FDB: A Query Engine for Factorised Relational Databases

Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný
Department of Computer Science, University of Oxford, OX1 3QD, UK
{nurzhan.bakibayev, dan.olteanu, jakub.zavodny}@cs.ox.ac.uk

ABSTRACT

Factorised databases are relational databases that use compact factorised representations at the physical layer to reduce data redundancy and boost query performance.

This paper introduces FDB, an in-memory query engine for select-project-join queries on factorised databases. Key components of FDB are novel algorithms for query optimisation and evaluation that exploit the succinctness brought by data factorisation. Experiments show that for data sets with many-to-many relationships FDB can outperform relational engines by orders of magnitude.

1. INTRODUCTION

This paper introduces FDB, an in-memory query engine for select-project-join queries on factorised relational data.

At the outset of this work lies the observation that relations can admit compact, factorised representations that can effectively boost the performance of relational processing. The relationship between relations and their factorised representations is on a par with the relationship between logic functions in disjunctive normal form and their equivalent nested forms obtained by algebraic factorisation.

Example 1. Consider a database of a grocery retailer with delivery orders, stock availability at different locations, availability of dispatcher units for each location, and suppliers with items and locations they supply to (Figure 1).

The query Q_1 returns all orders with their respective items, possible locations to retrieve them from, and dispatchers available to deliver them (shown only partially):

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

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

This query result can be expressed as a relational expression built using singleton relations, union, and product, whereby each singleton relation $\langle v \rangle$ holds one value v, each tuple is a product of singleton relations, and the relation is a union of products of singleton relations:

$$\langle 01 \rangle \times \langle \text{Milk} \rangle \times \langle \text{Istanbul} \rangle \times \langle \text{Adnan} \rangle \cup$$
$$\langle 01 \rangle \times \langle \text{Milk} \rangle \times \langle \text{Istanbul} \rangle \times \langle \text{Yasemin} \rangle \cup$$
$$\langle 01 \rangle \times \langle \text{Milk} \rangle \times \langle \text{Izmir} \rangle \times \langle \text{Adnan} \rangle \cup$$
$$\langle 01 \rangle \times \langle \text{Milk} \rangle \times \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle \cup \ldots$$

A more compact equivalent representation can be obtained by algebraic factorisation using distributivity of product over union and commutativity of product and union:

$$\langle \text{Milk} \rangle \times \langle 01 \rangle \times (\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \cup \langle \text{Izmir} \rangle \times \langle \text{Adnan} \rangle \cup \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle) \cup$$
$$\langle \text{Cheese} \rangle \times (\langle 01 \rangle \cup \langle 03 \rangle) \times (\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \cup \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle) \cup$$
$$\langle \text{Melon} \rangle \times (\langle 02 \rangle \cup \langle 03 \rangle) \times \langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle)$$

This *factorised representation* has the following structure: for each item, we construct a union of its orders and a union of its possible locations with dispatchers. This nesting structure together with the attribute names form the schema of the factorised representation, which we call a *factorisation tree*, or f-tree for short.

Figure 2 depicts several f-trees; the leftmost one ($\mathcal{T}_1$) captures the nesting structure of the above factorisation. The second f-tree ($\mathcal{T}_2$) is an alternative nesting structure for the same query result, where for each location, we construct a union of its items and orders and a union of dispatchers:

$$\langle \text{Istanbul} \rangle \times (\langle \text{Milk} \rangle \times \langle 01 \rangle \cup \langle \text{Cheese} \rangle \times (\langle 01 \rangle \cup \langle 03 \rangle) \cup \langle \text{Melon} \rangle \times (\langle 02 \rangle \cup \langle 03 \rangle)) \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \cup$$
$$\langle \text{Izmir} \rangle \times \langle \text{Milk} \rangle \times \langle 01 \rangle \times \langle \text{Adnan} \rangle \cup$$
$$\langle \text{Antalya} \rangle \times (\langle \text{Milk} \rangle \times \langle 01 \rangle \cup \langle \text{Cheese} \rangle \times (\langle 01 \rangle \cup \langle 03 \rangle)) \times \langle \text{Volkan} \rangle$$

The factorised result of the query $Q_2 = \text{Produce} \bowtie_{\text{supplier}} \text{Serve}$ over the f-tree $\mathcal{T}_3$ given in Figure 2 is:

$$\langle \text{Guney} \rangle \times (\langle \text{Milk} \rangle \cup \langle \text{Cheese} \rangle) \times \langle \text{Antalya} \rangle \cup$$
$$\langle \text{Dikici} \rangle \times \langle \text{Milk} \rangle \times (\langle \text{Istanbul} \rangle \cup \langle \text{Izmir} \rangle \cup \langle \text{Antalya} \rangle) \cup$$
$$\langle \text{Byzantium} \rangle \times \langle \text{Melon} \rangle \times \langle \text{Istanbul} \rangle \qquad \square$$

Factorisations are ubiquitous. They are most known for minimisation of Boolean functions [9] but can be useful in a number of read-optimised database scenarios. The scenario we consider in this paper is that of factorising large intermediate and final results to speed-up query evaluation on data sets with many-to-many relationships. A further scenario we envisage is that of compiled databases: these are static databases, such as databases encoding the human genome, that can be factorised to efficiently support a particular scientific workload. In provenance and probabilistic databases, factorisations of provenance polynomials [12] are

Orders

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

Store

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

Disp

dispatcher	location
Adnan	Istanbul
Adnan	Izmir
Yasemin	Istanbul
Volkan	Antalya

Produce

supplier	item
Guney	Milk
Guney	Cheese
Dikici	Milk
Byzantium	Melon

Serve

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

Figure 1: An example database for a grocery retailer.

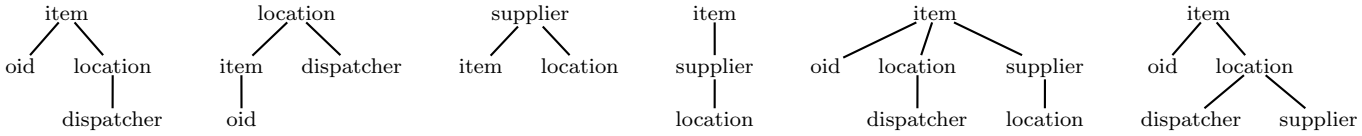


Figure 2: Examples of factorisation trees. From left to right: $\mathcal{T}_1$ and $\mathcal{T}_2$ for the result of query Q_1; $\mathcal{T}_3$ and $\mathcal{T}_4$ for the result of Q_2; $\mathcal{T}_5$ is obtained after joining $\mathcal{T}_1$ and $\mathcal{T}_4$ on item, and $\mathcal{T}_6$ is $\mathcal{T}_5$ after joining on location.

used for compact encoding of large provenance [18] and for efficient query evaluation [16, 21]. Factorisations are a natural fit whenever we deal with a large space of possibilities or choices. For instance, data models for design specifications, such as the AND/OR trees [15], are based on incompleteness and non-determinism and are captured by factorised representations. The world-set decomposition formalisms for incomplete information [4, 17] relies on factorisations of universal relations encoding very large sets of possible worlds. Outside data management scenarios, factorised relations can be used to compactly represent the space of feasible solutions to configuration problems in constraint satisfaction, where we need to connect a fixed finite set of given components so as to meet a given objective while respecting given constraints [5].

Factorised representations have several key properties that make them appealing in the above mentioned scenarios.

They *can be exponentially more succinct* than the relations they encode. For instance, a product of n relations needs size exponential in n for a flat relational result, but only linear in the size of the input relations for a factorised result. Recent work has established tight bounds on the size of factorised query results [19]: For any select-project-join query Q, there is a rational number $s(Q)$ such that for any database $\mathbf{D}$, there exists a factorised representation E of $Q(\mathbf{D})$ with size $O(|\mathbf{D}|^{s(Q)})$, and within the class of representations whose structures are given by factorisation trees, there is no factorisation of smaller size. The parameter $s(Q)$ is the fractional edge cover number of a particular subquery of Q, and there are arbitrarily large queries Q for which $s(Q) = 1$. Moreover, the exponential gap between the sizes of E and of $Q(\mathbf{D})$ also holds between the times needed to compute E and $Q(\mathbf{D})$ directly from the input database $\mathbf{D}$.

Further succinctness can be achieved using dictionary-based compression and null suppression of data values [20]. Compressing entire vertical partitions of relations as done in c-store [10] is not considered in our factorisation approach since it breaks the relational structure.

Notwithstanding succinctness, factorised representations of query results *allow for fast (constant-delay) enumeration of tuples*. More succinct representations are definitely possible, e.g., binary join decompositions [11] or just the pair of the query and the database [6], but then retrieving any tuple in the query result is already NP-hard. Factorised representations can thus be seen as compilations of query results that allow for efficient subsequent processing.

By construction, factorised representations *reduce redundancy in the data and boost query performance* using a mixture of vertical (product) and horizontal (union) data partitioning. This goal is shared with a large body of work on normal forms [2] and columnar stores [8] that considers join (or general vertical) decompositions, and with partitioning-based automated physical database design [3, 14]. In the latter case, the focus is on partitioning input data such that the performance of a particular workload is maximised.

