated and value  $a$  does not satisfy  $\mathcal{C}_A$  then
             $\mathcal{R} = \mathcal{R} \cup \langle A : a \rangle$ 
      return  $\mathcal{R}$ 
  if  $\mathcal{T}$  is a single rooted tree  $A(\mathcal{U})$  then
    Let  $E = \bigcup_{a \in \mathcal{A}} \langle A : a \rangle \times E_a$ 
    if  $A$  is not head attribute of any condition in metaconditions of  $\mathcal{M}$  then
      return  $\bigcup_{a \in \mathcal{A}} \langle A : a \rangle \times$  buildFragment( $E_a, \mathcal{U}, \mathcal{M}$ )
    if  $A$  is head attribute of a condition in a metacondition  $M \in \mathcal{M}$  then
      Let  $\mathcal{C}_A$  be the condition whose head attribute is  $A$ 
       $\mathcal{R} = \emptyset$ 
      for each singleton  $\langle A : a \rangle$  in  $E$  do
        if  $A$  is a non-head attribute of  $M$  then
          if  $M$  is positively annotated and value  $a$  satisfies  $\mathcal{C}_A$  then
             $\mathcal{R} = \mathcal{R} \cup \langle A : a \rangle \times$  buildFragment( $E_a, \mathcal{U}, \mathcal{M}$ )
          if  $M$  is negatively annotated then
            if value  $a$  satisfies condition  $\mathcal{C}_A$  then
               $\mathcal{R} = \mathcal{R} \cup \langle A : a \rangle \times$  buildFragment( $E_a, \mathcal{U}, \mathcal{M}$ )
            else
               $\mathcal{R} = \mathcal{R} \cup \langle A : a \rangle \times E_a$ 
        if  $A$  is the head attribute of  $M$  then
          if  $M$  is positively annotated and value  $a$  satisfies  $\mathcal{C}_A$  or
             $M$  is negatively annotated and value  $a$  does not satisfy  $\mathcal{C}_A$  then
               $\mathcal{R} = \mathcal{R} \cup \langle A : a \rangle \times E_a$ 
      return  $\mathcal{R}$ 

```

Figure 4.6: Construction of a fragment f-representation

current metacondition. Since we already have value a that violates a condition in the current metacondition, we need to keep the entire f-representation E_a .

- If value a of singleton $\langle A : a \rangle$ satisfies the condition on A , then the fragment f-representation will contain the product $\langle A : a \rangle \times F(E_a, \mathcal{M})$. Since the current metacondition is satisfied so far, we need to put singleton $\langle A : a \rangle$ in a product with a factorisation representing tuples that violate this metacondition.

- If A is the head attribute of a negatively annotated metacondition, we keep only A -singletons in the current union that do not satisfy the condition on A . If we reached a union of such singletons, it means that all other conditions in the current metacondition are satisfied. In order for the fragment f-representation we are building to represent only tuples that violate the current metacondition, we need to eliminate all A -singletons that satisfy the condition on A .

Notice that procedure *buildFragment* can return \emptyset as a result if none of the singletons in the current union is selected. We can use the following rules to remove all \emptyset occurrences from the result representation, if there are any:

- Any product containing \emptyset as a factor is entirely replaced by \emptyset .
- If \emptyset is placed inside a union that contains other non-empty items, then the empty item is simply removed. If the union consists only of empty items, then we replace the union entirely with \emptyset .

The time complexity of procedure *buildFragment* is $O(|E|)$, where E is the f-representation for which we build the fragment.

Definition 4.6. Let \mathcal{T} be an f-tree, E be an f-representation over \mathcal{T} and U be a value modification statement on E . Consider also a set of metaconditions \mathcal{M} of U and \mathcal{T} .

- For an empty set \mathcal{M} , we define the *f-set* of E and \mathcal{M} as $\mathcal{F} = \{E\}$.
- For a non-empty set $\mathcal{M} = \{M_1, M_2, \dots, M_n\}$, we define the *f-set* of E and \mathcal{M} as a set of fragment f-representations of E , $\mathcal{F} = \{F_0, F_1, \dots, F_n\}$, where

$$\begin{aligned}
 F_0 &= F(E, \{M_1^+, M_2^+, M_3^+, \dots, M_{n-2}^+, M_{n-1}^+, M_n^+\}) \\
 F_1 &= F(E, \{M_1^+, M_2^+, M_3^+, \dots, M_{n-2}^+, M_{n-1}^+, M_n^-\}) \\
 F_2 &= F(E, \{M_1^+, M_2^+, M_3^+, \dots, M_{n-2}^+, M_{n-1}^-\}) \\
 F_3 &= F(E, \{M_1^+, M_2^+, M_3^+, \dots, M_{n-2}^-\}) \\
 &\dots
 \end{aligned}$$

$$F_{n-2} = F(E, \{M_1^+, M_2^+, M_3^-\})$$

$$F_{n-1} = F(E, \{M_1^+, M_2^-\})$$

$$F_n = F(E, \{M_1^-\})$$

Example 4.6. Consider f-tree \mathcal{T}_{16} and f-representation E_{11} shown in Figure 4.5. Consider also the value modification statement U_6 from Example 4.5 and the minimal metacondition partition $\mathcal{P}_1 = \{\{\mathcal{C}_E\}, \{\mathcal{C}_B, \mathcal{C}_D\}\}$ of U_6 and \mathcal{T}_{16} . The f-set of E and \mathcal{P}_1 is $\mathcal{F} = \{F_0, F_1, F_2\}$, where

$$F_0 = F(E, \{\{\mathcal{C}_E\}^+, \{\mathcal{C}_B, \mathcal{C}_D\}^+\})$$

$$\begin{aligned} &= \langle A : 4 \rangle \times (\langle B : 5 \rangle \times (\langle D : 7 \rangle \cup \langle D : 9 \rangle)) \cup \langle B : 7 \rangle \times \langle D : 11 \rangle \\ &\quad \times (\langle C : 7 \rangle \times \langle E : 8 \rangle \cup \langle C : 11 \rangle \times \langle E : 12 \rangle) \end{aligned}$$

$$F_1 = F(E, \{\{\mathcal{C}_E\}^+, \{\mathcal{C}_B, \mathcal{C}_D\}^-\})$$

$$\begin{aligned} &= \langle A : 4 \rangle \times (\langle B : 2 \rangle \times (\langle D : 1 \rangle \cup \langle D : 2 \rangle \cup \langle D : 3 \rangle \cup \langle D : 4 \rangle)) \cup \\ &\quad \langle B : 5 \rangle \times (\langle D : 2 \rangle \cup \langle D : 3 \rangle) \cup \langle B : 7 \rangle \times (\langle D : 4 \rangle \cup \langle D : 5 \rangle) \cup \\ &\quad \times (\langle C : 7 \rangle \times \langle E : 8 \rangle \cup \langle C : 11 \rangle \times \langle E : 12 \rangle) \end{aligned}$$

$$F_2 = F(E, \{\{\mathcal{C}_E\}^-\})$$

$$\begin{aligned} &= \langle A : 4 \rangle \times (\langle B : 2 \rangle \times (\langle D : 1 \rangle \cup \langle D : 2 \rangle \cup \langle D : 3 \rangle \cup \langle D : 4 \rangle)) \cup \\ &\quad \langle B : 5 \rangle \times (\langle D : 2 \rangle \cup \langle D : 3 \rangle \cup \langle D : 7 \rangle \cup \langle D : 9 \rangle) \cup \\ &\quad \langle B : 7 \rangle \times (\langle D : 4 \rangle \cup \langle D : 5 \rangle \cup \langle D : 11 \rangle) \cup \\ &\quad \times (\langle C : 6 \rangle \times (\langle E : 2 \rangle \cup \langle E : 3 \rangle \cup \langle E : 5 \rangle)) \cup \\ &\quad \langle C : 7 \rangle \times (\langle E : 4 \rangle \cup \langle E : 6 \rangle) \cup \langle C : 11 \rangle \times (\langle E : 9 \rangle \cup \langle E : 10 \rangle) \quad \square \end{aligned}$$

Theorem 4.7. Let \mathcal{T} be an f-tree, E be an f-representation over \mathcal{T} and U be a value modification statement on E . Consider also a set of metaconditions \mathcal{M} of U and \mathcal{T} , $\mathcal{M} = \{M_1, M_2, \dots, M_n\}$ and the f-set $\mathcal{F} = \{F_0, F_1, \dots, F_n\}$ of E and \mathcal{M} . The following statements are true:

1 \mathcal{F} represents the same relation \mathbf{R} that E represents.

2 \mathcal{F} represents each tuple of relation \mathbf{R} exactly once.

