Queries with Order-by Clauses and Aggregates on Factorised Relational Data

Tomáš Kočiský
Magdalen College
University of Oxford

A fourth year project report submitted for the degree of
Masters of Mathematics and Computer Science

May 21, 2012

Acknowledgements

I am truly indebted and thankful to my supervisor Dan Olteanu for his expert guidance and inspiring discussions.

I would also like to thank Jakub Zavodny for participating in our discussions, and Nurzhan Bakibayev for his implementation of the FDB that I extended.

Finally, I would like to thank my family for constant support and encouragement.

Abstract

Factorised representation form a succinct encoding of relational data, whose size can be exponentially smaller than equivalent representation. The nesting structure of the factorised representations is described by factorisation trees. For a query, only valid factorisation trees, characterised previously, can describe factorised representations over any database. This thesis aims to extend the theory of optimisation and evaluation of conjunctive queries with an order-by clause, and a group-by clause with an aggregate.

For queries with an order-by clause, I present a characterisation of factorisation trees, such that factorised representations with the nesting structure of these factorisation trees allow enumeration in sorted order with constant delay and constant precomputation time. Moreover, I present a polynomial-time optimisation heuristic for finding one of the characterised factorisation trees in the space of all valid factorisation trees for a given query. Optimisation with this heuristic was experimentally evaluated and produces results very close to the optimal solution.

For queries with group-by and an aggregate, I present a characterisation of all factorisation trees whose nesting structure can represent the result of a query over any database.

Contents

Chapter 1

Introduction

Relational databases represent the most commonly used data management paradigm for structured information. It can be found behind major websites, and a lot of systems that need to store, process, and retrieve information. The demand on these systems is often high and, therefore, common requirements are for them to be fast and space efficient.

Databases often store data in relations as a list of tuples. However, it is likely there is more hidden structure in the relations, and especially in relations representing results of queries and materialised views. Therefore, a natural question to ask is: can we exploit this structure to obtain more succinct representations of the data? And thereby, also work with it faster? One answer to these questions, that we pursue in this thesis, is that the recently introduced representation system of *factorised representations* of relations can be both more succinct than relations they encode and allow for faster processing [4]. In particular, we show how this framework of factorised representations can be extended with two key features that are ubiquitous: sorting and aggregation as present in SQL ORDER-BY and GROUP-BY clauses. This is a key contribution since many real-world queries need sorting and aggregation. For instance, all 22 queries in the TPC-H benchmark [1] use at least one of these two features. In the sequel, we exemplify factorised representations and their nesting schemas.

Factorised representations (f-representations) can be thought of as an algebraic factorisation of a relation, whereby relations are unions of tuples and each tuple is a product of unary relations representing a single value. Classes of f-representations that share the same nesting structure are defined by so-called *factorisation trees* (f-trees) — undirected rooted forests of trees.

Example 1.1. In Figure 1.1 we can see the relation P and one of its equivalent f-representations. For this particular f-representation there is an f-tree; not all f-representations have an f-tree, but converse is true. The f-tree describes the nesting structure of the f-representation — starting from the root A, first we have union over values of A, and each of them

A	B
1	1
1	3
2	1
2	4

(a) Relation P

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

(b) f-representation of P

$$\begin{tikzcd} A \arrow[d, no head] \\ B \end{tikzcd}$$

(c) f-tree for the f-representation

Figure 1.1: Example of an f-representation with its f-tree for a given relation.

allows values from the subtrees, in this case only from attribute B (see Example 1.2 for more details). □

There is a precise and complete characterisation of all f-trees for a given database schema and a given conjunctive query, for which there is an f-representation that can represent the result of the query over the database schema for *any* data contained in the relations [4]. Also an asymptotically tight bound on the size of the f-representation is given that can be inferred from the f-tree.

Example 1.2. In Figure 1.2 we have relations P, R, with attributes $\{A, B_P\}$, $\{B_R, C\}$. Let's consider the query $Q = \sigma_{B_P=B_R}(P \times R)$. Figure 1.2 shows two possible f-trees $\mathcal{T}$ and $\mathcal{U}$ describing possible nesting structures of the factorised representation of the query result and also one such representation over the f-tree $\mathcal{T}$. I have omitted the multiplication signs for clarity.

Notice that the f-tree $\mathcal{T}$ describes the nesting structure of the f-representation — first we have a union over values of the equated attributes B_P, B_R and for each of those there is a product of f-representations of subtrees for these attributes, in particular, product of possible values of A and C for the corresponding B_P, B_R.

The f-tree $\mathcal{U}$ (Figure 1.2e) is not an f-tree that can represent result of query Q for all possible data, because B_P, B_R and C don't factorise for each value of A in general, and this example is an example of that. □

Given relations and a query, we want to evaluate the query. We can represent the join of the relations using an f-representation with its f-tree, and find a sequence of operations that manipulate the two, to find the result of the query with equalities [2]. These operations, for example, are swapping of two nodes in the f-tree, or merging two nodes (equating the corresponding attributes), with the corresponding adjustment to the f-representation.

This thesis further explores operations on these f-representations, and shows how to do order-by and group-by clauses on them.

A	B_P
1	1
1	3
2	1
2	4

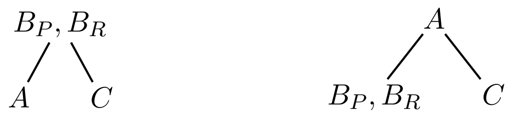
(a) Relation P

B_R	C
1	1
3	2
3	3
4	4

(b) Relation R

$$\langle B:1\rangle\,(\langle A:1\rangle \cup \langle A:2\rangle)\,\langle C:1\rangle \cup$$
$$\langle B:3\rangle\,\langle A:1\rangle\,(\langle C:2\rangle \cup \langle C:3\rangle)\cup$$
$$\langle B:4\rangle\,\langle A:2\rangle\,\langle C:4\rangle$$

(c) f-representation of $Q(P, R)$



(d) good f-tree $\mathcal{T}$ (e) bad f-tree $\mathcal{U}$

Figure 1.2: Example of an f-representation with good and bad f-trees for query Q.

1.1 Motivation and Challenges

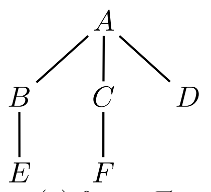
After joining relations and selecting from the result, we are often left with too many tuples to display to the user. Choosing only some of them is what we want to do. But which tuples to keep and which to discard? Ordering by, for example by relevancy, is one of fundamental operations in any information retrieval system, and it allows us to easier view the results we are most interested in — which we specify by an order. Group-by clause, on the other hand, provides further information on specific classes of tuples, for example, number of items that share a specific location and a manufacturer, and such.

For order-by clauses, I focus on the lexicographic ordering by a sequence of attributes. I will *precisely characterise* f-trees, for which we can enumerate tuples, in the order specified, with *constant delay* between tuples. By constant delay we will mean that for enumerating each next tuple we want to spend at most time proportional to the size of a tuple, i.e. $O(|\mathcal{S}|)$.

Example 1.3. If we consider the f-representation in Example 1.2, enumerating tuples ordered by B is straightforward, because the elements of the union have the singletons of B at the beginning, which corresponds to B being at the root of the f-tree, and moreover, they are already ordered in ascending order, which will be the case for each list of values of an attribute in f-representations. However, to order by C, we need to first find the smallest value of C, and that is only possible by looking at three lists of ordered values of C, each corresponding to a value of B, i.e. a valuation of ancestors of C. $\square$

This suggest that not all f-trees are convenient, or even admit the possibility, for ordering by any sequence of attributes with constant-delay enumeration and constant precomputation time, i.e. to enumerate the first tuple it take the same time as the rest. I will later show that enumeration with constant delay and constant precomputation time of tuples ordered by C is not possible in this case in the example.

For group-by clauses, we want, for each valuation of group-by attributes, to apply an aggregate function to the remaining attributes, or project them away. To do this effectively, we need an access to the sets of tuples corresponding to each valuation of group-by attributes, and I will show, for which f-trees we can access these tuples, again, with constant delay and constant precomputation time. And I will again precisely characterise f-trees that describe f-representation that can represent the result of a query with a group-by clause. As we have seen in Example 1.2, not all f-trees could be used as a structure to represent results of the query in the example — attributes B and C were dependent in a way, by coming from the same relation, which can't be factorised in general, and so they needed to be on the same root-to-leaf path. This is what is restricting the possible f-trees.



(a) f-tree $\mathcal{T}$

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

(b) an f-representation over $\mathcal{T}$, only some of the times operators shown

$$\begin{aligned}
&\langle A:1\rangle\,(((\langle B:1\rangle\,(\langle CNT:2\rangle)) \times (\langle C:1\rangle\,\langle F:2\rangle) \times \langle D:2\rangle\\
&\qquad(\langle B:2\rangle\,(\langle CNT:3\rangle))\\
&\qquad\qquad\times(\langle C:3\rangle) \times \langle D:2\rangle)\\
&\langle A:2\rangle\,(((\langle B:1\rangle\,\langle CNT:1\rangle) \times (\langle C:1\rangle) \times \langle D:3\rangle)
\end{aligned}$$

(c) the f-representation after count(E) as the CNT attribute

Figure 1.3: The aggregate value in the f-representation. (Database schema is $\{\{A,B,E\},\{A,C,F\},\{A,D\}\}$.)

Example 1.4. Consider the f-tree $\mathcal{T}$ and the f-representation over it from Figure 1.3 for database schema with relations $R_1 = \{A,B,E\}$, $R_2 = \{A,C,F\}$, $R_3 = \{A,D\}$. Let us explore two queries on these relations.

First consider the query $Q_1 = \sigma_\emptyset(R_1 \times R_2 \times R_3); \text{group-by} = A,B,C,D; \text{count}(E)$. Similarly to Example 1.3 where enumerating tuples in order of a root node attribute was fast, because we did not have to merge lists, we can here access tuples for the group-by attributes easily, because they are all above the non-group-by attributes. And to evaluate the aggregate, for each valuation of A,B,C,D (or just A,B is enough) we count values of E, and we discard values for F. The result of the aggregate can be placed in the f-representation instead of union of values of E (Figure 1.3c), which corresponds to placing the new attribute at the place of E in $\mathcal{T}$. And this works for any input database.

However, if we wanted to count(E, F), which is the same as count($*$), for the same group-by clause, it is unclear where the resulting number of the count aggregate should be placed in the f-representation. The Chapter 5 explores this case. $\square$

By having characterised the f-trees we are interested in, we can search for an f-tree which will have the asymptotically smallest f-representation, as defined by the function $s(T)$ [4], which can be inferred from an f-tree. This $s(T)$ is a rational number, a solution to a linear program based on the f-tree. It is related to the fractional edge cover number of a hypergraf of the query. Further, we can search for an optimisation plan — path of f-trees where edges represent operations, like swapping, — that, for example, minimises the maximal $s(T)$ over the path. And then, we can compare execution time of this plan for f-representations with execution time of the same query with an engine that uses the flat representation of the database.

Related Work

There has been no previous work on enumeration in sorted order or aggregates on the factorised representations. There has been research into enumeration of flat relations in sorted order. For example, for acyclic conjunctive queries, there are characterised several orders, including the lexicographic, for which it is possible to do enumeration in sorted order with polynomial delay in terms of query and the relation size [3]. In comparison, with the factorised representation of size n we can do[1] this with precomputation time $O(n \log n)$ and then constant delay and, moreover, the n can be exponentially smaller that the relation size in the above. So the factorised representation has an advantage to, basically, do the sorting only on a smaller data set.

1.2 Contributions

The contributions of this work are as follows:

- I will precisely characterise the class of f-trees whose f-representation admits enumeration in the lexicographical order for a given sequence of order-by attributes, schema, and *any* database over this schema, with constant delay and constant precomputation time.
- Given a conjunctive query with an order-by clause, the space of all valid f-trees that can represent the result of the query is in general exponential and finding an f-tree with minimal factorised representation that allows constant-delay enumeration is a hard

[1]the following complexity is for transforming the input representation into a form enumerable with constant delay, which I will characterise

problem. I will therefore propose a polynomial-time greedy heuristic, that explores part of this space, yet can still find almost optimal f-trees. We experimentally verify the effectiveness and efficiency of this heuristic and compare it to the full-space search. I will also compare the actual optimisation plan execution time, i.e. execution time of the path found in the space starting from a given f-tree, on an f-representation to the execution time of the query on a database with flat representation.

