Storage layer for factorized databases

Antonio Lombardo

New College

University of Oxford

A thesis submitted for the degree of

MSc in Computer Science

Trinity 2016

Acknowledgements

plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle.

Abstract

The representation for factorized data implemented in the state-of-art **FDB** is too heavy and not cache friendly [2] [14].

In this work, we propose a succinct representation for factorized data by improving upon the FDB representation in three ways.

First, we consider d-representations, factorized representations with definitions that can be exponentially more succinct than the simple factorized representations supported by the state-of-art. Our representation leverages caching of fragments of data to avoid useless repetition.

Second, the representation is implemented by means of a contiguous sequence of bytes, which contrasts with the heavy tree implementations used previously. Our representation reports savings in number of bytes of different orders of magnitude compared to the state-of-art.

Third, it exploits more efficiently modern CPUs by being more cache friendly than the previous representation for typical operations like like common aggregates such as counting and summation and intersection of ordered list of values. For instance, JOIN processing can be efficiently supported via a sequence of such list intersections. Our representation, F-MEM, clusters values for the same attribute within compacted (and compressed for integer values) arrays of sorted values, thereby enabling caching of entire sequences of such values and efficient list intersection.

An algorithm for JOIN queries, taking as input flat relations, that returns a F-MEM representation of the result is proposed. The algorithm is a variant of the Leapfrog Triejoin algorithm extended to support partial variable orders and to leverage caching whenever possible.

We investigate the benefits of F-MEM for query processing. In that regard, we design query processing algorithms over F-MEM representations for COUNT queries and a restricted class of GROUP BY aggregate queries. Finally, we demonstrate that COUNT queries over F-MEM representations outperform FDB by orders of magnitude.

Contents

Chapter 1

Introduction

Nowadays, relational databases are ubiquitous and constitute the principal ingredient behind the core business of many companies. At the heart of relational databases is the relational data model, a well studied mode in both academic and industrial settings. The relational data model entails a high degree of redundancy, hence motivating the design of compression schemes for relational data. Different solutions to decrease the redundancy brought by the relational data model have been proposed in the recent years. Standing out from the crowd, are factorized representations, a class of succinct representations for relational data [10, 2, 11, 8].

In this work, we present a new succinct representation for factorized data. The state-of-art database engine for factorized data, **FDB**, implements factorized representations. FDB has been shown to outperform off-the-shelf relational databases like PostgreSQL and SQLite [2]. However, the data structures bundled in FDB, for factorized representations, are heavy and do not exploit well modern CPU caches. This motivates the need for the design of a new compact and cache friendly representation. Factorized representations can be exponentially more succinct compared to flat representations. Take for instance the example database in Figure 1. The $Players \bowtie Teams$ relation can be expressed with the following flat expression:

$$\langle p_1 : P\rangle \times \langle t_1 : T\rangle \times \langle c_1 : C\rangle \cup \langle p_2 : P\rangle \times \langle t_1 : T\rangle \times \langle c_1 : C\rangle \cup \langle p_3 : P\rangle \times \langle t_1 : T\rangle \times \langle c_1 : C\rangle$$

Players

Player	Team
p_1	t_1
p_2	t_1
p_3	t_1

Teams

Team	Country
t_1	c_1

Players $\bowtie$ Teams

Player	Team	Country
p_1	t_1	c_1
p_2	t_1	c_1
p_3	t_1	c_1

Figure 1: An example database of 2 tables: Players, Teams. The rightmost table is the result of the JOIN of Players and Teams.

However, by exploiting distributivity of the cartesian product over union, we can obtain the following factorized representation:

$$(\langle p_1 : P\rangle \cup \langle p_2 : P\rangle \cup \langle p_3 : P\rangle) \times \langle t_1 : T\rangle \times \langle c_1 : C\rangle.$$

In this work, we improve upon the representation for factorized representations present in FDB in different ways:

- We target d-representations, which are factorized representations with definitions that can be exponentially more succinct than the simple factorized representations. Targeting d-representations allows us to cache fragments of data and hence, enables to minimize the overall redundancy in the representation.
- Factorized representations are represented through a sequence of bytes instead of the heavy tree representation previously used.
- Our proposed representation is more cache friendly than the previous representation for typical operation such as sum of values or intersection of ordered list of values. JOIN processing on factorized data can be efficiently supported with such list intersection.

We propose a new succinct representation for factorized data called **F-MEM**. F-MEM clusters values of the same attribute within an array of compacted data. If integer data is supplied they are also compressed. Therefore F-MEM representations enable caching of entire sequences and efficient list intersections, which can be exploited for competitive query processing. As a consequence of the heritage from factorized representations, compared to classical compression algorithms like GZIP, LZW, etc., in order to process data we do not have the overhead of a decompression task.

In this work, we also propose an algorithm for JOIN queries that, taken as input flat relations, returns the result in form of F-MEM representations. The algorithms for JOIN queries is a variant of the Leapfrog Triejoin, which is known to be worst-case optimal, extended to support partial orders and to leverage caching whenever possible.

As final result of our work we build F-MEM, and we show that it outperforms FDB in space, and in time when we can leverage caching of fragments of data for JOIN queries.

This work makes the following key contributions:

- We design a new succinct representation for factorized data called F-MEM.
- We design algorithms for building F-MEM representations out of single flat relations.
- We design an algorithm for JOIN queries, that taken as input flat relations, returns the result in form of a F-MEM representation. The algorithm is a variant of the Leapfrog Triejoin algorithm extended to support partial variable orders and to leverage caching whenever possible.
- By leveraging caching whenever possible for JOIN queries we show that F-MEM outperforms FDB.
- We show that F-MEM outperforms FDB in space by several orders of magnitude.
- We design and implement a query processing algorithm over F-MEM representation for COUNT queries.
- We design a query processing algorithm for a restricted class of aggregate queries over grouping of attributes.
- In the context of aggregate queries, we investigate the cache friendliness brought by F-MEM representations in terms of the total number of memory transfers during the computation of aggregates.
- We investigate the performance in operating on F-MEM representations stored in the disk. To not confuse the reader we will denote F-MEM over the disk as **F-DISK**.

1.1 Outline

In this chapter, we introduced F-MEM and F-DISK representations.

Chapter 2 provides an introduction to factorized representations by referring to the current literature.

Chapter 3 describes the limitations of the data-structure for factorized representations implemented in state-of-art factorized database engine **FDB**. We then design and propose F-MEM representations, a more succinct representation for factorized data. We also design an algorithm that, given as input a flat relation, returns the corresponding F-MEM representation.

Chapter 4 outlines methods to build factorized representations in the form of F-MEM representations of the result of JOIN queries. This is motivated by the fact it is a very common use case which needs to be accommodated.

Chapter 5 gives insights about the design of query processing algorithms over F-MEM representations. We provide compelling arguments in support of the claim that query processing is speeded up in the context of F-MEM representations because of its succinctness and inherent cache friendliness. We also back up the argument that F-MEM representations provide a sound framework for the design of query processing algorithms.

Chapter 6 outlines the experimental evaluation of F-MEM and F-DISK against FDB on a range of d-trees and datasets. We show how leveraging d-trees (and hence caching whenever possible) can lead to better performance.

Chapter 7 positions our work in the realm of compression with a different flavour, as we exploit structural properties of queries. A short review of the current literature on the matter is given.

Chapter 8 concludes our dissertation. We briefly describe the results obtained in this dissertation and we also discuss potential directions for future work.

Chapter 2

Factorized Representations: A Primer

This chapter draws the connection between the theory behind the algorithms and the implementation of **F-MEM** while giving historical background of the old, **FDB**, implementation. It is meant as a medium to make this work self-contained, so we just outline the relevant details. The reader, interested in a more rigorous treatment about factorized representations, should look at [11].

Factorized representations are a class of succinct representations for relational data exploiting the algebraic distributive property of the cartesian product over union for reducing data redundancy. Take for instance the following relation over the schema A, B:

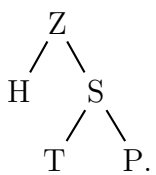
$$R = \langle A : 1\rangle \times \langle B : 2\rangle \cup \langle A : 1\rangle \times \langle B : 3\rangle \cup \langle A : 1\rangle \times \langle B : 4\rangle \cup \langle A : 1\rangle \times \langle B : 5\rangle \quad (1)$$

The above can be more succinctly represented as:

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

This intuition contributed to the development of the *f-trees* and *f-representations*, concepts which we shall explore in detail. An f-tree is simply a nesting structure like the one given below, describing an arrangement of attributes of relational data. It can be thought of a partial order on the attributes:

(3)



An f-representation is an algebraic expression consisting of unions ($\cup$), cartesian

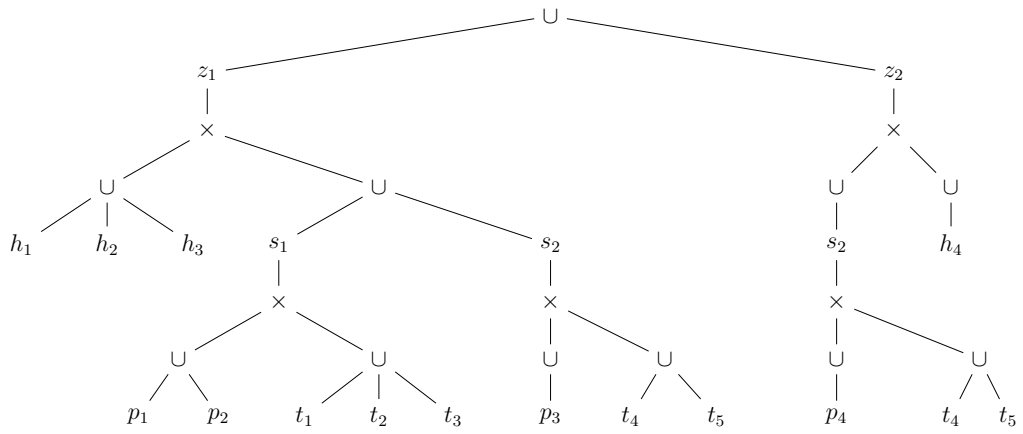
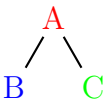


Figure 2: Parse-tree of an example f-representation with uniform nesting structure given by the f-tree (3).

products ($\times$) and singleton expressions ($\langle A : 1\rangle$). The expressions (1, 2) are f-representations. An f-representation can be represented by means of a parse-tree like the one in Figure 2. The size $|E|$, of an f-representation E, is denoted by the total number of singletons present in the f-representation. We are interested in the class of f-representations with uniform nesting structure backed by a given f-tree [11].

Let's start by developing the necessary intuition:



(4)

An f-representation with the above nesting procedure can be succinctly denoted with:

$$\bigcup_{a\in A}\langle A : a\rangle \times \left(\left(\bigcup_{b\in B}\langle B : b\rangle\right) \times \left(\bigcup_{c\in C}\langle C : c\rangle\right)\right) \tag{5}$$

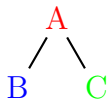
A sample instance of an f-representation with the above nesting procedure is shown below:

$$\begin{aligned} R = \langle A : 1\rangle \times (\langle B : 1\rangle \cup \langle B : 2\rangle) \times (\langle C : 3\rangle) \cup \\ \langle A : 2\rangle \times (\langle B : 3\rangle \cup \langle B : 4\rangle) \times (\langle C : 5\rangle \cup \langle C : 6\rangle) \end{aligned} \tag{6}$$

With the above f-tree, we are saying that the attributes B and C are conditionally independent of A. This is a powerful concept as it means that, if we are given an f-representation with such a nesting procedure then, given an A-singleton, we can build a flat tuple $\langle$A,B,C$\rangle$ by picking any B-singleton and C-singleton dependent on that A-singleton.

F-trees, in a nutshell, provide a mean to factorize a relation or a query result by making explicit the dependencies of the attributes: an attribute A is dependent on the set of ancestors $anc(A)$. In the above f-tree (4) $anc(B) = \{A\}$, $anc(C) = \{A\}$ and $anc(A) = \emptyset$.

A given f-tree cannot factorize every relation exhibiting the same schema. Let's be more specific with an example of a relation not factorizable with a given f-tree despite exhibiting the same schema:



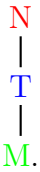
$\neq$

A	B	C
1	2	1
1	2	2
1	3	1

In the above example, it is false that B and C are unconditionally independent of A. The B-singleton 3 is paired just with the C-singleton 1. For B to be independent from C, we should be able to use any C-singleton dependent on the same A-singleton. This is not the case as we cannot pair the B-singleton 3 with the C-singleton 2. Hence it cannot be that B and C are conditionally independent.