Finally, factorised representations *are relational algebra expressions* and factorisation trees are nested join dependencies. Their relational nature sets them apart from XML documents, object-oriented databases, and nested objects [2], where the goal is to avoid the rigidity of the relational model. Moreover, in our setting, a query result can admit several equivalent factorised representations and the goal is to find one of small size. The Verso project [1] and earlier work on compacted relations [7] pointed out compactness and modelling benefits of non-first-normal-form relations and considered hierarchical data representations that are special cases of factorised representations. These works did not focus on the search for factorisations of small sizes.

A factorised database presents relations at the logical layer but uses succinct factorised representations at the physical layer. The FDB query engine can thus not only compute factorised query results for input relational databases, but can evaluate queries directly on input factorised databases.

Example 2. Consider now the query $Q_1 \bowtie_{\text{location,item}} Q_2$ on factorised representations: Find possible suppliers of ordered items. Joining the above factorisations over the f-trees $\mathcal{T}_1$ and $\mathcal{T}_3$ on the attributes location and item is not immediate, since tuples with equal values for location and item appear scattered in the factorisation over $\mathcal{T}_3$. If we restructure the factorisation of Q_2's result to follow the f-tree $\mathcal{T}_4$ so that tuples are grouped by item first, we obtain

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

which can be readily joined with the factorisation over $\mathcal{T}_1$ on the attribute item, since both factorisations have items

as topmost values. The factorisation of the join on item follows the f-tree $\mathcal{T}_5$, where we simply merged the roots of the two f-trees. An excerpt of this factorisation is

$$\langle \text{Milk} \rangle \times \langle 01 \rangle \times (\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \cup \langle \text{Izmir} \rangle \times \langle \text{Adnan} \rangle \cup \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle) \times (\langle \text{Guney} \rangle \times \langle \text{Antalya} \rangle \cup \langle \text{Dikici} \rangle \times (\langle \text{Istanbul} \rangle \cup \langle \text{Izmir} \rangle \cup \langle \text{Antalya} \rangle)) \cup \dots,$$

To perform the second join condition on location, we first rearrange for each item the subexpression for suppliers and locations, so that it is grouped by locations as opposed to suppliers. This amounts to swapping supplier and location in $\mathcal{T}_5$. The join on location can now be performed between the possible locations of each item. The obtained factorisation follows the schema $\mathcal{T}_6$ in Figure 2. □

Examples 1 and 2 highlight challenges involved in computing factorised representations of query results.

Firstly, a query result may have different (albeit equivalent) factorised representations whose sizes can differ by an exponential factor. We seek f-trees that define succinct representations of query results for input (flat or factorised) databases. Such f-trees can be statically derived from the query and the input schema and are independent of the database content. Query optimisation thus has to consider two objectives: minimising the cost of computing a factorised query result and minimising the size of this output representation. In addition to the standard query operators selection, projection, and product, the search space for a good query and factorisation plan, or f-plan for short, needs to consider specific operators for restructuring schemas and factorisations. We propose two such operators: a swap operator, which exchanges a given child with its parent in an f-tree, and a push-up operator, which moves an entire subtree up in the f-tree. For instance, the swap operator is used to transform the f-tree $\mathcal{T}_3$ into $\mathcal{T}_4$ in Figure 2. The selection operator is used to merge the item nodes in the f-trees $\mathcal{T}_1$ and $\mathcal{T}_4$ and create the f-tree $\mathcal{T}_5$. The transformation of $\mathcal{T}_5$ into $\mathcal{T}_6$, which corresponds to a join on location, needs a swap of supplier and location and a merge of the two location nodes.

Secondly, we would like to compute the factorised result as efficiently as possible. This means that we must avoid the computation of intermediate results in flat form. Our query engine has algorithms for each f-plan operator (selection, projection, product, swap, push-up) that use time quasilinear in the sizes of input and output representations.

The main contributions of this paper are as follows:

- We introduce exhaustive and heuristic optimisations for finding f-plans that compute factorised results for select-project-join (aka conjunctive) queries.

  As cost metric, we use selectivity and cardinality estimates and a compile-time parameter that defines tight bounds on the sizes of factorised results.

- We describe efficient algorithms for the evaluation of each f-plan operator on factorised data.

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

- We report on an extensive experimental evaluation showing that FDB can outperform a home-bred in-memory and two open-source (SQLite and PostgreSQL) relational query engines by orders of magnitude.

2. F-REPRESENTATIONS AND F-TREES

We next recall the notions of factorised representations and factorisation trees, as well as results on tight size bounds for factorised representations over factorisation trees [19].

Factorised representations of relations are algebraic expressions constructed using singleton relations and the relational operators union and product.

Definition 1. A *factorised representation* E, or f-representation for short, over a set $\mathcal{S}$ of attributes and domain $\mathcal{D}$ is a relational algebra expression of the form

- $\emptyset$, the empty relation over schema $\mathcal{S}$;
- $\langle\rangle$, the relation consisting of the nullary tuple, if $\mathcal{S} = \emptyset$;
- $\langle A:a \rangle$, the unary relation with a single tuple with value a, if $\mathcal{S} = \{A\}$ and a is a value in the domain $\mathcal{D}$;
- (E), where E is an f-representation over $\mathcal{S}$;
- $E_1 \cup \dots \cup E_n$, where each E_i is an f-representation over $\mathcal{S}$;
- $E_1 \times \dots \times E_n$, where each E_i is an f-representation over $\mathcal{S}_i$ and $\mathcal{S}$ is the disjoint union of all $\mathcal{S}_i$.

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

Any f-representation over a set $\mathcal{S}$ of attributes can be interpreted as a database over schema $\mathcal{S}$. Example 1 gives several f-representations, where singleton types are dropped for compactness reasons. For instance, $\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle)$ represents a relation with schema {location, dispatcher} and tuples (Istanbul, Adnan), (Istanbul, Yasemin).

F-representations form a representation system for relational databases. It is *complete* in the sense that any database can be represented in this system, but not injective since a database may have different f-representations. The space of f-representations of a database is defined by the distributivity of product over union. Under the RAM model with uniform cost measure, the tuples of a given f-representation E over a set $\mathcal{S}$ of attributes can be enumerated with $O(|\mathcal{S}|)$ additional space and delay between successive tuples.

Factorisation trees define classes of f-representations over a set of attributes and with the same nesting structure.

Definition 2. A *factorisation tree*, or f-tree for short, over a schema $\mathcal{S}$ of attributes is an unordered rooted forest with each node labelled by a non-empty subset of $\mathcal{S}$ such that each attribute of $\mathcal{S}$ labels exactly one node.

Given an f-tree $\mathcal{T}$, an f-representation over $\mathcal{T}$ is recursively defined as follows:

- If $\mathcal{T}$ is a forest of trees $\mathcal{T}_1, \dots, \mathcal{T}_k$, then
$$E = E_1 \times \dots \times E_k$$
where each E_i is an f-representation over $\mathcal{T}_i$.
- If $\mathcal{T}$ is a single tree with a root labelled by $\{A_1, \dots, A_k\}$ and a non-empty forest $\mathcal{U}$ of children, then
$$E = \bigcup_a \langle A_1:a \rangle \times \dots \times \langle A_k:a \rangle \times E_a$$
where each E_a is an f-representation over $\mathcal{U}$ and the union $\bigcup_a$ is over a collection of distinct values a.
- If $\mathcal{T}$ is a single node labelled by $\{A_1, \dots, A_k\}$, then
$$E = \bigcup_a \langle A_1:a \rangle \times \dots \times \langle A_k:a \rangle.$$
- If $\mathcal{T}$ is empty, then $E = \emptyset$ or $E = \langle\rangle$.

Attributes labelling the same node in $\mathcal{T}$ have equal values in the represented relation. The shape of $\mathcal{T}$ provides a hierarchy of attributes by which we group the tuples of the represented relation: we group the tuples by the values of the attributes labelling the root, factor out the common values, and then continue recursively on each group using the attributes lower in the f-tree. Branching into several subtrees denotes a product of f-representations over the individual subtrees. Examples 1 and 2 give six f-trees and f-representations over them.

For a given f-tree $\mathcal{T}$ over a set $\mathcal{S}$ of attributes, not all relations over $\mathcal{S}$ have an f-representation over $\mathcal{T}$. However, if a relation admits an f-representation over $\mathcal{T}$, then it is unique up to commutativity of union and product.

Example 3. The relation $R = \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 2,2\rangle\}$ over schema $\{A, B\}$ does not admit an f-representation over the forest of f-trees $\{A\}$ and $\{B\}$, since there are no sets of values a and b such that R is represented by $(\bigcup_a \langle A:a\rangle) \times (\bigcup_b \langle B:b\rangle)$. Its f-representation over the f-tree with root A and child B is $\langle A:1\rangle \times (\langle B:1\rangle \cup \langle B:2\rangle) \cup \langle A:2\rangle \times \langle B:2\rangle$. □

For any f-tree $\mathcal{T}$ that is a path of nodes each labelled by one attribute of a schema $\mathcal{S}$, any relation over $\mathcal{S}$ has an f-representation over $\mathcal{T}$. Given a flat relation R over $\mathcal{S}$, it is sufficient to sort R by its attributes following the root-to-leaf order in $\mathcal{T}$, and group its tuples by the root attribute first, then by its child, and so on, to obtain an f-representation of R over $\mathcal{T}$ in time $O(|\mathcal{S}| \cdot |R| \cdot \log |R|)$. Conversely, since the tuples of any f-representation can be enumerated with delay $O(|\mathcal{S}|)$, we can retrieve R in time $O(|\mathcal{S}| \cdot |R|)$. Therefore, f-representations over f-trees allow for fast conversion to and from flat relations.