- I have implemented the above two kinds of search and enumeration in sorted order on top of an existing implementation of f-representation and flat representation database systems.
- For queries with group-by clause, I will characterise the f-trees whose f-representation admit access to the classes of tuples to be aggregated with constant delay between tuples and constant precomputation time.
- I will precisely characterise all the f-trees whose f-representation can represent the result of a given query with group-by clause, zero or one aggregate function applied to an attribute, given schema, and *any* database over this schema. Implications of this allow us to do further operations and plan funding on the result relation, in the same manner as before applying the aggregate, and all this, with having the tree dependency sets restricted as little as possible during the application of the aggregate function, and thereby allowing the most room possible for further optimisation.

1.3 Outline

- Chapter 2 introduces theory of f-trees and f-representations.
- Chapter 3 introduces enumerations of f-representations, gives statement and proof of the characterisation on f-trees that admit constant-delay enumeration in sorted order with constant precomputation time, and gives proof of existence of an f-representations over these f-trees for a given query. It introduces several search methods for finding of such f-trees. Chapter 4 describes the implementation of evaluation of queries with order-by, and Chapter 6 presents experiments of the optimisation plan search and query with order-by clause evaluation.
- Chapter 5 gives characterisation of f-trees that admit access to tuples to be aggregated with constant delay and constant precomputation time. And further, explains how to compute the aggregates. It also gives characterisation of f-trees that can be used to represent results of queries with group-by clause and aggregates.

Chapter 2

Prerequisites

I am going to present some background needed to understand my thesis. All of the definitions and theorems in this chapter closely follow or cite previous work [4, 2].

2.1 Factorised Representations and Trees

This section gives few basic definition about databases and queries considered in this work, and formal definitions of f-representations and f-trees.

Let us first have a look at some basic relation database concepts.

Definition 2.1 (Schema, Tuple, Relation, Database, Database Schema)**.** A schema $\mathcal{S}$ is a set of named attributes. A tuple is a mapping from a schema $\mathcal{S}$ to a domain $\mathcal{D}$. Relation R with a schema $\mathcal{S}$ is a set of tuples with this schema. A database $\mathbf{D}$ is a collection of relations. A database schema if a set of schemas of a collection of relations. $\square$

Definition 2.2 (Conjunctive Query, Equi-join Query)**.** A conjunctive query Q on relations $R_1, \dots, R_n$ with schemas $\mathcal{S}_1, \dots, \mathcal{S}_n$ is $Q = \pi_{\mathcal{P}}(\sigma_{\varphi}(R_1 \times \cdots \times R_n))$ with φ conjunction of equalities between attributes of the relations, and $\mathcal{P} \subseteq \bigcup_i \mathcal{S}_i$ is the projection list. Queries with $\mathcal{P} = \bigcup_i \mathcal{S}_i$ are called equi-join queries. As each order-by and group-by clause we will consider an ordered sequence and a set, respectively, of attributes from these relations such that no two are transitively related by equalities in φ. Further, we will also consider possibility of adding one aggregate function of an attribute or star. $\square$

The factorised representation represents tuples of a relation. It uses unions and products, and tuples are product of singletons after the representation is fully expanded. Or it can be enumerated without expanding, as we will see later.

Definition 2.3 (Factorised representation [4])**.** Let $\mathcal{S}$ be a relational schema. A factorised representation, or f-representation for short, over $\mathcal{S}$, is a relational algebra expression of the form

- $\emptyset$, the empty relation over schema S,
- $\langle\rangle$, the relation consisting of the nullary tuple, if $S = \emptyset$,
- $\langle A : a\rangle$, 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), where E is an f-representation over $\mathcal{S}$,
- $E_1 \cup \cdots \cup E_n$, where each E_i is an f-representation over $\mathcal{S}$,
- $E_1 \times \cdots \times E_n$, where each E_i is an f-representation over $\mathcal{S}_i$ and $\mathcal{S}$ is the disjoint union of all $\mathcal{S}_i$. □

A factorisation trees represent a nesting structure of some f-representations. We will use them to reason about manipulation of the corresponding f-representation and to infer bound on size of the f-representation.

Definition 2.4 (Factorisation tree [4])**.** A factorisation tree, or f-tree for short, over a possibly empty 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. □

The following definition defines more formally when an f-tree describes the nesting structure of an f-representation, we define an f-representation over an f-tree.

Definition 2.5 (f-representation over f-tree [4])**.** An f-representation E over a given f-tree $\mathcal{T}$ is recursively defined as follows:

- If $\mathcal{T}$ is a forest of trees $\mathcal{T}_1, \dots, \mathcal{T}_k$, then

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

  where each E_i is an f-representation over T_i.
- If $\mathcal{T}$ is a single tree with a root labelled by $\{A_1, \dots, A_k\}$ and a non-empty forest $\mathcal{U}$ of children, then

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

  where each E_a is an f-representation over $\mathcal{U}$ and the union $\bigcup_a$ is over a collection of distinct values a.
- If $\mathcal{T}$ is a single node labelled by $\{A_1, \dots, A_k\}$, then

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

- If $\mathcal{T}$ is empty, then $E = \emptyset$ or $E = \langle\rangle$. □

For simplicity, we could write instead of equal class of attributes $\langle A_1 : a\rangle \times \cdots \times \langle A_k : a\rangle$ only $\langle A : a\rangle$, when from context we understand A to represent the attribute class.

Remark 2.1. In f-representation we will enforce that values of each attribute are ordered in ascending order, i.e. in each of the unions in the Definition 2.5, for each valuation of its ancestors. This was so in the previous work as well, and I will make use of this later.

2.2 Dependency Sets and Path Condition

The concepts of dependency sets and path condition will be important later for restricting f-trees that can represent any database over a given database schema.

2.2.1 Characterisation of Valid F-trees

As we saw in Chapter 1, not all f-trees can represent the result of a given query. We saw that, for example, attributes from the same relation can't be siblings, and it turns out that even more needs to be true: they need to be on the same root-to-leaf path.

Given a query with a set of equalities, we can define classes of attributes. Let two attributes belong to the same class if they are transitively equated by the equalities in the query.

For a given query, let us consider f-trees where attributes from the same class are at the same node. Then the following proposition describes all the valid f-trees for the query.

Proposition 2.2 ([4]). *Given an equi-join query Q, the result relation has an f-representation over an f-tree $\mathcal{T}$ for any database $\mathbf{D}$ iff for each relation in Q its attributes lie along a root-to-leaf path in $\mathcal{T}$.* □

We can think of sets of attributes belonging to the same relation as *dependency sets.* And the requirement for the attributes to lie along the same root-to-leaf path is called the *path condition.*

NB: *It is important to keep in mind for later, that the concept of dependency sets together with the path condition were defined to precisely characterise the valid f-trees for the result relation of a query, based on the query. It is these two concepts that we will try to maintain when reasoning about the f-tree and f-representation.*

There are a few more trees that can represent the result of the query. In particular, trees such that attributes of some of the attribute classes are split into several nodes. In these cases, for each attribute class C, there needs to be a node labelled with some of the attributes of C that is also an ancestor to all other nodes labelled with attributes of C [4]. However, bringing these split nodes together can only decrease the representation size, and therefore, all the interesting f-trees have been described before, and we will consider only those.

2.2.2 Projections and Their Effect on Dependency Sets

Most of the time I will be considering equi-join queries, but in chapter on aggregates the fact how projections act on dependency sets will be useful. When attributes are projected away, for each attribute A that is projected away and its attribute class $A, A_1, \dots, A_n$, all dependency sets containing some of the attributes of the class are merged together, i.e. unioned as sets.

Example 2.1. In Example 1.2, after the query is evaluated, we have these dependency sets: $\{\{A, B_P\}, \{B_Q, C\}\}$. After projecting away attributes A, $B_P = B_Q$, $\{B_Q, C\}$ we would get, respectively, dependency sets $\{\{B_P\}, \{B_Q, C\}\}$, $\{\{A, C\}\}$ (with a new f-tree with A, C as parent-child is some order), $\{\{A\}\}$. $\square$

2.3 Size of F-representations

From an f-tree $\mathcal{T}$, it is possible to derive a rational number $s(\mathcal{T})$ that together with the database size gives asymptotic bounds on size of the corresponding f-representation [4]. For a given query Q, let $s(Q)$ be the smallest $s(\mathcal{T})$ over all valid f-trees $\mathcal{T}$ for the query T. In particular, if the size of the query is constant, we get the bound $\Theta(|\mathbf{D}|^{s(Q)})$ on size of the f-representation, where $|\mathbf{D}|$ is the sum of numbers of tuples of all relations in $\mathbf{D}$.

Definition 2.6 (F-representation size [4])**.** The size $|E|$ of an f-representation E is the number of singletons plus the number of empty relations in E. $\square$

The following is an aggregated definition of $s(Q)$ given in previous work [4].

Definition 2.7 (The important number $s(Q)$)**.** Let Q be a given query over a given database schema $\mathcal{S}$.

Let $\mathcal{T}$ be a valid f-tree for the query and let $L_{\mathcal{T}}$ be the set of all leafs of the forest $\mathcal{T}$. For each leaf $l \in L_{\mathcal{T}}$ and the root-to-leaf path p_l ending at l, consider the database schema $\mathcal{S}_l$ as a restriction of S by removing the attributes outside p_l. Define $\rho^*(p_l)$ as the solution to the linear program with variables $\{x_{R_i}\}$ for $R_i \in \mathcal{S}_l$:

$$\begin{array}{rll}\text{minimise} & \sum_{i:R_i \in \mathcal{S}_l} x_{R_i} & \\ \text{subject to} & \sum_{i:A \in R_i \in \mathcal{S}_l} x_{R_i} \geq 1 & \text{for each attribute } A \text{ on path } p_l \\ & x_{R_i} \geq 0 & \text{for all } i : R_i \in \mathcal{S}_l.\end{array}$$

The idea of this linear program is that it tries to cover the root-to-leaf path attributes with their relations, possibly assigning fractional number to each, and the cover of each node must be at least one. Note that the solution is a rational number.

Define

$$s(\mathcal{T}) := \max_{l \in L_{\mathcal{T}}} \rho^*(p_l), \qquad s(Q) := \min_{\text{valid } \mathcal{T} \text{ for } Q} s(\mathcal{T}).$$

$\square$

Example 2.2. Consider the f-tree $\mathcal{T}$

$$\begin{tikzcd}
& A \arrow[dl, dash] \arrow[dr, dash] & & \\
B \arrow[d, dash] & & D \arrow[d, dash] & \\
C & & E \arrow[dl, dash] \arrow[dr, dash] & \\
& F & & G
\end{tikzcd}$$

with relation schemas $\{A, B, C\}$, $\{A, D\}$, $\{D, E\}$, $\{E, F\}$, $\{D, G\}$. We need to consider all root-to-leaf paths: A, B, C; A, D, E, F; A, D, E, G. We need 1, 2, 3, relations, respectively, to cover these paths. Maximum being 3, we get $s(\mathcal{T}) = 3.0$. $\square$

2.4 Operations on F-trees and Normalised F-trees

This section presents operations on f-trees [2]. They are used to find evaluation plan for the query.

2.4.1 Query evaluation

An evaluation plan for a query can be found, which doesn't depend on data in the database. It works like this:

First, we construct an f-tree for the input database schema that represents the join of the relations. We are not considering the attribute classes as inferred from the equalities in the query at this point — all attributes are at their own node in the tree, and the equally named attributes from different relation, which we are joining on, are considered to be the same attribute, and so are represented by exactly one node in the tree. This is also illustrated by Example 2.2. We can construct from the database schema the f-tree shown. This is described in more detail in the previous work. The f-tree is one of the several possibilities of the result tree for a query with no equalities, which just joins the relations.

Further, if the query includes equalities, we can move nodes in the f-tree with the swap operation, described below, and when two attributes that we want to equate are positioned, for example, as siblings, we can use the merge operation, which combines the two nodes, and restricts the data in the f-representation.