It is now time to introduce the notion of *path contraint* that states that, in an f-tree, the attributes belonging to the same relation must lie on the same root-to-leaf path. The above example would have been indeed factorizable with the f-tree:

Now let's take as example the conjunctive query `Player(N, T), Player'(T, M)` with the result of the query factorized by the following f-tree:



(7)

Suppose that the result of the query leads to the following f-representation:

$$\begin{aligned} R = &\langle N : Messi\rangle \times \langle T : Barcellona\rangle \times (\langle M : Neymar\rangle \cup \langle M : Iniesta\rangle \cup \langle M : Messi\rangle) \cup \\ &\langle N : Neymar\rangle \times \langle T : Barcellona\rangle \times (\langle M : Neymar\rangle \cup \langle M : Iniesta\rangle \cup \langle M : Messi\rangle) \cup \\ &\langle N : Iniesta\rangle \times \langle T : Barcellona\rangle \times (\langle M : Neymar\rangle \cup \langle M : Iniesta\rangle \cup \langle M : Messi\rangle) \end{aligned} \tag{8}$$

Imagine that you are standing in front of your little brother explaining f-representations and how cool they are when, suddenly, he comes up with a question: *Why are we repeating the members of the same team so many times, could not we do that once as the*

members depend only on the team? Our little brother is right, we could have avoided that repetition by caching the recurring sub-expression $X = \langle M : Neymar\rangle \cup \langle M : Iniesta\rangle \cup \langle M : Messi\rangle$) ending up with:

$$\begin{aligned} R = \langle N : Messi\rangle \times \langle T : Barcellona\rangle \times X \cup \\ \langle N : Neymar\rangle \times \langle T : Barcellona\rangle \times X \cup \\ \langle N : Iniesta\rangle \times \langle T : Barcellona\rangle \times X. \end{aligned} \tag{9}$$

We now introduce *d-representations*, an extension of f-representations. A d-representation is an algebraic expression consisting of unions ($\cup$), cartesian products ($\times$), singleton expressions ($\langle A : 1\rangle$) and named symbolic references (i.e., X in the previous example) to refer to subexpressions. The expression (9) is a d-representation. Analogously to f-representations, d-representations can be represented by means of a parse-tree like the one in Figure 2, in which a subexpression may be pointed by multiple edges. The size $|E|$, of an d-representation E, is denoted by the number of singletons, unions, products and instances of named symbolic references [11]. D-representations, contrary to f-representations, allow for caching of recurring sub-expressions, as denoted by the dashed lines in Figure 2.

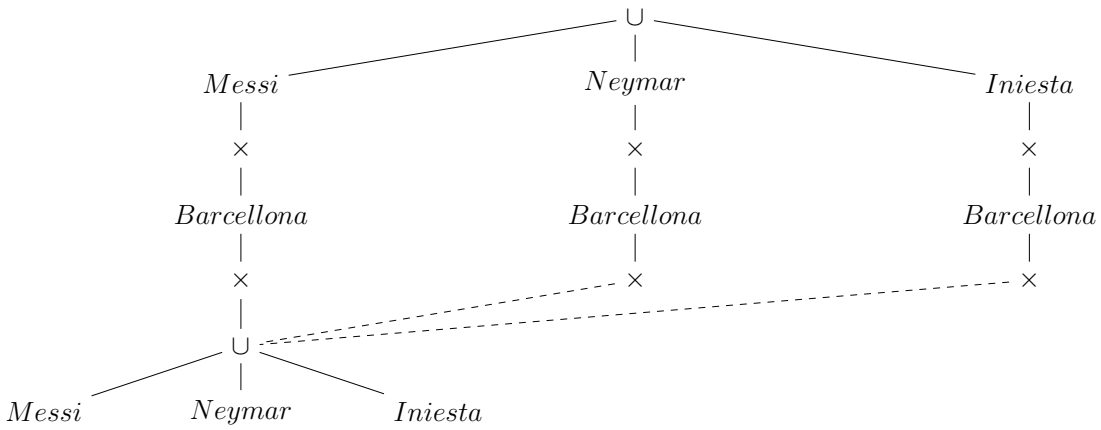


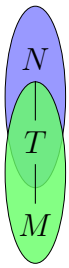
Figure 3: Parse-tree of the factorized representation with caching (9) and its uniform nesting structure given by the f-tree (7), dashed edges represent a pointer to the cached sub-expression.

We shall now proceed in understanding why we could cache the M attribute of the above conjunctive query `Player(N, T), Player'(T, M)` with respect to the factorization tree (7). In the f-tree (7) we stated that the attribute M was dependent on its ancestors $anc(M) = \{N, T\}$ but we can make an intriguing observation: N and M belong to different relations so we can restrict M to be dependent only on $\{T\}$.

Moreover, since it depends only on a *strict subset* of ancestors, it can be cached. The proof is simple. If a given attribute A depends on the set of the ancestors, then we know that a given assignment over the ancestors attributes occurs once. Hence we cannot cache that. Vice versa, if it depends on a subset of the ancestors, we know that a given assignment over a proper subset of ancestors can occur multiple times. Therefore, we can cache the attribute A.

Turning back to the example in the Figure 2, if we had said that the attribute M was dependent on the set of ancestors $\{N, T\}$ then we would have known that a given assignment over the ancestors: (N=Messi, T=Barcellona) is unique. Hence, we do not have the opportunity to cache the attribute M. However, since we have restricted the set of dependent attributes to the subset of ancestors $\{T\}$, we know that a given assignment over a subset of ancestors can occur multiple times. In fact, this is the case with the assignment (T=Barcellona), hence we can cache the attribute M.

Now, let $key(A)$ denote the set of depending ancestors of A, we can cache the attribute A if $key(A) \neq anc(A)$. In the case of the factorization tree (7) we can cache the attribute M as we have $key(M) = \{T\}$ which implies $key(M) \neq anc(M)$. We denote d-tree as a variant of f-tree augmented with $key(A)$ information for each attribute A.



$$key(N) = \{\}$$
$$key(T) = \{N\}$$
$$key(M) = \{T\}$$

Figure 4: A d-tree for d-representation (9) and the corresponding $key(A)$ information for every attribute A

A *d-tree* is an f-tree in which each attribute A is annotated by a set of attributes $key(A)$ such that:

- $key(A) \subseteq anc(A)$
- for B being a child of A the following holds: $key(B) \subseteq key(A) \cup A$

Chapter 3

F-MEM: A compact in-memory representation for factorized data

In this chapter, we start by giving the historical background of the existing in-memory representation of factorized implementation as implemented in the **FDB** engine. Then we analyze the shortcomings by referring to prior work [14]. We propose a new compact representation for factorized representations with definitions **F-MEM**, and its disk counterpart **F-DISK**. F-MEM and F-DISK representations are equivalent, the reason we differentiate them is to denote the context in which they operate. We also give examples of instances of d-representations converted into our new compact representations.

A common use case is to convert a flat relation in a factorized representation. In order to accommodate this use case, we also propose an algorithm that, given as input a flat representation and a d-tree, emits the equivalent F-MEM representation. Finally, we also give precise complexity analysis of the procedure.

3.1 Why a new in-memory representation?

The key to success in the handling of factorized data is an efficient representation of the parse tree of a factorized representation. We are building upon past experience in the development of the **FDB** engine and reflecting on the actual limitations of its representation and then proposing an alternative.

During the development of a distributed query engine for FDB the need of a serialization scheme came up. Different shortcomings of the in-memory data structure were identified [14]. We discovered that the in-memory data structure for f-representations in FDB did not bring any substantial compression advantage, due to the high redun-

```
class Node
{
        int mOperationType;      // 4 bytes
        string mAttributeName;   //32 bytes
        int mNodeType;           // 4 bytes
        Node *pNext;             // 8 bytes
        Node *pPrev;             // 8 bytes
        int mAttributeID;        // 4 bytes
        int mValueType;          // 4 bytes
        int mChildrenCount;      // 4 bytes
        Node *pFirstChild;       // 8 bytes
        Node *pLastChild;        // 8 bytes
        Value* pValue;           // 8 bytes
};
```

Figure 5: The data structure proposed by the FDB implementation denoting a node in the parse-tree of a factorized representation

dancy in the used data structure. We shall analyze in more detail the in-memory representation of FDB for factorized representations.

FDB represents every node of the parse-tree of a factorized representation ($\cup$, $\times$ and leaf) with the data-structure depicted in Figure 5.

The `pNext` and `pPrev` fields are, respectively, the next and the previous node which share the same parent of the current node, it is a double linked list. The `pFirstChild` and `pLastChild` fields are, respectively, the first and the last child of the current node. The `mNodeType` field tells us whether it is an internal node (Operation) or a leaf node (Operand). The `mOperationType` field identifies the operation ($\cup$, $\times$). We have attribute related fields `mAttributeName`, `mValueType` and `mValueType`. Finally, we have a pointer to a `Value` node being an abstraction to represent a generic value. To tell the truth, the leaf nodes are represented with a stripped down data structure similar to the above with the pertinent fields removed. To simplify the explanation, we assume that the above data-structure is used for leaf nodes.

Lambros Petrou identified the first (and main) bottleneck of the above data structure in the excessive book-keeping of pointers deriving from the double linked list structure [14]. Another point highlighted is that as FDB operates on f-representations with uniform nesting dictated by a given f-tree, then we can directly exploit the f-tree to traverse the f-representation. At the same time, we do not need to attach attribute specific fields at each node as they add redundancy and they can be inferred by the f-tree.

Lambros Petrou's first attempt at serialization of the above data structure relied on the usage of facilities for doing a portable memory dump of the data structure. He noted that the size of the above representation was almost the same as the flat one. In fact, FDB allocates **92** bytes for each instance of the above data-structure and hence for every node in the parse-tree (this is assuming that a *string* takes 32 bytes). Hence, the benefits of the high compressing nature of f-representations go away. Despite this disappointment, Lambros Petrou made a plethora of useful observations which led to the development of a competitive serialization scheme [14] with the following properties:

- $\times$ nodes can be inferred by the f-tree hence they do not need to be included in the serialization.
- The values reside only in $\cup$ nodes hence we are interested in serializing just the $\cup$ nodes.

At the end of the journey, Lambros Petrou managed to develop a competitive serialization scheme [14]. However, a competitive serialization scheme does not fix the root problem. The query processing is still done on the flat data-structure. This motivates the need for a new implementation of factorized representations. We shall also note that FDB in-memory data-structure targeted f-representations, which, can sometimes show some redundancy.

3.2 Design desiderata

The goals of our representation are the following:

1. It must specifically target d-representations. This means that we should design a facility for describing references.
2. It must be compact and cache friendly. The FDB in-memory data structure is very prone to cache misses as each node in the parse-tree is allocated ad-hoc and the memory allocator may store the values having the same $\cup$ node parent in disparate locations of memory. A cache-friendly representation is also vital for competitive query processing.
3. It should be effortless to move the in-memory representation to disk. Also, such a representation should be easy to distribute over a network with few work.
4. The total number of pointers in our data-structure should be reduced to the bare minimum.

3.3 F-MEM representation

After careful consideration, we opted for building up on the serialization scheme proposed in [14], insisting on the fact that we should simply use that as the data-structure for factorized representations. We thereby propose a representation based on the serialization scheme proposed in [14] extended to support caching in the context of d-representations.

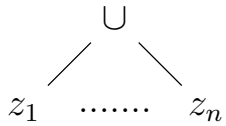
The benefits in using a serialization scheme as the underlying data-structure are many:

- It is designed for easy distribution so succinctness of the representations is one of the goals when it was conceived.
- It is easy to move out from memory to disk and vice versa. It requires just a simple copy operation.
- We can build a class of algorithms operating on the aforementioned scheme and have them ready to operate both in a memory and disk setting. Depending on the setting of course a diverse range of optimizations can be applied.

The main ingredient of our in-memory representation is the concept of *block*, which can be succinctly defined as a structure consisting of 2 fields:

`Header | Content`.

An $\cup$ node in the parse tree is represented as:



$\Longrightarrow$ n `|` z_1`,` ..., z_n.

A reference is simply encoded by setting the `Header` field of a block to 0 and by filling the `Content` field with the offset of the block of the pointed node: `0 | Offset`.

We can, therefore, sketch an algorithm that given a parse-tree of a d-representation, outputs the above representation obtained through a traversal of the tree in a depth-first fashion combined with a pattern matching driven procedure as sketched in the Figure 6.

