Cost-based Query Optimisation for Factorised Relational Databases

4th Year Project Report

Szymon Wyleżoł, Merton College

Supervisor: Dan Olteanu

May 20, 2012

Abstract

Factorised representations of relational data introduced by Olteanu and Závodný [9] allow for succinct representations of relations. FDB, a recently introduced in-memory query engine for queries on factorised databases, has shown significant performance gains in query execution compared to a typical relational engine [1].

This project introduces the concept of cost-based query opmitisation to factorised representations and extends FDB with a query optimiser based on query result size estimation techniques. The report begins with background information on the concept of factorised representations and a review of selectivity and cardinality estimation techniques. The focus of the project is to find good query plans for queries on factorised databases by using size estimation of factorised representations in query optimisation. The result is a query optimiser based on a size-estimation cost model which can find efficient factorisation plans for queries. An overview of design and implementation of the optimiser is presented. A range of tests is performed to assess the quality of the estimates and the performance of the optimiser. The findings indicate that for uniformly distributed data the esimates are of high accuracy and that the estimates-based optimiser finds plans with lower cost compared with the state-of-the-art FDB optimiser.

Contents

1 Introduction

Database management systems, and their relational variant in particular, have been a subject of extensive research since early 1970s. Above all, two areas of study — *data models* and *query optimisation* — have received extensive treatment as they are fundamental components of any database management system (*DBMS*) [2, 7, 8].

Relational model is the most commonly used database model. It organises data in tables (two-dimensional arrays), with each table consisting of a set of tuples in arbitrary order. Each tuple is a set of unique attributes mapped to values. Consequently, a database query result is also represented as a set of tuples in tabular form. However, query results can be represented more succinctly using a factorised form. One such form, called *factorisation trees*, was introduced by Olteanu and Závodný in 2011 [9]. This innovative approach to data representation not only can provide a space-efficient data storage, but may also significantly improve the performance of processing queries on relational data. Indeed, experimental evaluation performed on FDB, a main-memory query engine for conjunctive queries on factorised representations, shows exponential spatial and performance gains compared to a query engine for flat data representation [1].

A given query can be processed and evaluated by a DBMS via many so-called query plans. The query plans give the same result, but their execution has a varying cost. The objective of *query optimization* is to find a plan with a minimal cost, where the cost is most often simply the time required by the engine to compute the query result from the given input database. In the context of f-representations, we need to consider an additional objective of query optimization, that is, space efficiency. Ideally, we want to represent the query result over an f-tree which would give a factorised representation optimal in size. FDB currently uses a statically inferred (and worst-case optimal) bound on the size of f-representations to find an efficient query plan, employing an exhaustive search or a greedy heuristic [1].

1.1 Motivation

The f-trees derived in the paper on factorised representations [9] are statically optimal in size, but independent of the database content. This is an unsatisfactory approach since it does not take into account how commonly certain values appear in the relations and how selective the query joins are. Instance-based bounds may be arbitrarily off from the statically-inferred ones, especially when the data distribution is skewed and the selectivities of join conditions differ significantly. Additionally, we always want to factor the most common values first, and the most distinct ones (e.g. keys) last. This requires an approach to query optimization based on catalog information, which we consider in this paper.

1.2 Goal

Our goal is to improve efficiency of query evaluation on factorised representations by applying cost-based query optimization techniques adjusted to fit the factorized model of data representation.

1.3 Contributions

The main contributions of this project are:

- query cost estimation function based on estimates on sizes of f-representations
- a modular framework built on top of FDB for finding cost-effective f-plans
- experimental evaluation of the impact of cost-based query optimisation on efficiency of queries on factorised relational databases
- analysis of future work involving query optimisation for factorised representations

2 Background

2.1 Preliminaries

2.1.1 Databases

In our discussion we consider relational databases with named attributes under set semantics. A database is a set of relations, with each relation R_i defined over a schema S_i (a schema is a set of attributes) consisting of a set of tuples t_i compliant with S_i. We assume that the relation symbols and attribute names are all distinct, although, for simplicity, our examples will occasionally use the same name for distinct attributes.

Example 1. An example database for a league management system is presented in Figure 1. The relation TeamColor is defined over schema (team, color), the relation TeamLoc is defined over schema (team, city), and the relation LocArena is defined over schema (city, arena). $\square$

team	colour
LA Lakers	black
LA Lakers	gold
NY Knicks	black
NY Knicks	blue
NY Liberty	blue

(a) TeamColor data sample

team	city
LA Lakers	Los Angeles
NY Knicks	New York
NY Liberty	New York

(b) TeamLoc data sample

city	arena
Los Angeles	Staples Centre
Los Angeles	The Forum
New York	Madison Square Garden
New York	Prudential Arena
Los Angeles	LongBeach Arena

(c) LocArena data sample

Figure 1: Data sample of the league management database in tabular form

2.1.2 Queries

In our analysis, we consider conjunctive queries expressed in relational algebra of the form:

$$Q = \pi_{\mathcal{P}}(\sigma_{\varphi}(R_1 \times \ldots \times R_n))$$

where each R_i is a relation over schema $\mathcal{S}_i$, $\mathcal{P}$ is a list of attributes, and φ is a conjunction of equalities of attributes of the form $A_1 = A_2$. An equi-join query Q is a special case of the above, with $\mathcal{P} = \bigcup \mathcal{S}_i$. In this case the projection operator can be removed from Q.