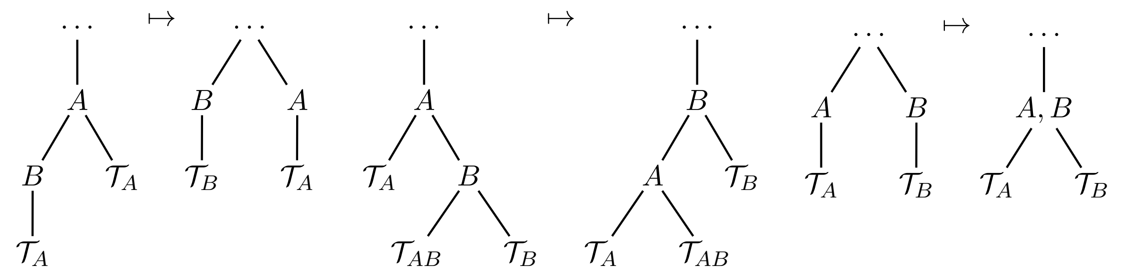


Figure 2.1: Operation on f-trees [2]. Push-up on general f-trees, swap and merge act on and produce normalised f-trees.

As we can see, this kind of evaluation has different f-trees and their corresponding f-representations as intermediate steps. For each we can calculate the $s(\mathcal{T})$ and, thereby, get a bound on the intermediate representation size, which we want to minimise.

Let's look at some the operations in more detail. They are explained here a bit less formally, for more formal description see previous work [2].

2.4.2 The Push-Up Operation

The push-up operation basically factors out common subexpression. For a node A and its child B, we can push-up B, as shown in Figure 2.1a, if none of the descendant nodes of B, or B itself, depend on A. If this is the case, we can move B with its children to be a sibling of A. Note that, since none of these moved nodes depended on A, we didn't violate the path condition. In the process we decreased the lengths of root-to-leaf path, and so the $s(\mathcal{T})$ decreased or remained unchanged.

Example 2.3. Consider two relation P, R in Figure 2.2 and f-tree $\mathcal{T}$ valid for the query $Q = P \times R$, a query with no equalities, with its corresponding f-representation. We can push-up node C, since B doesn't depend in it. And indeed we can see in the f-representation that we can factor out the attribute C. We can see the result in Figure 2.3. Notice also that the representation size decreased. □

2.4.3 Normalised F-trees

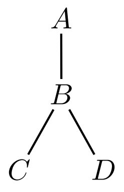
If we can factor out terms, we should do it. It has no immediate advantages to leave the f-tree in this state. And, moreover, pushing up nodes when possible, can't increase the f-representation size, it can only help. We want the f-trees to be as wide and branched as possible, as opposed to deep, because the former have potential for smaller $s(\mathcal{T})$. So we define normalised f-trees.

A	B	D
1	1	2
1	2	3
2	1	1

(a) Relation P

A	C
1	2
2	1

(b) Relation R

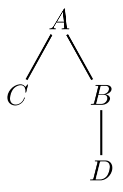


(c) f-tree $\mathcal{T}$

$$\begin{aligned}&\langle A:1\rangle\,(\langle B:1\rangle\,\langle C:2\rangle\,\langle D:2\rangle\\ &\qquad\cup\langle B:2\rangle\,\langle C:2\rangle\,\langle D:3\rangle)\\ &\cup\langle A:2\rangle\,\langle B:1\rangle\,\langle C:1\rangle\,\langle D:1\rangle\end{aligned}$$

(d) f-representation over $\mathcal{T}$

Figure 2.2: Example of two relations, valid f-tree for $P \times R$, and the corresponding f-representation.



(a) f-tree $\mathcal{T}$

$$\begin{aligned}&\langle A:1\rangle\,\langle C:2\rangle\,(\langle B:1\rangle\,\langle D:2\rangle\\ &\qquad\cup\langle B:2\rangle\,\langle D:3\rangle)\\ &\cup\langle A:2\rangle\,\langle C:1\rangle\,\langle B:1\rangle\,\langle D:1\rangle\end{aligned}$$

(b) f-representation over $\mathcal{T}$

Figure 2.3: f-tree and f-representation after a push-up operation on node C from Figure 2.2.

Definition 2.8 (Normalised f-tree [2]). An f-tree $\mathcal{T}$ is normalised if no node in $\mathcal{T}$ can be pushed up without violating the path constraint. □

Notice that the f-tree in Figure 2.3a is normalised, because we can't push up any of the nodes. (Each of the nodes depends on its, in this case, immediate descendant.)

2.4.4 The Swap Operation

The swap operation takes two nodes of a *normalised* f-tree and produces a *normalised* f-tree. It is depicted in Figure 2.1b. It swaps the two nodes and places the subtrees based on dependency on the two swapped nodes — the subtrees in $\mathcal{T}_{ns}$ depend on nodes ns. It requires a little more thought on why the resulting f-tree is actually normalised.

One thing to note is that, although the operation in the figure looks like a rotation in a binary tree, it is not. The subtrees in the figure can be forests, and the partition of children of B into $\mathcal{T}_{AB}$ and $\mathcal{T}_B$ needs to be computed.

2.4.5 The Merge Operation

The merge operation acts on two sibling nodes of a normalised f-tree. It basically does a cross product of the two attributes and keeps only the pairs with two same values — it equates the attributes. This operation is depicted in Figure 2.1c. The result f-tree is normalised, again.

There is another operation, called absorb, used for the same purpose. It acts on a node and a descendant, equates them, and removes the descendant. Moreover, it normalises the f-tree with push-up operations.

2.5 Constant Delay and Constant Precomputation Time Enumeration

Here are defined concepts that define time complexity of enumeration.

Definition 2.9. *Constant delay* enumeration is enumeration where to output the next tuple we need time $O(|\mathcal{S}|)$, where $|\mathcal{S}|$ is size of the database schema of the current tree, i.e. number of attributes of the result relation, i.e. size of the tuple itself. This is the best possible time for enumerating one tuple, since it is its size. And *constant precomputation* time is $O(1)$. We can alternatively have $O(|\mathcal{S}|)$, but we can absorb the latter into the computation of the first tuple, so we will consider the former. $\square$

Chapter 3

Queries with Order-by Clause

We already know how to evaluate a conjunctive query using f-representations. We would like to evaluate such a query with an order-by clause. In particular, we would like to get an f-representation of the result such that we can enumerate the tuples of the result quickly in the *lexicographic* order specified by the query. And more precisely, this chapter gives an f-representation for which we can enumerate the result tuples with constant delay and constant precomputation as defined in Definition 2.9.

As order-by clause I take an ordered sequence of attributes. Without loss of generality I assume that no two are transitively related by equalities in the query, for if they are, their values are the same for each tuple, so we can trivially ignore one of them. We want to order by the first attribute first and then the rest. For each we could specify whether we want ascending or descending order, but since these cases are so similar, it is enough to solve this problem for ascending orders of each attribute, as we will see later.

We can further assume that we have an intermediate result where the equalities of the query are already enforced, and so we get this problem:

The problem: Given an f-representation over an f-tree, and a sequence of attributes, we want to find an equivalent f-representation with its f-tree, such that we can enumerate the tuples in lexicographic order, given by the sequence of attributes, each tuple with time complexity $O(|\mathcal{S}|)$ and constant precomputation time, i.e. the first takes only $O(|\mathcal{S}|)$ too.$\square$

We want to do enumeration of an f-representation with specific ordering of the enumerated tuples. Let us first have a look at how to do it with arbitrary order, and then find what properties we need for the enumeration to have the right order and complexity.

3.1 Parse Tree and Enumeration

This section first explains how the f-representation is stored — the parse tree —, and then how to do enumeration in some order. This has already been done before, and also implemented, but let us recall it [4].

Parse tree: The parse tree is a data structure that holds the natural parse tree of the f-representation (which is associated with a f-tree). More precisely we can imagine it as follows. There are two kinds of nodes: union nodes and product nodes. The product nodes hold also a value of some type and on the root-to-leaf paths they alternate. Let me explain this on an example.

Example 3.1. For a f-tree with only a single node, the parse tree would be a union node with children product nodes each holding one value for the attribute of the single node of the f-tree.

If the f-tree is single tree, i.e. not a forest, with a root node A, then the parse tree is like the above tree with *each* product node with a value of attribute A in the parse tree having union node children, one for *each* children attribute of A in the f-tree.

Lastly, if the f-tree is a forest, parse tree has as root a product node with no value and each tree in the forest is then under this product node in the form explained above. □

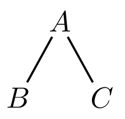
Enumeration: First for illustration, imagine that we want to enumerate 5 digit numbers in base 3. We do this by setting each digit at its first value, namely zero, and then we advance the last digit. When we have no more further values for the last digit we advance the second to last digit and set the last digit to the first permitted value, as determined by previous digits, which in this case will always be zero. And we continue in this manner until the first digit can't be advanced anymore.

For f-representations it is similar. However, the structure of advancing is not linear, but branches, and it is given by the f-tree. The branching gives us a choice which leaf to advance next. And we will encode this choice by a sequence nodes of the f-tree and call it the *priority list*.

Definition 3.1 (Enumeration attribute priority list)**.** Given an f-tree, let a priority list be a sequence of all nodes of the f-tree, such that for each node n, all of its ancestors appear before n in the list. The list can also be of labels of nodes. □

It is easy to see, that given a priority list, we can enumerate the f-representation by advancing values of the attributes in reverse order of the list. This enumeration outputs each next tuple in $O(|\mathcal{S}|)$ time. And, moreover, it describes all such orders.

For the above base 3 example, the priority list would be digits starting with the most significant digit and up to the least significant.



(a) f-tree $\mathcal{T}$

$$\langle A:1\rangle\,(\langle B:1\rangle \cup \langle B:2\rangle)(\langle C:1\rangle \cup \langle C:2\rangle)$$
$$\langle A:2\rangle\,(\langle B:1\rangle \cup \langle B:2\rangle)(\langle C:1\rangle \cup \langle C:2\rangle)$$

(b) f-representation over $\mathcal{T}$

A	B	C
1	1	1
1	1	2
1	2	1
1	2	2
2	1	1
2	1	2
2	2	1
2	2	2

(c) Enumeration for priority list $[A, B, C]$.

A	B	C
1	1	1
1	2	1
1	1	2
1	2	2
2	1	1
2	2	1
2	1	2
2	2	2

(d) Enumeration for priority list $[A, C, B]$.

Figure 3.1: Example of two enumerations for priority lists for given f-tree and f-representation.

Example 3.2. Consider the f-tree and f-representation from Figure 3.1. There are precisely two priority lists for this f-tree: $[A, B, C], [A, C, B]$. Their corresponding enumerations are shown in Figures 3.1c and 3.1d.

Notice that, both enumerations are ordered by A; first enumeration is ordered by A, B and A, B, C; and second enumeration is ordered A, C and A, C, B. □

Proposition 3.1 (Priority list induces lexicographic order)**.** *Given an f-representation over an f-tree, and an attribute priority list for the f-tree, then the enumeration of the f-representation according to this priority list is ordered by any prefix sequence of the priority list in ascending order of each of the attributes.* □

Remark 3.2. To get a descending order for an attribute, instead of ascending, traverse the increasingly sorted lists of values of the attribute in the parse tree in reverse. □

Proof. See Appendix A □

3.2 The F-tree Characterisation

This section characterises all the f-trees that admit enumeration of each tuple in $O(|\mathcal{S}|)$ time with constant precomputation time.

Theorem 3.3 (The Characterisation)**.** *Let $\mathcal{T}$ be an f-tree and U a sequence of attributes. F-representation over $\mathcal{T}$ can be enumerated in sorted lexicographic order by U, with constant* *precomputation time and time $O(|\mathcal{S}|)$ for each tuple if and only if each attribute X of U is at a node in $\mathcal{T}$ which is either a root, or a child of a node labelled with an attribute appearing before X in U.* □

The "If" Part of The Characterisation Theorem 3.3

Let us first observe how the attributes of a priority list are positioned in an f-tree and, thereby, relate them to position of order-by attributes required in the Theorem 3.3.