Pattern	serialize($parse-tree$, $varMap$, $cache$)
$\langle A : a \rangle$	**return**
$\langle A : a \rangle$ $\times$ E_1	$varMap[A] \leftarrow a$ SERIALIZE(E_1, $varMap$, $cache$)

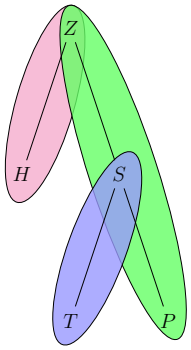
$\cup$
 a_1 ..... a_n
	$A \leftarrow ATTRIBUTE(a_1)$
	 if $key(A) \neq anc(A)$ **then**
	 $context \leftarrow \pi_{key(A)} varMap$
	 if $\exists$ $cache[A][context]$ **then**
	 $offset \leftarrow$ cache[A][context]
	 EMIT($0 \mid offset$)
	 return
	 else
	 $cache[A][context] \leftarrow$ current offset
	 EMIT($n \mid a_1, \ldots, a_n$)
	 for all $a_i \in (a_1,..., a_n)$ **do**
	 SERIALIZE(a_i, $varMap$, $cache$)

$\times$ E_1 ..... E_n	**for all** $E_i \in (E_1,..., E_n)$ **do** SERIALIZE(E_i, $varMap$, $cache$)

Figure 6: Pattern matching driven procedure for building a F-MEM representation out of a parse-tree of a d-representation

3.3.1 F-MEM by example

Suppose now that we have the following d-tree Δ. It resembles the hypergraph of a query to make it clear the set of dependencies of each attribute (H depends on Z, T depends on S, P depends on S and Z):



and the parse-tree depicted in Figure 7 of a d-representation with nesting procedure dictated by the d-tree Δ.

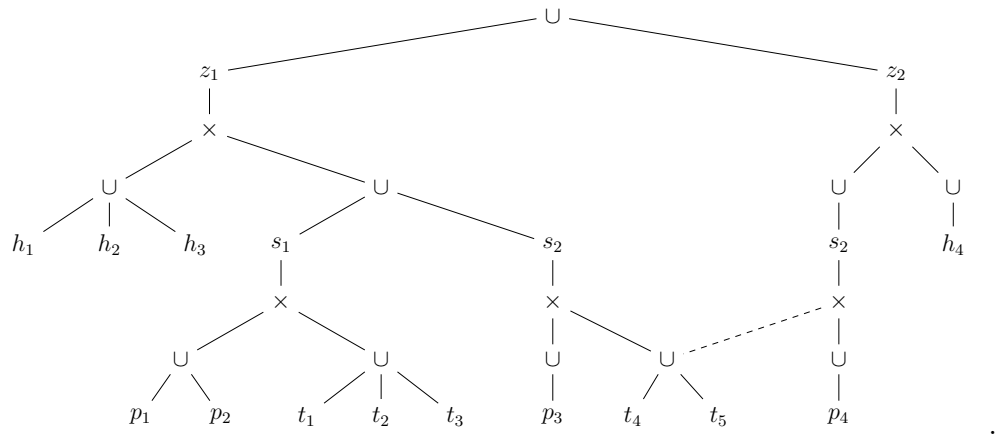
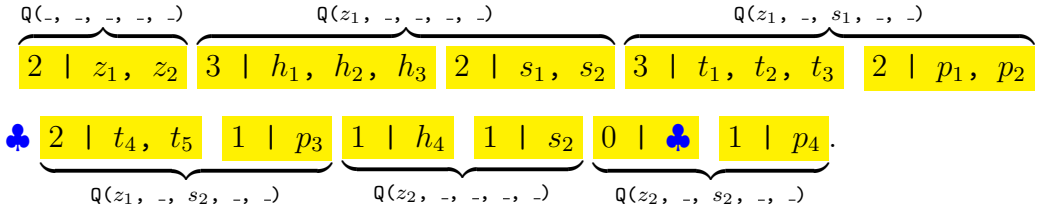


Figure 7: Parse-tree of a d-representation

The attribute T in the d-tree Δ can be cached because $key(T) \neq anc(T)$. Our goal is now to provide an example of the in-memory representation of the above d-tree and d-rep. The symbols (♣) outside of blocks indicate the current offset of the next block for illustration purposes (hence they are not present in the actual representation). If they are used inside blocks, they represent the offset pointed by the symbol. Here is the equivalent in-memory representation of the above d-representation:

2 | z_1, z_2 3 | h_1, h_2, h_3 2 | s_1, s_2 3 | t_1, t_2, t_3 2 | p_1, p_2 ♣ 2 | t_4, t_5 1 | p_3 1 | h_4 1 | s_2 0 | ♣ 1 | p_4

More clearly, let Q denote the result of the query culminating in the above d-representation. Another way of viewing the representation is like a collection of partial assignments over `Q(Z, H, S, T, P)` in a Datalog fashion:



Our aim is now to introduce an interesting scenario in which we have a d-tree exhibiting an attribute B child of an attribute A such that $key(A) \neq anc(A)$. It turns out that this scenario gives us tremendous opportunity for more compression: we will show that the entire sub-tree rooted at A can be cached as well. Firstly, we proceed in giving a formal and simple proof of the fact that if an attribute A can be cached then the whole sub-tree rooted at attribute A can be cached as well. We anticipated this in the background reading with an informal tone. Secondly, we give

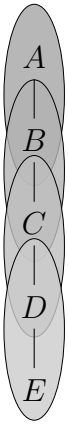
another, more compelling, example of a d-representation where some attribute B is the child of another attribute A such that $key(A) \neq anc(A)$.

Proof. Let Δ be a d-tree such that attribute A exhibits $key(A) \neq anc(A)$ and B is a child of A. We mentioned that an attribute A can be cached if the set $key(A)$ is a strict subset of $anc(A)$ which by consequence means that $|key(A)| < |anc(A)|$.

For B child of A it follows that $anc(B) = anc(A) \cup \{A\}$, $|anc(B)| = |anc(A)| + 1$ and $key(B) \subset key(A) \cup \{A\}$. Suppose that $key(B) = key(A) \cup \{A\}$ (being the biggest $key(B)$ that can be tagged for attribute B) then by consequence it follows that:

$$\underbrace{|key(A) + 1|}_{key(B)} < \underbrace{|anc(A) + 1|}_{anc(B)}$$

Hence, $key(B)$ is a proper subset of $anc(B)$ and B can be cached so the entire sub-tree rooted at A can be also cached since the above argument can be applied recursively for an eventual attribute C child of B. □



A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	d_1	e_1
a_3	b_3	c_2	d_1	e_1
a_4	b_3	c_2	d_1	e_1

We now proceed in building the F-MEM representation of the above query result with the above d-tree. In the above d-tree, C can be cached and, as a consequence, D and E can be cached as well. Once we find that a given block has been cached then we do not need to recurse over the children attributes as we just follow the trail to find out the children attributes during the traversal. As in the previous section, the symbols (♣, ♠, ★) outside of blocks just indicate the address of the subsequent blocks while inside, they represent the address pointed by that symbol:

4 | a_1, a_2, a_3, a_4 1 | b_1 1 | c_1 1 | d_1 ♣ 1 | e_1 1 | b_2 1 | c_2 ♠ 1 | d_1 0 | ♣ 1 | b_3 ★ 1 | c_2 0 | ♠ ♢ 1 | b_3 0 | ★.

$\underbrace{\qquad\qquad}_{\mathtt{Q}(a_4,\_,\_,\_,\_)}$

Let's start by traversing the blocks exploiting the d-tree Δ and by traversing the block in a depth-first fashion. Let's see how a traversal of the child blocks of $A = a_4$ would go, supposing that we have already traversed the a_1, a_2 and a_3 child blocks:

1. We are at attribute B at the offset $\diamondsuit$, we read the block $1 \mid b_3$, we traverse and update the assignment map ($A = a_4$, $B = b_1$).

2. We are at attribute C, we read the block $0 \mid \bigstar$ finding out that it is a reference, hence we move to offset $\bigstar$ reading the block $1 \mid c_2$, we traverse and update the assignment map ($A = a_4$, $B = b_1$, $C = c_2$).

3. We are at attribute D, we read the block $0 \mid \spadesuit$ finding out that it is a reference, hence we move to offset $\spadesuit$ reading the block $1 \mid d_1$, we traverse and update the assignment map ($A = a_4$, $B = b_1$, $C = c_2$, $D = d_1$).

4. We are at attribute E, we read the block $0 \mid \clubsuit$ finding out that it is a reference, hence we move to offset $\clubsuit$ reading the block $1 \mid e_1$, since it is a leaf node we are done and we traverse each value of the block.

5. We backtrack at attribute D, find out that we have read all the values hence we backtrack again.

6. ....

7. We have traversed all the child blocks of $A = a_4$ and we are done since there no more A-values to visit.

The practical implication is that whenever a value is found in the cache, we do not need to recurse over its children, since a reference represents not just an attribute, but an entire sub-tree rooted at that attribute. The benefits from our F-MEM representation really show in the above example. We avoid lots of repetitions thanks to references. We will see in Chapter 5 how we can use references perversely to speed up aggregates.

3.4 From flat relations to F-MEM representations

In this section, we propose an algorithm with precise runtime guarantees to convert a flat relation to a F-MEM representation. The algorithm takes as input a flat relation over a schema $\mathcal{S}$ and a d-tree Δ over the schema $\mathcal{S}$ satisfying the path constraint. The algorithm is sketched in Algorithm 1.

The complexity of the proposed algorithm is $O(\overbrace{|R| \ log(|R|)}^{sorting} + |R||\Delta|)$ where $|R|$ denotes the size of the relation and $|\Delta|$ the number of attributes in the d-tree Δ. The intuition behind this result is that each row of the relation R is traversed twice

Algorithm 1 Algorithm for building F-MEM representation of a flat relation

function BUILD-F-MEM-REP(R, Δ)
  SORT R by variable order Δ
  BUILD-F-MEM-NODE(R, Δ, 1, $|R|$)
function BUILD-F-MEM-NODE(R, Δ, $START$, END)
  if EMPTY-SUBTREE(Δ) **then return**
  $A \leftarrow$ ROOT-ATTRIBUTE(Δ)
  a_1, ..., $a_n \leftarrow$ FETCH A-VALUES IN ROW RANGE $\in [START, END]$
  EMIT(n | a_1, ..., a_n)
  $\Delta_C \leftarrow$ SUBTREE ROOTED AT CHILD OF A IN Δ
  for all $a_i \in (a_1,..., a_n)$ **do**
    $(START_{a_i}, END_{a_i}) \leftarrow$ RANGES OF ROWS HAVING $A = a_i$
    BUILD-F-MEM-NODE(R, Δ_C, $START_{a_i}$, END_{a_i})

for each attribute (the first time to find the unique A-values and the second time for finding the ranges of a given A-value), hence, yielding $O(2|R||\Delta|) = O(|R||\Delta|)$. It is of course possible to traverse it once and store the ranges in an auxiliary variable for subsequent use.

3.5 Some practical considerations about F-MEM representation and introducing F-DISK representations

We have defined the concept of a block being a data structure consisting of 2 fields `Header | Content`. We shall now give more rigorous information for implementors regarding this data structure.

The `Header` field has fixed size which shall be greater or equal than the number of bytes needed to represent the maximum number of values inside a $\cup$ node in the parse-tree. This number of bytes can be attribute dependent as it could be that a given attribute has the biggest block holding a small number of values meaning that we can use few bytes for the `Header` field for that attribute.

In the case that block is a reference (`0 | Content`) the `Content` must have fixed size: the number of bytes to represent the address space (in our implementation we use 4 bytes meaning that we get an address space of 2^{32} bytes). In the opposite case, the block represents a collection of values the size of the block is by $N \times V$ bytes where N is the number of values inside the block and V the number of bytes needed to represent each attribute value (hence V can be attribute dependent).

Our implementation, in order to achieve maximum compression, finds the optimal size for the `Header` field and the values of each attribute. Prepended to the in-memory representation of blocks, there is a data-structure that specifies the number of bytes needed to represent a value and the header for each attribute.

We can finally introduce **F-DISK** representations which are the counterpart of F-MEM representations operating on the disk. They are equivalent and there is no difference in the two formats. Hence everything we mention about F-MEM applies to F-DISK as well.

Chapter 4

JOINs for fun and profit

In this chapter, we discuss how to build F-MEM representations of the result of JOIN queries. We propose an algorithm to evaluate JOIN queries that returns the result as F-MEM representation. Our algorithm is a variant of the Leapfrog Triejoin, extended to partial orders in form of d-trees. We then proceed in giving the intuition regarding the JOIN algorithm by exploiting known results. We then list some open questions and concerns regarding the building of F-MEM representations of the result of a JOIN query.

4.1 An algorithm for JOIN queries for F-MEM

We consider first the case of building d-representations and then we extend it to build F-MEM representations. Before diving in the explanation we lay down some notation:

- $rel(A)$: denotes a mapping function which given the attribute A returns the set of the relations which contain the attribute A.
- A JOIN query is denoted in the datalog form, the datalog query `Player(A,B), Team(A,C)` denotes a JOIN query on the attribute A among the relations `Player` and `Team` and the schema of the result is $\{A, B, C\}$.