Proof. Let $\mathcal{P}_{\mathcal{F}}$ be the set of tuples represented by \mathcal{F} and \mathcal{P}_E be the set of tuples represented by E . We will prove the first statement of the theorem by double inclusion for sets $\mathcal{P}_{\mathcal{F}}$ and \mathcal{P}_E .

The inclusion $\mathcal{P}_{\mathcal{F}} \subset \mathcal{P}_E$ is evident. The tuples represented by any representation $F_i, 0 \leq i \leq n$ are represented by E also since F_i is a fragment f-representation of E .

Let's also prove the inclusion $\mathcal{P}_E \subset \mathcal{P}_{\mathcal{F}}$. Consider a tuple $t \in \mathcal{P}_E$. If t satisfies all metaconditions in \mathcal{M} , then t is represented by F_0 . It is not represented by any other fragment f-representation of \mathcal{F} , because any other factorisation represents tuples that do not satisfy at least one metacondition.

If t does not satisfy all metaconditions, let \mathcal{M}^- be the set of metaconditions in \mathcal{M} that t does not satisfy. Consider also metacondition $M_i \in \mathcal{M}^-, i = \min\{j; M_j \in \mathcal{M}^-\}$. The tuple t is represented by the fragment f-representation $F_{n-i+1} = F(E, \{M_1^+, M_2^+, \dots, M_{i-1}^+, M_i^-\})$. The tuples represented by F_{n-i+1} satisfy the first $i - 1$ metaconditions and do not satisfy metacondition M_i . F_{n-i+1} does not have any restrictions for metaconditions $M_{i+1}, M_{i+2}, \dots, M_n$, which means that it includes both tuples that satisfy these metaconditions and tuples that do not satisfy them.

Consider now any fragment representation $F_j, 0 \leq j < n - i + 1$. Such an f-representation cannot represent t because all tuples represented by this representation satisfy metacondition M_i . Consider also any fragment representation $F_{n-j+1}, 1 \leq j < i$. F_j cannot represent t because all tuples represented by this representation do not satisfy metacondition j , which t satisfies. As a result, F_{n-i+1} is the only fragment representation that represents t .

To conclude, any tuple $t \in \mathcal{P}_E$ is represented by a fragment f-representation in \mathcal{F} . As a result $\mathcal{P}_E \subset \mathcal{P}_{\mathcal{F}}$.

We also showed that any tuple $t \in \mathcal{P}_E$ is represented by a single fragment representation of \mathcal{F} , which means that each tuple of relation \mathbf{R} is represented exactly once by \mathcal{F} . \square

Definition 4.8. Consider an f-tree \mathcal{T} , a value modification statement U on an f-representation over \mathcal{T} and a minimal metacondition partition \mathcal{P} of \mathcal{T} and U . A *free metacondition* M of partition \mathcal{P} is a metacondition that satisfies the following constraints for each target attribute A of statement U :

- Attribute A and all attributes involved in conditions of M are on the same root-to-leaf path of \mathcal{T} .
- Attribute A has a lower level in \mathcal{T} than all attributes involved in conditions of M .

Figure 4.7 shows procedure *splittingValueModification* which performs a value modification U on an f-representation E over f-tree \mathcal{T} and produces a result that consists of a union of f-representations over the same f-tree. The procedure can be applied only if the conditions stated in Proposition 4.2 are satisfied for U and \mathcal{T} . It builds a minimal metacondition partition \mathcal{P} of U and \mathcal{T} and removes exactly one free metacondition from it, if the

```

procedure SPLITTINGVALUEMODIFICATION( $\mathcal{T}$ ,  $E$ , value modification  $U$ )
  Build a minimal metacondition partition  $\mathcal{P} = \{M_1, \dots, M_n\}$  of  $\mathcal{T}$  and  $U$ 
  Let  $M = \emptyset$ 
  if  $\mathcal{P}$  has at least one free metacondition then
    Choose a free metacondition  $F \in \mathcal{P}$ 
     $M = F$ 
     $\mathcal{P} = \mathcal{P} \setminus \{F\}$ 
   $\mathcal{R} = \emptyset$ 
  for  $1 \leq i \leq n$  do
     $\mathcal{R} = \mathcal{R} \cup F(E, \{M_1^+, \dots, M_{i-1}^+, M_i^-\})$ 
  Build  $E_0 = F(E, \{M_1^+, \dots, M_n^+\})$ 
  Build a new value modification statement  $U'$  on  $E_0$  that
    has the same set clause as  $U$  and
    its where clause consists of all conditions of metacondition  $M$ 
  valueModification( $\mathcal{T}$ ,  $E_0$ ,  $U'$ )
   $\mathcal{R} = \mathcal{R} \cup E_0$ 
  return  $\mathcal{R}$ 

```

Figure 4.7: Value modification procedure that generates a result consisting of a union of f-representations over the same f-tree

partition has such a metacondition. Then it builds the f-set of E and \mathcal{P} , which represents the basis of the union of result f-representations computed by the procedure. The procedure also constructs a new value modification statement U' and performs it on the fragment f-representation E_0 of E that represents only tuples that satisfy all metaconditions of \mathcal{P} . The set of conditions of U' is either empty, or consists of the conditions of a free metacondition of \mathcal{P} . In both cases, all conditions of Proposition 4.1 are satisfied for U' and \mathcal{T} such that U' is supported by f-tree \mathcal{T} without restructuring and produces a single result representation. Procedure *valueModification* shown in Figure 4.1 is used to perform it.

The time complexity of procedure *splittingValueModification* is $O(n \cdot |E|)$, where n is the size of the minimal metacondition partition \mathcal{P} of statement U and f-tree \mathcal{T} . The time complexity is dominated by the construction of the f-set of \mathcal{P} and E . This f-set contains $O(n)$ fragment f-representations of E and the time needed to compute any such fragment representation is $O(|E|)$.

The following proposition characterises the number of f-representations in the union representing the result of procedure *splittingValueModification*.

Proposition 4.4. *Consider an f-tree \mathcal{T} , an f-representation E over \mathcal{T} and a value modification statement U on E . Let \mathcal{P} be a minimal metacondition partition of \mathcal{T} and U , $|\mathcal{P}| = n$.*

Suppose f-tree \mathcal{T} supports statement U without restructuring and the statement generates a result that consists of a union of f-representations over the same f-tree.

- If statement U has an empty list of conditions, then the number of f-representations in the result union is 1.
- If \mathcal{P} has at least one free metacondition, then the number of f-representations in the result union is n .
- If \mathcal{P} does not have any free metaconditions, then the number of f-representations in the result union is $n + 1$.

If statement U has no conditions, then \mathcal{P} is empty. The f-set generated by an empty metacondition partition and any f-representation E contains a single f-representation which is precisely E .

If statement U has at least one condition, then \mathcal{P} is not empty. If \mathcal{P} does not contain any free metaconditions, then the union of result representations is generated by the f-set computed using a set of metaconditions of size n . We have seen in Definition 4.6 that the size of such an f-set is exactly $n + 1$. If \mathcal{P} contains at least one free metacondition, exactly one free metacondition is removed from \mathcal{P} . In this case, the union of result representations is generated by the f-set computed using a set of metaconditions of size $n - 1$ and has size n .

Consider the case where \mathcal{P} has a single metacondition, which is a free metacondition. In this particular case, f-tree \mathcal{T} and statement U satisfy all conditions in Proposition 4.1, so the statement can be performed using procedure *valueModification* which produces a single result f-representation. Notice that in this case, the procedure *splittingValueModification* produces a union containing a single f-representation. The f-set computed by this procedure is generated by an empty set of metaconditions, such that it will contain only the f-representation we want to update. Then, the procedure constructs statement U' , which, for this particular case, is exactly statement U . Finally, the *splittingValueModification* procedure uses the *valueModification* procedure to perform statement U' on the only f-representation present in the union built. As a result, for this particular case, the *splittingValueModification* procedure will produce a union with a single f-representation which is exactly the f-representation produced by procedure *valueModification*. In fact, free metaconditions were introduced precisely for this reason.

The resulting union of procedure *splittingValueModification* supports new value modification statements. These can be performed by applying either procedure *valueModification*, or procedure *splittingValueModification* (depending on which is supported by the f-tree) on

each f-representation in the result union. Notice that after repeatedly performing value modification statements on the same relation using the *splittingValueModification* procedure, the number of f-representations in the result union will be exponential in the number of statements performed.

4.2 Supporting Value Modifications by Restructuring

We have discussed so far two techniques of performing a value modification statement. For a given value modification statement, the first technique can be applied to only a restricted class of f-trees. For the same statement, the second technique is supported by a larger class of f-trees which includes the class of f-trees supporting the first technique. However, for a given statement, there may be f-trees that do not support either of these two techniques. In such cases, we first need to restructure the f-tree (together with the f-representation defined by the f-tree) using the swap operator such that the new f-tree supports one of the techniques described above. There are two possible approaches:

- We restructure the f-tree such that the new f-tree supports procedure *valueModification* which produces a result consisting of a single f-representation.