Lemma 3.4 (Prefix of a priority list in the f-tree). *Given an f-tree $\mathcal{T}$ and a priority list U for this f-tree, then for any prefix of U, each attribute A of the prefix is either a root in $\mathcal{T}$, or a child of a node labelled with an attribute appearing before A in the prefix. In particular, the first attribute of U is a root in $\mathcal{T}$. Conversely, for any sequence of attributes V such that all $X \in V$ are either a root, or a child of a node labelled with an attribute appearing before X in V, there exists a priority list with prefix V.* □

Proof. Immediate from the definition of a priority list. □

Proof of the "if" part of the Theorem 3.3. We are given an f-tree $\mathcal{T}$ and a sequence of attributes U. Assume that each attribute X' of U is in $\mathcal{T}$ either a root, or a child of a node labelled with an attribute appearing before X' in U. Then by Lemma 3.4 there is a priority list V for $\mathcal{T}$ that has U as a prefix, and by Proposition 3.1 the enumeration with the priority list V is ordered by U and the enumeration can be done in time required. □

The "Only If" Part of The Characterisation Theorem 3.3

This section proves the converse: that if we assume the order and the time complexity of the enumeration, then we need the tree structure of attributes of the order-by sequence, which is described in the theorem.

Lemma 3.5 (The first attribute at a root). *If we can enumerate an f-representation over f-tree $\mathcal{T}$ in sorted lexicographic order by attributes of a non-empty sequence U, with constant precomputation time and delay $O(|\mathcal{S}|)$ for each tuple, then the first attribute of U labels a root in $\mathcal{T}$.* □

Proof. Idea is that there is no way to avoid merging otherwise. See Appendix A. □

Proof of the "only if" part of the Theorem 3.3. The proof is by strong induction on (n, m), $n \geq m$, n number of nodes of an f-tree and m number of elements in the sequence U. We are assuming here that the enumeration can be done in the order and complexity specified in the theorem, and we want to show the structure of the f-tree.

Base case $(n, 0)$**:** Immediate.
Induction step (n, m)**:** Assume the hypothesis is true for all pairs of (s, t), $s \geq t$, $s < n$, $t < m$. By Lemma 3.5 the first element of U is a root in the f-tree forest $\mathcal{T}$. Let U' be a sequence of attributes of U, except the attribute A, and let $\mathcal{T}'$ be the restriction[1] of $\mathcal{T}$ without the root node labelled with attribute A.

We assumed an f-representation over $\mathcal{T}$, call it F, can be enumerated in the order and the complexity, and so for each value a of A the tuples can be enumerated in such way, because we can select the subtree of a in a parse tree of F in $O(1)$, and so each of the restricted f-representations F_a over $\mathcal{T}$ can be enumerated in the order and the complexity. So we can apply the induction hypothesis to $\mathcal{T}'$ and U. So for $\mathcal{T}'$ and U', by induction hypothesis, every node X' of U' in $\mathcal{T}'$ is either a root, or a child of a node labelled by an attribute appearing before X' in U'.

Now in $\mathcal{T}$, $A \in U$ is a root. The attributes X'' of U', a subsequence of U, that were children of nodes labelled by an attribute appearing before X'' in U' have this property also in context of $\mathcal{T}$ and U. The rest, that were roots in $\mathcal{T}'$, are going to be roots in $\mathcal{T}$, or children of the node labelled with A, which comes before all of them in U, since A is the first attribute in U. $\square$

This completes the proof of Theorem 3.3.

3.3 Existence of a Valid F-tree

I have described the f-trees that allow enumeration with constant delay. However, for a given query and database schema, as we have seen in examples previously, not all f-trees can represent the result. In particular, only those that satisfy the path condition. Now, the question is, does one of those f-trees that can represent the result have the structure we require for constant delay enumeration? I show this in two ways. Firstly, by an interesting observation of uniqueness of structure of the top parts of the in-sorted-order-enumerable f-tree by constructing it and it will satisfy the path condition by construction. Secondly, by exhibiting f-tree operations that can transform a given f-tree to one that allows the enumeration, and this in Section 3.4.2.

The following proof of the following lemma provides a way to construct all f-tree that admit enumeration in sorted order with constant delay and constant precomputation time. It further shows that the relative placement in the f-tree of attributes of the order-by clause is the same among all these f-trees, i.e. they form connected trees at roots of the trees in the f-tree.

[1]remove the node with attribute A and promote its subtrees to be trees in the forest $\mathcal{T}'$

Lemma 3.6. *Given a database schema $\mathcal{S}$ and a sequence of attributes U that we want to order by, then there is an f-tree that satisfies the condition for the f-tree to be constant-delay enumerable, and the path condition based on dependency sets from $\mathcal{S}$, and all such f-trees have the same tree structure of attributes of U.* □

Proof. See Appendix A □

3.4 Query Optimisations

Now we are getting to answering the main problem, stated in the introduction of this chapter — given f-representation over an f-tree $\mathcal{T}$, we want to find an equivalent f-representation over some f-tree that can be enumerated with constant delay and precomputation time.

It is easy to see that from an f-tree that satisfied the path condition for a database schema, we can get to all other f-trees that satisfy the path condition for the database scheme using only the swap operation. (This is a previous result, but was not presented in detail.)

Our aim in this section is to find a sequence of such swap operations. This is obviously hard, because the search space is exponential in number of the attributes.

The following sections describe several kinds of search for the f-tree we desire.

3.4.1 Full Search: The Brute Force Way

I use the same full search as was proposed in the previous work [2], and also implemented. It uses the Dijkstra's algorithm to search the *exponential* full space of f-trees, generating the following ones using the swap operation. We need just a different goal condition that describes the f-trees enumerable in sorted order with constant delay.

3.4.2 Greedy-Heuristic Search: The Simple But Effective Way

I am proposing a simple, but as we will see in Chapter 6, very effective heuristic how to choose nodes for the next swap operation.

In turn, for each node in the order-by attribute sequence, say U, we will keep swapping the node with its parent until it becomes a root, or becomes a child of a previously brought-up node, i.e. a node sooner in the sequence of order-by attributes.

This method requires only about $O(\text{depth(f-tree)} \cdot |U|)$ f-trees to be explored, which is considerably less that the exponential full-search space size.

3.4.3 Distance-Measure-Heuristic Search: The Failed Attempt

Since, by Lemma 3.6, the attributes of the order-by attribute sequence U have a predetermined position in the f-tree for a database schema $\mathcal{S}$, I have tried to define a distance measure between the current f-tree $\mathcal{T}$ and the target f-tree as

$$d_{U,\mathcal{S}}(\mathcal{T}) = \sum_{A \in U} |\text{currentDepth}(A) - \text{targetDepth}(A)| \, .$$

Then the search would consider only paths of f-trees where this distance is non-increasing.

This has an advantage compared to the greedy-heuristic search, that it could potentially achieve better results by doing some modifications to other parts of the f-tree, other that just the nodes with attributes of U. In fact, we will see in Example 6.1 later that the greedy-heuristic search will not find an optimal solution in general, so some fixes to other parts of the f-tree are really necessary.

Moreover, compared to the full search, this limits the search space, which is an advantage when the space is exponential.

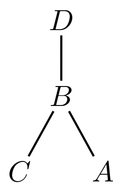
However, this distance-measure heuristic does not find a solution in all cases. There are cases, where the distance needs to increase first before it can decrease to 0, i.e. get to the goal.

Example 3.3. Consider database schema $\{\{A, B\}, \{B, C\}, \{B, D\}\}$ and the f-tree $\mathcal{T}$ in Figure 3.2. The figure shows all possible swaps on this tree and illustrates that the distance needs to increase for the goal to be reached.

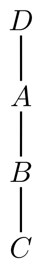
I have considered to augment the search criteria to allow an increase in the distance to the target, but in general, the increase required would be about as large as the size of the attribute order-by sequence, which is disadvantageous.

3.5 Summary

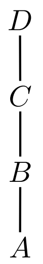
In this chapter we have seen how to do enumeration on a parse tree and priority lists that specify the enumeration order to some extent. We have seen the characterisation of *all* the f-trees, for a given order-by attribute sequence U and a given database schema $\mathcal{S}$, that allow enumeration in sorted order by the sequence with delay $O(|\mathcal{S}|)$ and constant precomputation time. Moreover, we have seen two existence proof for such f-trees: a construction that shows also the necessary structure of attributes of U in such a tree, and a proof by swapping and bringing nodes to the top of the tree. We have also seen two search methods for finding the desired tree, which we will evaluate in Chapter 6.



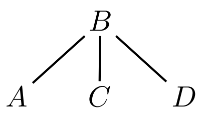
(a) f-tree $\mathcal{T}$, $\mathbf{d} = \mathbf{2}$



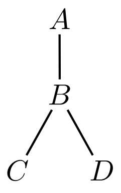
(b) after $\chi_{A,B}$, $\mathbf{d} = \mathbf{3}$



(c) after $\chi_{B,C}$, $\mathbf{d} = \mathbf{5}$



(d) after $\chi_{B,D}$, $\mathbf{d} = \mathbf{3}$



(e) target for order-by A, B, C, e.g. from (d) by $\chi_{A,B}$, $\mathbf{d} = \mathbf{0}$

Figure 3.2: All possible swaps on f-tree $\mathcal{T}$, a *unique* target f-tree, d is the distance measure to the target. (Database schema is $\{\{A, B\}, \{B, C\}, \{B, D\}\}$.)

Chapter 4

Implementation of Order-by

I have implemented plan finding and enumeration for queries with an order-by clause. There are two search methods implemented, the full search and the search with the greedy heuristic, as described in 3.4.1 and 3.4.2, respectively. And also the enumeration in sorted order on both the factorised and flat representations. Let me first describe the implementation of factorised representation database engine that I was extending.

4.1 Existing FDB Implementation

Query-evaluation plan finding and also the evaluation for f-representation was previously implemented. Evaluation on flat representation for the same kind of queries was also implemented before. The implementation is written in C++.

It has different classes for f-representation, parse tree, few optimisers that find the evaluation plan, a class that executes the plan on an f-representation and thereby evaluates a query. Also a class for computing the $s(\mathcal{T})$ is present.

The optimiser for the full search uses Dijkstra's algorithm. It uses string representations of f-trees as identifiers for the f-trees, and remembers the cost as a double precision floating point number. The optimiser also records various statistics like the number of goals, size of the search space, and such.

The query parsing, search algorithm logic, next f-tree computation, goal f-tree detection, and statistics, are all in the same optimiser class and mostly in the same method.

4.2 Improved Implementation

I have cleaned up the implementation somewhat by splitting the optimiser into smaller parts.

I have implemented from scratch a separate search class, using also the Dijkstra's algorithm but parametrised heavily. It has a template argument Cost, which is expected to behave like a double, for example. It is used to store the current path cost and has some comparison operators implemented on it. Previously a simple double was used, but I use a class which also records length of the path, and can encapsulate more metrics to asses the paths. Note, for the Dijkstra's algorithm it is required that the cost monotonically increases.

Second very important parameter to the general search class is the Options class, which are search options. Namely, the Options class, also with a Cost template argument, includes methods like: getStartFTree(), which returns the f-tree of the initial f-representation; getNextFTrees(ftree), which performs possible operations on the f-tree given and returns list of operation and next f-tree pairs; getCost(oldCost, move, newFTree), which returns the next cost as a Cost class; isGoal(ftree), which checks if the tree has the desired properties; recordGoal(ftree, cost), which keeps some statistics.

I have further created a Query class which is constructed from a string representation of the query, and can return list of parsed equalities or list of order-by attributes.

4.3 Evaluation of Order-by Queries

For the plan finding, I have defined a subclass of the Options class, basically reusing all the code from the full search implemented before, and, in addition to previous goal that equalities need to be resolved, I have added a check for the order-by attribute position in the tree, in particular, whether they are at a root, or a child of an earlier attribute in the list.

I have modified the enumeration method of parse tree and also of the flat representation. Which were constructing kind of priority list, similar to what I have defined in the previous chapter. It was done using a depth-first search and the nodes were added to the list as they were first visited. I have modified this to use an explicit stack and I am adding children in their reverse order on the stack. Each time I want pop of the stack and there is a node on the stack that has an attribute in the order-by list, it is prioritised, and thereby a priority list for enumeration in sorted order is produced.