F-trees of a query. Given a query $Q = \pi_{\mathcal{P}}\sigma_{\varphi}(R_1 \times \cdots \times R_n)$, we can derive the f-trees that define factorisations of the query result $Q(\mathbf{D})$ for any input database $\mathbf{D}$, which we call *f-trees of* Q. We consider f-trees where nodes are labelled by equivalence classes of attributes in $\mathcal{P}$. The equivalence class of an attribute A is the set of A and all attributes transitively equal to A in φ.

The f-trees need to satisfy a so-called *path constraint*: all *dependent* attributes can only label nodes along a same root-to-leaf path. The attributes of a relation are dependent, since in general we cannot make any independence assumption about the structure of a relation, cf. Example 3. Attributes from different relations can also be dependent: if we join two relations, then their non-join attributes are independent conditioned on the join attributes, but if these join attributes are not in the projection list $\mathcal{P}$, then the non-join attributes of these relations become dependent.

The path constraint is key to defining which f-trees represent valid nesting structures for factorised query results.

PROPOSITION 1. *Given a query Q, an f-tree $\mathcal{T}$ is an f-tree of Q if and only if it satisfies the path constraint.*

Tight size bounds for f-representations over f-trees. Given any f-tree $\mathcal{T}$, we can derive tight bounds on the size of f-representations over $\mathcal{T}$ in polynomial time. We next sketch our approach to finding such bounds; a full treatment is given in prior work [19].

For any root-to-leaf path p in $\mathcal{T}$, consider the hypergraph whose nodes are the attributes classes of nodes in p and whose edges are the relations containing these attributes. The edge cover number of p is the minimum number of edges necessary to cover all nodes in p. We can lift edge covers to their fractional version in which the edges are assigned non-negative rational weights so that each node is covered by edges with total weight at least 1. The fractional edge cover number of p is then the minimum sum of weights of all edges necessary to cover all nodes in p [13].

For an f-tree $\mathcal{T}$, we define $s(\mathcal{T})$ as the maximum such fractional edge cover number of any root-to-leaf path in $\mathcal{T}$.

Example 4. Each f-tree $\mathcal{T}$ except for $\mathcal{T}_3$ in Figure 2 has $s(\mathcal{T}) = 2$, while $s(\mathcal{T}_3) = 1$. In $\mathcal{T}_3$, both root-to-leaf paths supplier–item and supplier–location can be covered by assigning weight 1 to relations Produce and Serve respectively. □

For any database $\mathbf{D}$ and f-tree $\mathcal{T}$, the size of the f-representation of the query result over $\mathcal{T}$ is at most $|\mathcal{P}| \cdot |\mathbf{D}|^{s(\mathcal{T})}$, and there exist arbitrarily large databases $\mathbf{D}$ for which the size of the f-representation over $\mathcal{T}$ is at least $(|\mathbf{D}|/|Q|)^{s(\mathcal{T})}$. Moreover, given $\mathbf{D}$ and $\mathcal{T}$, the f-representation of the query result over $\mathcal{T}$ can be computed in time $O(|Q| \cdot |\hat{\mathcal{T}}| \cdot |\mathbf{D}|^{s(\hat{\mathcal{T}})+1})$, where $\hat{\mathcal{T}}$ is an extension of $\mathcal{T}$ with nodes for all attributes of the input schema and not only those in the projection list. The parameter $s(\mathcal{T})$ thus dictates the asymptotic size of f-representations over $\mathcal{T}$, and provides an important quality measure for f-trees. The f-trees with smallest $s(\mathcal{T})$ produce most succinct f-representations. They can be obtained by decreasing the length of root-to-leaf paths or equivalently by increasing branching while preserving the path constraint.

We next define $s(Q)$ as the minimal $s(\mathcal{T})$ over all f-trees $\mathcal{T}$ of Q. Then, for any database $\mathbf{D}$, there is an f-representation of $Q(\mathbf{D})$ with size at most $|\mathcal{P}| \cdot |\mathbf{D}|^{s(Q)}$, and this is asymptotically the best upper bound for f-representations over f-trees.

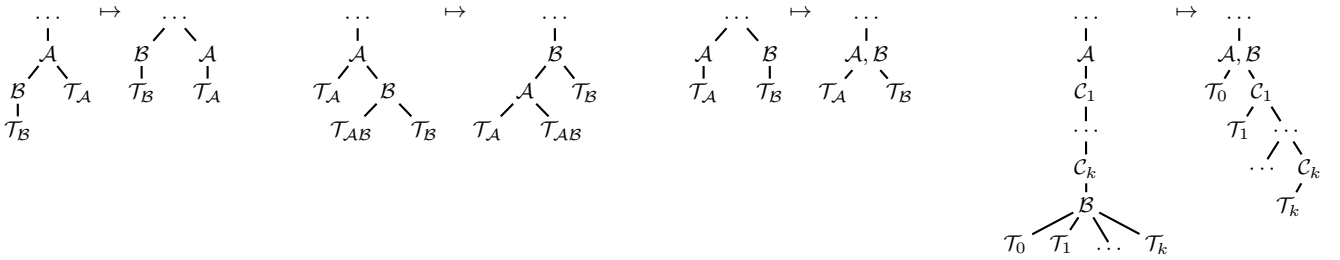
Example 5. In Example 1, we have $s(Q_1) = 2$ since Q_1 admits no f-tree with $s(\mathcal{T}) < s(\mathcal{T}_1) = 2$. However, $s(Q_2) = 1$, since $\mathcal{T}_3$ is an f-tree of Q_2 and $s(\mathcal{T}_3) = 1$. □

The size bound $|\mathcal{P}| \cdot |\mathbf{D}|^{s(Q)}$ can be asymptotically smaller than the size of the query result $Q(\mathbf{D})$. For such queries, computing and representing their result in factorised form can bring exponential time and space savings in comparison to the traditional flat representation as a set of tuples.

3. QUERY EVALUATION

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

We consider f-representations over f-trees as defined in Section 2. F-trees conveniently represent the structure of factorisations as well as attributes and equality conditions on the attributes. An f-tree uniquely determines (up to commutativity of $\cup$ and $\times$) the f-representation of a given relation. Therefore, the semantics of each of our operators may be described solely by the transformation of f-trees $\mathcal{T} \mapsto \mathcal{T}'$. We also present efficient algorithms to carry out the transformations on f-representations. These algorithms are almost



(a) push-up $\psi_{\mathcal{B}}$ (b) swap $\chi_{\mathcal{A},\mathcal{B}}$ (c) merge $\mu_{\mathcal{A},\mathcal{B}}$ (d) absorb $\alpha_{\mathcal{A},\mathcal{B}}$ (w/o full normalisation)

Figure 3: Transformations performed by f-plan operators depicted on f-trees.

optimal in the sense that they need at most quasilinear time in the sizes of both input and output f-representations.

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

We assume that for any union expression $\bigcup_a$ in the input f-representations, the values a occur in increasing order, and that the path constraint holds for the input f-tree. Our algorithms preserve these two constraints.

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

3.1 Restructuring Operators

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

Consider an f-tree $\mathcal{T}$, a node $\mathcal{A}$ and its child $\mathcal{B}$ in $\mathcal{T}$. If $\mathcal{A}$ is not dependent on $\mathcal{B}$ nor on its descendants, the subtree rooted at $\mathcal{B}$ can be brought one level up (so that $\mathcal{B}$ becomes sibling of $\mathcal{A}$) without violating the path constraint. Proposition 1 guarantees that there is an f-representation over the new f-tree. Lifting up a node can only reduce the length of root-to-leaf paths in $\mathcal{T}$ and thus decrease the parameter $s(\mathcal{T})$ and the size of the f-representation, cf. Section 2. The transformation only alters the structure of the factorisation, the represented relation remains unchanged.