We define the equivalence class $\mathcal{A}$ for an attribute A as the set of all attributes transitively equal to A in φ.

The result $Q(\mathbf{D})$ of the query Q evaluated on the database $\mathbf{D}$ is the set of tuples t which satisfy φ. The *size* of $Q(\mathbf{D})$, denoted $|(Q(\mathbf{D}))|$, is the number of tuples in the result set.

Example 2. Say we are interested in which shirt colours can be seen in which cities and at which sport arenas. We can get this result by executing the query Q_1 on the relations presented in Example 1 that joins the three relations, where

$$Q_1 = \text{TeamColour} \bowtie_{\text{team}} \text{TeamLoc} \bowtie_{\text{city}} \text{LocArena}$$

and is analogous to

$$Q_1 = \sigma_{\varphi}(\text{TeamColour} \times \text{TeamLoc} \times \text{LocArena})$$

$$\text{where} \quad \varphi = TeamColour.team = TeamLoc.team \land TeamLoc.city = LocArena.city$$

□

2.2 Factorised Forms

Factorised representations form a *complete* system for representing relational databases and query results. By exploiting the fact that a value may appear within multiple tuples in relational databases, we represent the data in a factorised form where the value is stored only once. This brings an asymptotically exponential reduction in both storage size and evaluation time of query results. This gain is most prominent in data exhibiting many-to-many relationships. In this section, we formalise f-representations for both relations and query results.

2.2.1 F-Representations of relations

We express factorised representations using a subset of relational algebra consisting of union ($\cup$) and product ($\times$) operators as well as singleton tuples. By $\langle A\!:\!a\rangle$ we mean a relation with a single attribute A and a single tuple with value a.

Definition 1. *[9] 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* $\mathcal{S}$;
- $E_1 \cup \cdots \cup E_n$, *where each* E_i *is an f-representation over* $\mathcal{S}$;
- $E_1 \times \cdots \times E_n$, *where each* E_i *is an f-representation over* $\mathcal{S}_i$ *and* $\mathcal{S}$ *is the disjoint union of all* $\mathcal{S}_i$.

Example 3. We can interpret any factorised representation over a set of attributes $\mathcal{S}$ as a database over a schema $\mathcal{S}$. For instance, $\langle city\!:\!NewYork\rangle \times (\langle team\!:\!NYKnicks\rangle \cup \langle team\!:\!NYLiberty\rangle)$ represents a subset of the relation TeamLoc from Example 1. We usually drop the product signs and singleton types for compactness, so we can just write $\langle NewYork\rangle(\langle NYKnicks\rangle \cup \langle NYLiberty\rangle)$. □

Definition 2. *[9] A* factorisation tree, *or* f-tree *for short, over a possibly empty schema S is an unordered rooted forest with each node labelled by a non-empty subset of S such that each attribute of S occurs in exactly one node.* □

For any relation R there may exist many f-trees defining nesting structures over which we can represent R.

Example 4. The f-representation in Example 3 is over the f-tree $\mathcal{T}_1$ presented in Figure 2. The f-tree $\mathcal{T}_2$ provides an alternative factorisation structure for this relation. Figure 3 presents complete f-representations of the LocTeam relation over the two alternative f-trees. □

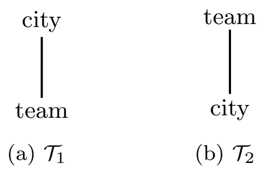


Figure 2: F-Trees over which we can represent the relation LocTeam

The nesting structure of an f-representation of a relation over a given f-tree is uniquely determined by its shape. We start the factorisation through grouping the tuples of the represented

$$\langle NewYork\rangle \times (\langle NYKnicks\rangle \cup \langle NYLiberty\rangle)\cup$$
$$\langle LosAngeles\rangle \times \langle LALakers\rangle\cup$$

(a) $\mathcal{T}_1(LocTeam)$

$$\langle NYKnicks\rangle \times \langle NewYork\rangle\cup$$
$$\langle NYLiberty\rangle \times \langle NewYork\rangle\cup$$
$$\langle LALakers\rangle \times \langle LosAngeles\rangle\cup$$

(b) $\mathcal{T}_5(Q_1)$

Figure 3: F-Representations of the relation LocTeam over corresponding f-trees from Figure 2

relation by attributes that label the root node. We then factor out the common values and recursively factorise each group over the subtrees. The formal definition follows.