I have implemented another subclass of Options, reusing all the above, just modifying the getNextFTrees(. . .) to return only one swap, precisely as described in Section 3.4.2. This implements the greedy-heuristic search.

In addition to modifying the priority list of the flat representation, which controls the order of attributes and sorting, I have sorted the relation using the standard stable sort provided by the Standard Template Library of C++.

The experiments are reusing the experiment generation infrastructure. Evaluating the experiment queries is done using my implementation and the result processing is done by a small Python script I wrote.

The Chapter 6 presents the experiments in more detail.

Chapter 5

Queries with Group-by Clause and Aggregates

This chapter explores queries with a group-by clause and an aggregate for f-representations. We will first see what is needed to access the groups of tuples with constant delay and constant precomputation time. This will follow from the results about enumerating in sorted order. Then we will consider high level view of the computation of the most common aggregates.

The main result of this chapter is the characterisation of the result f-trees of a query with a group-by clause and one aggregate. We will see that it is easy to overestimate the dependency sets, which together with the path condition restrict the admissible f-trees for a given query, but we will characterise the dependency sets precisely to allow the most room possible for further optimisation of the evaluation plans considering result from the aggregated query as an input.

As a group-by clause we will consider (unordered) set of attributes.

5.1 Accessing the Groups of Tuples

To access each group of tuples with constant delay and constant precomputation time is the same as being able to enumerate them in an order where the groups of tuples form consecutive tuples.

Theorem 5.1 (Accessing the groups of tuples)**.** *Given an f-representation over an f-tree $\mathcal{T}$ and a set of group-by attributes G, we can access the groups of tuples with constant delay and constant precomputation time if and only if all attributes of G are either a root in $\mathcal{T}$, or a child of another attribute of G.* □

Proof. Follows from The Characterisation Theorem 3.3. See Appendix A. □

The sequel talks about f-tree with the following property.

Definition 5.1. An f-tree, or and f-representation over an f-tree, is *grouped by G* if the f-tree has the property from the above Theorem 5.1. Further, for an f-tree $\mathcal{T}$ grouped by G, the set of *f-trees below G* is a set of all subtrees[1] of $\mathcal{T}$ that have a parent labelled with an attribute of G and don't contain any nodes labelled with an attributes of G.

5.2 Computation of Aggregates

A way to compute an aggregate is to enumerate the tuples for each valuation of G for an f-representation over an f-tree grouped by G. The computation of the aggregate is done in the usual way. It is possible to do this in better time complexity, for sketch of this see Appendix B. For purposes of this chapter assume we can compute aggregates.

5.3 Dependency Sets of the Aggregate Attribute

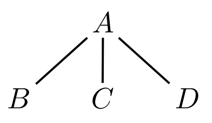
Previously the dependency sets together with the path condition characterised the f-trees that could represent the result of a given conjunctive query [4]. Having a precise characterisation, as opposed to an overestimate, allowed us to explore all the f-trees, together with the operations on them, and thereby allowed for the greatest possibility of optimising the size of f-representation.

We know how to compute the value on an aggregate. Now, the question is: where to put the result in the f-representation and the attribute in the f-tree? What should be the new dependency set, which we need for further plan finding in the future use of the result? This is what I am going to answer here.

5.3.1 Overestimating

Aggregate provides a value for each valuation of a set of group-by attributes G. We could express this data as a new relation having attributes $G \cup \{C\}$, where C is the attribute representing the aggregate value, and join it with the previous f-representation. This would effectively add a new dependency set $G \cup \{C\}$ that would force all these attributes to be on some root-to-leaf path. This is a huge overestimation.

Example 5.1. Consider the f-tree



[1]only trees, not forests

with dependency sets $\{A, B\}, \{A, C\}, \{A, D\}$, and group-by $\{A, B, C\}$ and count D. The above overestimation would result in a linear f-tree. But for each value of A the value of the aggregate is the same for all combinations of values of B and C. We see this from the f-tree. So it would be enough to replace B in the f-tree and also in the dependency set by the aggregate attribute. □

We could just join a relation, instead of $G \cup \{C\}$, that has the attributes

$$(\text{ancestors}(\text{f-trees below } G) \cap G) \cup \{C\}.$$

The reason this works is as follows. We can imagine splitting the representation to parts that has these attributes and the trees below G, and the rest, do the aggregation on the former, and then join them back. In the example above this would mean detaching A, C, D and doing aggregation on A, B only. We can see that this works from simple relational-algebra-like thinking. We will use a similar idea in a a proof later.

5.3.2 The Characterisation

In the previous section we were trying to join on a relation that specified the value of the aggregate for the groups. The problem was that we considered too many attributes for this relation and, thereby, we restricted the f-tree too much. Can the relation that joins on the values of the aggregate have smaller number of attributes?

The groups described by the group-by clause describe, for an attribute, multisets of values of the attribute. An aggregating function can then be applied to this multiset of values.

Let us define a subset of group-by attributes, which we will prove is the minimal set of group-by attributes that is needed to differentiate the multisets of values described above. And all the groups have one of these multisets and are obtained by joining with relations that are not needed for computation of the aggregate. This is in the same way as B, C were not needed in Example 5.1.

For min/max we need one attribute to compute the aggregate and there is a minimal subset of group-by attributes that can describe these multisets, and they form a relation together with a new aggregate attribute that can be later joined on the relation we want to aggregate. This is in the same way as when we were overestimating in the previous section. For sum, avg, and count, all non-group-by attributes are required for computation.

The following is the definition of the minimal subset.

Definition 5.2. Given attributes $\mathcal{A}$ and attributes G from a database schema $\mathbf{S}$, define $\mathcal{M}_{\mathbf{S},\mathcal{A},G} \subseteq G$ as the set of all $A \in G$ such that there exists a sequence $\mathcal{S}_1, \ldots, \mathcal{S}_j$ in $\mathbf{S}$ with $\mathcal{A} \cap S_1 \neq \emptyset$ and $A \in S_j$. □

The following theorem proves the properties of this set. Let me give an intuitive description of the two subsets of G described at the end of the following theorem. Given attributes needed for computation of the aggregate, call these attributes K together with all non-group-by attributes in the f-tree reachable from them. These form full f-trees below G. We take attributes in G from all the dependency sets that have an attribute in K, to get the sets.

Theorem 5.2 (Minimal subset of group-by attributes)**.** *Given a database schema* $\mathbf{S}$*, a set of relations* $R_1, \ldots, R_n$*, a set of group-by attributes* G*, an aggregate* f*, an attribute* A *to aggregate over, an f-tree* $\mathcal{T}$ *over schema* $\mathbf{S}$ *and grouped by* G*, we want to evaluate a query with only this group-by set and this aggregate. Let* $\mathcal{S}$ *be the smallest set of attributes of the database schema* $\mathbf{S}$*, such that for all databases and all valuations* g *of* G*, the valuation* $g' := g|_{\mathcal{S}}$ *of* $\mathcal{S}$*, the multiset of values to aggregate over in* $R_1 \times \cdots \times R_n$ *for* g *and in* $R = \pi_{\mathcal{S}}(R_1 \times \cdots \times R_n)$ *(with tuple set semantics) for* g' *is the same. Let* R_C *be the extension of relation* R *containing attributes* $\mathcal{S}$*, and a fresh attribute* C*, that represents the value of the aggregate* f*.*

Then the result of the query is $\pi_{G\cup\{C\}}(R_C \times R_1 \times \cdots \times R_n)$ *and the set* $\mathcal{S}$ *is as follows:*

- *if* f *is min or max, then* $\mathcal{S} = \mathcal{M}_{\mathbf{S},\{A\},G}$*;*
- *if* f *is sum, avg, or count, then for* $G^{\complement}$ *complement of* G*,* $\mathcal{S} = \mathcal{M}_{\mathbf{S},G^{\complement},G}$*.* □

Proof. See Appendix A. □

Theorem 5.3 (The Characterisation)**.** *Given an f-representation over an f-tree, a conjunctive query with only a group-by with a set* G*, an aggregate* f *over an attribute* A*. Then the result of the query can be an f-representation over f-tree which has dependency sets* only *as follows. The dependency sets are as after a projection on* G *together with the dependency set* $\mathcal{S}$*:*

- *if* f *is min or max, then* $\mathcal{S} = \mathcal{M}_{\mathbf{S},\{A\},G}$*;*
- *if* f *is sum, avg, or count, then for* $G^{\complement}$ *complement of* G*,* $\mathcal{S} = \mathcal{M}_{\mathbf{S},G^{\complement},G}$*.* □

Proof. The attributes need to be projected away. So that changes the dependency sets as described in Section 2.2.2 and the new dependency set is due to values of the aggregate, and there is no smaller one, by Theorem 5.2. □

Remark 5.4. The description of the dependency sets is not unique. The above can also be described as merging all dependency sets reaching into f-trees below G that contains attributes we need for computation of the aggregate (A or $G^{\complement}$) with added C, attribute of the aggregate. □

5.4 Step by Step Aggregate Evaluation

When evaluating a min/max aggregate of an attribute A the added dependency set will lie on an existing root-to-leaf path and we don't need to do any more rebuilding of the tree, only put the value of the aggregate at the end of this path instead of the f-tree below G that contained A.

For sum, avg, count, this can be more complicated, since the dependency set spans different root-to-leaf path in the f-tree. Therefore, I propose to evaluate the aggregate in steps and the evaluation is illustrated on the aggregate function count($*$).

For an f-tree, a set of group-by attributes G, and the aggregate count($*$) we can compute it as follows. For each f-tree $\mathcal{T}_i$ below G do a count while grouping on everything except $\mathcal{T}_i$ and previously computed counts. We will end up with counts of each of the f-trees below G and the result count($*$) is the product of them.

We can use optimisation techniques for moving nodes in the f-tree, and we know how the dependency sets work from the previous section.

I have explored two new operation that will be detailed in a future paper. The operations are a cross product of two neighbouring nodes in an f-tree, and applying functions, such as multiplication or summation, on pairs of a cross product. This would allow us to bring together the distributed counts and combine them together.

5.5 Summary

In this chapter I have presented a very important characterisation of f-trees for representation of the results of a query with a group-by clause. This will allow for future work to explore optimisation on such result f-trees. And I have proposed a step-by-step evaluation of aggregates that avoids rebuilding the whole representation after the aggregate is computed.

Chapter 6

Experimental Evaluation

This chapter presents experimental evaluation of the techniques proposed in this work, in particular: optimisation and evaluation of order-by queries. In the first experiment, I compare the full search and the greedy-heuristic search proposed earlier, in terms of the maximal $s(\mathcal{T})$ of the path, final f-tree $s(\mathcal{T})$, execution time, and the search space size. In the second experiment, I use the evaluation plans found in the first experiment to do the evaluation on randomly generated relations to compare the evaluation time. These experiments are based on the experiments that evaluate conjunctive queries with equalities [2] and they are modified for queries with order-by.

We consider queries with order-by and with no projection or select conditions, i.e. no equalities. It would be possible to combine the full search for plan finding for equalities with plan finding for order-by. The modification would be just to change the goal condition. And since the goal condition is an extension of the previous one (that the equated attributes are merged) the result plan will have the same cost as if we did the plan finding one after another. So we choose not to have equalities, and the same for the greedy-heuristic search for them to be comparable.

6.1 Experiment One: Evaluation Plan Finding

In this experiment we will compare the efficiency of the full search and the greedy-heuristic search.

6.1.1 Input Data and Parameters

The input data are: a schema, an optimal f-tree based solely on the schema, and a query with only an order-by clause.

The generation of these is reused from the previous implementation with some minor adjustments.

The parameters used are, first of all, number of attributes A and number of relations R. A completely disjoint schema is generated with the most evenly distributed attributes. For example, for 4 relation and 10 attributes a schema with relations with 3,3,2,2 attributes, respectively, is generated. Another parameter, number of equalities in the schema K describes the attributes the relations will be joined on in the query. In particular, it chooses K pair of attributes and changes the schema so that they have the same name. I have enforced, which was not in previous work, that as pairs of attributes we want attributes from different relations only, and, moreover, in process of choosing the equalities, i.e. pairs, if two attributes were already transitively equated, they will not be equated again. This reduces the noise in the results to some extent. Based on this, a new schema is produced and an optimal f-tree for this schema, with respect to joins, computed.