- We restructure the f-tree such that the new f-tree supports procedure *splittingValueModification* which produces a result consisting of a union of f-representations over the same f-tree.

Each approach has its advantages and disadvantages. In order to apply procedure *valueModification* for complex value modification statements, we will need in general a more costly restructuring step, with many attribute swaps which could lead to a new f-representation that has a large size. The advantage of this technique is that it produces a single result representation. The second technique requires in general a more shallow restructuring step which might generate an f-representation that has a smaller size than the f-representation obtained after the restructuring step required by the first technique. The disadvantage is that procedure *splittingValueModification* produces a result union that contains, in general, more than one f-representation.

Example 4.7. Consider f-tree \mathcal{T}_{17} shown in Figure 4.8a, an f-representation E_{12} over \mathcal{T}_{17} and the following value modification statement:

$$U_7 = \text{Update } E_{12} \text{ set } B = B+1 \text{ where } B \leq E \text{ and } E \leq F$$

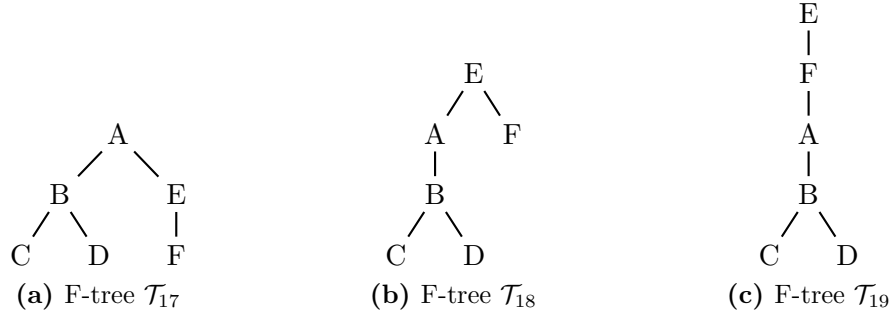


Figure 4.8: F-trees \mathcal{T}_{17} , \mathcal{T}_{18} and \mathcal{T}_{19}

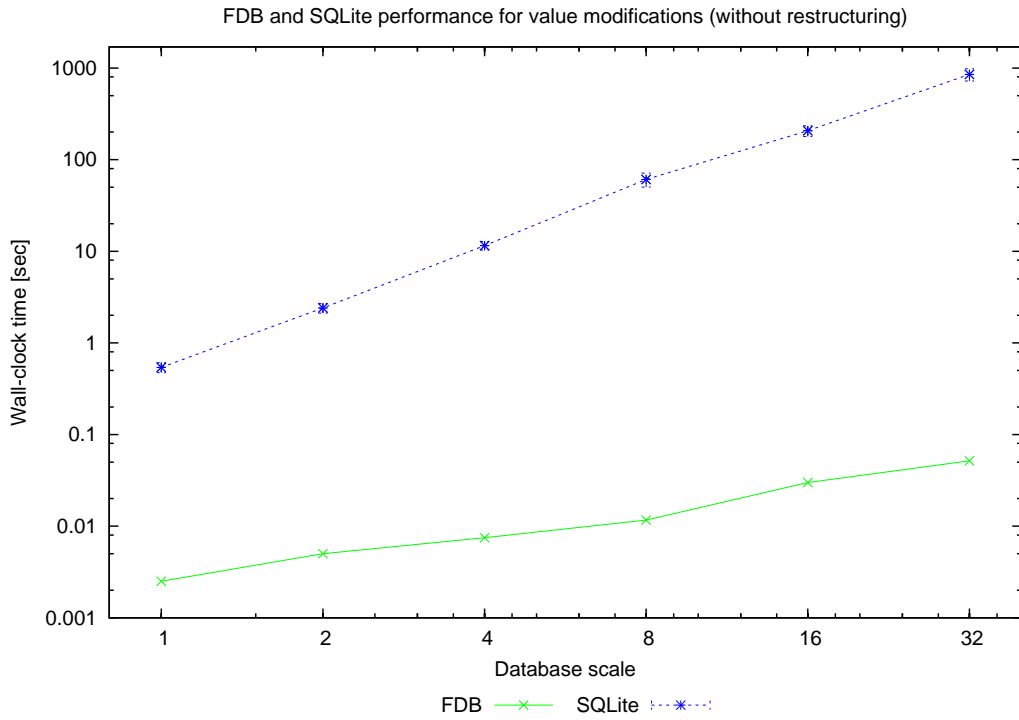
F-tree \mathcal{T}_{17} does not support statement U_7 without restructuring. We will assume that the nodes of \mathcal{T}_{17} labelled by attributes A and F are independent. Under this assumption, by applying the swap operator for nodes A and E on \mathcal{T}_{17} , we obtain f-tree \mathcal{T}_{18} shown in Figure 4.8b that defines f-representations that support statement U_7 using the *splittingValueModification* procedure. The procedure generates a union of two result representations. Notice that this f-tree does not support procedure *valueModification* for statement U_7 because the condition attributes of the statement (B , E and F) are spread over two paths of f-tree \mathcal{T}_{18} .

If we apply the swap operator for nodes E and F on f-tree \mathcal{T}_{18} , we obtain f-tree \mathcal{T}_{19} shown in Figure 4.8c which supports both value modification procedures for statement U_7 . Notice that \mathcal{T}_{19} has a lower branching factor than \mathcal{T}_{18} . For this reason, the f-representation of a relation over \mathcal{T}_{18} is, in general, more succinct than the f-representation of the same relation over \mathcal{T}_{19} , assuming that the relation is representable over both f-trees. \square

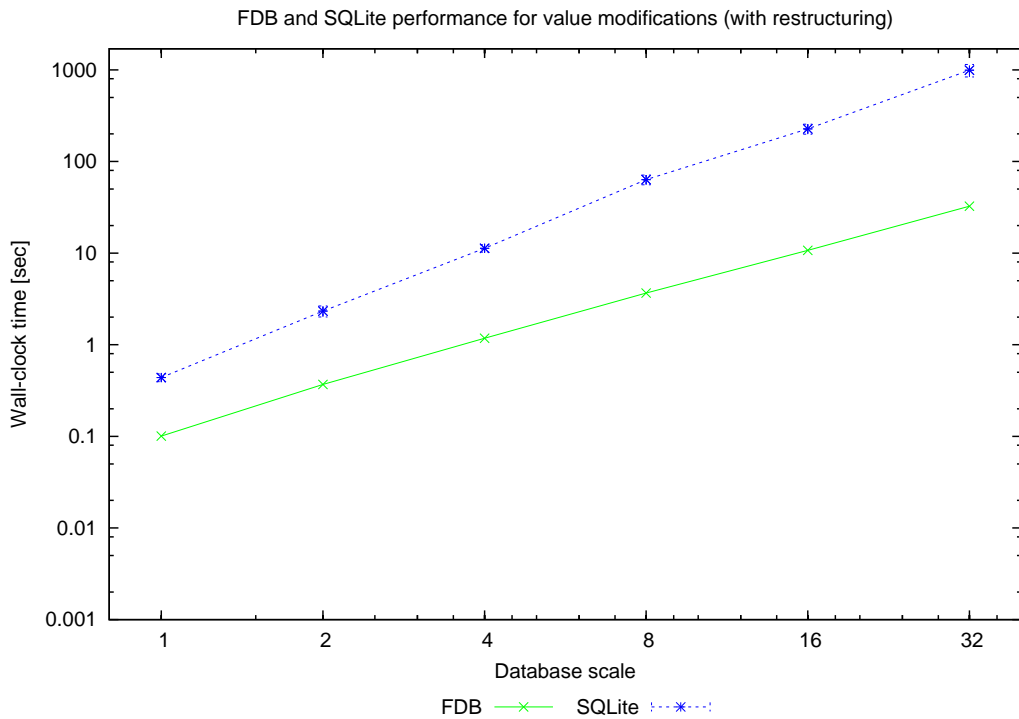
4.3 Experimental Evaluation

We implemented the first value modification technique in FDB and evaluated its performance against SQLite updates. We performed two experiments. The first experiment considers value modification statements supported by the f-representation given without restructuring. The second experiment considers value modifications statements that cannot be performed directly on the representation given and require an additional restructuring step. Our main finding is that FDB value modifications outperform SQLite updates in both cases.

Experimental Design For both experiments, we use the same dataset used in Section 3.2.5. SQLite performs the updates on relation R representing the natural join of relations Orders, Packages and Items, while our value modification algorithm is performed on an f-representation E of R over f-tree \mathcal{T}_{10} shown in Figure 3.11a. All value modification



(a) FDB and SQLite performance for value modification statements that do not require restructuring



(b) FDB and SQLite performance for value modification statements that require restructuring

Figure 4.9: FDB and SQLite performance for value modifications

statements considered update the *price* attribute. For the first experiment, we used four different value modification statements:

- The first statement has two conditions: one on *price* and the other on *package*.