Definition 3. *[9] An f-representation over a given f-tree $\mathcal{T}$ is defined recursively:*

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

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

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

If $\mathcal{T}$ is a single tree with a root labelled by $\{A_1, \ldots, A_k\}$ and a non-empty forest $\mathcal{U}$ of children, then

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

where each E_a is an f-representation over $\mathcal{U}$ and the union $\bigcup_a$ is over a collection of distinct values a.

If $\mathcal{T}$ is a single node labelled by $\{A_1, \ldots, A_k\}$, then

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

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

An f-representation Φ of a relation R over a given f-tree $\mathcal{T}$ and database $\mathbf{D}$ is unique up to commutativity of union and product. Hence we can speak of *the* f-representation $\mathcal{T}(\mathbf{R})$.

2.2.2 F-Representations of query results

We now extend the notion of f-representations to query results. We are interested in f-trees that define factorisation of query results for any input database $\mathbf{D}$, regardless of its contents.

Definition 4. *Let Q be an equi-join query and $\mathcal{T}$ an f-tree over which Q can be represented. The* path condition *is satisfied if and only if for each relation in Q its attributes lie along a root-to-leaf path in $\mathcal{T}$.*

F-trees which satisfy the path condition are precisely the ones that define f-representations of query results, and we only focus on them in our discussion.

Further, we focus on f-trees whose nodes are labelled by equivalence classes of attributes, derived from the input query Q.

Example 5. Consider the binary relations from Example 1 and the query Q_1 from Example 2. Recall $Q_1 = \text{TeamColour} \bowtie_{\text{team}} \text{TeamLoc} \bowtie_{\text{city}} \text{LocArena}$.

All three f-trees in Figure 4 capture a factorisation structure over which we can represent the result of Q_1. The corresponding f-representations are presented in Figure 5. □

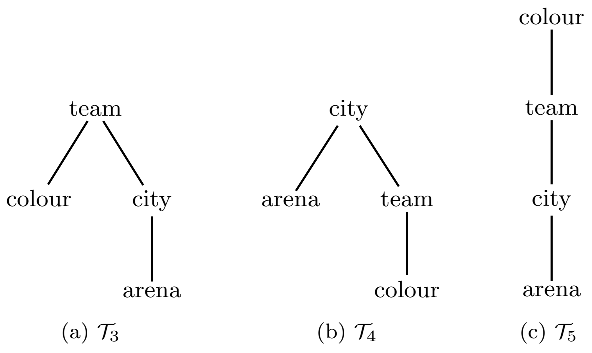


Figure 4: F-Trees over which we can represent the result of Q_1

As before, for any conjunctive query Q there may exist multiple distinct f-trees defining a nesting structures over which we can represent the result of Q. Furthermore, an f-representation Φ of the result of a query Q over a given f-tree $\mathcal{T}$ and database $\mathbf{D}$ is unique up to commutativity of union and product. Hence we can speak of *the* f-representation $\mathcal{T}(Q(\mathbf{D}))$.

2.3 Auxiliary definitions

Let A be an attribute labelling a node in an f-tree $\mathcal{T}$. We define *anc(A)* as the set of all attributes which label nodes that are ancestors of the node labelled by A in $\mathcal{T}$. We define *path(A)* as the

$$\langle LALakers\rangle \times (\langle black\rangle \cup \langle gold\rangle) \times (\langle LosAngeles\rangle \times (\langle StaplesCentre\rangle \cup \langle TheForum\rangle \cup \langle LongBeachArena\rangle))\cup$$
$$\langle NYKnicks\rangle \times (\langle black\rangle \cup \langle blue\rangle) \times (\langle NewYork\rangle \times (\langle MadisonSG\rangle \cup \langle PrudentialArena\rangle))\cup$$
$$\langle NYLiberty\rangle \times \langle blue\rangle \times (\langle NewYork\rangle \times (\langle MadisonSG\rangle \cup \langle PrudentialArena\rangle))$$

(a) $\mathcal{T}_3(Q_1)$

$$\langle NewYork\rangle \times (\langle MadisonSG\rangle \cup \langle PrudentialArena\rangle) \times (((\langle NYKnicks\rangle \times (\langle black\rangle \cup \langle blue\rangle)) \cup (\langle NYLiberty\rangle \times \langle blue\rangle))\cup$$
$$\langle LosAngeles\rangle \times (\langle StaplesCentre\rangle \cup \langle TheForum\rangle \cup \langle LongBeachArena\rangle) \times (\langle LALakers\rangle \times (\langle black\rangle \cup \langle gold\rangle))$$