Further, parameter L is number of equalities to include in the query. This experiment uses $L = 0$. And the parameter M is number of order-by attributes to include in the clause. Here I enforce that latter attributes are not equated transitively with any of the previous one. (In case if $L > 0$.) To have attributes like this would be pointless as explained in chapter on order-by.

I generate these experiment cases for $A = 10$, $R = 4$, $K = [1..9]$ and $M = [1..9]$ with $K + M \leq A$. The last condition is due to only $A - K$ different attributes being present in the schema, for a given K. And, for each valid pair (K, M) I do $T = 500$ test, over which I average.

6.1.2 Setup

I am using an instance on Amazon EC2. I am using a Large instance, from the Standard instance category, with dual core Intel Xeon CPU E5507 @ 2.27GHz[1] and 7.5GB of memory. The test were run only on one core so that the timing is more accurate.

The memory consumption of these experiments is considerably lower, about a few hundred megabytes, compared to experiments in previous work [2], because search spaces are much smaller due to not considering equalities, which require the merge operation, in addition to swap operation.

6.1.3 Results

6.1.3.1 Plan Cost

The plan cost is the maximal $s(\mathcal{T})$ over f-trees on the path. The second criterion in the length, but that doesn't affect these experiments, because we evaluate only the $s(\mathcal{T})$ and the search space is not affected. The second statistic, which is recorded for each of the

[1]as stated by the /proc/cpuinfo

searches considered, is the best, i.e. minimal, $s(\mathcal{T})$, of the final f-tree of paths leading to a goal.

We can see the result in Figure 6.1. Each set of columns represents average of 500 tests.

First of all, we see that the greedy-heuristic search produces results very close to the full search. Recall that this s parameter is an exponent in f-representation size bound, so the smaller the better.

Full search produces consistently better results, as expected. But often very close, or even equal, to the heuristic search. In the figure, there are 17 pairs (K, M) where the full search found better path on average, i.e. with asymptotically smaller intermediate f-representation sizes. And out of all the 22,500 tests in total, in 66.47% test the path found by the two search methods was the same.

So where does the greedy-heuristic go wrong? Why does it sometimes perform worse?

Example 6.1. The greedy-heuristic search focuses its attention locally and only to attributes of the order-by clause. However, the size bound and the $s(\mathcal{T})$ are based and depend on entire root-to-leaf paths. So a natural question to ask is, would changes to other, more distant possibly, parts of the f-tree help? And this seems to be the case as illustrated by two different paths in Figure 6.2. First is the one returned by the full search and the second by the greedy heuristic. The swap of attributes E, F helps. So there is a global interaction, not just local. □

We can also see a trend of increasing path cost with increasing number of order-by attributes M, when K is considered constant. This is due to the order-by attributes partially forcing the shape of the f-tree, and so better factorisations can not be achieved. However, as we saw before, this is necessary for the constant-delay enumeration with constant precomputation time.

6.1.3.2 Search Time and Search Space Size

This section presents both these statistics first as a series representing each M as K varies (for some of the M's only that have enough data points) and then for series representing each of K as M varies.

We can see the plan finding time in Figures 6.3 and 6.4 and the search space explored size in Figures 6.5 and 6.6. (Note the log scale.) We can see that the graphs behave very similarly, which is due to time needed being closely related to the number of f-trees explored.

With $K = 1$ the relations are mostly separate trees in the f-tree so the space is smaller compared to when they interact more for $K = 3$. However then, larger K means less different

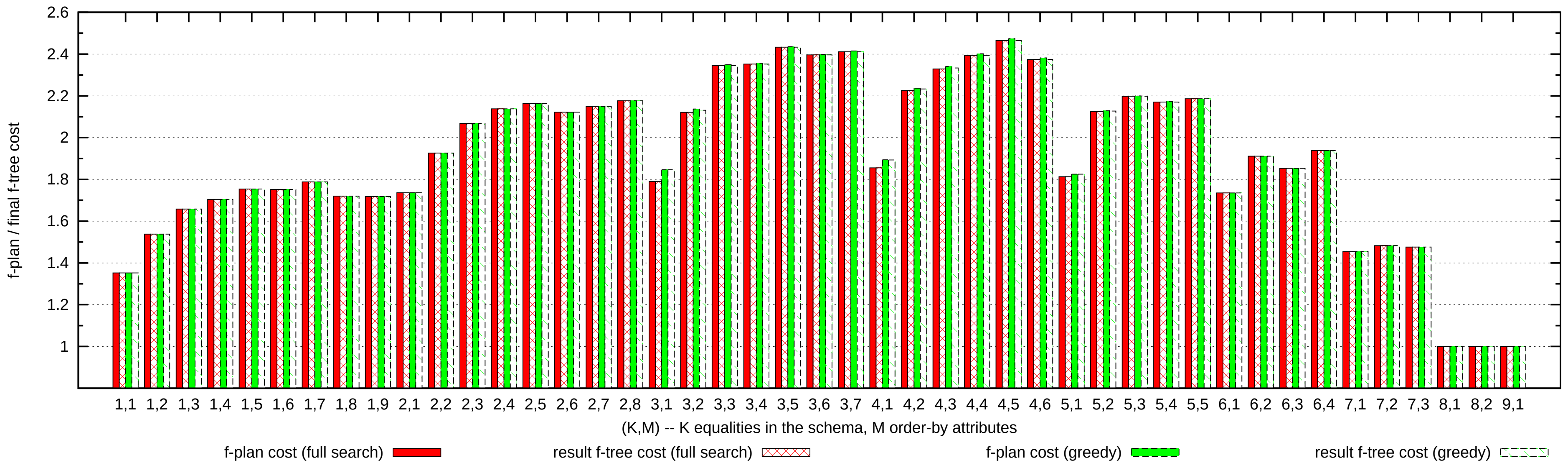


Figure 6.1: Average maximal $s(\mathcal{T})$ and average final $s(\mathcal{T})$ on a path in optimisation using the full search and the greedy-heuristic search for 10 attributes and 4 relations.

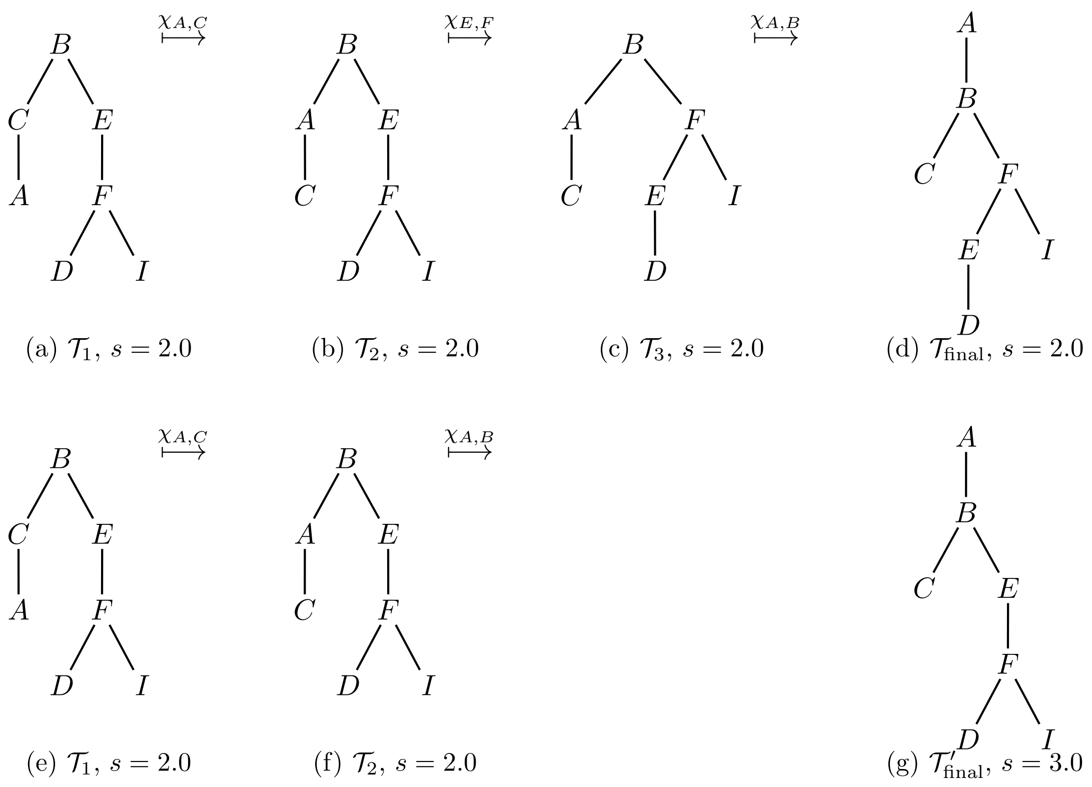

$\xrightarrow{\chi_{A,C}}$
$\xrightarrow{\chi_{E,F}}$
$\xrightarrow{\chi_{A,B}}$
(a) $\mathcal{T}_1$, $s = 2.0$
(b) $\mathcal{T}_2$, $s = 2.0$
(c) $\mathcal{T}_3$, $s = 2.0$
(d) $\mathcal{T}_{\text{final}}$, $s = 2.0$
$\xrightarrow{\chi_{A,C}}$
$\xrightarrow{\chi_{A,B}}$
(e) $\mathcal{T}_1$, $s = 2.0$
(f) $\mathcal{T}_2$, $s = 2.0$
(g) $\mathcal{T}'_{\text{final}}$, $s = 3.0$

Figure 6.2: Example of a case where the full search (a-d) finds a better path than the greedy-heuristic search (e-g). The path A, B, E, F, I in $\mathcal{T}'_{\text{final}}$ needs 3 relations to cover the path, hence the $s = 3.0$. (Database schema is $\{\{A, B, C\}, \{D, E, F\}, \{E, B\}, \{I, F\}\}$.)

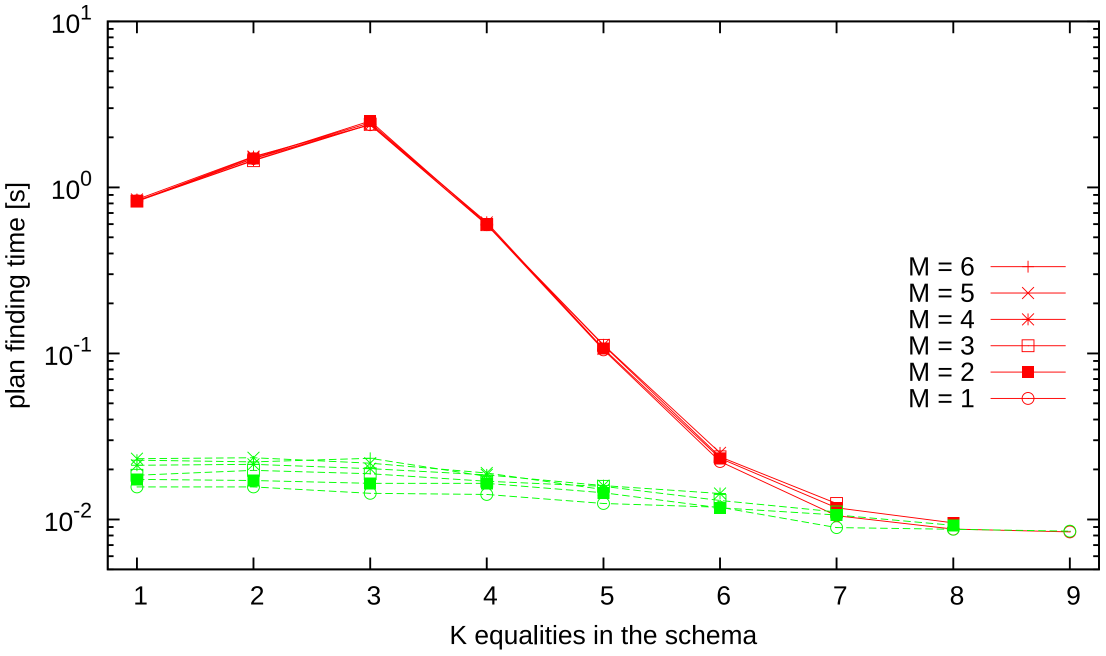


Figure 6.3: Average plan finding time, as K varies, for the full search (top set of red series) and the greedy-heuristic search (bottom set of green series) for 10 attributes and 4 relations.