- The second statement has also two conditions: one on *package* and the other on *item*.
- The third statement has two conditions: one on *item* and the other on *price*.
- The last statement has only one condition on *price*.

For the second experiment, we also used four different value modification statements:

- The first statement has two conditions: one on *price* and the other on *customer*.
- The second statement has also two conditions: one on *customer* and the other on *package*.
- The third statement has two conditions: one on *item* and the other on *customer*.
- The last statement has three conditions: on *customer*, on *item* and on *package*.

In order to be able to perform the value modification statements designed for the second experiment, we need to apply the swap operator two times on the f-representation E to be updated: first we swap *customer* and *date*, then we swap *customer* and *package*. The structure of the f-representation obtained after applying the swap operators is defined by f-tree \mathcal{T}_{12} shown in Figure 3.11c.

For both experiments, we run our value modification algorithm and the SQLite update three times for each value modification statement and report the average wall-clock time of all 12 runs. The times reported do not include the time to load the relation from plain-text files on disk. Our value modification algorithm was implemented in FDB for execution in main memory. To have an accurate comparison, the lightweight statement engine SQLite was also tuned for main memory operation by turning off the journal mode and synchronisations and by instructing it to use in-memory temporary store.

Experiment 1: Value modifications without restructuring

Figure 4.9a compares the performance of FDB and SQLite for value modification statements that do not require the restructuring of the f-representation updated by FDB. This f-representation is more compact than the relation considered by SQLite, which explains the gap between the times to perform the value modifications. The gap widens as we increase the scale factor and raises from two orders of magnitude for scale 1 to four orders of magnitude for scale 32. The reason we have an increasing gap is that the gap between the size of the f-representation considered by FDB and the size of the relation updated by SQLite increases also as we increase the scale of the database.

Experiment 2: Value modifications with prior restructuring

Figure 4.9b shows that FDB outperforms SQLite for value modification statements that require the restructuring of the representation updated by FDB. The times reported for FDB include the restructuring time. We notice, however, that the gap between FDB and SQLite is smaller than the gap observed in the previous experiment. There are two reasons for this:

- Before performing the value modification statement, FDB has to restructure the f-representation.
- The size of the f-representation obtained after restructuring is larger than the size of the original f-representation.

Chapter 5

Deletions

This chapter presents two techniques to delete values from an f-representation. Behind these two techniques lies the same idea that led to the development of the two value modification techniques described in the previous chapter: the first technique produces a result that consists of a single f-representation, while the second technique generates a result consisting of a union of f-representations over the same f-tree. For both techniques, we precisely characterise all f-trees that allow a given deletion statement without restructuring on any f-representation following their structure and discuss the algorithms that build the result of the deletion from the f-representation to be updated. The first technique was also implemented on top of an existing implementation of a factorised database and experimentally evaluated against SQLite.

We consider delete statements of the following form:

$$D = \text{Delete from } E \text{ where } \bigwedge_{1 \leq i \leq m} A_i \theta_i f_i(A_{i_1}, \dots, A_{i_{s_i}}, c_{i_1}, \dots, c_{i_{t_i}})$$

where

- E is an f-representation of a relation \mathbf{R} over schema \mathcal{S}
- $A_i, A_{i_k}, 1 \leq k \leq s_i, 1 \leq i \leq m$ are attributes of schema \mathcal{S} of the same type
- $c_{i_k}, 1 \leq k \leq t_i$ are constant values of the same type as attribute $A_i, 1 \leq i \leq m$
- $f_i, 1 \leq i \leq m$ are functions that take as arguments either values of attributes of \mathcal{S} , or constants
- $\theta_i, 1 \leq i \leq m$ are comparison operators

The *where* clause of the deletion statements considered has the same form as the *where* clause of the value modification statements discussed in the previous chapter. This is the reason why the techniques developed for value modifications on factorised representations can be applied for deletions also, with only minor changes that relate to the fact that we remove singletons instead of changing their values. However, unlike a value modification

statement, a deletion statement does not have a *set* clause. For this reason, the constraints that the f-tree needs to satisfy in order to support a deletion statement are less tight than the constraints imposed on the same f-tree in order to support a value modification statement with the same *where* clause.

Several concepts introduced in the previous chapter to describe value modifications techniques on factorised representations, such as the head/non-head attribute of a condition, a metacondition, a minimal metacondition partition, the fragment f-representation of a given f-representation and the f-set of a representation and a set of metaconditions, will be used throughout this chapter also.

5.1 Nesting Structures Supporting Efficient Deletions

This section studies the case of deletion statements that can be performed directly on the f-representation given. Since they do not require any restructuring step, they can be performed very efficiently.

5.1.1 The Case of a Result Consisting of One Factorised Representation

The following proposition characterises f-trees that support without restructuring a given deletion statement whose result consists of a single representation.

Proposition 5.1. *Consider an f-tree \mathcal{T} and a deletion statement D on an f-representation over f-tree \mathcal{T} . Statement D is supported by \mathcal{T} without restructuring and produces a result consisting of a single f-representation if the nodes labelled by attributes involved in all conditions of D are found on the same-root-to-leaf path of \mathcal{T} .*

The attributes of a certain condition of the delete statement D need to be on the same root-to-leaf path in order to be able to check the condition. Consider the case where two attributes A and B involved in the same condition are found on different paths of \mathcal{T} . If A and B are siblings, then an f-representation over \mathcal{T} may contain a product between a union of A -singletons and a union of B -singletons. If they are not siblings, then these unions will be part of larger partial f-representations found in a product. The condition on A and B cannot be checked in either of these cases because we may have a union with more than one A -singleton in a product with a union containing more than one B -singleton.

If the conditions are spread over several paths of the f-tree, then they can be checked only independently of each other. In order to perform a deletion statement whose *where* clause is a conjunction of several conditions, we need to be able to check all these conditions before deciding to remove singletons or partial representations. This can be done only if their attributes are on a single path of the f-tree.

```

procedure DELETE(f-tree  $\mathcal{T}$ , f-representation  $E$ , statement  $D$ )
  if statement  $D$  has an empty where clause then
     $E = \emptyset$ 
  else
    if  $\mathcal{T}$  is empty then return
    if  $\mathcal{T}$  is a forest  $\mathcal{T}_1, \dots, \mathcal{T}_k$  then
      Let  $E = E_1 \times \dots \times E_k$ 
      Let  $\mathcal{T}_i$  be the f-tree containing the attributes of conditions of  $D$ 
      deletion( $\mathcal{T}_i, E_i, D$ )
    if  $\mathcal{T}$  is a single rooted tree  $A(\mathcal{U})$  then
      Let  $E = \bigcup_a \langle A : a \rangle \times E_a$ 
      for each singleton  $\langle A : a \rangle$  in  $E$  do
        if  $A$  is not head attribute of any condition of statement  $D$  then
          delete( $\mathcal{U}, E_a, D$ )
        if  $A$  is head attr. of a condition  $\mathcal{C}_A$  of  $D$  and value  $a$  sat.  $\mathcal{C}_A$  then
          if node  $A$  is the lowest node in  $\mathcal{T}$  labelled by a head attribute then
            remove  $\langle A : a \rangle \times E_a$  from  $E$ 
          else
            delete( $\mathcal{U}, E_a, D$ )

```

Figure 5.1: Deletion procedure that generates a result consisting of a single f-representation

Figure 5.1 shows procedure *delete* which performs a deletion statement D on an f-representation E that generates a single result representation. It updates E in-place and can be performed only if statement D and the f-tree \mathcal{T} defining the structure of E satisfy the condition stated in Proposition 5.1. The procedure scans only unions of A -singletons, where A is either a head attribute of a condition of D , or an ancestor of a head attribute. Whenever the procedure reaches a union of A -singletons, where A is a head attribute, it checks condition \mathcal{C}_A for each singleton $\langle A : a \rangle$. If the condition is not satisfied, we do not need to take any further steps. If the condition is satisfied, we have two cases:

- Node A is not the lowest node in \mathcal{T} labelled by a head attribute. In this case, we recursively call the *delete* procedure for the f-representation found in a product with singleton $\langle A : a \rangle$. This f-representation contains unions of singletons for other head attributes. Before removing any singletons or partial f-representations, the conditions on these attributes must be checked.
- Node A is the lowest node in \mathcal{T} labelled by a head attribute. In this case, we can remove all A -singletons (together with the f-representations they are found in a product with) whose values satisfy \mathcal{C}_A because all other conditions of statement D have already been checked.