Let's start with the scenario in which, given flat relations, a mapping function rel and a d-tree Δ we build d-representations. A simple approach could be the following: for every attribute A in Δ compute the JOIN query of the relations $rel(A)$ in the attributes $key(A) \cup \{A\}$, do some book-keeping, by means of an associative map of sub-expressions containing the values of A grouped by unique assignments over the $key(A)$ attributes, and then build the d-representation in an incremental way exploiting the d-tree Δ combined with lookups in the associative map of sub-expressions.

The previously proposed approach is interesting but we can do better by leveraging a simple fact: suppose we are at attribute A and JOIN returns the first value a_1 then for the child attributes of A it makes sense to reduce the successive evaluation of the JOIN to the rows of the relations satisfying `Q(`a_1`, _, _, ..., _)`. The idea is to do book-keeping of the ranges of rows satisfying the current assignment and pass that to the child attributes. The trick to keep record of the ranges of rows satisfying a partial assignment follows in the spirit the Leapfrog Triejoin algorithm [18].

Our JOIN algorithm can be summed in the following way: traverse the d-tree Δ in a depth-first fashion, at each attribute A compute the intersection of the A-values among the relations involved in the JOIN. As soon we hit a new A-value a_i do book-keeping of the row ranges satisfying the partial assignment $(A = a_1)$. The JOIN of the child attributes of A will operate solely on the row ranges satisfying $(A = a_1)$. The idea is that as we traverse the d-tree and compute the intersection of A-values at each step we are restricting the admissible row ranges for the subsequent child attributes. The A-value a_i is part of the result set if for none of the child attributes B of A the intersection of B-values was empty. Evaluation of JOIN at a given attribute stops when we exhausted the admissible ranges of rows of some relation then we backtrack.

We have explored in depth how relational data can have lots of rendudancy and how factorizations provide us a foundation aimed at mitigating this problem. We can do better by exploiting factorizations during the evaluation of JOIN: suppose we are about to evaluate JOIN on a range of variables at attribute A such that $key(A) \neq anc(A)$. We do book-keeping of the current assignment over the $key(A)$ attributes to avoid re-evaluating branches of the tree leveraging the same assignment over the $key(A)$ attributes.

We build F-MEM representations of the result of JOIN queries by first maintaining a 2-level associative map of sub-expressions of A-values indexed by the attribute A and an unique assignment over $key(A)$ attributes. The associative map has a two-fold purpose: besides assisting us in the building of the F-MEM representation it gives us a facility to avoid revisiting branches of the factorization with the same assignment over $key(A)$ attributes, hence speeding up the overall query evaluation.

The pseudocode of the sketched procedure can be found in the Algorithm 2. The second step is to build the F-MEM representation from the associative map of sub-expressions created during the join as laid out in Algorithm 3.

We need to remark that our approach is heavily derived by the Leapfrog Triejoin algorithm [18]. We shy away from Leapfrog Triejoin as the former expects a variable total order resembling a d-tree with a single root-to-leaf path. The variable order

expected by the Leapfrog Triejoin algorithm is less powerful as it does not capture independence of attributes for instance with the downside that we may revisit countless times the same branch of the factorization. We should note nonetheless, that our algorithm degrades to Leapfrog Triejoin when we are given in input a d-tree which exhibits a single root-to-leaf path with no caching of the attributes (equivalent to having an f-tree). The similarities with Leapfrog Triejoin will ease the analysis of the complexity as we can reuse proven results.

Algorithm 2 Algorithm to compute the JOIN query of relations $R_{\{1,\dots,n\}}$ and given a d-tree Δ

Sort the relations $R_{\{1,\dots,n\}}$ by variable order Δ
$D \leftarrow$ Empty 2-level associative map (`Attribute -> Key`)
$varMap \leftarrow$ Empty map storing the current assignment of variables
$ranges_{i=\{1,\dots,n\}} \leftarrow [1, |R_i|]$
FMEM-JOIN(D, $varMap$, Δ, $ranges$)

function NEXT(A, $ranges$) ▷
The relations $rel(A)$ are sorted by their lowest value in the admitted $ranges$ and the *seek* function finds the row ranges satisfying a minimum threshold in the row $ranges$ of a relation, *seek* is assumed to have $O(log(n/m))$ amortised complexity where m is the number of the keys looked up for
    $M \leftarrow$ maximum value among the relations at the beginning
    $p \leftarrow$ first relation $\in rel(A)$
    while !ATEND(ranges) **do**
        $M' \leftarrow seek(M, p, ranges)$
        if $M' = M$ **then return** M'
        else
            $M \leftarrow M'$
            $p \leftarrow$ successive relation
    return $\emptyset$

function FMEM-JOIN(D, $varMap$, Δ, $ranges$)
    $A \leftarrow$ ROOT-ATTRIBUTE(Δ)
    $context \leftarrow \pi_{key(A)} varMap$
    if $key(A) \neq anc(A) \wedge \exists D[A][context]$ **then return** nonempty
    $values \leftarrow$ Empty list
    for all $a_i \leftarrow$ NEXT(A, $ranges$) **do**
        $varMap[A] \leftarrow a_i$
        $anyChildEmpty \leftarrow FALSE$
        for all $B \leftarrow child(A)$ **do**
            $\Delta' \leftarrow$ sub-tree rooted at B
            if FMEM-JOIN(D, $varMap$, Δ', $ranges$) $= \emptyset$ **then**
                $anyChildEmpty \leftarrow TRUE$
                break
        if anyChildEmpty = FALSE **then**
            $values \leftarrow values || a_i$
        Move $ranges$ to the next value.
    if $values = \emptyset$ **then**
        return $\emptyset$
    else
        $D[A][context] \leftarrow values$
        return nonempty

Algorithm 3 Algorithm to build the resulting F-MEM representation given an associative map and d-tree Δ, C denotes an initally empty cache for storing the offset of the cached blocks

```
function BUILD-FMEM-REP(D, Δ, varMap, C)
    A ← ROOT-ATTRIBUTE(Δ)
    context ← π_{key(A)} varMap
    if key(A) ≠ anc(A) then
        if ∃C[A][context] then
            ♣ ← C[A][context]
            EMIT( 0 | ♣ )
            return
        else
            ♣ ← current offset
            C[A][context] ← ♣
    a_1, ..., a_n ← D[A][context]
    if HAS-CHILD(A) then
        EMIT( n | a_1, ..., a_n )
        for all a_i ∈ {a_1, ..., a_n} do
            varMap[A] ← a_i
            for all B ← child(A) do
                Δ' ← sub-tree rooted at B
                BUILD-FMEM-REP(D, Δ', varMap, C)
```

4.2 Pushing the building of F-MEM representations over the JOIN

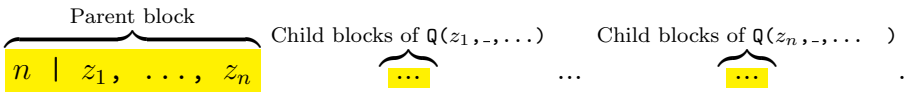
In the previous section we have proposed an algorithm for building F-MEM representations of JOIN results. The key to success has been the decoupling of the actual evaluation of the JOIN query with the process of writing the representation at a later stage. This decoupling is fundamental for supporting references.

In this chapter we will see that if we are willing to:

- give up on references in F-MEM representations,
- fix *a-priori* the number of the bytes for the `Header` and the values inside each box

then we can push the process of building the individual blocks of F-MEM representations of the results of JOIN queries inside the evaluation of the JOIN by exploiting an insight about F-MEM representations.

We shall remember that F-MEM representations are built in a depth-first fashion and we have the following pattern:



From the above pattern we can evince that we can immediately append the child blocks and at the end when we are done evaluating the block containing the partial assignment `Q(`z_n`,_,...)` we may prepend the parent block to the sequence of appended child blocks.

We modify accordingly Algorithm 2 to make it return a block and an empty block in case of a branch being empty, the proposed algorithm is Algorithm 4.

Algorithm 4 Algorithm to push the construction of F-MEM representation in the evaluation of the JOIN query of relations $R_{\{1,\ldots,n\}}$ and given a d-tree Δ, it uses the *next* procedure from Algorithm 2

Sort the relations $R_{\{1,\ldots,n\}}$ by variable order Δ
$cache \leftarrow$ Empty 2-level associative map (`Attribute -> Key`)
$varMap \leftarrow$ Empty map storing the current assignment of variables
$ranges_{i=\{1,\ldots,n\}} \leftarrow [1, |R_i|]$
FMEM-JOIN-ON-STEROIDS($cache$, $varMap$, Δ, $ranges$)

function FMEM-JOIN-ON-STEROIDS($cache$, $varMap$, Δ, $ranges$)
  $A \leftarrow$ ROOT-ATTRIBUTE(Δ)
  $context \leftarrow \pi_{key(A)} varMap$
  if $key(A) \neq anc(A) \wedge \exists cache[A][context]$ **then return** $cache[A][context]$
  $values \leftarrow$ Empty list
  $childBlocks \leftarrow$ empty sequence of child blocks
  for all $a_i \leftarrow$ NEXT(A, $ranges$) **do**
    $varMap[A] \leftarrow a_i$
    $currentChildBlocks \leftarrow$ empty sequence of child blocks of a_i
    for all $B \leftarrow child(A)$ **do**
      $\Delta' \leftarrow$ sub-tree rooted at B
      $block \leftarrow$FMEM-JOIN-ON-STEROIDS($cache$, $varMap$, Δ', $ranges$)
      if $block = \emptyset$ **then**
        $currentChildBlocks \leftarrow \emptyset$
        break
      else
        $currentChildBlocks \leftarrow currentChildBlocks \,||\, block$
    if LENGTH($currentChildBlocks$) > 0 **then**
      $values \leftarrow values||a_i$
      $childBlocks \leftarrow childBlocks \,||\, currentChildBlocks$
    Move $ranges$ to the next value.
  if $values = \emptyset$ **then**
    return $\emptyset$
  else
    $a_1, a_2, ..., a_n \leftarrow values$
    $builtBlock \leftarrow$ `n | a1, a2, ..., an` || childBlocks
    if $key(A) \neq anc(A)$ **then**
      $D[A][context] \leftarrow builtBlock$
    return $builtBlock$

The algorithm proposed is indeed elegant and very appealing from a textbook standpoint. In practice, it is not preferred and the reason is simple: over-reliance on the memory allocator. Memory allocation is generally a very hard problem and the implication of the above algorithm means that we should resize the buffer when we

are about to append a new child block. Also the process of prepending a memory block is problematic as we have to shift the child blocks to make room for the parent block. Resizing a memory block may fail in our circumstances as we contribute in allocating and freeing a plethora of sparse and fragmented portions of memory. And the allocator may not be able to grow the current block in some cases and as a consequence moves the old content to a newly allocated portion of memory to allow the resizing operation.

4.3 Intuition regarding the complexity of the JOIN algorithm

In the previous sections we have outlined how by leveraging Algorithms 2 and 3 we can build F-MEM representations of the result of a JOIN query. Our goal is to now draw a big picture of the complexity of the algorithm in analyzing Algorithm 2. We note that Algorithm 3 is linear in the size of the factorization, hence it does not contribute much to the complexity. The complexity of Algorithm 4 isthe same of Algorithm 2. We will not be giving a thorough analysis as it is beyond the scope of this work but we feel that the reader will benefit from knowing the complexity of the algorithm.

Central to our analysis is the fact that our algorithm is a variant of the Leapfrog Triejoin with the difference that it accounts for partial variable orders. Partial variable ordering captures independence between two attributes. If A and B are independent attributes then the branches in which they lie are evaluated separately. Contrary to Leapfrog Triejoin we also account for caching and we avoid revisiting branches of the factorization.

The complexity of the Leapfrog Triejoin algorithm is $O(q(n) * log(M(n)))$ where $q(n)$ is the worst-case size of the JOIN query and $M(N)$ is the size of the biggest relation [18]. Given that the leading factor in the complexity of the Leapfrog Triejoin is $q(n)$ we say that it is worst-case optimal. Our next step is then to have a pictorial representation of the $q(n)$ term as it is the governing factor of the complexity of the Leapfrog Triejoin and Algorithm 2.

Suppose we have the relations $R_{\{1,\ldots,N\}}$ involved in a JOIN query then a trivial upper bound of the size of the result set is $\prod_{i=1}^{N} |R_i|$. While this upper bound works, in practice it turns out that it can sometime overestimate the result.

For example suppose we have the query `R(A,B,C),S(A)` in which we have a single relation containing all the attributes of the result with the consequence that in such

case a better upper bound is given by the size of the relation R only, in this scenario we say that we have a relation that *covers* all the attributes. To ease the subsequent explanation (and the formulas especially) I assume from here to the end of this section that the relations involved in a JOIN have equal size.