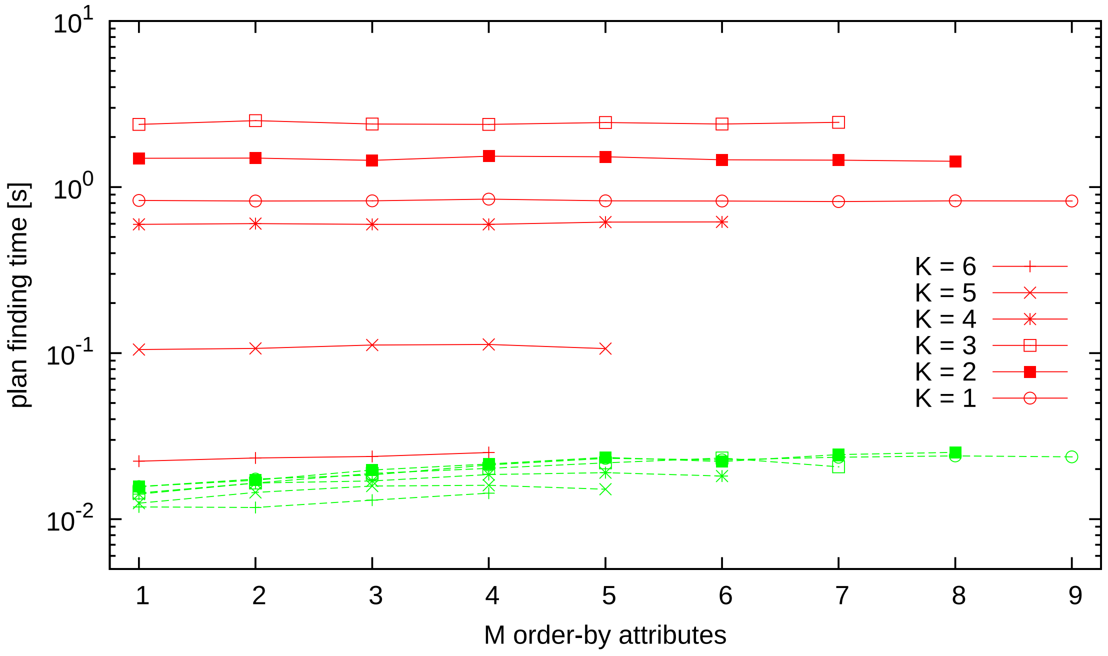


Figure 6.4: Average plan finding time, as M varies, for the full search (top set of red series) and the greedy-heuristic search (bottom set of green series) for 10 attributes and 4 relations.

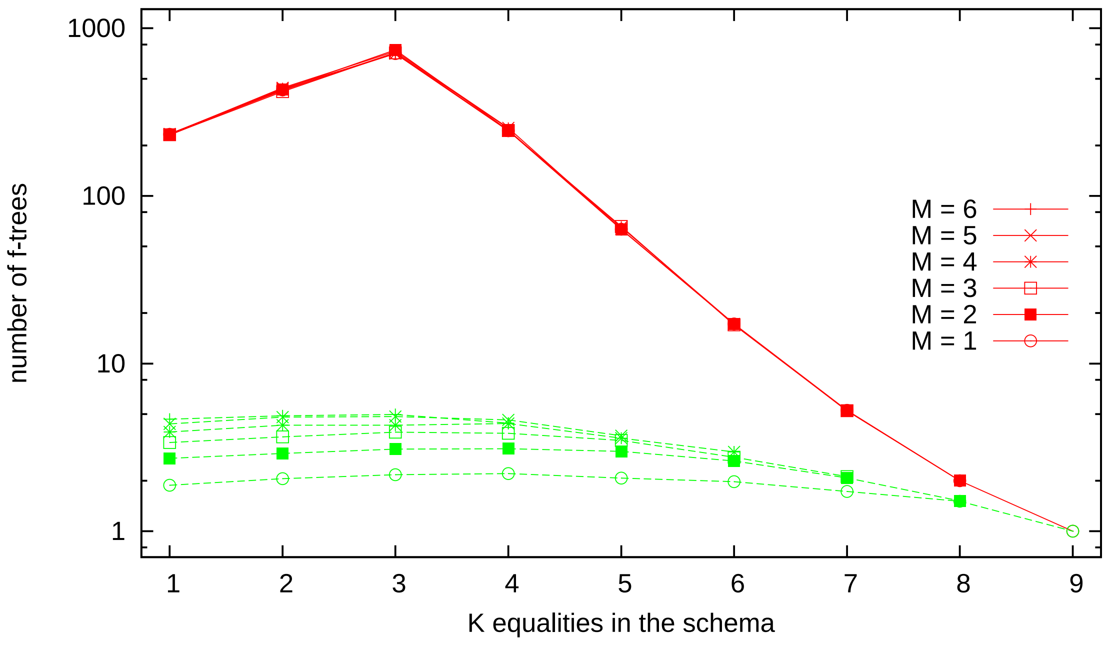


Figure 6.5: Average search space size, as K varies, for the full search (top set of red series) and the greedy-heuristic search (bottom set of green series) for 10 attributes and 4 relations.

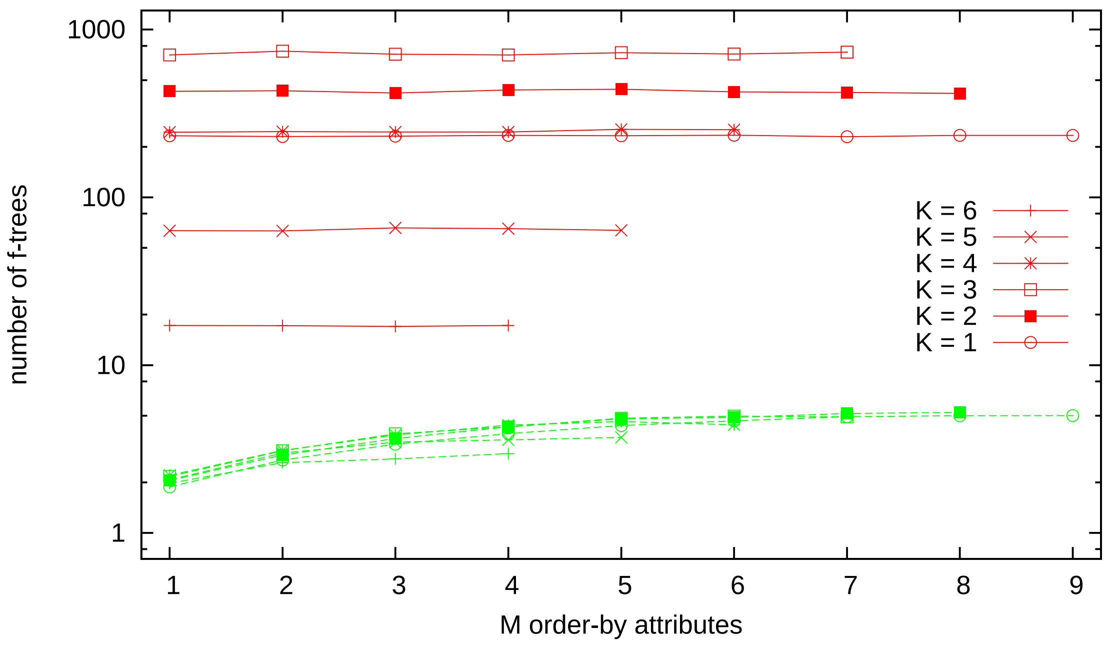


Figure 6.6: Average search space size, as M varies, for the full search (top set of red series) and the greedy-heuristic search (bottom set of green series) for 10 attributes and 4 relations.

attributes in the schema and the search space goes down exponentially as expected, with the number of attributes.

The time and search space size for the greedy-heuristic search is considerably smaller, this is due to it being dependent only on the tree depth and number of order-by attributes. With higher K both of these go down, so the time and space decrease.

6.1.4 Conclusion

It seems that the greedy-heuristic is very effective. It produces path costs very close to the full search and runs exponentially faster.

6.2 Experiment Two: Query Evaluation

This experiment compares the evaluation of the query using evaluation plan found in the previous experiment from both the searches, in turn, with execution of the query with a flat relation database engine.

6.2.1 Input Data and Parameters

Input data are the same as for the previous experiment, together with random relations generated based on two parameters: a parameter C that specified number of tuples to generate for each relation, in particular, to generate $C^{\text{relation arity}}$ tuples; and a parameter B

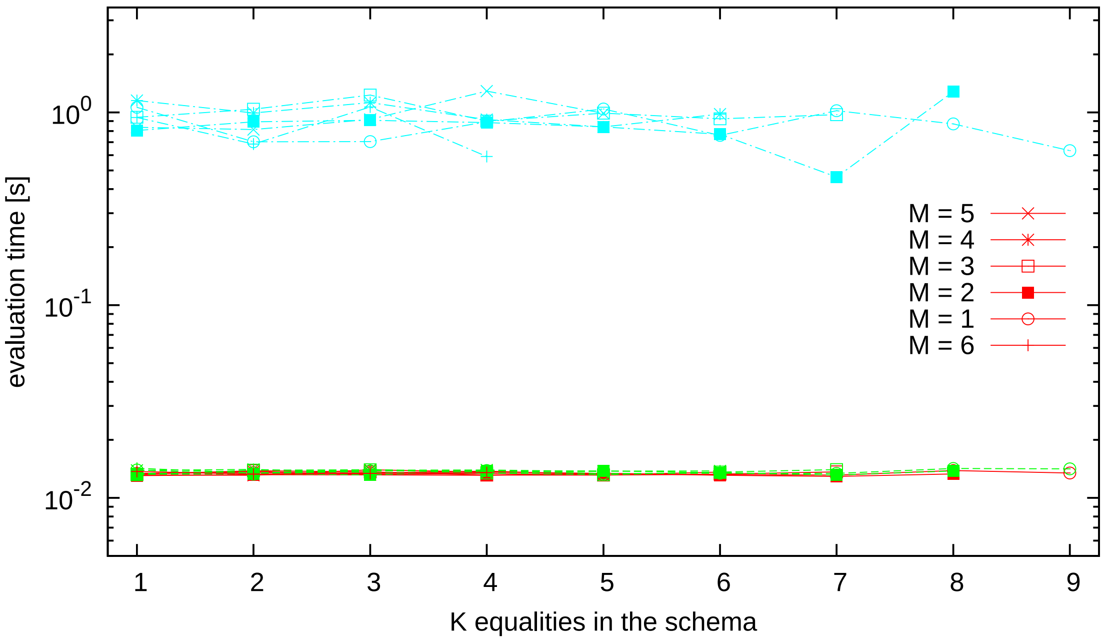


Figure 6.7: Average query evaluation time, as K varies, needed to execute the plan found by the full-search and the greedy-heuristic search (bottom set of red and green series), and the average time to evaluate the same query with flat-representation engine (top set of blue series) for 10 attributes and 4 relations.

that gives a maximal value for values in the tuple, in particular, we generate integers from range $[0..B]$ uniformly at random.

I use $C = 7$ and $B = 100$.

6.2.2 Setup

The setup for this experiment is the same as in the previous experiment.

6.2.3 Results

From Figures 6.7 and 6.8, which are as K, M varies, respectively, we can clearly see that to evaluate the query with a flat representation takes much longer that evaluating with factorised representation.

The evaluation of the plans found by the full search and the greedy-heuristic search takes very similar time. We would expect that the execution of the plan found by the full search to be faster, since the path found by it is sometimes better and has the intermediate representations smaller. However, we have too small relations to see this, I believe. We will likely explore this in the future.