There are cases when all items of a union are removed. Empty unions should always be removed from the result f-representation. In order to do this, we can replace each empty union with the symbol \emptyset and then propagate all occurrences of this symbol in a bottom-up fashion using the following two rules:

- Any product containing \emptyset as a factor is entirely replaced by \emptyset .
- If \emptyset is placed inside a union that contains other non-empty items, then the empty item is simply removed. If the union consists only of empty items, then we replace the union entirely with \emptyset .

By applying these two rules, the removal of subexpressions can be propagated up to the top-most union of the original f-representation. In this case, all tuples represented by the factorisation to be updated satisfy the conditions of the statement and the new f-representation will be empty.

The time complexity of the *delete* procedure is in the worst case $O(|E|)$. In most of the cases it is sublinear since the procedure scans only unions of A -singletons, where A is either a head attribute of a condition of the deletion statement, or an ancestor of a head attribute.

5.1.2 The Case of a Result Consisting of a Union of Factorised Representations over the Same Factorisation Tree

The main idea of the second approach we propose for the deletion task is similar to the idea behind the technique described in section 4.1.2: split the f-representation to be updated into several smaller f-representations such that one of them represents only the tuples of the relation considered that satisfy all conditions of the given deletion statement. All other f-representations should represent only tuples that violate at least one such condition. The deletion statement can be performed by removing entirely the first f-representation mentioned and keeping in the resulting union only factorisations representing tuples that violate at least one condition of the deletion statement. The class of f-trees supporting a deletion statement without restructuring using this technique is larger than the class of f-trees that support the same statement using the previous technique described, but this comes at the cost of a result that consists of a union of possibly more than one f-representation over the f-tree of the f-representation to be updated.

Example 5.1. Consider f-tree \mathcal{T}_{20} and f-representation E_{13} , both shown in Figure 5.3. Consider also the deletion statement

$$D_2 = \text{Delete from } E_{13} \text{ where } B < C \text{ and } D < 5$$

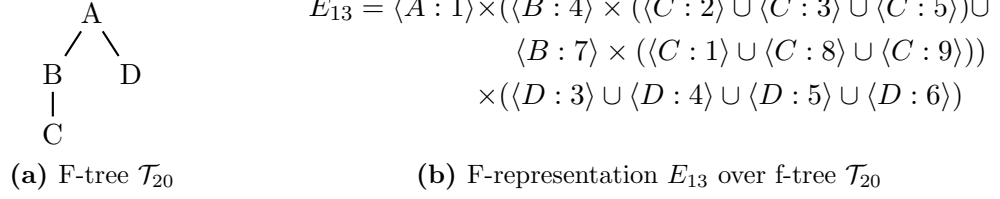


Figure 5.3: F-tree \mathcal{T}_{20} and f-representation E_{13}

The result of D_2 is a union of two fragment f-representations of E_{13} , F_1 and F_2 , where

$$\begin{aligned}
 F_1 &= F(E_{13}, \{\{\mathcal{C}_{D_2, C}\}^+, \{\mathcal{C}_{D_2, D}\}^-\}) \\
 &= \langle A : 1 \rangle \times (\langle B : 4 \rangle \times \langle C : 5 \rangle \cup \langle B : 7 \rangle \times (\langle C : 8 \rangle \cup \langle C : 9 \rangle)) \times (\langle D : 5 \rangle \cup \langle D : 6 \rangle) \\
 F_2 &= F(E_{13}, \{\{\mathcal{C}_{D_2, C}\}^-\}) = \langle A : 1 \rangle \times (\langle B : 4 \rangle \times (\langle C : 2 \rangle \cup \langle C : 3 \rangle) \cup \langle B : 7 \rangle \times \langle C : 1 \rangle) \\
 &\quad \times (\langle D : 3 \rangle \cup \langle D : 4 \rangle \cup \langle D : 5 \rangle \cup \langle D : 6 \rangle)
 \end{aligned}$$

The fragment f-representation F_1 represents all tuples that satisfy condition $B < C$ and do not satisfy condition $D < 5$, while F_2 represents all tuples that do not satisfy condition $B < C$. These are all the tuples that do not satisfy both conditions at the same time. \square

The following proposition characterises f-trees that support without restructuring a given deletion statement whose result consists of a union of f-representations over the same f-tree.

Proposition 5.2. *Consider an f-tree \mathcal{T} and a deletion statement D on an f-representation over f-tree \mathcal{T} . Statement D is supported by \mathcal{T} without restructuring and produces a result consisting of a union of f-representations over the same f-tree if the nodes labelled by attributes involved in each condition of D are found on the same-root-to-leaf path of \mathcal{T} .*

The attributes of a condition \mathcal{C} of D still need to be on the same root-to-leaf path of \mathcal{T} in order to be able to check condition \mathcal{C} for any f-representation over \mathcal{T} . However, we no longer need to have the attributes of all conditions of D on the same path of the f-tree. The result union of a deletion statement can be computed by checking the conditions independently of each other, so they can be spread over several paths of \mathcal{T} .

Figure 5.4 shows procedure *splittingDelete* which performs a deletion statement D on an f-representation E and produces a union of result representations. This procedure can be performed only if statement D and the f-tree \mathcal{T} defining the structure of E satisfy the condition stated in Proposition 5.2. For a deletion statement with a non-empty *where* clause, the procedure builds a minimal metacondition partition $\mathcal{P} = \{M_1, \dots, M_n\}$ of D and \mathcal{T} .

```

procedure SPLITTINGDELETE(f-tree  $\mathcal{T}$ , f-representation  $E$ , statement  $D$ )
  if statement  $D$  has an empty where clause then
     $E = \emptyset$ 
    return  $E$ 
  else
    Build a minimal metacondition partition  $\mathcal{P} = \{M_1, \dots, M_n\}$  of  $\mathcal{T}$  and  $D$ 
     $\mathcal{R} = \emptyset$ 
    for  $1 \leq i \leq n$  do
       $\mathcal{R} = \mathcal{R} \cup F(E, \{M_1^+, M_2^+, \dots, M_{i-1}^+, M_i^-\})$ 
    return  $\mathcal{R}$ 

```

Figure 5.4: Deletion procedure that generates a result consisting of a union of f-representations over the same f-tree

Then, it constructs the resulting union of fragment f-representations of E , $\mathcal{R} = F_1 \cup \dots \cup F_n$, where

$$\begin{aligned}
 F_1 &= F(E, \{M_1^+, M_2^+, \dots, M_{n-1}^+, M_n^-\}) \\
 F_2 &= F(E, \{M_1^+, M_2^+, \dots, M_{n-1}^-\}) \\
 &\dots \\
 F_{n-1} &= F(E, \{M_1^+, M_2^-\}) \\
 F_n &= F(E, \{M_1^-\})
 \end{aligned}$$

The set $\mathcal{F} = \{F_1, \dots, F_n\}$ together with the fragment f-representation $F(E, \{M_1^+, M_2^+, \dots, M_{n-1}^+, M_n^+\})$ forms the f-set of representation E and partition \mathcal{P} . Theorem 4.7 showed that the f-set of a representation and a set of metaconditions represents all the tuples that the original factorisation represents. Notice that \mathcal{F} contains all representations in the f-set of E and \mathcal{P} , except for the factorisation that represents tuples satisfying all metaconditions in \mathcal{P} . For this reason, it represents all tuples of the relation represented by E that violate at least one metacondition.

The time complexity of the procedure is $O(n \cdot |E|)$, where n is the size of a minimal metacondition partition of statement D and f-tree \mathcal{T} . The time complexity is dominated by the computation of n fragment f-representations of E and the construction of each such representation takes $O(|E|)$ time.

The following proposition characterises the number of f-representations in the union representing the result of procedure *splittingDelete*.

Proposition 5.3. *Consider an f-tree \mathcal{T} , an f-representation E over \mathcal{T} and a deletion statement D on E . Let \mathcal{P} be a minimal metacondition partition of \mathcal{T} and D , $|\mathcal{P}| = n$. Suppose statement D is supported by f-tree \mathcal{T} without restructuring and generates a result consisting of a union of f-representations over the same f-tree.*

- *If statement D has an empty set of conditions, then the result union contains a single empty f-representation.*
- *If statement D has a non-empty set of conditions, then the result union contains n fragment representations of E .*

Notice that if statement D and f-tree \mathcal{T} satisfy the condition stated in Proposition 5.1, all conditions of the statement form a single metacondition and procedure *splittingDelete* produces a union that contains a single result f-representation. Procedure *delete* could also be used to perform such a deletion statement and would produce the same result representation.