We can now introduce the concept of *edge cover* from graph theory, an edge cover is a subset of the edges of a graph such that it *covers* all the nodes of a graph. In our case we modify the scenario of our problem and make the edges being the relations involved in a query and the nodes being the attributes of the result of a query. An edge cover assigns a value $w_i \in \{0, 1\}$ to each relation R_i, thus we we can now finally give an improved upper bound: $\prod_{i=1}^{N} |R_i|^{w_i}$. Our goal is then to minimize the edge cover which can be done with the following linear program,:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} w_i \\ \text{subject to} \quad & \sum_{j:A \in R_j} x_j \geq 1, \qquad \forall \text{ attribute A of the result of the query} \\ & x_j \in \{0, 1\}, \quad i = 1, ..., N \end{aligned}$$

In [1] the authors had an intuition: what if we remove the integrity constraints from the above linear program? They found out that by computing that, a better upper bound of the size of the result of a JOIN query is obtained and they also proved the correctness of the bound using an information-theoretic approach [1]. We now introduce the *fractional edge cover* $\rho^*(Q)$ which can be obtained with the above linear program with the integrity constraints removed:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} w_i \\ \text{subject to} \quad & \sum_{j:A \in R_j} x_j \geq 1, \quad \forall \text{ attribute A of the result of the query} \\ & x_j \geq 0, \quad i = 1, ..., N \end{aligned}$$

The upper bound of the result of the query Q over database D is then given by $|D|^{\rho^*(Q)}$. Thus Leapfrog Triejoin algorithm has $O(q(n) * log(M(n)))$ complexity where $q(n) = |D|^{\rho^*(Q)}$.

Our algorithm (Algorithm 2) at its heart, traverses the d-tree and for each attribute performes a restricted JOIN over the $key(A) \cup \{A\}$ attributes and for each A-value does some book-keeping of the ranges of the rows of relations which satisfy any partial assignment having that A-value. Thus an upper bound on the worst-case number of A-values is given by $\rho^*(Q_{key(A) \cup A})$. The worst-case size of our factorized

JOIN is given by the maximum fractional edge cover over the restricted JOIN queries to execute for each attribute. The same tactic has been used to denote the maximum size of the d-representation of a JOIN query Q over a d-tree Δ ([8]).

$$s(\Delta) = max(\{\rho^*(Q_{key(A)\cup\{A\}})|\text{A is attribute in } \Delta\})$$

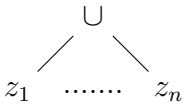
Given a d-tree Δ and a database D then it admits a d-representation of size $|D|^{s(\Delta)}$ ([8]).

Coming back to the analysis of our algorithm, our algorithm like Leapfrog Triejoin has complexity $O(q(n) * log(M(n)))$ where $q(n) = |D|^{s(\Delta)}$.

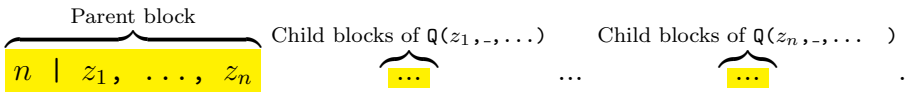
4.4 Open questions and concerns

In this section, we outline a current limitation of F-MEM representations.

Suppose that we have



This leads to the subsequent F-MEM representation:



Suppose that we have a JOIN which has already evaluated the branch $\texttt{Q}(z_1,\_,\_,\ldots)$. We cannot start writing the F-MEM representation as we have yet to evaluate other Z-values. Practically it means that if Z is root then we can start the process of writing down the F-MEM representation only after the complete evaluation of the JOIN query. This limitation was the motivating factor behind the decision to make the Algorithm 2 maintain an associative map of values and behind the trick to push the process of building F-MEM representations inside the evaluation of the JOIN query to immediately append the child blocks and then prepend the parent block when we are done evaluating the last Z-value.

This is what differentiates F-MEM representations from the FDB data-structure, FDB has the perk of having a data-structure resembling a tree meaning that can write on a branch immediately.

While this may sound a big limitation we show, in our experiments later that indeed the fact of exploiting structural properties of the query, by means of d-trees, overcomes the described limitation. We also demonstrate that F-MEM representations make up for very competitive query processing of aggregates which really shadow the above limitation.

Chapter 5

Aggregates on F-MEM

In this chapter, we show how F-MEM representations offer an appealing framework for the design of query processing algorithms. Query processing in inherently speeded up by the usage of factorized representations as they do not bring the redudancy often present in relational data. Factorized representations are beneficial especially for aggregate queries. We show how to exploit d-trees in such contexts. Firstly, we design an algorithm for COUNT queries over F-MEM representations. Secondly, we design an algorithm for a restricted class of GROUP BY queries.

5.1 COUNT: a case study

A COUNT query returns the total number of flat tuples present in a factorized representation. We explain how it is possible to do counting over factorized representations in general and then we propose an algorithm for doing COUNT over F-MEM representations.

The general approach to do counting over an f-representation is to replace every singleton with 1. Let the $\times$ nodes be an arithmetic multiplication and let $\cup$ nodes be an arithmetic sum. An equivalent way to view this is to conduct an in-order traversal of the parse-tree of a d-representation E with the following pattern matching driver procedure `count`:

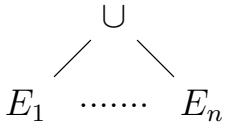
- **If** $E = \langle A : a \rangle$ **then return** 1
- **If** $E = \bigcup_i E_i$ **then return** $\sum_i \texttt{count}(E_i)$
- **If** $E = \times_i E_i$ **then return** $\prod_i \texttt{count}(E_i)$

The reasoning behind at heart of counting is the following, given the following portion of f-representation:

$$\begin{array}{c} \langle A : a \rangle \\ \times \\ E_1 \end{array}$$

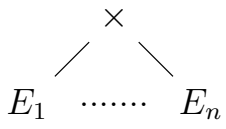
the count is given by the number of suffixes of $\langle A : a \rangle$ present in the flat representation. Hence is given by the COUNT of E_1.

Given



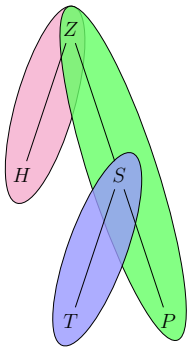
the total count is given by the sum of the counts of the invidual sub-expressions E_i

Given



the total count is given by the total number in which we can combine values from each sub-expression (E_1, ..., E_n). The total number of combinations is given by the product of the number of combinations of each sub-expression. Suppose we have two bags, an A-bag ($\langle A : a_1 \rangle$, $\langle A : a_2 \rangle$) and a B-bag ($\langle B : b_1 \rangle$, $\langle B : b_2 \rangle$, $\langle B : b_3 \rangle$). It is easy to see that the total number of way in which we can combine a value from the first bag with one from the second bag is given by $|A| * |B|$.

We now extend the above approach to d-representations to exploit caching whenever possible. Suppose that we have the following d-tree:



where we can cache the attribute T. We can observe that the Z-values depend solely on the assignment over $key(Z)$ attributes, in this case $\{S\}$. For an unique S-value we have the same collection of Z-values. Conversely, for each S-value we have an unique count of Z-values. This means that we can cache the aggregate count of Z-values so at the next occurence of the same S-value we do not recompute the count of the Z-values dependent on that S-value. Caching counts of *cachable* fragments of the factorized representation can be done by means of an associative map of the form (`Attribute -> Key -> Count`) where `Key` represents an unique assignments over key(`Attribute`) attributes. In the associative map we store the total COUNT of the entire sub-tree rooted at a given cachable attribute.

We now move to the task of counting the number of tuples over F-MEM representations. We start by discussing an awesome trick, made possible by F-MEM representations, to get rid of the 2-level associative map `Attribute -> Key -> Count` and instead have a single-level associative map with superior performance. Intuitively, when we traverse the factorized representation, we keep record of the current assignment over the ancestor $anc(A)$ attributes for a given attribute A in a variable map, with F-MEM representations sometimes we may not need to do that.

So, suppose we have in a F-MEM representation, the block n `|` z_1`, ...,` z_n of Z-values at offset ♣ and that $key(Z) \neq anc(Z)$. The idea is that for uniquely identifying the block of Z-values instead of using the assignment over $key(Z)$ attributes we can use the offset ♣ directly. Thus we compute the COUNT over the sub-tree rooted at Z and then store that in an associative map with key ♣. Whenever we find a reference in a F-MEM representation 0 `|` ♣ we inspect the associative map for the key ♣. In practical terms, this means that, instead of having a 2-level associative map `Attribute -> Key -> Count`, we may empower a provably faster single-level associative map `Offset -> Count`. The offset of a block for attribute Z equals an unique assignment over $key(Z)$ attributes. We do not need to store the attribute as it cannot be that two blocks belonging to two different attributes have the same offset and we identify the ownership of a block to a given attribute by means of the d-tree.

For doing COUNT, the only valuable part of the block is the first field containing the number of values residing in that block, while in the case of references (`Header` $=$ 0) we are interested in the second field which contains the key to look up for in the associative map.

We can now put together all these observations and generate Algorithm 5. The running time of the algorithm is linear in the number of the blocks of a F-MEM representation.

Algorithm 5 COUNT over F-MEM representations given d-tree Δ

function COUNT-F-MEM-REP(Δ, *cache*)
- $\clubsuit \leftarrow$ current offset
- $n \mid z_1, \ldots, z_n \leftarrow$ read block
- **if** $key(A) \neq anc(A) \wedge \clubsuit \in cache$ **then**
  - **return** $cache[\clubsuit]$
- $A \leftarrow$ ROOT-ATTRIBUTE(Δ)
- $count \leftarrow \sum_{1}^{n} \left(\prod_{c \in child(A)} \text{COUNT}(c, cache) \right)$
- **if** $key(A) \neq anc(A)$ **then**
  - $cache[\clubsuit] \leftarrow count$
- **return** *count*

While COUNT queries are speeded up in F-MEM as you will see in the experiments section and part of the success is due to F-MEM representations they do not fully exploit caches and for this reason in the next case study we focus on aggregates and GROUP BY.

COUNT queries are speeded up in F-MEM as we will see in Chapter 6 and part of the success is due to F-MEM representations. However we need to note that COUNT queries do not fully exploit caches as the only value they need of a block is the `Header` field. For this reason, in the next case study, we focus on aggregate queries.

5.2 Aggregates: a case study

One of the goal of F-MEM representations is to make it easy to design cache-oblivious query processing algorithms. The trick, in order to better exploit caches, is to minimize the total number of memory transfers, this can be done by reasoning over the pattern of memory accesses. We aim at increasing *operational intensity*: intuitively the ratio of operations executed and the number of memory accesses [20].

In this chapter, we start with the computation of aggregates over blocks. Then we give argument on the maximum number of memory transfers. Finally we propose an algorithm for computation of aggregates over GROUP BY.

Let V the number of bytes used to encode a value, let H denotes the number of bytes to encode the header and let R the number of bytes used to encode a reference, then the total number of bytes of a block is given by:

$$size(n \mid \texttt{Content}) = \begin{cases} H + N * V & n \neq 0 \\ H + R & n = 0 \end{cases}$$

We introduce now some aggregates over blocks:

- $count(n \mid z_1, \ldots, z_n) = n$
- $sum(n \mid z_1, \ldots, z_n) = \sum_{i=1}^{n} z_i$

Suppose that we have a cache where a single cache line fits B bytes. We can then place an upper bound on the total number of memory transfers for each aggregate.

The *count* aggregate has an upper bound of 2 memory transfers. In the worst-case size the input block is not aligned in memory hence requiring in the worst-case 2 memory transfers. The *count* aggregate does not really exploits caches as it does not do a big number of operations in relation to the number of memory transfers as it just loads the first value in the block as mentioned in the chapter 5.1.

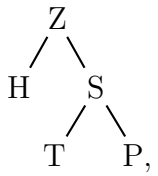
The *sum* aggregate, on the other side, touches a sweet spot on caches as it does not just read the first part of the block to read the number of values deposited in the block as successively scans each value to accumulate the sum. Caches are fully exploited because B can be big enough to store more than one value. A theorem states that for scanning N contiguous locations of memory on a cache with block size B it requires at most $\lceil N/B \rceil + 1$ memory transfers [6]. Henceforth for computing the *sum* aggregate we require at most $\left\lceil size(n \mid z_1, \ldots, z_n)/B \right\rceil + 1$ memory transfers. It is easy to see how the *sum* aggregate has higher operational intensity compared to the *count* as for each memory transfer computes at most $\lceil B/V \rceil$ additions as opposed to the single load for *count* operations.

Now let's imagine how the aggregate *sum* would behave in FDB where a single node takes **92** bytes. This means that for n values it takes $92 * n$ bytes to which we should add another level of indirection as it stores a pointer to the value contained in the node to be deferenced. In the real world, cache lines are usually smaller than 92 bytes. Hence even in the best case the allocator does a good job in putting the nodes in contiguous portions of memory we incur in a number of cache misses linear in the number of the values contained in the node.