Figure 3(a) shows the transformation of the relevant fragment of $\mathcal{T}$, where $\mathcal{T}_{\mathcal{A}}$ and $\mathcal{T}_{\mathcal{B}}$ denote the subtrees under $\mathcal{A}$ and $\mathcal{B}$. F-representations over this fragment have the form

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

and change into

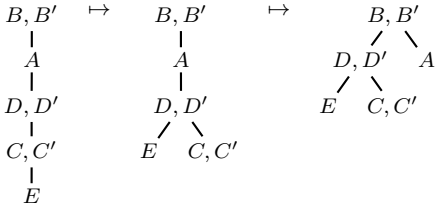
$$\Phi_2 = (\bigcup_b \langle \mathcal{B}{:}b\rangle \times F_b) \times (\bigcup_a \langle \mathcal{A}{:}a\rangle \times E_a),$$

where each E_a is over $\mathcal{T}_{\mathcal{A}}$, each F_b is over $\mathcal{T}_{\mathcal{B}}$, and $\langle \mathcal{A}{:}a\rangle$ stands for $\langle A_1{:}a\rangle \times \cdots \times \langle A_n{:}a\rangle$ in case A_1 to A_k are the attributes labelling node $\mathcal{A}$; the case of $\langle \mathcal{B}{:}b\rangle$ is similar. Since neither $\mathcal{B}$ nor any node in $\mathcal{T}_{\mathcal{B}}$ depend on $\mathcal{A}$, all copies of $(\bigcup_b \langle \mathcal{B}{:}b\rangle \times F_b)$ in Φ_1 are equal, so the transformation amounts to factoring out subexpressions over the subtree rooted at $\mathcal{B}$. In any f-representation over $\mathcal{T}$, the change shown above occurs for all unions over $\mathcal{A}$, and can be executed in linear time in one pass over the f-representation.

Definition 3. An f-tree $\mathcal{T}$ is *normalised* if no node in $\mathcal{T}$ can be pushed up without violating the path constraint.

Any f-tree $\mathcal{T}$ can be turned into a normalised one as follows. We traverse $\mathcal{T}$ bottom up and push each node $\mathcal{B}$ and its subtree upwards as far as possible using the operator $\eta_{\mathcal{B}}$. In case a node $\mathcal{A}$ is pushed up, we mark it so that we do not consider it again. If it is marked, so are all the nodes in its subtree, and at least one of them is dependent on the parent of $\mathcal{A}$ (or $\mathcal{A}$ is a root). The parent of $\mathcal{A}$ and the subtree of $\mathcal{A}$ do not change anymore after $\mathcal{A}$ is marked, so $\mathcal{A}$ cannot be brought upwards again. All nodes are marked after at most $|\mathcal{T}|^2$ applications of the push-up operator, so the resulting f-tree is normalised. Since the size of the f-representation over $\mathcal{T}$ decreases with each push-up, the time complexity of normalising an f-representation is linear in the size of the input f-representation. This procedure defines the normalisation operator η. In the remainder we only consider normalised f-trees and operators that preserve normalisation.

Example 6. Let us normalise the left f-tree below with relations over schemas $\{A, B\}, \{B', C\}, \{C', D\}, \{D', E\}$.



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

The Swap Operator $\chi_{\mathcal{A},\mathcal{B}}$ exchanges a node $\mathcal{B}$ with its parent node $\mathcal{A}$ in $\mathcal{T}$ while preserving the path constraint and normalisation of $\mathcal{T}$. We promote $\mathcal{B}$ to be the parent of $\mathcal{A}$, and also move up its children that do not depend on $\mathcal{A}$. The effect of the swapping operator $\chi_{A,B}$ on the relevant fragment of $\mathcal{T}$ is shown in Figure 3(b), where $\mathcal{T}_{\mathcal{B}}$ and $\mathcal{T}_{\mathcal{AB}}$ denote the collections of subtrees under $\mathcal{B}$ that do not depend, and respectively depend, on $\mathcal{A}$, and $\mathcal{T}_A$ denotes

foreach expression S_{in} over the part of $\mathcal{T}$ in Figure 3(b) **do**
  create a new union S_{out}
  let Q be a min-priority-queue
  foreach $\langle \mathcal{A}{:}a \rangle \times E_a \times \bigcup_b (\langle \mathcal{B}{:}b \rangle \times F_b \times G_{ab})$ in S_{in} **do**
    let U_a be the union $\bigcup_b (\langle \mathcal{B}{:}b \rangle \times F_b \times G_{ab})$
    let p_a be the first value b in the union U_a
    insert value a with key p_a into Q
  while Q is not empty **do**
    let b_{min} be the minimum key in Q
    create a new union $V_{b_{min}}$
    foreach a in Q with key b_{min} **do**
      append $\langle \mathcal{A}{:}a \rangle \times E_a \times G_{ab}$ to $V_{b_{min}}$
      remove a from Q
      if p_a is not the last value in U_a **then**
        update p_a to be the next value b in the union U_a
        insert value a with key p_a into Q
    append $\langle \mathcal{B}{:}b_{min} \rangle \times F_{b_{min}} \times V_{b_{min}}$ to S_{out}
  replace S_{in} by S_{out}

Figure 4: Algorithm for the swap operator $\chi_{\mathcal{A},\mathcal{B}}$.

the subtree under $\mathcal{A}$. Separate treatment of the subtrees $\mathcal{T}_{\mathcal{B}}$ and $\mathcal{T}_{\mathcal{AB}}$ is required so as to preserve the path constraint and normalisation. The resulting f-tree has the same nodes as $\mathcal{T}$ and the represented relation remains unchanged.

Any f-representation over the relevant part of the input f-tree $\mathcal{T}$ in Figure 3(b) has the form

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

while the corresponding restructured f-representation is

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

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

The swap operator $\chi_{\mathcal{A},\mathcal{B}}$ thus takes an f-representation where data is grouped first by $\mathcal{A}$ then $\mathcal{B}$, and produces an f-representation grouped by $\mathcal{B}$ then $\mathcal{A}$. Figure 4 gives an algorithm for $\chi_{\mathcal{A},\mathcal{B}}$ that executes this regrouping efficiently. We use a priority queue Q to keep for each value a of attributes in $\mathcal{A}$ the minimal values b of attributes in $\mathcal{B}$. This minimal value occurs first in the union U_a due to the order constraint of f-representations. We then extract the values b from the priority queue Q in increasing order to construct the union over them, and for each of them we obtain the pairing values a. When a value a is removed from Q, we insert it back into Q with the next value b in its union U_a.

Except for the operations on the priority queue, the total time taken by the algorithm in any given iteration of the outermost loop is linear in the size of the input S_{in} plus the size of the output S_{out}. For each a in S_{in} and b in U_a, the value a is inserted into the queue with key b once and removed once. There are at most $|S_{in}|$ such pairs (a, b) and each of the priority queue operations runs in time $O(\log |S_{in}|)$.

Example 7. The tree $\mathcal{T}_1$ in Figure 2 is transformed into $\mathcal{T}_2$ by the operator $\chi_{\text{item,location}}$. The effect of the operator on the f-representation amounts to regrouping it primarily by location instead of item, as illustrated in Example 1. □

3.2 Cartesian Product Operator

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

3.3 Selection Operators

We next present operators for selections with equality conditions of the form $A = B$. Since equality joins are equivalent to equality selections on top of products, and the product of f-representations is just their concatenation, we can evaluate equality joins in the same way as equality conditions on attributes of the same relation, and do not distinguish between these two cases in the sequel.

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

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

The Merge Selection Operator $\mu_{\mathcal{A},\mathcal{B}}$ merges the sibling nodes $\mathcal{A}$ and $\mathcal{B}$ of $\mathcal{T}$ into one node labelled by the attributes of $\mathcal{A}$ and $\mathcal{B}$ and whose children are those of $\mathcal{A}$ and $\mathcal{B}$, see Figure 3(c). This operator preserves the path constraint, since the root-to-leaf paths in $\mathcal{T}$ are preserved in the resulting f-tree. Also, normalisation is preserved: merging two nodes of a normalised f-tree produces a normalised f-tree. To preserve the value order constraint, node merging is implemented as a sort-merge join. Any f-representation over the relevant part of $\mathcal{T}$ has the form

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

and change into

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

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

Example 8. Consider an f-tree that is the forest of $\mathcal{T}_1$ and $\mathcal{T}_4$ from Figure 2. The two attributes with the same name item are siblings (at the topmost level). By merging them, we obtain the f-tree $\mathcal{T}_5$. Example 1 shows f-representations over the input and output f-trees of this merge operation.□

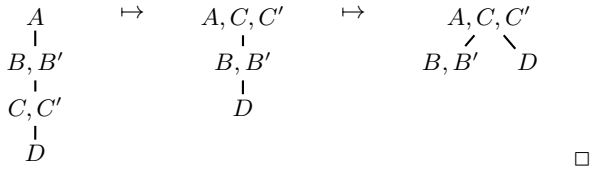
The Absorb Selection Operator $\alpha_{\mathcal{A},\mathcal{B}}$ absorbs a node $\mathcal{B}$ into its ancestor $\mathcal{A}$ in an f-tree $\mathcal{T}$, and then normalises the resulting f-tree. The labels of $\mathcal{B}$ become now labels of $\mathcal{A}$.

The absorption of $\mathcal{B}$ into $\mathcal{A}$ preserves the path constraint since all attributes in $\mathcal{B}$ remain on the same root-to-leaf paths. By definition, the absorb operator finishes with a normalisation step, thus it preserves the normalisation constraint. Similar to the merge selection operator, it employs sort-merge join on the values of $\mathcal{A}$ and $\mathcal{B}$ and hence creates f-representations that satisfy the order constraint.

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

To normalise the f-tree after absorbing $\mathcal{B}$ into $\mathcal{A}$, we use the normalisation operator η. If the original tree was normalised, it is sufficient to push up the subtrees of $\mathcal{B}$ as shown in Figure 3(d), and possibly also push up some of the nodes $\mathcal{C}_1, \ldots, \mathcal{C}_k$ on the path between $\mathcal{A}$ and $\mathcal{B}$.