The union of result f-representations computed by procedure *splittingDelete* supports new deletions. New statements can be performed using either procedure *delete*, or procedure *splittingDelete* on each f-representation in the result union separately. Notice that if we apply procedure *splittingDelete* to perform a new deletion, each f-representation in the result union may be split into several other smaller f-representations such that after a sequence of m deletion statements (all performed using procedure *splittingDelete*), the number of f-representations that the result union contains will be exponential in m .

5.2 Supporting Deletions by Restructuring

In the previous section, we proposed two approaches to the task of deletions on factorised representations. The first procedure is supported by a rather narrow class of f-trees, but has the advantage that its result consists of a single f-representation. The second procedure is supported by a larger class of f-trees (for the same given delete statement), but its result is a union of one or more f-representations over the same f-tree. If the f-tree defining the structure of the f-representation to be updated does not support the deletion with any of these techniques, then the f-tree (together with the f-representation to be updated) must first be restructured using the swap operator such that the new f-tree supports the statement using one of these two techniques. Section 4.2 discusses two restructuring approaches in case a value modification statement is not supported by a given f-tree. These two approaches can be applied for deletion statements also and the reasoning behind them is the same.

5.3 Experimental Evaluation

We implemented in FDB the deletion technique that generates a result consisting of a single representation and evaluated it against SQLite updates. We performed two experiments. The first experiment considers a class of four different deletion statements that do not require the restructuring of the f-representation to be updated. The second experiment considers a class of four different deletion statements that cannot be performed directly on the f-representation to be updated and require a prior restructuring step. In both cases, FDB outperformed SQLite.

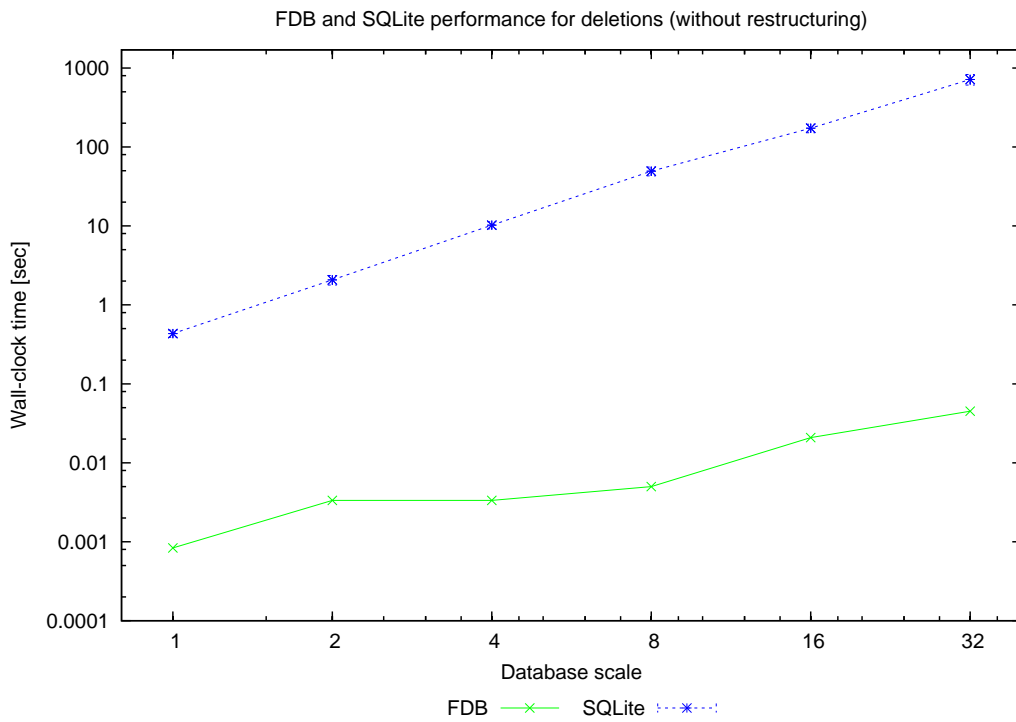
Experimental Design Both experiments use the synthetic dataset used also in Sections 3.2.5 and 4.3. Similarly to value modification experiments, SQLite performs the deletions on a relation R representing the natural join of relations Orders, Packages and Items, while our deletion algorithm is performed on an f-representation E of R over f-tree \mathcal{T}_{10} shown in Figure 3.11a. The deletion statements considered for both experiments have the same *where* clause as the value modification statements considered in the two experiments performed in Section 4.3. The restructuring step performed before applying the deletions designed for the second experiment consists of two swap operations: we first swap *customer* and *date*, then we swap *customer* and *package*.

Experiment 1: Deletions without restructuring

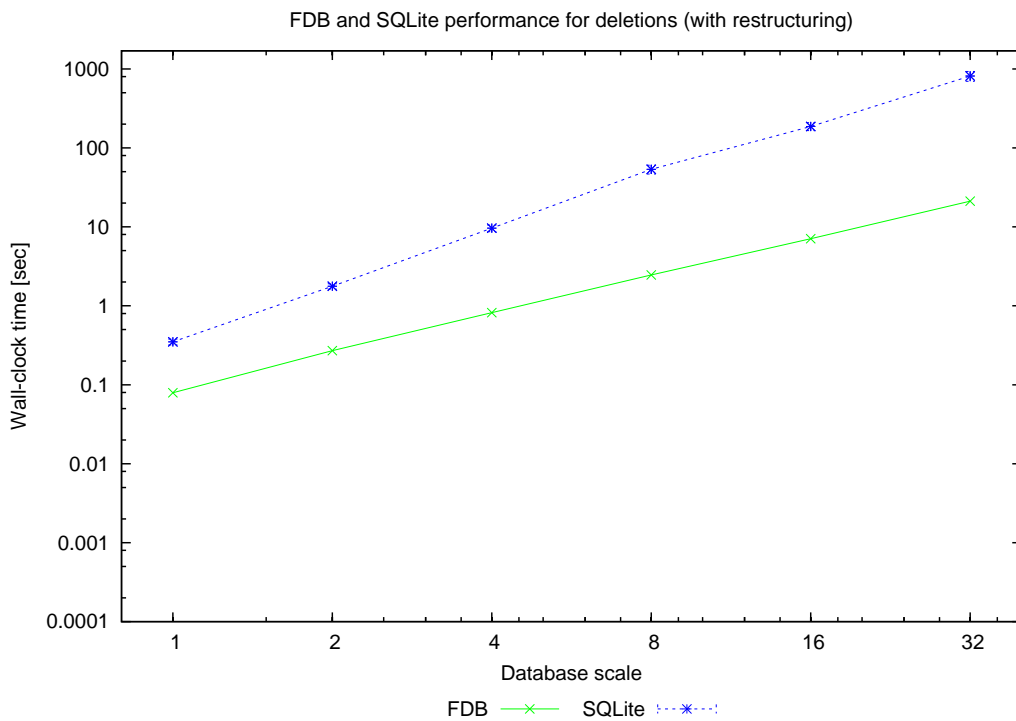
Figure 5.5a shows that for deletion statements that do not require the restructuring of the f-representation, FDB outperforms SQLite by up to four orders of magnitude for large database scales. FDB considers a more compact representation of the relation updated than SQLite, which explains the performance gap. The gap is increasing with the database scale because the gap between the sizes of two relation representations considered by FDB and SQLite is also increasing.

Experiment 2: Deletions with prior restructuring

Figure 5.5b shows that FDB outperforms SQLite for deletion statements that require the restructuring of the f-representation considered by FDB, but the performance gap for these types of deletions is smaller than the gap observed in the previous experiment (only two orders of magnitude for the largest database scale considered). The reason we have a smaller gap is that FDB has to restructure the f-representation before performing the deletion and the restructuring leads to an f-representation less succinct than the original f-representation.



(a) FDB and SQLite performance for delete statements that do not require restructuring



(b) FDB and SQLite performance for delete statements that require restructuring

Figure 5.5: FDB and SQLite performance for deletions

Chapter 6

Implementation

This chapter provides details about the implementation of the algorithms discussed in previous chapters. We implemented all three bulk insertion algorithms presented and the technique that generates a result consisting of a single f-representation for both value modifications and deletions. These algorithms were implemented on top of an existed framework for factorised databases implemented in the C++ programming language. The implementation of our algorithms uses intensely two main components of the existing framework: the component implementing a factorisation tree as a standard rooted tree and the component implementing a factorised representation as a parse tree.

6.1 Bulk Insertions

The main classes implementing the bulk insertion algorithms discussed have the same base class called *FRepFTreeBuilder*. This class implements two methods used by all three algorithms: a method that loads the relation to be factorised in main memory and another method used to sort the tuples of the relation.