We now show how we can build an algorithm to build resulting F-MEM representations of aggregates over GROUP BY queries with F-MEM representations as inputs. To make things simpler, we focus on a restricted family of GROUP BY queries and operate on F-MEM representations with no references.

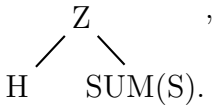
In the design of an algorithm for combining GROUP BY queries and aggregates we exploit some prior result [7]. We exploit the fact that factorization trees are already grouped in the above attributes. More specifically we can list groupings with

constant delay over the set of attributes G such that G contains the root of the factorization tree and each other attribute is a child of another attribute in G [7]. Suppose we have the following factorization:



then it is already grouped in the attributes $\{Z, H, S\}$ or $\{Z, S, P\}$ or yet $\{Z, H, S, T\}$.

Suppose now that we want to run the aggregate sum of S grouped by $\{Z, H\}$ on the input factorization in above. We note that the $SUM(S)$-values are dependent on the Z-values as were the S-values and that like S is indipendent with respect to the H attribue. The output factorization is then:



. As noted in [7] the dependence set of an attribute does not change with its eventual aggregation.

We now propose an algorithm that given in input a F-MEM representation with no caching, a set of grouping attributes G which allows for costant delay enumeration, an attribute A child of another attribute in G and F being an aggregate function, returns a corresponding F-MEM representation equivalent to the result of the aggregate query.

We build F-MEM representation of aggregate query by traversing the F-MEM representation using the given d-tree Δ. For enumerating the grouping attributes we ought to copy the corresponding blocks, copying takes a number of memory transfers asymptotic to the size of the corresponding blocks. When we at an attribute C that need to be aggregated, we, first, read the C-values inside the block and then we compute the aggregate function F of the read C-values. Then since the aggregate is a single value we emit the following block in the output F-MEM representation: `1 | F(`C`-values)`. If an attribute is not part of the output factorization we just skip the corresponding blocks in the output F-MEM representation.

In the truth the algorithm executes a very restricted class of GROUP BY queries. However the goal of this chapter is to show how we can design query processing algorithm under the framework casted by F-MEM representations. The proposed algorihm is presented in Algorithm 6.

Algorithm 6 Aggregates over F-MEM representations with no references given d-tree Δ, grouping attributes G, attribute on which to aggregate X, the aggregate function F

```
function F-MEM-GROUP-BY(Δ)
    n | z_1, ..., z_n ← read block
    A ← ROOT-ATTRIBUTE(Δ)
    if A ∈ G then
        EMIT(n | z_1, ..., z_n)
        for all i ∈ [1, n] do
            for all B ← child(A) do
                F-MEM-GROUP-BY(B)
    else if A = X then
        aggregate ← F(z_1, ..., z_n)
        EMIT(1 | aggregate)
        skip the entire sub-tree rooted at A in the F-MEM representation.
    else
        skip the entire sub-tree rooted at A in the F-MEM representation.
```

The algorithm is indeed limited as it supports a restricted class of aggregates but we have shown how with little effort we supported aggregation over F-MEM representation. We have also shown how aggregation over F-MEM representations can really benefit from its inherent cache friendliness which was the point we wanted to state.

Chapter 6

Experimental evaluation

In this chapter, we run a full-fledged experimental evaluation of F-MEM and F-DISK against FDB. Firstly, we design a suite of experiments and describe the purpose of each experiments. Secondly, we describe the setting used to run the experiments. Thirdly, we collect the results of the experiments and we give a thorough analysis.

F-MEM and F-DISK, opposedly to FDB, leverage d-trees which in turn allow us to support references. We show that F-MEM/F-DISK representations are more succinct than the in-memory data structure of FDB. We also show that leveraging d-trees speeds up JOIN queries. The compactness of our representation proves beneficial for query processing as we show that COUNT queries are definitely speeded up.

6.1 Experiment suite

The adversary in our experiments suite is FDB, an in-memory factorized database. FDB is built upon f-representations and f-trees and, as such, it is not able to process the more powerful d-representations and d-trees. D-trees, as opposed to f-trees, are richer in the sense that they embed information about structural properties of the query, which we exploit for the purposes of caching whenever possible. As we have shown, leveraging d-trees can speed up queries to the point that, in some cases, we can completely skip the evaluation of some branches of the factorization.

Firstly, we compare F-MEM against FDB in the process of building representations of the result of a JOIN query. We compare the total time taken to execute the JOIN query and to build the representations of the result. We also compare the size in memory of F-MEM representations against the FDB data-structure. The purpose of the first experiment is to understand the impact of the limitation affecting F-MEM representations given that we cannot immediately write branches of the factorization as shown in the Chapter 4.4. An investigation is made on the size of the resulting

representations as one of our battle horses was the fact that the FDB data-structure was heavy. We show that F-MEM representations are more succinct. Then, F-MEM is compared against F-DISK to investigate the overhead induced by the I/O requests.

Secondly, we evaluate the performance of COUNT queries for F-MEM and FDB. We have seen how F-MEM representations offer us a nice framework to design query processing algorithms on the top of that, we shall now assess the competitivity of query evaluation over F-MEM representations. Then, we compare F-MEM against F-DISK to investigate the overhead induced by I/O requests. Motivated by early results, we also propose and test an alternative to F-DISK limited to COUNT queries dubbed F-DISK$_2$ to validate a hypothesis of ours. F-DISK$_2$ is basically F-MEM operating on memory mapped files which is a facility offered by some operating systems to map regions of virtual memory to files.

Each experiment in the suite is run 5 times and the average wall-clock time in seconds is plotted.

6.2 Experimental setting

The experiments are being run on an Intel Core i7-4712HQ CPU @ 2.30GHz 8 cores with the following cache info and sizes:

- **L1 cache**: 256kb size and 8-way associative
- **L2 cache**: 1024kb size and 8-way associative
- **L3 cache**: 6144kb size and 12-way associative.

The external memory was a WDC WD10SPCX-75K HDD having a read speed $\sim$ `1gb/s` and write speed $\sim$ `109mb/s`. The experiments are being run on a Linux machine with the 4.4.0.34 kernel version mounted. FDB, F-MEM and F-DISK have been compiled with g++5.4.0 using the same optimization flags.

6.2.1 Datasets

We run the experiment over two datasets: **Housing** and **LastFM**.

The Housing dataset is a synthetic dataset meant to reproduce a common use case of input for typical regression tasks. It is very common, especially in the industry, to give as input for regression tasks the result of the JOIN query of multiple tables [9].

It is comprised of 6 tables: `House` (postcode, livingarea, price, nbbedrooms, nbbathrooms, kitchensize, house, flat, unknown, garden, parking), `Shop` (postcode, openinghoursshop, pricerangeshop, sainsburys, tesco, ms), `Institution` (postcode, typeeducation, sizeinstitution), `Restaurant` (postcode, openinghoursrest, pricerangerest), `Demographics`(postcode, averagesalary, crimesperyear, unemployment, nbhospitals) and `Transport`(postcode, nbbuslines, nbtrainstations, distancecitycentre).

The LastFM dataset is publicicly released and it has been generally used for social network analysis and prediction. It it is made of 3 tables: `Userfriends` (userid, friendid), `Usertaggedartiststimestamps` (userid, artistid, tagid, timestamp), `Userartists` (userid, artistid, weight) [4]. The artists liked by user id are joined with the artists liked by the friends. This setting offers room for learning tasks to learn if a given user is an "influencer" among his friends.

The chosen datasets have been recently used in the work of learning linear regression models over factorized joins [9].

6.2.2 Queries and d-trees

In the Appendix A we give the d-trees which have been used for the LastFM dataset (**LQ1**, **LQ2**) and for the Housing dataset (**HQ1**, **HQ2**, **HQ3**). The blue attribute can be cached which means that the entire sub-tree rooted at that attribute can be cached as well.

In 4.3 chapter we discussed about the $s(\Delta)$ measure which can be used in order to obtain an upper bound on the size of a d-representation over d-tree Δ and database D being $|D|^{s(\Delta)}$ [8]. For a query Q there exists a multitude of d-trees Δ, then:

$$s^{\uparrow}(Q) = min(\{s(\Delta)|\forall\Delta \text{ d-tree of Q}\})$$

denotes the *d-tree width* of a query Q, which is given by the minimum $s(\Delta)$ attained by the total set of d-trees Δ for query Q. A hierarchical query Q has $s^{\uparrow}(Q) = 1$ which is the best d-tree width attainable.

The LQ1, LQ2 d-trees both denote hierarchical queries on the LastFM dataset and their d-trees are asymptotically the best $s^{\uparrow}(Q) = 1$. We can think of a d-tree being like a query plan and the d-trees LQ1 and LQ2 denote the same query but they have different evaluation plans, the intention was to evaluate how much the ordering of attributes would have affected the overall evaluation plan.

The LQ1 and LQ2 d-trees leverage caching. Hence, they will be an interesting playground to investigate the general benefit provided by d-trees, as we can avoid revisiting some branches. The JOIN is done on the attributes *userid* and *friendid*

Dataset	Relations	JOIN
LastFM	Userartists (25.434 tuples), Userfriends (186.479 tuples), Userartists (92.834 tuples)	59.079.380 tuples
Housing	Demographics (25.000 tuples), House (125.000 tuples), Institution (50.000 tuples), Restaurant (75.000 tuples), Shop (125.000 tuples), Transport (25.000 tuples)	3.750.000 tuples

Figure 8: The number of tuples of the relations of the datasets used for experiments. Reported is also the total number of tuples given by their resulting JOIN on the d-trees LQ1, LQ2 for the LastFM dataset and HQ1, HQ2 and HQ3 for the Housing dataset.

and the artists liked by both *userid* and *friendid* are joined, the artist related fields of *friendid* are denoted with a final x. A typical use case of this query is to find out similarity of song taste with friends in a network or perhaps to identify some trend where people liking a specific artist also like another specific one. Different d-trees lead to different factorized representation.

The HQ1, HQ2 and HQ3 d-trees denote hierarchical queries on the housing dataset and again since the queries are hierarchical their d-tree width is $s^{\uparrow}(Q) = 1$. The queries are join on the *postcode* field, a typical use case of this query is to obtain samples for typical regression tasks for predicting the value of a house given characteristics of the neighborhood.

The HQ1 and HQ2 d-tree are similar as we have the joining attribute being the root attribute and the branches represent the different relations. The HQ3 d-tree on the other side is a bit different as we investigate the case in which the attribute $openinghoursshop$ exhibits $key(openinghoursshop) \subset key(nbtrainstations) \cup \{nbtrainstations\}$ and does does not depend on the parent $nbtrainstations$. We note that while the attribute *openinghourshop* can be theoretically cached in the d-tree HQ3, this is not the case as it depends on *postcode* which is a root attribute. HQ1, HQ2 and HQ3 d-trees do not leverage caching. The intention is to run experiments in a scenario which is more near to FDB which does not understand d-trees.

6.3 Results

6.3.1 FDB vs F-MEM: the beginning of the battle

The first experiment highlights the performance of building the factorized representation of the result of a JOIN query. The time to execute the JOIN query is also taken in account.

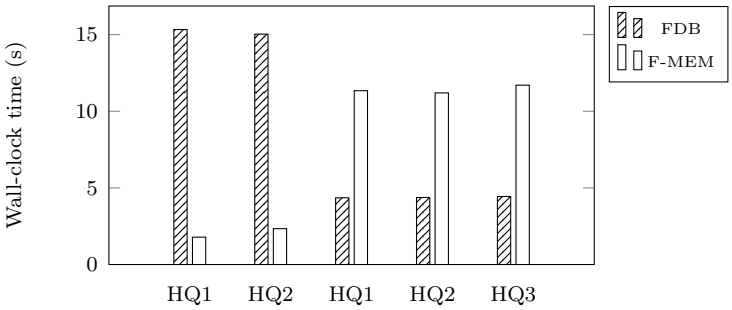


Figure 9: FDB vs F-MEM: Time needed to compute the factorized representation of a range of JOIN queries

As highlighted by the results in Figure 9, F-MEM outperforms FDB by a factor of 3 on the LastFM dataset where the d-trees leverage caching. This means that the factorized JOIN is more efficient, as we avoid to visit redundant branches of the factorization. D-trees with caching definitely touch a sweet spot.

Unfortunately, the news are not as good with the Housing dataset (where the d-trees do not leverage caching). It turns out that there is a fundamental reason that explains the behaviour of F-MEM. The problem is that in F-MEM we do not push the process of building F-MEM representations inside the JOIN led by the insights in chapter 4.4. The process of building F-MEM representation can be summed with the following multi-step process:

1. It executes the JOIN query and maintains an associative map `Attribute -> Key -> Values` which is needed to then build the blocks of F-MEM representations.