Example 9. Consider the selection $A = C$ on the leftmost f-tree below with relations over schemas $\{A, B\}$, $\{B', C\}$ and $\{C', D\}$. Since A and C correspond to ancestor and respectively descendant nodes, we can use the absorb operator to enforce the selection. When absorbing $\{C, C'\}$ into A (middle f-tree), the nodes $\{B, B'\}$ and D become independent and D can be pushed upwards (right f-tree):



□

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

For a comparison θ different from equality, the f-tree remains unchanged. In case of equality, we can infer that all A-values in the f-representation are equal to c and thus the node $\mathcal{A}$ labelled by A is independent of the other nodes in the f-tree and can be pushed up as the new root. When computing the parameter $s(\mathcal{T})$, we can ignore $\mathcal{A}$ since the only f-representation over it is the singleton $\langle A\!:\!c\rangle$.

3.4 Projection Operator

We define a projection operator π_{-A}, which projects away the attribute A, i.e., it acts as a projection $\pi_{\mathcal{S}\setminus\{A\}}$ on the represented relation of schema $\mathcal{S}$. A relational projection $\pi_{\mathcal{P}}$ can be evaluated by a sequence of projection operators, one for each attribute in $\mathcal{S} \setminus \mathcal{P}$.

In an f-representation E, π_{-A} replaces all A-singletons $\langle A\!:\!a\rangle$ with the nullary singleton $\langle\rangle$. Subsequently, any union of nullary singletons is replaced by one nullary singleton and any nullary singleton in a product with other singletons is removed from E. This procedure can be performed in one scan over the input f-representation E and trivially preserves the order constraint. In the corresponding f-tree $\mathcal{T}$, π_{-A} removes the attribute A. If there is no further attribute labelling the node of A, we also remove that node.

We only permit the operator π_{-A} when there are further attributes labelling the node of A or when A labels a leaf. By removing this node, any two attributes B and C previously dependent on A now become dependent on each other. If A is a leaf, such attributes lie on a path from A to root, and the path constraint is thus preserved. If A is at an inner node, then B and C may be on different branches under A and the path constraint is violated. In this latter case, π_{-A} is therefore not permitted and the f-representation must be first restructured using the swap operator.

4. QUERY OPTIMISATION

In this section, we discuss the problem of query optimisation for queries on f-representations. In addition to the optimisation objective present in the standard (flat) relational case, namely finding a query plan with minimal cost, the nature of factorised data calls for a new objective: from the space of equivalent f-representations for the query result, we would like to find a small, ideally minimal, f-representation.

The operators described in Section 3 can be composed in f-plans that implement select-project-join queries on f-representations over f-trees. Several f-plans may exist for a given query. In this section we introduce different cost measures for f-plans and algorithms for finding optimal ones.

The products and selections with constant are the cheapest and can be evaluated first. Projections can always be evaluated last, but evaluating a projection earlier (if permitted) may speed up subsequent operators. Most expensive are the equality selection operators and the restructuring operators that make selections and projections possible. Their evaluation order is addressed in this section.

A selection $A = B$ can only be executed on an f-representation over an f-tree $\mathcal{T}$ if the attributes A and B label nodes $\mathcal{A}$ and respectively $\mathcal{B}$ that are either the same, siblings, or along a same path in $\mathcal{T}$. Otherwise, we first need to transform the f-representation. If $\mathcal{A}$ and $\mathcal{B}$ are in the same tree, we can e.g. repeatedly swap $\mathcal{A}$ with its parent until it becomes an ancestor of $\mathcal{B}$. Alternatively, we can push up both $\mathcal{A}$ and $\mathcal{B}$ until they become siblings (or both roots in separate f-trees). To complete the evaluation, we apply a merge or absorb selection operator on the two nodes $\mathcal{A}$ and $\mathcal{B}$.

There are several choices involved in the search for a good f-plan: For each selection, should we transform the input f-tree, and consequently the f-representation, such that the nodes $\mathcal{A}$ and $\mathcal{B}$ become siblings or one the ancestor of the other? Is it better to push up $\mathcal{A}$ or $\mathcal{B}$? What is the effect of a transformation for one selection on the remaining selections? What should the relative order of selections and projections be? Finally, if the input is given in flat form, which initial f-tree is best for its conversion into an f-representation? The aim of FDB's optimiser is to find an f-plan for the given query such that the maximal cost of the sequence of transformations is low and the query result is well-factorised.

4.1 Cost of an F-Plan

We next define two cost measures for f-plans. One measure is based on the parameter $s(\mathcal{T})$ that defines size bounds on factorisations over f-trees for any input database. The

second measure is based on cardinality estimates inferred from the intermediate f-trees and on the database catalogue with information on relation sizes and selectivity estimates. Both measures can be used by the exhaustive search procedure and the greedy heuristic for query optimisation presented later in this section.

Cost Based on Asymptotic Bounds. As discussed in Section 2, the size of any f-representation over an f-tree $\mathcal{T}$ depends exponentially on the parameter $s(\mathcal{T})$, i.e., its size is in $O(|\mathbf{D}|^{s(\mathcal{T})})$. Since the cost of each operator is quasilinear in the sum of sizes of its input and output, the parameter $s(\mathcal{T})$ dictates this cost. For an f-plan $f = \omega_1, \ldots, \omega_k$ that performs the following sequence of transformations:

$$\mathcal{T}_{\text{initial}} = \mathcal{T}_0 \overset{\omega_1}{\mapsto} \mathcal{T}_1 \overset{\omega_2}{\mapsto} \ \ldots \ \overset{\omega_k}{\mapsto} \mathcal{T}_k = \mathcal{T}_{\text{final}},$$

the evaluation time is $O(|\mathbf{D}|^{s(f)} \cdot \log |\mathbf{D}|)$, where

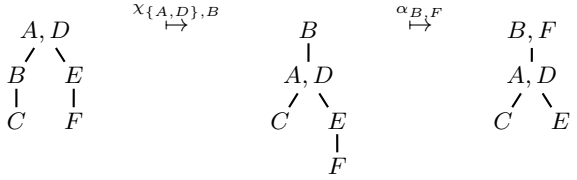
$$s(f) = \max(s(\mathcal{T}_0), s(\mathcal{T}_1), \ldots, s(\mathcal{T}_k)).$$

The sizes of the intermediate f-representations thus dominate the execution time. Using this cost measure, a good f-plan is one whose intermediate f-trees $\mathcal{T}_i$ have small $s(\mathcal{T}_i)$.

In defining a notion of optimality for f-plans, we would like to optimise for two objectives, namely minimise the f-plan cost $s(f)$ and the cost of the final result $s(\mathcal{T}_{\text{final}})$. However, it might not be possible to optimise for both objectives $<_{s(f)}$ and $<_{s(\mathcal{T})}$ at the same time. Instead, we prioritise $s(f)$ over $s(\mathcal{T}_{\text{final}})$ and use the lexicographic order $<_{s(f)} \times <_{s(\mathcal{T})}$ on f-plans: given f-plans f_1 and f_2, we consider f_1 better than f_2 and write $f_1 <_{s(f)} \times <_{s(\mathcal{T})} f_2$ if either (1) the most expensive operator in f_1 is less expensive than the most expensive operator in f_2, or (2) their most expensive operators have the same cost but the cost of the result is smaller for f_1. An f-plan f_1 for a query Q is *optimal* if there is no other f-plan f_2 for Q such that $f_2 <_{s(f)} \times <_{s(\mathcal{T})} f_1$.

This notion of optimality is over f-plans consisting of operators defined in Section 3. Since these operators preserve f-tree normalisation, this also means that we consider optimality only over the space of possible normalised f-trees.

Example 10. Consider the following f-plan evaluating the selection $B = F$ on the leftmost f-tree, with dependencies $\{A, B, C\}$ and $\{D, E, F\}$.



The input f-tree and the output f-tree have both cost 1, as each root-to-leaf path is covered by a single relation. However, the intermediate f-tree has cost 2 (as on the path from B to F each of B and F must be covered by a separate relation), so the cost of the f-plan is 2. An alternative f-plan starts by swapping F with its parent to obtain an intermediate f-tree with cost 1, and then merges F with B.

Although both f-plans result in an f-tree with cost 1, the latter f-plan has cost 1 while the former has cost 2. □

Cost Based on Estimates. We can also estimate the cost of an f-plan using cardinalities and join selectivities for the input f-representation E. One approach is to keep cardinalities and selectivities for the relation $\mathbf{R}_E$ represented by E, in which case we fall back to relational catalogues. A different approach would take advantage of the conditional independence of attributes, as specified by the input f-tree, to more accurately record join selectivities and cardinalities (e.g., how many A-singletons are in average under a B-singleton). This latter approach is subject to future work.