Class *FRepOverFTreeAndFlatDataBuilder* implements the TF algorithm which takes as input a relation and an f-tree. After loading the relation in main memory, we perform a depth-first search on the f-tree and store the order of the attributes labelling the nodes of the f-tree as we scan them. The sequence of attributes computed during the depth-first search is used as a priority array to sort the relation. By performing this sorting prior to the factorisation step, we avoid the computation of all projections that the TF algorithms requires. We build the f-representation during a complete scan of the f-tree and a possibly partial scanning of the relation. At each recursive call corresponding to a rooted f-tree, we need to find all distinct values of the attribute labelling the root of the f-tree. This operation that can be done very efficiently as the tuples are sorted. Whenever we reach a forest with more than one f-tree, the sorted list of values for an attribute labelling a root of

one of these f-trees may occur several times in the relation, but the procedure implemented stops immediately after the first occurrence of such a list. For this reason, the procedure implemented scans only partially the relation during the factorisation step.

The implementation of the PF and RCF algorithms required the implementation of a class called *Factor* that behaves as a wrapper around an f-representation. This class stores a pointer to the root of the parse tree of an f-representation and the hash value of this parse tree. The hash value of the parse tree is computed as follows:

- For a parse tree T consisting of a single node labelled by a singleton $\langle A : a \rangle$, we use the following formula:

$$\text{hash}(T) = (B \cdot id_A + C \cdot h(a)) \% R$$

where B , C and R are integer values, id_A is an integer value associated to attribute A and $h(a)$ represents the hash value of the value stored in singleton $\langle A : a \rangle$.

- For a rooted parse tree T with children parse trees T_1, T_2, \dots, T_k , we use the following formula:

$$\text{hash}(T) = (M \cdot (\dots (M \cdot (M \cdot h_r + N \cdot \text{hash}(T_1)) + N \cdot \text{hash}(T_2)) + \dots) + N \cdot \text{hash}(T_k)) \% R$$

where M , N and R are integer values and h_r represents the hash value associated to the operation labelling the root of parse tree T .

This hash function allows the computation of the hash value of a parse tree incrementally: we can start with a single node which is later added as a child to a small parse tree, which, in turn, is added to a larger parse tree and so on. Whenever we add a child to a larger parse tree, we do not need to recompute the hash value of the entire resulting parse tree, we only need to update it based on its current value and the hash value of the child added.

The PF algorithm is implemented by three classes: *GeneralFRepTreeBuilder*, *FactorProcessor* and *LeapFrogIntersector*. They work similarly to a pipeline, but information is passed between them in both directions: a *GeneralFRepTreeBuilder* object sends information to a *FactorProcessor* object, which will send information to a *LeapFrogIntersector* object. The *LeapFrogIntersector* object sends back the result of its computation to the *FactorProcessor* object, which, after further computation, will send new information back to the *GeneralFRepTreeBuilder* object. The main responsibilities of each of these classes are described below:

- The *GeneralFRepTreeBuilder* class reads the relation from the input file and sorts its tuples using the input path as a priority array. The factorisation procedure recursively divides the relation into smaller fragments and builds the list of factors for such fragments of data. At each recursive level, it interacts with the *FactorProcessor* class by giving it the list of distinct singletons at the current level and the lists of factors of all fragments computed at a lower recursion level and expecting in return a single list of factors of the entire relation processed at the current level.
- The *FactorProcessor* class receives a list \mathcal{A} of distinct A -singletons (where A is an attribute of the relation processed) and a list of lists of factors. It builds the list of distinct factors \mathcal{D} in all these lists of factors and assigns identifiers (using integer values) to all distinct factors. It also builds the A -factor matrix for \mathcal{A} and \mathcal{D} . Since this matrix can be very sparse, we store it internally as a list \mathcal{L} of lists of integers. Each list in \mathcal{L} represents a row of the factor matrix and contains the ids of all factors for which the row has a non-zero element. It interacts with the *LeapFrogIntersector* class by sending list \mathcal{L} and expecting in return a list containing the ids of factors present in all lists of \mathcal{L} . It removes all these factors from the lists of \mathcal{L} and sends back to the *GeneralFRepTreeBuilder* instance a list containing pointers to all common factors and one additional factor built from the factors remaining in the lists of \mathcal{L} .
- The *LeapFrogIntersector* class receives a list of lists containing integer values and computes their intersection using the LeapFrog Join algorithm described in [21].

The PF and RCF algorithms share many steps such that, in order to avoid code duplication, the implementation of the RCF algorithm reuses the three classes implementing the PF algorithm. A special argument is passed to the *GeneralFRepTreeBuilder* instance to specify the type of factorisation we want to build. The main difference arises in class *FactorProcessor*, when we compute the list of factors of the currently processed relation fragment. In both cases, this list contains pointers to factors that are present in all the lists of list \mathcal{L} storing the A -factor matrix and one additional factor representing the rest of the data. In the case of the RCF algorithm, this factor is built using rectangle coverings of the factor matrix. The main procedure of class *RectangleExtractor* computes a set of rectangles that covers completely a subset of rows of the A -factor matrix. To speed up the computation of this set, we represent it internally as a list \mathcal{R} of lists. At each round of expansion, instead of creating new rectangles, we simply add a new list to \mathcal{R} , representing ids of all common factors of the factor matrix row added in this round of expansion and the factor matrix row added in the previous round of expansion. By repeatedly calling the main procedure of class

RectangleExtractor, the *FactorProcessor* instance computes incrementally the factorisation representing all the factors remaining in the lists of \mathcal{L} .

To check the correctness of the f-representations produced by the RCF algorithm, we implemented the class *FactorIterator* which enumerates tuples represented by a *Factor* object. The framework already implemented tuple enumeration for an f-representation, but the structure of the f-representations produced by the RCF algorithm is more general than the f-representations that the framework used before and required a tuple enumeration procedure that can capture more general cases.

6.2 Value Modifications and Deletions

For both value modifications and deletions, we implemented the technique that produces a result consisting of a single f-representation. The two algorithms have many similarities, such that a large part of the code implementing value modifications was used for the implementation of deletions also. The classes implementing their common parts are described below:

- Class *QueryParser* reads a value modification/deletion statement from a text file. The value modification statements parsed must have the following form:

```

Statement ::= UPDATE Rel SET AssignmentStatement WHERE WhereStatement
AssignmentStatement ::= Attr = RHS
RHS ::= Value | Value Op Value
Value ::= Attr | Const
Op ::= '+' | '-' | '*' | '/'
WhereStatement ::= Value CompOp Value (AND Value CompOp Value)* |  $\emptyset$ 
CompOp ::= '<' | '<=' | '==' | '>=' | '>'

```

A deletion statements parsed by this class has a *where* statement of the same exact form. The shape of such a deletion statement is

```

Statement ::= DELETE FROM Rel WHERE WhereStatement

```

- Class *UpdateItem* is the base class of classes *Target* (which encapsulates the structures storing the information needed to process the *set* clause) and *Condition* (which encapsulates the structures storing the information needed to check a condition of the *where* clause). The *UpdateItem* class provides storage for two *Value* instances. Each such instance can be either an attribute, or a constant. In addition to this, the *Target* class also stores the attribute that the statement updates and an operator, while class

Condition stores only a comparison operator. In both cases, the operator is stored as a pointer to a function.

- Class *UpdateFRep* is the base class of the classes that actually perform the value modification/deletion statement on the input f-representation. It contains only a few methods that process the objects storing the conditions of the statement performed and deallocate memory.

Class *ValueModification* implements a simplified version of the value modification statements discussed. There are two main simplifications:

- The *set* clause of the statements supported by the current implementation contains exactly one target attribute.
- The additional step that follows the update of the f-representation and merges singletons occurring several times in a union was not implemented.

A *ValueModification* instance receives the parse tree of the f-representation to be updated and a text file specifying the value modification statement. It builds a *QueryParser* object which reads the statement from the file and builds a *Target* object and the *Condition* objects. The main procedure of the *ValueModification* instance then scans the parse tree received and updates the value stored by any node labelled by a target attribute if all conditions of the statement are satisfied on the path from the root to the updated node of the parse tree.

Finally, class *Delete* performs a deletion statement on the parse tree of an f-representation. Its implementation is similar to the *ValueModification* class, except that the *QueryParser* object constructed will read and build only *Condition* objects and the main procedure of a *Delete* instance removes subtrees of the parse tree we update instead of updating the values stored by its nodes.

Chapter 7

Related Work

Factorised representations were originally introduced in [18]. Equivalent to factorised representations over factorisation trees are generalised hierarchical decompositions (GHDs) and compacted relations over compaction formats. Existing work establishes the correspondences of GHDs to functional and multi-valued dependencies [9], and characterises selection conditions with disjunctions that can be performed on the compacted relations in one sequential pass [6], but questions of succinctness have not been addressed. Nested and non-first normal form relations [16, 14, 1] are also structurally equivalent to factorised representations over factorisation trees, but are proposed as an alternative data model and not as a representation system for standard relation.