2. Then inspects the associative map of blocks to find out the optimal number of bytes for representing the header and the values of a block for each attribute.

3. Finally writes the representation.

FDB on the other side builds its representation closely intertwined with the JOIN procedure. FDB opposedly to F-MEM can write a single A-value in the representation immediately which is the perk of having a tree-like structure as described in the chapter 4.4. On the other side, F-MEM cannot start write a single A-value but instead it needs all the A-values before starting doing that. This is the main reason the process of writing the representation comes after the JOIN.

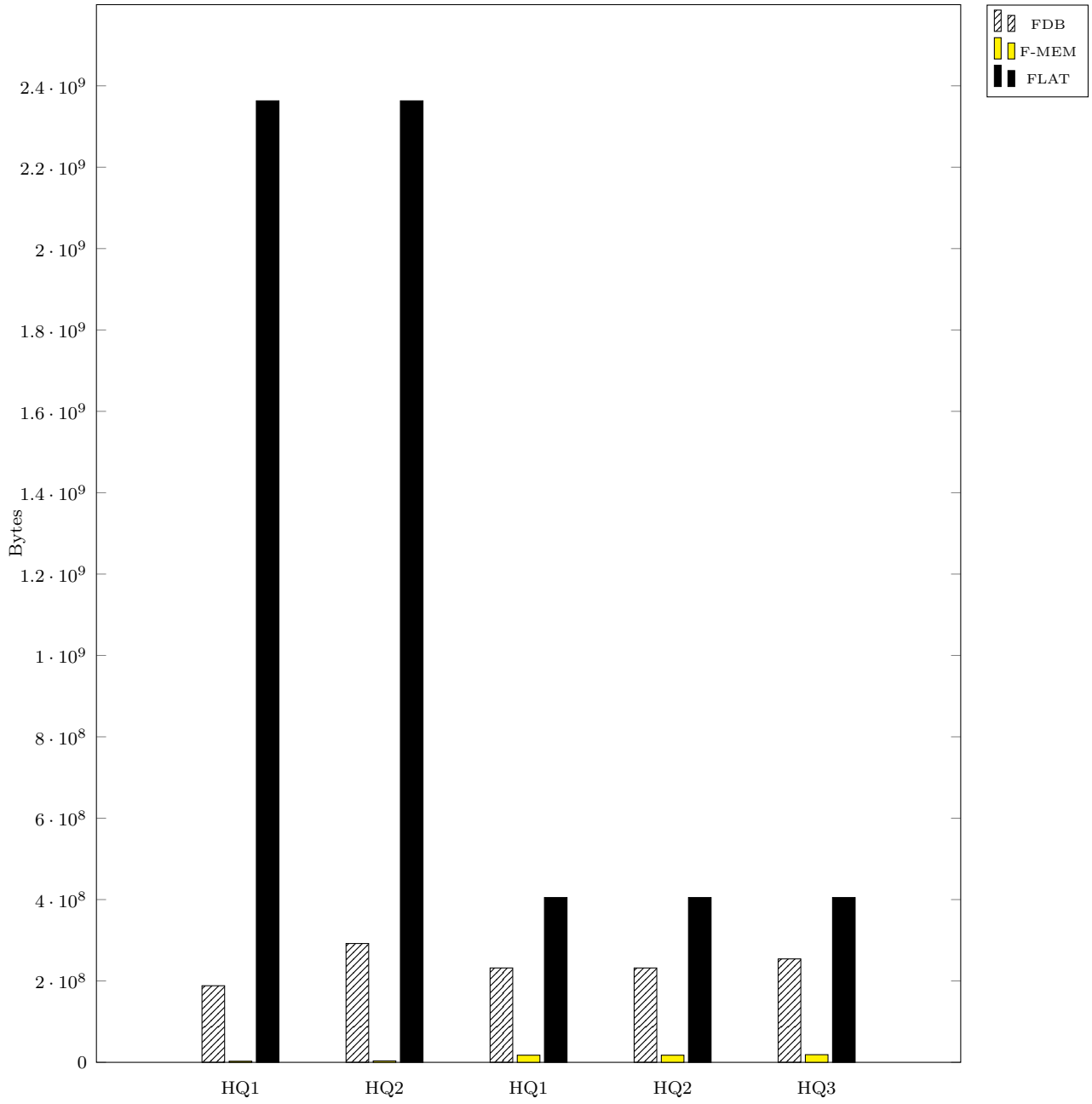


Figure 10: FDB vs F-MEM: size of the resulting factorized representation. For better showing you the power of factorized representations, we show you the size of the flat representation of tuples. In the Housing and LastFM datasets the values are floating point values, we assume in the size of the flat representation that each value takes 4 bytes in memory ($\#tuples * \#attributes * 4$).

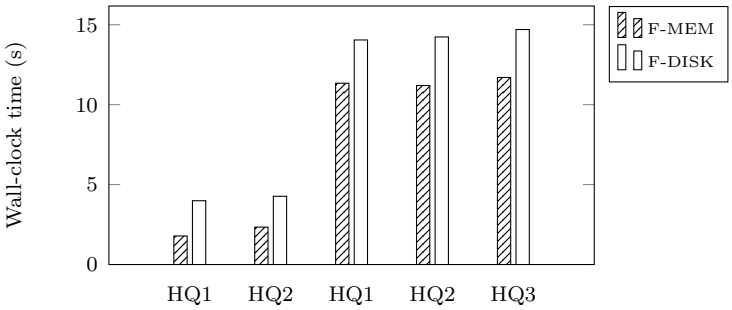


Figure 11: F-MEM vs F-DISK: Time needed to compute the factorized representation of a range of JOIN queries

F-MEM representations are more compressed compared to FDB representations as shown in Figure 10 which is a great news as the size of the representation is a fundamental ingredient for competitive query processing. Also F-MEM representation blocks are contiguous meaning that can be exploited to make great use of the CPU caches. What we lose with d-trees with no caching, as it happens with the experiments involving the Housing dataset, we gain in subsequent query processing of the materialized result. This strengthens the case for materialized views in the analytics context where usually the queries are not settled a-priori.

6.3.2 F-MEM vs F-DISK: the bottleneck to write on the disk

It is common knowledge that reading and writing from disk is more expensive than reading and writing to memory. Given that we are at the storage layer we are interested in understanding the performance degradation caused by the disk. Disk capacity is cheaper compared to the memory counterpart, this motivates our interest in investigating the performance of F-DISK.

We benchmark the time taken by F-DISK against F-MEM for computing the F-MEM representation of a JOIN query, like we did in the previous section against FDB. We note that the size of our representations is very small hence our experiments do not give a complete picture of the difference between F-MEM and F-DISK.

In Figure 11 reported is the performance of F-DISK against F-MEM. There is little surprise as the experiments show that F-DISK reports slower performance compared to F-MEM.

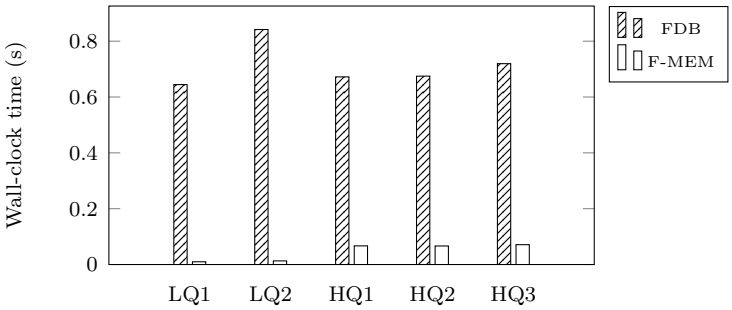


Figure 12: Benchmarking the execution of COUNT queries in FDB and F-MEM

6.3.3 COUNT query: a case study

We have shown in the previous sections how F-MEM representations have small memory footprint compared to FDB. In our work we also showed how one could design algorithms operating on our representation for competitive query processing. We shall now verify the impact of F-MEM representation in the context of a query processing task. In our setting we are going to test the performance of COUNT queries. In this experiment the results of the previous JOIN queries are already materialized in memory and they constitute the input for the COUNT query.

In the Figure 12 we show how F-MEM is faster in handling COUNT queries by different orders of magnitude compared to FDB. This verifies our claims about how succinctness of representations is fundamental for competitive query processing. We can postulate that the extra time taken by FDB is due to cache misses due to the abuse of pointers.

We want to note that F-MEM COUNT queries are faster on the LQ1 and LQ2 result set, in spite of containing a higher number of tuples compared to HQ1, HQ2 and HQ3. This is because of d-trees and caching. Caching really does wonders as it leads to more compressed F-MEM representations as witnessed in Figure 10 which in turn affect the performance of COUNT queries.

In Figure 13 reported is the performance of COUNT queries between F-MEM and F-DISK and it shows that operating on the disk is not efficient. The slow down is expected because of the cost of `seek` operations for moving the disk arm. But we note that this is not the whole story as thanks to some profiling of F-DISK we found out that lot of time was spent in copying the read data from kernel-space memory to user-space memory. Every time we issue a disk request the kernel deposits the read data in a buffer which needs to be then copied to a user-accessible location of memory. This copying operation has indeed some overhead and motivated by this we made a

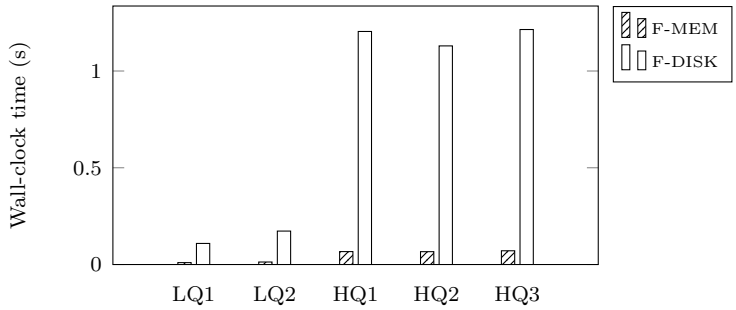


Figure 13: Benchmarking the execution of COUNT queries in F-MEM and F-DISK

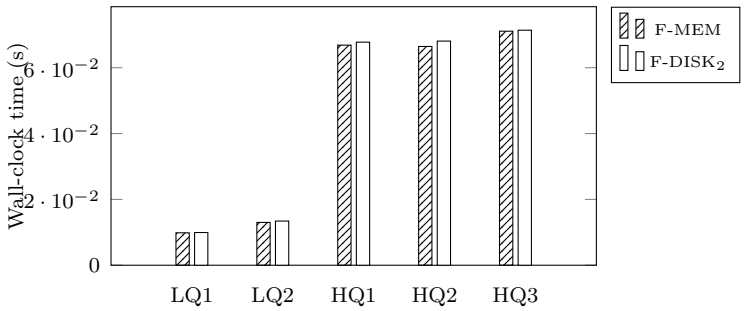


Figure 14: Benchmarking the execution of COUNT queries in F-MEM and F-DISK$_2$

quick alternative implementation of F-DISK labelled F-DISK$_2$. F-DISK$_2$ makes use of memory-mapped files which is a facility to map regions of virtual memory to files. The kernel takes care of transparently issuing I/O requests whenever needed based on our accesses to the portions of mapped memory. The advantage of memory mapping files is that we can operate on files like we are operating with portions of memory. This allows us to reuse F-MEM, in fact F-DISK$_2$ under the hood reuses F-MEM.

In Figure 14 reported are the results of F-MEM against F-DISK$_2$. From the result it looks like that our previous analysis was correct and indeed some of the overhead was due to the expensive copying happening from the kernel memory to the process accessible memory. Again, we are not parroting the idea that operating to the disk is more efficient, in spite of similar performance especially for LQ1 and LQ2 queries. We postulate that the similarity of performance for LQ1 and LQ2 queries is due to the size of the representation which is not big enough to make the overhead of operating on the disk noticeable.

The disadvantage of F-DISK$_2$ is that is limited by the total amount of virtual memory which limits the use cases.

Chapter 7

Related work

Our contribution lies in between databases and compression. We showed how our compression scheme other than achieving excellent compression ratio at the same time it also makes up for competitive query processing. In the last years there has been a surge in bundling compression schemes in databases but it is still surprising how this problem received very little treatment in the academic community and yet is the key to tackle the scalability challenges posed in the handling of enormous amount of data. We position our work in the realm of succinct data structures for relational data.

7.1 Compression

Factorized representations are a different flavour of compression in the sense that compared to value-based data compression which dominate the information retrieval field (ie. [19] [12] [21]) they exploit structural properties of the query, in fact we position factorized representation in the family of compression algorithm which exploit structural properties of the input data.

In information retrieval field different flavours of value-based data compression schemes have been proposed to store monotone sequences of integers, at their heart many of them are based on differential coding schemes.