Given a query Q on the input f-representation E, we can estimate the size of the f-representation F of the query result over an f-tree $\mathcal{T}$ as follows. Let $\mathbf{R}_F$ be the relation represented by F, i.e., $\mathbf{R}_F = Q(\mathbf{R}_E)$. The number of A-singletons in F is equal to the size of the relation $\pi_{\text{path}(A)}\mathbf{R}_F$, where $\text{path}(A)$ is the set of attributes along the path from the root to the node of A in $\mathcal{T}$ [19]. The cardinality of $\pi_{\text{path}(A)}\mathbf{R}_F = \pi_{\text{path}(A)}(Q(\mathbf{R}_E))$ can now be computed using known techniques for relational databases, or alternatively using factorisation-aware cardinalities and selectivities as mentioned above. The size of the f-representation of $Q(\mathbf{R}_E)$ over $\mathcal{T}$ is then estimated as $\sum_{A \in \mathcal{P}} |\pi_{\text{path}(A)}\mathbf{R}_F|$, where $\mathcal{P}$ is the projection list of Q, and the cost $e(f)$ of an f-plan f can be estimated as the sum of the size estimates for the intermediate and final f-trees created by f [22].

4.2 Exhaustive Search

To find an optimal f-plan for a select-project-join query we search the space of all possible normalised f-trees and all possible operators between the f-trees. We represent this space by a directed graph where f-trees are nodes and operators are edges. Given a query Q on an input over an f-tree $\mathcal{T}_{in}$, any f-plan for Q is represented by a path from $\mathcal{T}_{in}$ to an f-tree $\mathcal{T}_{final}$ of Q, i.e., such that the equivalence classes of $\mathcal{T}_{final}$ are the classes of $\mathcal{T}_{in}$ extended by Q's equality conditions and restricted to Q's projection list. In the case of flat input, instead of one possible initial f-tree $\mathcal{T}_{in}$, there is a set of possible choices for the initial f-tree of the f-representation of the input, namely all paths of nodes labelled by single attributes of the input schema.

Both the asymptotic cost function and the cost estimate for f-plans define a distance function on the space of f-trees: the distance from $\mathcal{T}_1$ to $\mathcal{T}_2$ is the minimum possible cost of an f-plan from $\mathcal{T}_1$ to $\mathcal{T}_2$. We are thus searching for f-trees $\mathcal{T}_{final}$ of Q, which (1) are closest to $\mathcal{T}_{in}$ and (2) have the smallest possible cost. An optimal f-plan is then a shortest path from $\mathcal{T}_{in}$ to a $\mathcal{T}_{final}$ with smallest cost. We can use Dijkstra's algorithm to find distances from $\mathcal{T}_{in}$ to all f-trees: we explore the space starting with $\mathcal{T}_{in}$ using all allowed operators, processing the reached f-trees in order of increasing distance to $\mathcal{T}_{in}$. Then, among all f-trees $\mathcal{T}_{final}$ of Q with the shortest distance to $\mathcal{T}_{in}$, we pick one with smallest cost.

The complexity of the search is determined by the size of the search space. By successively applying operators to $\mathcal{T}_{in}$, we rearrange its nodes (swap) or modify the nodes (merge, absorb, projection). For each partition of attributes over nodes, there is a cluster of f-trees with the same nodes but different shape, among which we can move using the swap operator. By applying a merge or absorb operator, we move to a cluster whose f-trees have one fewer node. By applying a projection, we move to a cluster whose f-trees have one attribute less and possibly one node less. Consider a query on an input schema of n attributes with k equality conditions

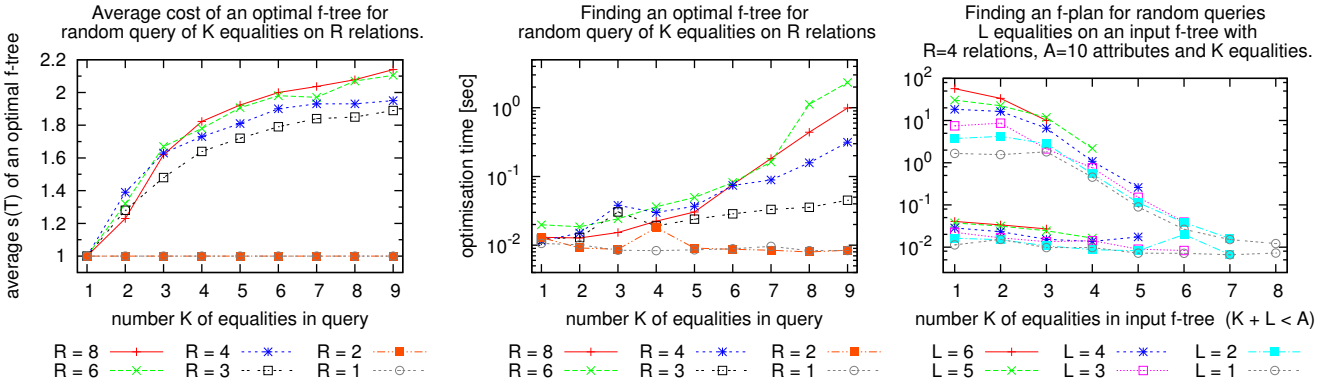


Figure 5: Experiment 1 (left and centre): Query optimisation on flat data, K equalities on R relations with $A = 40$ attributes. Experiment 2 (right): Query optimisation on factorised data, L equalities on f-tree with K equalities, time performance of full search (slower, top series) and greedy optimiser (faster, bottom series).

and $n-p$ attributes in the projection list. Any valid sequence of operators contains at most k of the at most $\binom{k+1}{2} \leq k^2$ possible merge or absorb operators, and any of the p projection operators. Since all sequences with the same operators (though in different orders) end up in the same cluster, we have $O(k^{2k}2^p)$ reachable clusters. In a cluster with m nodes there are at most m^m f-trees, and since $m, k, p \leq n$, the size of the search space is $O(k^{2k}2^p n^n) = O(n^{4n})$.

4.3 Greedy Heuristic

Our greedy optimisation algorithm restricts the search space for f-plans in two dimensions: (1) it only applies restructuring operators to nodes that participate in selections and projections, and (2) it considers a greedy approach to selection and projection ordering, whereby at each step it chooses an operator with the least cost. In case of flat input, the initial f-tree is chosen as any path of nodes labelled by the attributes in the input schema such that the attributes participating in selections form a prefix of the path and the attributes that do not participate in selections and are not in the projection list form a suffix of the path.

The algorithm constructs an f-plan f as follows. For each selection with two attributes labelling nodes $\mathcal{A}$ and $\mathcal{B}$, we consider three restructuring scenarios: swapping one of the nodes $\mathcal{A}$ and $\mathcal{B}$ until $\mathcal{A}$ becomes the ancestor of $\mathcal{B}$ or the other way around, or bringing both $\mathcal{A}$ and $\mathcal{B}$ upwards until they become siblings. We choose the cheapest f-plan for each selection. We consider the projection π_{-A} for each attribute A not in the projection list or further selections. In case A is the only attribute labelling its node $\mathcal{A}$, we precede π_{-A} by repeated swaps of $\mathcal{A}$ with its child with the smallest subtree until $\mathcal{A}$ becomes a leaf. We then order the projections and selections by the cost of their f-plans, choose the one with the cheapest f-plan, and append its f-plan to the overall f-plan f. We repeat this process with the remaining selections and projections until we finish them. The new input f-tree is now the resulting f-tree of the f-plan obtained so far.

In contrast to the full search algorithm, this greedy algorithm takes only polynomial time in the size of the input f-tree $\mathcal{T}$. For each selection and projection, there can be at most $O(|\mathcal{T}|)$ swaps. Each swap requires to look at all descendants of the swapped nodes to check for independence and to find the smallest subtree in case of projections. Computing the resulting f-tree thus takes time $O(|\mathcal{T}|^2)$.

5. EXPERIMENTAL EVALUATION

We evaluate the performance of our query engine FDB against three relational engines: one home-bred in-memory (RDB) and two open-source engines (SQLite and PostgreSQL). Our main finding is that FDB clearly outperforms relational engines for data sets with many-to-many relationships. In particular, in our experiments we found that:

- The size of factorised query results is typically at most quadratic in the input size for queries of up to eight relations and nine join conditions (Figure 5 left).
- Finding optimal f-trees for results of queries of up to eight relations and six join conditions takes under 0.1 seconds (Figure 5 middle). Finding optimal f-plans for queries on factorised data is about an order of magnitude slower. In contrast, the greedy optimiser takes under 5 ms (Figure 5 right) without any significant loss in the succinctness of factorisation (Figure 6).
- For queries on flat data, factorised query results are two to six orders of magnitude smaller than their flat equivalents and FDB outperforms RDB by up to four orders of magnitude (Figure 7 middle). The gap increases with increasing data size (Figure 7 left). For the same workload SQLite performed about three times slower than RDB, and PostgreSQL performed three times slower than SQLite; both systems have additional overhead of fully functioning engines. The gap between RDB and SQLite increases with the number of joins (Figure 7 bottom right) as RDB implements an optimised multi-way merge-sort join.
  This holds for uniform and Zipf data distributions.
- The evaluation of subsequent queries on input data representing query results has the same time performance gap, since the new input is more succinct as factorised representation than as flat relation (Figure 8).
- For one-to-many relationships, the performance gap is smaller, since even the flat result sizes for one-to-many joins only depend linearly on the input size and the possible gain brought by factorisation is less dramatic. For instance, in the TPC-H benchmark all joins are on keys and therefore the sizes of the join results do not exceed that of the relation with foreign keys. Factorised query results are still more succinct than their relational representations, but only by a factor bounded by the query size (Figure 7 top right).