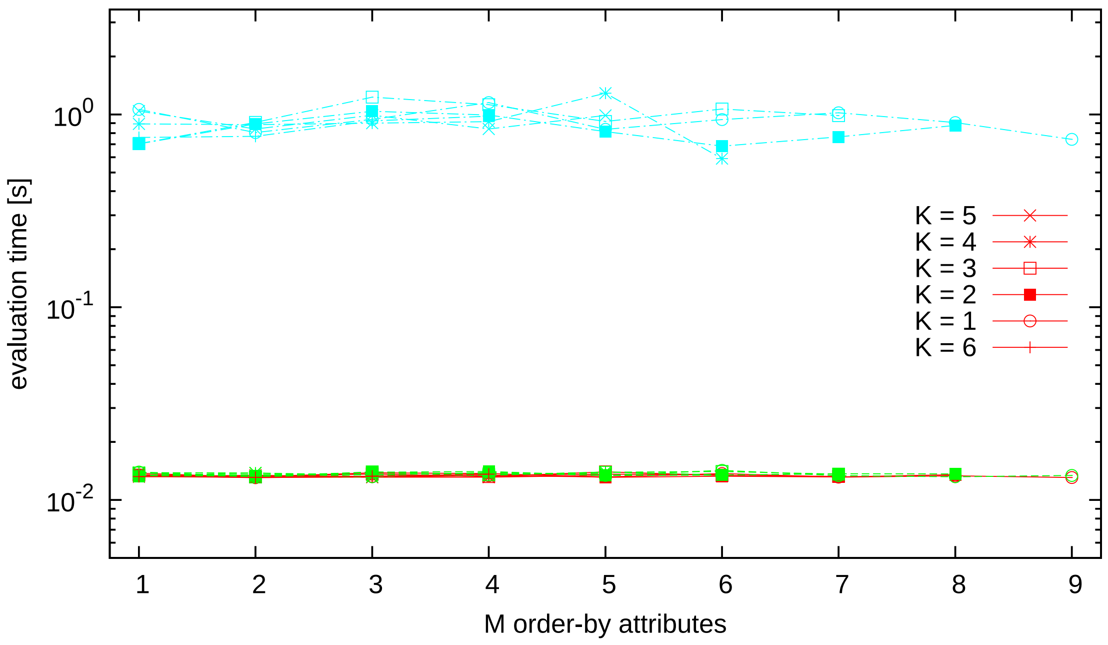


Figure 6.8: Average query evaluation time, as M varies, needed to execute the plan found by the full-search and the greedy-heuristic search (bottom set of red and green series), and the average time to evaluate the same query with flat-representation engine (top set of blue series) for 10 attributes and 4 relations.

Chapter 7

Conclusions

This thesis extended optimisation and evaluation of queries on factorised relational databases for queries with an order-by clause, and a group-by clause with an aggregate.

The major contribution that allow us to optimise and evaluate queries with order-by are characterisation of factorised trees that allow enumeration with constant delay and constant precomputation, and the greedy optimisation heuristic for search in the space of the factorised trees that were characterised. I have implemented these optimisations and the evaluation and evaluated the heuristic experimentally. I have found that the heuristic performed surprisingly well. Contribution to optimisation of queries with group-by is the characterisation of result factorised trees for queries with an aggregate.

This thesis proposed a step-by-step evaluation of queries with aggregates, advantageous because it can reuse previous optimisation techniques, with further operators on factorised trees, which will be part of a future paper in more detail.

I was hoping to find a better than the presented heuristic for optimisation of queries with order-by, that would consider more of the structure of the f-tree, but I was unsuccessful. And it would have been good for completeness to finish the ideas of step-by-step evaluation of aggregates, formalise the new operations on f-trees, and explore the plan finding and evaluation time, but there was insufficient time.

Finally, the experiments we saw suggest that the factorised representations are a promising idea, because of much faster evaluation times on larger data.

Future research can explore the following:

- Heuristics for optimisation for queries with order-by that consider more properties and structure of the f-tree (similar to the distance-measure heuristic attempt) and at same time need to search less than the full search space.
- Explore the optimisation and evaluation of queries with group-by further, by considering the proposed step-by-step evaluation and also a one step evaluation that would

rebuild the f-tree. The former requires the new operations on an f-tree, for which a heuristic for optimisation is needed, because the search space is exponential and it is likely a hard problem to find the best path in polynomial time.

Appendix A

Proofs

Proposition 3.1

Proof of Proposition 3.1. First of all, by previous results we know, that this enumeration produces all tuples of the f-representation.

If the result has zero or one tuple, the proposition is trivially true.

The following shows that the enumeration is ordered by the entire sequence of attributes of the priority list.

Consider two consecutive tuples and let the attribute A be such attribute that it is the first attribute in the priority list on which the two tuples differ. We output values of each attribute in increasing order, or we output a possibly smaller value only if we advance the attribute before A in the priority list (and never for the first attribute). But by the definition of A, the latter case can't be true. Hence, these tuples are ordered by the attributes of the priority list.

Each two consecutive tuples are ordered in increasing order as defined by the priority list, so all must be in the correct order by transitivity. And also, they are ordered by any prefix of the priority list. This concludes the proof. □

Lemma 3.5

Proof of Lemma 3.5. Let A be the first attribute of U, let the f-representation represent a relation R and denote the f-representation by $\mathcal{T}(R)$. Assume for contradiction that A is not at a root of $\mathcal{T}$. Then there are m lists of values of attribute A in the parse tree of $\mathcal{T}(R)$, for some m, all of which are assumed to be ordered in increasing order (Remark 2.1).

In the bad (possible even the worst) case, there are as many of these list as there are unions in the f-representation $\mathcal{T}(R)$, of which there are in the bad case $\Theta(|\mathcal{T}(R)|)$, where $|\mathcal{T}(R)|$ denotes the sizeof the f-representation, and the bad case being, for example, an f-tree with root B and one child A. So $m = \Theta(|\mathcal{T}(R)|)$, which is more that $O(|\mathcal{S}|)$.

Alternatively, a more precise count of the m lists would be to consider all valuations of ancestors of A, of which there can in general be more than $O(|\mathcal{S}|)$. □

Lemma 3.6

Proof of Lemma 3.6. This is a constructive proof. We can partition attributes of the database schema based on the dependency sets[1] as follows. Two attributes are in the same class if they are in the same dependency set, or there is an attribute that is in a dependency set with the first and also the second attribute, or the previous applied transitively. If attributes depend on each other in this way, then they need to be in the same subtree to satisfy the path condition[2].

Each of these classes represents set of nodes of subtrees, and in this first case of trees of the forest f-tree. For each we have a choice of a root from all the attributes in the class. However, some of the choices are forced by the sequence U. Then we would remove this attribute from the class, and repartition the attributes of the said class using dependency sets without the removed root attribute. And again, we got sets of nodes that represent nodes of the subtrees.

We do this recursively until an f-tree is chosen. The resulting f-tree satisfies the path condition, and no other do by construction, and has the attributes of U at the top as required for constant-delay enumeration. And the choice for position of the attributes of U was always forced, so the tree structure they are in is the same for all f-trees for the given database schema and sequence U.

Note that this method can be used to enumerate all of these f-trees. □

Theorem 5.1

Proof of Theorem 5.1. The theorem follows from our definition of access a group of tuples by being possible to enumerate them in an order where the group tuples are consecutive, and The Characterisation Theorem 3.3. To apply the theorem we want to order-by any sequence of attributes of G. Note that we have more room for optimisation since we do not need a precise sequence. □

Theorem 5.2

Proof of Theorem 5.2. To see that the result of the query is $\pi_{G\cup\{C\}}(R_C \times R_1 \times \cdots \times R_n)$ we need that the relation R_C contains the value of the aggregate for all the groups in

[1]here it is the set of sets of attributes of each relation

[2]Recall: the path condition states that each dependency set needs to lie on some root-to-leaf path, and this is needed so that the f-tree and its f-representation can represent any data in the database.

$R_1 \times \cdots \times R_n$, i.e. for all valuations of G. This is easily true by definition of R_C and $\mathcal{S}$ a and easy properties of relational algebra.

Case min/max: To compute the value of this aggregate over an attribute A we only need the values of this attribute. We further need to know which groups the values of A belong to. So we need some attributes from G. No other attributes are necessary for $\mathcal{S}$. The claim is that these attributes are $\mathcal{S}' := \mathcal{M}_{\mathbf{S},\{A\},G}$.

Assume that one of such attributes, say $H' \in \mathcal{S}'$ is missing from $\mathcal{S}$, then we want to show that, for some database, there is a valuation h' of $\mathcal{S}' - \{H'\}$ for which not all extensions[3] to a valuation h of G have the same multiset of values of A in $\pi_{\mathcal{S}'}(R_1 \times \cdots \times R_n)$ for h' and in $R_1 \times \cdots \times R_n$ for h. To see this take any sequence $\mathcal{S}_1, \ldots, \mathcal{S}_j$, as defined in definition of $\mathcal{M}_{\mathbf{S},\{A\},G}$, for $H' \in \mathcal{S}_j$. Precisely this sequence of joins through the $G^{\complement}$ attributes allows for the possibility that the multisets of groups merge after the projection to a different multiset of values A.

Now I show that no more than the $\mathcal{S}'$ attributes are needed for $\mathcal{S}$, i.e. that $\mathcal{S}'$ is sufficient, i.e. $\mathcal{S} \subseteq \mathcal{S}'$. Take any valuation h of G and $h' := h|_{\mathcal{S}'}$. We want to show that the multisets are equal. The only way that the multisets can be unequal is if the multiset of h' in R is larger, i.e. some multisets merged. This is easy to see from how the projection in R works. So if the two multisets were unequal, then there is attribute in G, similarly to the above paragraph, that splits the multiset, and going back can merge it. The crucial part for the attribute to be able to merge groups of tuples was that it has that sequence of joins through non-G attributes, but there is no more of such attributes. So $\mathcal{S} \subseteq \mathcal{S}'$.

This can also be seen intuitively, as in Example 5.1, joining on relations with attributes only from G just clones the multisets of values of A. There are also other sets to consider, those that have attributes in $G^{\complement}$ but were not in sequences of relations in definition of $\mathcal{S}'$. Those are projected away in the query by the group-by clause and for aggregating by min/max can be projected away beforehand, and then this intuition is easier to see.

Case sum, avg, count: For these aggregates the argument is exactly the same, only the set of attributes they need for their evaluation is the whole of $G^{\complement}$, instead of just A. $\square$

[3]we can have extensions to multiple valuations of G, but all need to have this same multiset of values, otherwise the aggregate values might not be the same

Appendix B

Computation of Aggregates

Given an f-tree $\mathcal{T}$ grouped by a set G and an f-representation over $\mathcal{T}$ we can compute aggregates on it. The most common aggregates are: min, max, sum, avg, count. We will consider them to be over one attribute. Count(*) is simply Count(A) where A is any non-group-by attribute.

For a valuation of G, we could just compute the aggregate from the listing of tuples. We can do better if we use that we have a factorised representation. This is often by counting the number of tuples represented by subtrees in the parse tree and combining it with values of an attribute. All the cases are fairly straightforward. Just as an example I will present one of them.

Given an f-representation over an f-tree $\mathcal{T}$ grouped by a set G. Let V be the set of trees below G and let A be an attribute that is as a root of $\mathcal{T}_A \in V$. In other words, an attribute outside G that is a child of a node labelled with attribute of G. Consider summing over all values of A (not just distinct). For each valuation of G the aggregate has the value

$$\sum_{a \in A} \left(a \cdot size(a) \cdot \prod_{\mathcal{T}' \in V - \{\mathcal{T}_A\}} size(\mathcal{T}') \right)$$

where $size(a)$ denotes the number of tuples of a subtree in the parse tree of the f-representation rooted at the value a, and $size(\mathcal{T}')$ is number of tuples in the f-representation corresponding to that part of the f-tree $\mathcal{T}$. This can be computed in linear time complexity of the size of the f-representation.

Other aggregates are similar. If we use the keyword DISTINCT, the complexity would have an additional logarithmic factor. We can bring the corresponding attribute node below G, in same way as if we were sorting, and after that apply the aggregate. This will not hurt the complexity, by results about order-by.

Bibliography

[1] Tpc-h benchmark specification. http://www.tpc.org/tpch/.

[2] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. Fdb: A query engine for factorised relational databases. *VLDB*, 2012.

[3] Benny Kimelfeld and Yehoshua Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In *Proceedings of the 6th international conference on Next Generation Information Technologies and Systems*, NGITS'06, pages 141–152, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] Dan Olteanu and Jakub Zavodny. Factorised representations of query results: Size bounds and readability. *ICDT*, 2012.