PForDelta ([21]) is a simple differential coding scheme which works by first finding out a number of bytes b big enough to fit 90% of the differences in the data and by keeping patches for the remaining 10%.

Elias-Fano ([19]) index is a different mixture of differential coding scheme, let u the biggest number that can be hold, let λ denote a number $< u$. The approach is the following: store the lower $log(u/\lambda)$ bits of each member of the sequence explicitly, then encode the upper bits in unary code (ie. (0^k1) denotes the number k) and

compute the differences between the upper bits. It uses at most $2 + log(u/\lambda)$ bits per element [19]. Then there is a partitioned variant where the elements are partitioned in different folders and then in each folder an instance of the **Elias-Fano** algorithm is present [12].

Compression in the database realm is not yet a widely employed practice and this problem, as mentioned before, received very little treatment in the academic industry until recently where an intriguing intersection between the database and the machine learning fields crossed over[9] [15]. It has been noted that it is common case the input to learning tasks are the result of some JOIN queries hence they can entail unnecessary redundancy which is not needed for learning tasks.

Rendle proposed a limited form of factorization of the design matrix [15]. His approach does discovery of repeating patterns which is a hard problem, we do better in that regard as we exploit structural properties of the query.

The Google Adwords business is backed by the F1 database and key to their success is a flavour of factorization called *hierarchical clustered schema* which emphasizes data locality for a set of specific access patterns [17]. They scaled this factorization in a distributed setting where each node basically stores a fragment of the factorization rooted at a given tuple [17], [11].

At their heart factorizations are simply a way to denote dependencies of attributes which is similar in spirit to the concept of conditional independence emphasized in bayesian networks ([13]).

Chapter 8

Conclusion

In this work, we first designed a succinct representation for factorized data: F-MEM representations. Secondly, we accommodated a variety of use cases, like building a F-MEM representation out of a flat relation or of a result of a JOIN query. For accommodating the latter use case we extended the Leapfrog Triejoin algorithm [18] to handle partial orders and to avoid revisiting twice the same branches of the factorization.

We demonstrated that F-MEM representations give a simple yet powerful framework for the design of query processing algorithms. We designed algorithms for COUNT and GROUP BY queries.

In all of the work we pursued a line of attack marrying simple ideas to accomplish the goals. Like our compression scheme that is indeed simple and straightforward to implement. This is because F-MEM representations are best described as a serialization scheme. A serialization scheme is meant as a medium to store an instance of a data-structure in order to ease the distribution. At a latter time then the original data-structure gets reconstructed from the serialized data. In the case of F-MEM representations we have shown that this reconstruction process does not take place as we operate directly on the representation. The simplicity of our scheme also allows us to swap easily representations both from memory to disk effortlessly. Hence opening room for intriguing future opportunities.

The case studies conducted on the performance of query processing over F-MEM representations demonstrated that there is a significant speed up compared to the state-of-art FDB engine. FDB engine was demonstrated to be more performant than off-the-shelf databases [2].

The experiments best position our work in the materialization of a query result use case. Materialization of query results is key in analytics as it is typical in this setting to run different queries in order to conduct exploratory data analysis.

8.1 Future work

This work just scratched the surface and it is very far from being a finished one. In fact it opens a big room for future improvements and works.

Factorized representations have been recently leveraged for machine learning tasks [16]. It has been shown that they speed up learning tasks by orders-of-magnitude compared to different engines both commercial and not. They designed **F**, a learner for the regression family of models. **F** learns models out of the result of JOIN queries without explicitly materializing them. **F** could be extended to learn models over F-MEM representations. The advantage is given by the fact that multiple models over the same F-MEM representation could be explored efficiently given that this materialization takes place. This process, as a consequence, would take advantage from the succinctness and cache-friendliness characterized by the F-MEM representations. Linear regression can be implemented with the aggregate grouped over all the pairs of attributes H and S(`SUM(H * S)`) and by leveraging the rewritings of the linear regression described in [9]. We have sort of paved the way with aggregate queries in this work and it should be trivial to extend our work.

We have shown that swapping a F-MEM representation from disk to memory is trivial. This could be exploited by query processing over fragments of F-MEM representations, and then writing out the result to the disk. The concept of fragments of F-MEM representations follows in spirit what Google has done with the F1 database. F1 database employs a limited form of factorizations and is especially designed to operate in a distributed setting. Each node in a cluster in F1 is rooted at a given tuple. This can be achieved by properly splitting the F-MEM representations in fragments.

F-DISK is currently limited as it can manage only the family of datasets that fit in the main memory. A solution is to implement the associative map maintained by the JOIN query (Algorithm 2) with associative data-structures on disk like B-Trees. Leveraging tricks to minimize the number of disk transfers, as the ones employed by Log-Structured-Merge trees and fractal trees for instance [5] [3], will be beneficial for addressing this use case.

In a short this is a very intriguing time for the research in factorized databases as there is a whole new room of opportunities that have yet to be explored. We believe that F-MEM representations will be a key ingredient for future research.

References

[1] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 739–748. IEEE, 2008.

[2] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodnỳ. Fdb: A query engine for factorised relational databases. *Proceedings of the VLDB Endowment*, 5(11):1232–1243, 2012.

[3] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan Fogel, Bradley Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 81–92, San Diego, CA, USA, June9–11 2007.

[4] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[5] Shimin Chen, Phillip B Gibbons, Todd C Mowry, and Gary Valentin. Fractal prefetching b+-trees: Optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2002.

[6] Erik D Demaine. Cache-oblivious algorithms and data structures. 2002.

[7] Tomáš Kočiský. Queries with order-by clauses and aggregates on factorised relational data. Master's thesis, University of Oxford, 2015.

[8] Dan Olteanu. Factorized databases: A knowledge compilation perspective. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[9] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Record*, 45(2):5, 2016.

[10] Dan Olteanu and Jakub Závodnỳ. Factorised representations of query results: Size bounds and readability. In *Proceedings of the 15th International Conference on Database Theory*, pages 285–298. ACM, 2012.

[11] Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):2, 2015.

[12] Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 273–282. ACM, 2014.

[13] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.

[14] Lambros Petrou. Single-round vs multi-round distributed query processing in factorised databases. Master's thesis, University of Oxford, 2015.

[15] Steffen Rendle. Scaling factorization machines to relational data. In *Proceedings of the VLDB Endowment*, volume 6, pages 337–348. VLDB Endowment, 2013.

[16] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 3–18, 2016.

[17] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[18] Todd L Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[19] Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 83–92. ACM, 2013.

[20] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[21] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, pages 387–396. ACM, 2008.

Appendix A

d-trees used in the experiment

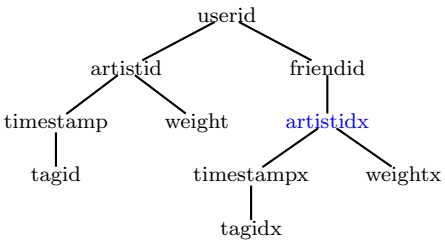


Figure 15: *LastFM Q1*

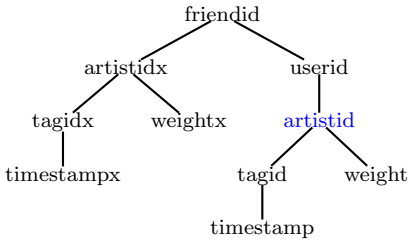


Figure 16: *LastFM Q2*

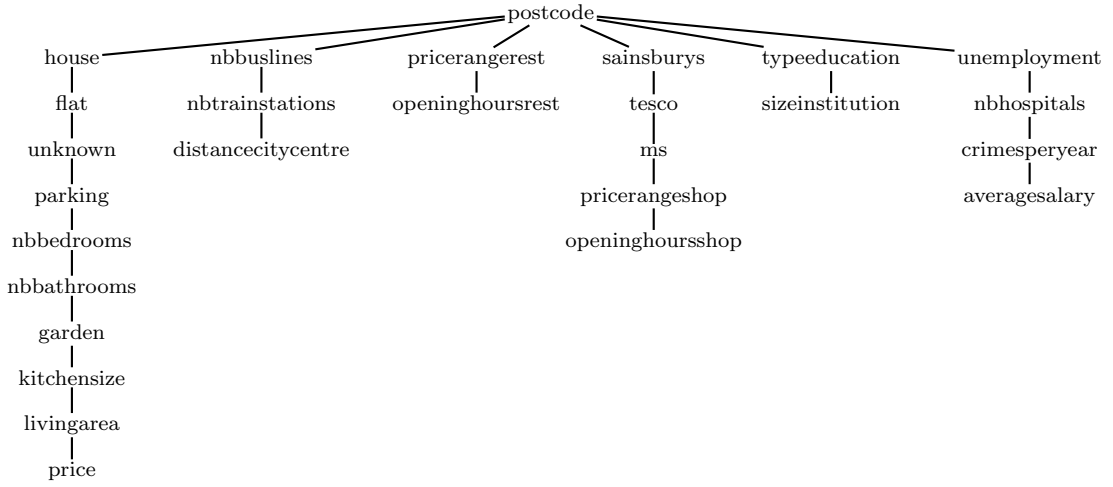


Figure 17: *Housing Q1*

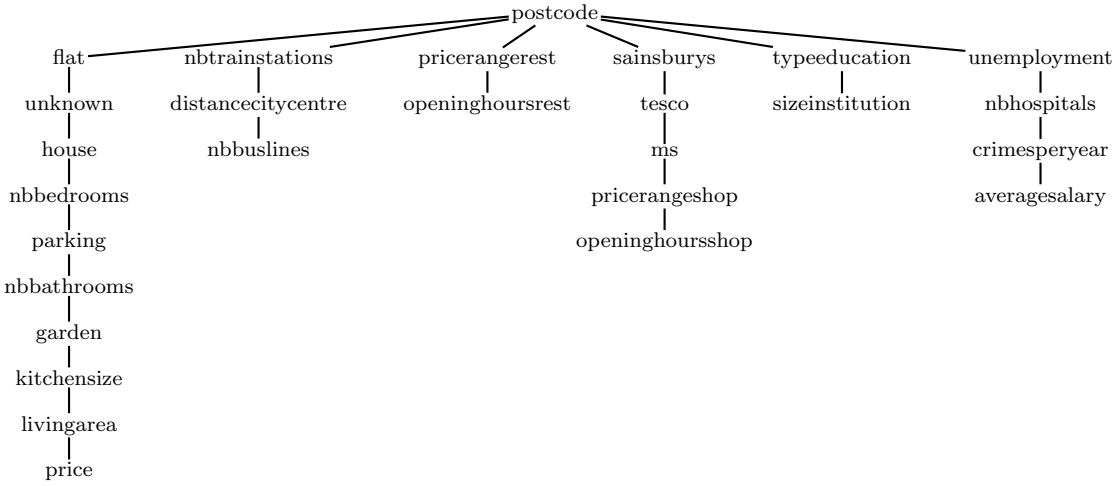


Figure 18: *Housing Q2*

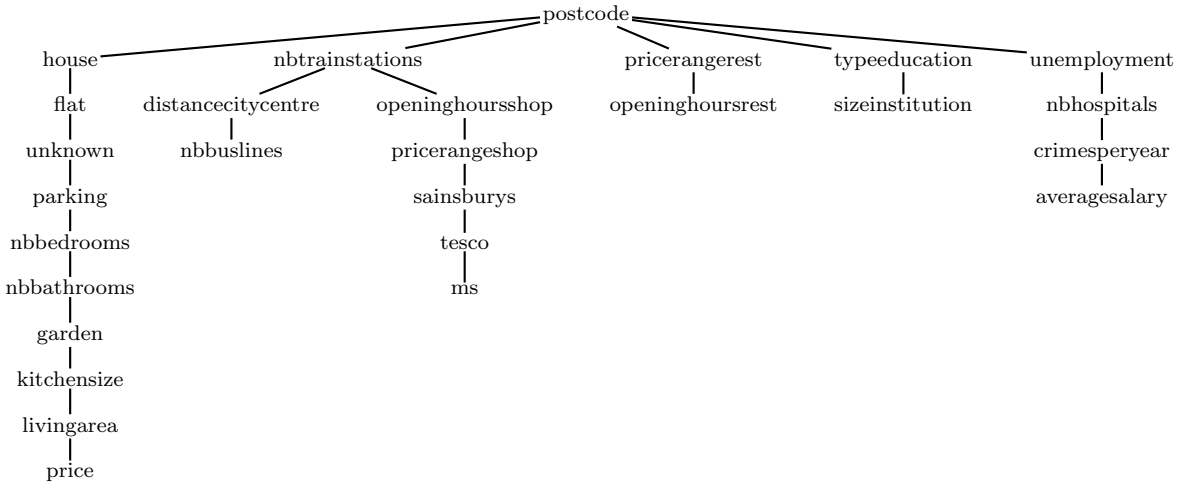


Figure 19: *Housing Q3*