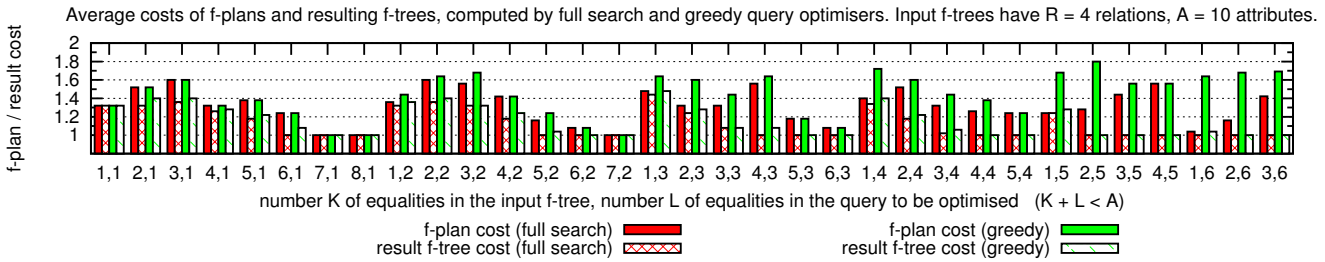


Figure 6: Experiment 2: Comparison of full-search and greedy query optimisers.

Implementation Details. We implemented FDB and RDB in C++ for execution in main memory. The parameter $s(\mathcal{T})$ is computed using the GLPK package v4.45 for solving linear programs. F-representations are stored as parse trees. The current version of FDB implements the evaluation and optimisation algorithms described in previous sections with the following exceptions. For evaluation on flat data it computes the result f-representation using a multi-way merge-sort join algorithm with the order of joins given by an optimal f-tree [19]. This join algorithm is also used by RDB, though producing a flat result this time. The FDB optimiser defers projections until the end (we only focus on the impact of joins). Both FDB and RDB sort flat input relations before executing merge-sort joins; the sorting time is included in the reported evaluation time.

Competing Engines. We also used SQLite 3.6.22 tuned for main memory operation by turning off the journal mode and synchronisations and by instructing it to use in-memory temporary store. Similarly, we run PostgreSQL 9.1 with the parameters: fsync, synchronous commit, and full page writes are off, no background writer, shared buffers and working memory increased to 12 GB. Both SQLite and PostgreSQL read the data in their internal binary format, whereas FDB and RDB use the plain text format.

Experimental Setup. All experiments were performed on an Intel(R) Xeon(R) X5650 Quad 2.67GHz/64bit/32GB running VMWare VM with Linux 2.6.32/gcc4.4.5. For all engines we report wall-clock times (averaged over five runs) to read data from disk and execute the query plans without writing the result to disk.

Experimental Design. The flow of our experiments is as follows. We generate random data and queries, then repeat a number of times four optimisation and evaluation experiments and report averages of optimisation time, execution time, representation sizes, and quality of produced f-plans.

We generate a schema of R relations and distribute uniformly A attributes over them. Each relation has a given number of tuples, each value is generated independently from $\{1,\ldots,M\}$ using a uniform or Zipf distribution. The queries are equality joins over all of these relations. We first generate an initial query Q with K non-redundant equality conditions and then a further query Q' with L additional non-redundant equality conditions.

For each generated initial query Q and database $\mathbf{D}$, we do the following. In Experiment 1, we run the FDB optimiser to find an optimal f-tree $\mathcal{T}$ for the query result and report the optimisation time and the value of the parameter $s(Q)$ that controls the size of the f-representation of $Q(\mathbf{D})$ over $\mathcal{T}$. In Experiment 3, we compute the result $Q(\mathbf{D})$ using RDB, SQLite, and PostgreSQL, and the factorised query result using FDB. We then report on both the evaluation time and size of the result as the number of its singletons; a singleton holds an 8 byte integer. We also unfold the factorised result of FDB into a flat relation for correctness check, and report the unfolding (or tuple enumeration) time.

In Experiments 2 and 4, we evaluate the second set of queries on top of the query results from Experiment 3. For each new query Q', we run the FDB optimiser to find an optimal f-plan to evaluate the query and the resulting f-tree of the query result. In Experiment 2, we report the optimisation time and cost of the found f-plans for both the exhaustive and greedy optimisation algorithms. Here, we consider the cost of the f-plan defined by the parameter $s(\cdot)$ of the intermediate and final f-trees; in our experiments, the alternative cost estimate $e(\cdot)$ would lead to very similar choices of optimal f-plans since our data generator treats attributes independently and introduces no dependencies or irregularities. In Experiment 4, we execute f-plans with FDB on the f-representations of the query results (and with RDB on the flat query results) computed in Experiment 3.

We also repeat Experiments 1 and 3 with TPC-H data and queries. We considered the conjunctive parts of TPC-H queries 2, 5, 7, 10 and compare the evaluation times of FDB, RDB, and SQLite on flat TPC-H databases of varying scale.

Experiment 1: Query optimisation on flat data.

The left and middle plots in Figure 5 show average times for optimising a query on flat data, and average costs $s(\mathcal{T})$ for the chosen optimal f-tree $\mathcal{T}$ of the query result. For schemas with $A = 40$ attributes over $R = 1,\ldots,8$ relations, we optimised queries of $K = 1,\ldots,9$ equality selections. The cost $s(\mathcal{T})$ of an optimal f-tree $\mathcal{T}$ is always 1 for queries of up to two relations. For $R \geq 3$ and a sufficient number of joins we often get queries with optimal $s(\mathcal{T}) = 2$ and in very rare cases $s(\mathcal{T}) > 2$. This means that the sizes of f-representations for the query results are in most cases quadratic in the size of the input database even in the case of 9 equality selections on 8 relations. The optimiser searches a potentially exponentially large space of f-trees to find an optimal one, but runs under 1 second for queries with less than 8 joins on up to 8 relations.

Experiment 2: Query optimisation on factorised data.

Figure 6 shows the behaviour of query optimisation for factorised data. It shows the costs of the computed f-plans as well as the costs of the f-tree of the result computed by the f-plans for our full-search and greedy optimisation algorithms.

The queries under consideration have $L \leq 6$ equality conditions on f-representations resulting after $K \leq 8$ equality conditions on $R = 4$ relations with $A = 10$ attributes. The greedy optimiser gives optimal or nearly optimal results in most cases (by comparison with the optimal outcome of full search). The exceptions are queries joining most attributes of an f-representation produced by a query with few joins

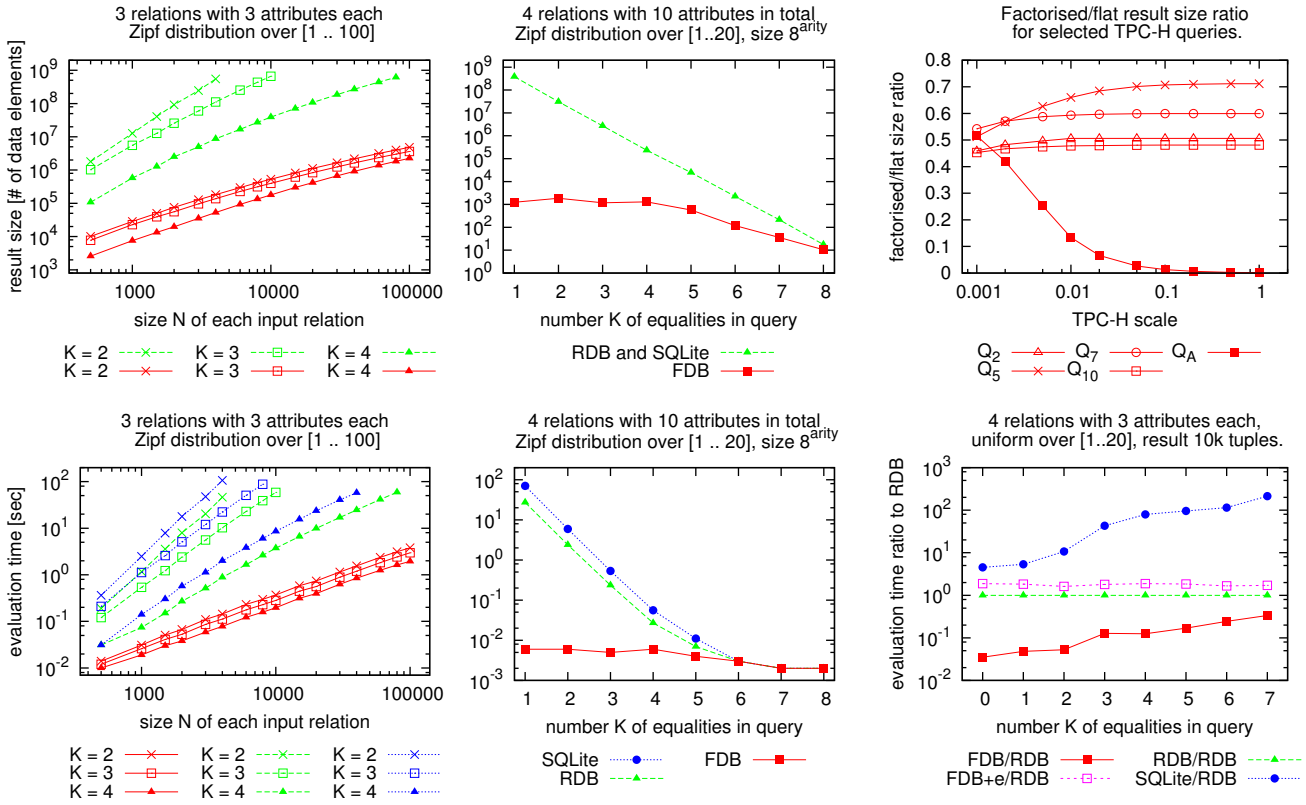


Figure 7: Experiment 3: Query evaluation on flat relational data: FDB in solid red, RDB in dashed green, SQLite in dotted blue, FDB with tuple enumeration in dashed pink. PostgreSQL is ca. 3 times slower than SQLite and not shown. Right column shows ratios to RDB (times and result sizes).