Subsumed by factorised representations are various representation systems equivalent to unions of products of unions of singletons, such as world-set decompositions in incomplete databases [17], GDNFs (generalised disjunctive normal forms) studied as succinct presentations of inputs to CSPs [8], formal concepts in data mining, and others.

In relational databases, eliminating redundancy caused by join dependencies and multi-valued dependencies is traditionally addressed by normalising the relational schema [15]. Representation systems for relations based on join decompositions include minimal constraint networks [12], but for these data retrieval (tuple enumeration) is NP-hard. Tuple enumeration is constant time for acyclic queries [2], in which case the input database together with the query already serve as a compact representation of the result. Decompositions of the query hypergraph, measuring the degree of acyclicity of the query, are traditionally used for classifying the tractability of Boolean queries and constraint satisfaction problems [13, 11].

Representations utilising algebraic factorisation are not restricted to relational data. In the context of relational databases, factorisation can also be applied to provenance polynomials [10] that describe how individual tuples of a query result depend on tuples of the

input relations [18]. Algebraic and Boolean factorisations were considered in succinct representations of Boolean functions [7] and are closely related to binary decision diagrams, Boolean circuits and other representations of Boolean functions.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis we have investigated updates on factorised databases, a problem previously unexplored in the context of such databases.

The insertion task was studied from two points of view: the insertion of one tuple in a given factorisation and bulk insertion. For the first type of insertion, we showed when and how a given tuple can be inserted in an f-representation without prior restructuring. We defined the task of bulk insertion on factorised databases as the construction of a factorisation of the relation representing the tuples to be inserted. We implemented and experimentally compared three algorithms to perform this task. Their behaviour reflects a trade-off between the amount of information we have about the hidden structure of the relation, the size of the result representation and the execution time. The first algorithm produces good quality factorisations with low computation time, but requires complete information about the structure of the f-representation it builds. The second algorithm is less efficient, but requires only partial information about the structure of the f-representation since it is able to infer some structures on its own. Finally, the last algorithm is the most computationally expensive, but experiments have shown that it can compute factorisations that are up to one order of magnitude more succinct than f-representations computed by the other two bulk insertion algorithms.

Value modifications and deletions on factorised databases were approached in a similar fashion. For both types of updates, we discussed two techniques. The first technique is supported without restructuring for a given update statement by a rather restricted class of f-trees, but generates as a result a single representation. The second technique described is supported without restructuring by a wider class of f-trees, which includes the class of f-trees supported by the first technique, but generates as a result a union that contains several f-representations over the same f-tree. For both types of updates, the first technique

was implemented and benchmarked against the open-source relational engine SQLite. In both cases, our algorithms outperformed SQLite updates by orders of magnitude.

8.2 Future Work

This work lies at the foundation of transaction processing, a very important part of any database, but which could not be tackled without support for updates. Besides this, several improvements discussed below could be brought to the bulk insertion algorithms proposed:

- We have seen that the path used by the PF and RCF algorithms for representation construction has an enormous impact on the structure and the size of the f-representation computed. The algorithms produce succinct f-representations if the path considered follows the topological sorting of the f-tree defining the hidden structure of the relation to be factorised, but often this structure is not known. Developing techniques that explore the data to be factorised and try to discover such paths could vastly improve the quality of the representations produced by both algorithms in cases where the structure of the relation is not known.
- The quality of the representations produced by the PF and RCF algorithms could be even further improved by considering different local paths for the factorisation of different fragments of the data, instead of a single global path that is applied to all these fragments. The idea is to generate these local paths as we advance at lower recursion levels of the algorithms using techniques similar to those developed to discover global paths (previously discussed). Notice that even though the representations computed using this approach can be much more compact, they will also have a more heterogeneous structure.
- The current version of all three bulk insertion algorithms makes the assumption that the data we want to factorise fits into main memory, but this is not always the case in real applications. Extensions of these algorithms that work with larger relations could be developed. The main idea of such extensions is to repeatedly read from the disk small fragments of the data, build their factorised representations and write back to disk the factorisations computed. In order to find factors that occur several times in a relation and build larger factorisations, the PF and RCF algorithms can store in main memory only the hash values of the factorised representations stored on disk. We first compare the hash values and only if they are equal, we load from the disk and compare the f-representations.

Improvements could also be brought to the value modification and deletion technique that generates a result consisting of a union of f-representations over the same f-tree. One possible extension of this technique uses the recently introduced d-representations [20]. These are f-representations where further succinctness is brought by explicit sharing of repeated subexpressions. The size of the result computed by our update technique can be greatly reduced by sharing subexpressions between the f-representations contained by the resulting union. Although we investigated this approach as part of the thesis, we decided to leave it out of its final version. We found that the f-set of an f-representation and a set of metaconditions (which lies at the basis of the result union of our update technique) has a very well defined structure which allows a precise characterisation of all the subexpressions that can be shared between any two distinct f-representations contained by it. The size of such an f-set is $O(\sum_{A \in \mathcal{S}} m_A \cdot N_A)$, where N_A is the number of A -singletons contained by the f-representation to be updated and m_A is the number of root-to-leaf paths of the tree rooted in A that contain condition attributes of metaconditions in the partition generating the f-set.

Bibliography

- [1] Serge Abiteboul, Nicole Bidoit. Non first normal form relations: An algebra allowing data restructuring. In *Journal of Computer and System Sciences*, Volume 33 Issue 3, December 1986, pages 361-393.
- [2] Guillaume Bagan, Arnaud Durand, Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, 2007, pages 208-222.
- [3] Nurzhan Bakibayev, Dan Olteanu, Jakub Závodný. FDB: A query engine for factorised relational databases. In *Very Large Data Bases (PVLDB)*, 5(12), 2012.
- [4] Nurzhan Bakibayev, Dan Olteanu, Jakub Závodný. Demonstration of the FDB Query Engine for Factorised Databases. In *Very Large Data Bases (PVLDB)*, 5(12), 2012.
- [5] Nurzhan Bakibayev, Tomáš Kočický, Dan Olteanu, Jakub Závodný. Queries with Order-by Clauses and Aggregates in Factorised Databases. In *Very Large Data Bases (PVLDB)*, 2013. (to appear)
- [6] François Bancilhon, Philippe Richard, Michel Scholl. On line processing of compacted relations. In *Proceedings of the 8th International Conference on Very Large Data Bases, VLDB '82*, pages 263-269.
- [7] R. K. Brayton. Factoring logic functions. In *IBM Journal of Research and Development - Mathematics and computing*, Volume 31 Issue 2, March 1987, pages 187-198.
- [8] Hubie Chen, Martin Grohe. Constraint satisfaction with succinctly specified relations. In *Journal of Computer and System Sciences*, Volume 76 Issue 8, December 2010, pages 847-860.
- [9] Claude Delobel. Normalization and hierarchical dependencies in the relational data model. In *ACM Transactions on Database Systems (TODS)*, Volume 3 Issue 3, September 1978, pages 201-222.

- [10] Todd J. Green, Grigoris Karvounarakis, Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 31-40.
- [11] Martin Grohe, Dániel Marx. Constraint solving via fractional edge covers. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, pages 289-298.
- [12] Georg Gottlob. On minimal constraint networks. In *Artificial Intelligence*, Volume 191-192, November 2012, pages 42-60.
- [13] Georg Gottlob, Nicola Leone, Francesco Scarcello. A comparison of structural CSP decomposition methods. In *Artificial Intelligence*, Volume 124 Issue 2, December 2000, pages 243-282.
- [14] G. Jaeschke, H. J. Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '82, pages 124-138.
- [15] William Kent. A simple guide to five normal forms in relational database theory. In *Communications of the ACM*, Volume 26 Issue 2, February 1983, pages 120-125.
- [16] Akifumi Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of the third international conference on Very Large Data Bases*, VLDB '77, Volume 3, pages 447-453.
- [17] Dan Olteanu, Christoph Koch, Lyublena Antova. World-set decompositions: Expressiveness and efficient algorithms. In *Theoretical Computer Science*, Volume 403 Issue 2-3, August 2008, pages 265-284.
- [18] Dan Olteanu, Jakub Závodný. Factorised Representations of Query Results: Size Bounds and Readability. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 285-298.
- [19] Dan Olteanu, Jakub Závodný. On Factorisation of Provenance Polynomials. In *3rd USENIX Workshop on the Theory and Practice of Provenance*, June 2011.
- [20] Dan Olteanu, Jakub Závodný. Size Bounds for Factorised Representations of Query Results. In *ACM Transactions on Database Systems*, Volume 1, No. 1, Article 1, July 2013.

- [21] Todd L. Veldhuizen. Leapfrog Triejoin: a worst-case optimal join algorithm. LogicBlox Technical Report, 2012.