(small K, large L). In all cases the average f-plan cost is between 1 and 2, which means that the f-plans take at most quadratic time for processing a join of up to 4 relations. The results show that for small queries (small L) the cost of the optimal f-plan is dominated by the cost of the final f-tree. As the number of equalities L grows, the result f-tree has less attribute classes and its cost is smaller than the cost of the f-plan, which is the maximum cost of intermediate f-trees needed to evaluate the query.

The right plot in Figure 5 shows the times for both exhaustive (full search) and greedy optimisers. The search space of possible f-trees grows exponentially with the number L of equalities and also with the size of the input f-tree (i.e., with decreasing K). The performance of the full-search algorithm is proportional to the size of the search space; we process about 1k f-trees/second. The greedy heuristic has running time polynomial in both K and L, and in our scenario is 2-3 orders of magnitude faster than full search.

Experiment 3: Query evaluation on flat data.

We compared the performance of FDB, RDB, SQLite, and PostgreSQL for query evaluation on flat input data. The left column of Figure 7 shows the result sizes and evaluation times for queries with up to four equality conditions on three ternary relations of increasing sizes, generated using a Zipf distribution over the range $[1, 100]$.

The size gap between factorised and relational results is largest for queries with fewer equality conditions, since the results are larger yet factorisable. The plots support the claim that the sizes are bounded by a power law, with a smaller exponent for FDB than for the relational engines.

The middle column in Figure 7 considers queries with increasing number of equality conditions over two binary relations of size $8^2 = 64$ and two ternary relations of size $512 = 8^3$, whose values are drawn from a Zipf distribution over $[1, 20]$. Each equality condition in the query decreases the number of result tuples approximately by a factor of 20, which is exhibited in the flat result size produced by RDB. FDB factorises the up to 500M data values into less than 4k singletons for all considered queries. The execution time for all engines is approximately proportional to their result sizes except for the millisecond region, where constant overhead dominates. For all plots, results for uniformly distributed data exhibit the same trends as those for Zipf-distributed data and are not shown due to lack of space.

The bottom right plot in Figure 7 considers a similar scenario, but the input size is increased with the number of equality conditions to keep the output size constant at about 10k tuples. We show the evaluation time ratio with respect to RDB. FDB outperforms RDB by one to two orders of magnitude. As the number of equality conditions increases, $s(Q)$ increases (cf. Figure 5 left), the query result becomes less factorisable and the size and time performance gaps decrease. Evaluation using FDB followed by enumerating all tuples of the factorised result is only by a small constant factor slower than direct evaluation using RDB. In both cases, most of the evaluation time is spent on writing the result to memory. SQLite performs significantly worse than RDB (and FDB) for queries with many equality conditions since

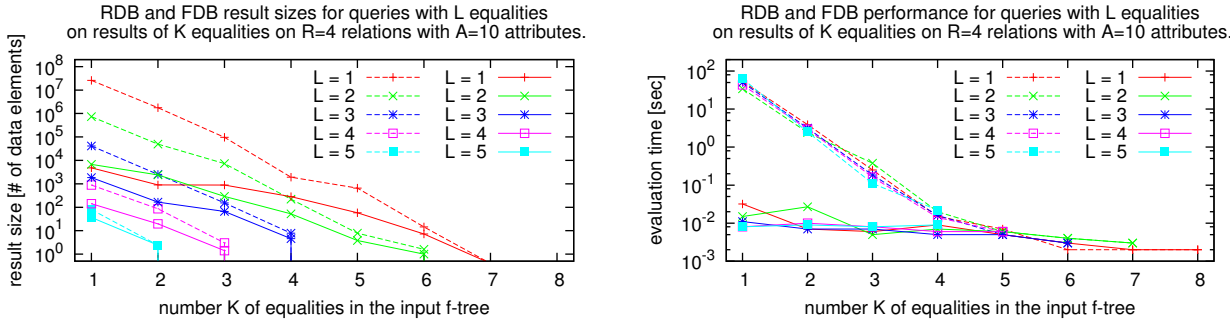


Figure 8: Experiment 4: Performance of FDB for query evaluation on factorised data (solid lines, bottom series in the right plot) and RDB on flat data (dashed lines, top series in the right plot). RDB needs one scan over the input relation, whereas FDB may need to restructure the factorised input. SQLite has similar performance to RDB.

it only implements in-memory nested-loops joins while RDB implements a multi-way merge-sort join.

For TPC-H data and queries, the compression factor for factorised output (versus flat output) remains roughly constant for each query even with increasing database size (top right plot in Figure 7). Since all joins are key-foreign key joins, the query result also has a primary key, and each distinct value of this key must appear in the factorisation: hence the factorisation cannot be asymptotically smaller than the result itself. This behaviour is contrasted by the query Q_A = Suppliers $\bowtie_{\text{Nation}}$ Customers whose join is many-to-many and the factorised/flat size ratio decreases inversely with database scale (reaches 1/800 for scale 1).

Experiment 4: Query evaluation on factorised data.

Figure 8 compares the performance of FDB and RDB for query evaluation on query results computed in Experiment 3 and with f-plans computed in Experiment 2. The behaviour of SQLite and PostgreSQL closely follows that of RDB.

FDB evaluates queries consisting of L selections on factorised representations. FDB uses the optimal f-plan found by the full-search optimiser. Additional experiments (not reported) reveal that the f-plans found by the greedy optimiser are up to 50% slower than the optimal f-plans.

RDB just evaluates a selection with a conjunction of L equality conditions on the attributes of the input relation. This can be done in one scan over the input relation. For FDB, the cost of the f-plan may be non-trivial: the more the f-plan needs to unfold the f-representation, the more expensive the evaluation becomes. Figure 8 suggests that FDB only unfolds the f-representations to a small extent. Similar to query evaluation on flat data, FDB shows up to 4 orders of magnitude improvement over RDB for both evaluation time and result size. The gap closes once the size of the input data decreases to about 1000 tuples and both FDB and RDB perform in under 0.1 seconds.

Experiments 2 and 4 show that using f-representations for data processing is sustainable in the sense that the quality of factorisations, in particular their compactness and sizes, does not decay with the number of operations on the data.

6. REFERENCES

[1] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *JCSS*, 33(3), 1986.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004.

[4] L. Antova, C. Koch, and D. Olteanu. 10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information. In *ICDE*, 2007.

[5] M. Aschinger, C. Drescher, G. Gottlob, P. Jeavons, and E. Thorstensen. Tackling the partner units configuration problem. In *IJCAI*, 2011.

[6] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, 2007.

[7] F. Bancilhon, P. Richard, and M. Scholl. On line processing of compacted relations. In *VLDB*, 1982.

[8] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[9] R. K. Brayton. Factoring logic functions. *IBM J. Res. Develop.*, 31(2), 1987.

[10] M. C. F. D. J. Abadi, S. R. Madden. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[11] G. Gottlob. On minimal constraint networks. In *CP*, 2011.

[12] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[13] M. Grohe and D. Marx. Constraint solving via fractional edge covers. In *SODA*, 2006.

[14] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - a main memory hybrid storage engine. *PVLDB*, 4(2), 2010.

[15] T. Imielinski, S. Naqvi, and K. Vadaparty. Incomplete objects — a data model for design and planning applications. In *SIGMOD*, 1991.

[16] D. Olteanu and J. Huang. Using OBDDs for efficient query evaluation on probabilistic databases. In *SUM*, 2008.

[17] D. Olteanu, C. Koch, and L. Antova. World-set decompositions: Expressiveness and efficient algorithms. *TCS*, 403(2-3), 2008.

[18] D. Olteanu and J. Závodný. On factorisation of provenance polynomials. In *TaPP*, 2011.

[19] D. Olteanu and J. Závodný. Factorised representations of query results: Size bounds and readability. In *ICDT*, 2012.

[20] M. Pöss and D. Potapov. Data compression in Oracle. In *VLDB*, 2003.

[21] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[22] S. Wyleżoł. Cost-based Query Optimisation for Factorised Relational Databases. Master's thesis, University of Oxford, 2012.