(b) $\mathcal{T}_4(Q_1)$

$$\langle gold\rangle \times \langle LALakers\rangle \times \langle LosAngeles\rangle \times (\langle StaplesCentre\rangle \cup \langle TheForum\rangle \cup \langle LongBeachArena\rangle)\cup$$
$$\langle black\rangle \times (((\langle LALakers\rangle \times \langle LosAngeles\rangle \times (\langle StaplesCentre\rangle \cup \langle TheForum\rangle \cup \langle LongBeachArena\rangle))\cup$$
$$((\langle NYKnicks\rangle \times \langle NewYork\rangle \times (\langle MadisonSG\rangle \cup \langle PrudentialArena\rangle)))\cup$$
$$\langle blue\rangle \times (((\langle NYKnicks\rangle \times \langle NewYork\rangle \times (\langle MadisonSG\rangle \cup \langle PrudentialArena\rangle))\cup$$
$$((\langle NYLiberty\rangle \times \langle NewYork\rangle \times (\langle MadisonSG\rangle \cup \langle PrudentialArena\rangle)))$$

(c) $\mathcal{T}_5(Q_1)$

Figure 5: F-Representations of the result of Q_1 over corresponding f-trees from Figure 4

set of attributes of ancestors of the node A in $\mathcal{T}$ together with attributes labelling the node A itself.

Furthermore, we define $anc(A) \restriction R$ to be the set anc(A) restricted to attributes in relation R. The set $path(A) \restriction R$ is defined symmetrically.

Example 6. Consider the f-tree $\mathcal{T}_3$ in Figure 4.

$$\text{anc(arena)} = \{\text{city}, \text{team}\}$$
$$\text{anc(arena)} \restriction LocArena = \{\text{city}\}$$
$$\text{path(arena)} = \{\text{arena}, \text{city}, \text{team}\}$$
$$\text{path(arena)} \restriction LocArena = \{\text{arena}, \text{city}\}$$

□

By the *size* of a node A of an f-representation $\mathcal{T}(\mathbf{D})$, denoted $|A|$, we mean the number of appearances of singletons of the form $\langle A\!:\!a\rangle$ in $\mathcal{T}(\mathbf{D})$. The *size* of the f-representation $\mathcal{T}(\mathbf{D})$ is then the sum of sizes of all its nodes, i.e.

$$|\mathcal{T}(\mathbf{D})| = \sum_{A\in\mathcal{T}(\mathbf{D})} |A|$$

Example 7. Consider the f-representations from Figure 5. We have

$$|\mathcal{T}_3(Q_1(\mathbf{D}))| = 18$$

$$|\mathcal{T}_4(Q_1(\mathbf{D}))| = 15$$

$$|\mathcal{T}_5(Q_1(\mathbf{D}))| = 25$$

$$\text{even though, algebraically, } \mathcal{T}_3(Q_1(\mathbf{D})) = \mathcal{T}_4(Q_1(\mathbf{D})) = \mathcal{T}_5(Q_1(\mathbf{D}))$$

□

2.4 Size bounds on f-representations

In the paper on factorised representations of query results [9] Olteanu and Závodný derive a bound on the size of f-representations of a given query Q over any database $\mathbf{D}$. First, they introduce a parameter $s(\mathcal{T})$[1] which gives an asymptotic bound on the size of factorised representations of the query result of $Q(\mathbf{D})$ over $\mathcal{T}$. This bound is given by $\left|\mathbf{D}^{s(\mathcal{T})}\right|$ and can be lifted to queries by finding

$$s(Q) = \min_{\mathcal{T}} s(\mathcal{T})$$

where $\mathcal{T}$ is any f-tree of Q. The asymptotic upper bound on f-representation of query results is then $\left|\mathbf{D}^{s(Q)}\right|$ for any database $\mathbf{D}$ and is optimal.

Consider the f-trees from Example 4. We have $s(\mathcal{T}_3) = s(\mathcal{T}_4) = 2$, whereas $s(\mathcal{T}_5) = 3$. This means that for any database the size of f-representation of the result of the query Q_1 is at most quadratic in $|\mathbf{D}|$ over the f-trees $\mathcal{T}_3$ and $\mathcal{T}_4$, and at most cubic in $|\mathbf{D}|$ over the f-tree $\mathcal{T}_5$. In fact, $\mathcal{T}_3$ and $\mathcal{T}_4$ are both asymptotically optimal for Q_1, hence $s(Q_1) = 2$.

2.5 F-Trees restructuring operators and f-plans

To evaluate a query on f-representations we require new operators for restructuring f-trees, introduced in [1]. Importantly, the *push-up* operator is used to normalise f-trees. Furthermore, all of the operators preserve both path constraint and normalisation. Hence, we will only consider normalised f-trees in our subsequent discussion. For each of the operators there exists

[1] $s(\mathcal{T})$ is the maximum fractional edge cover number of a hypergraph defined on T and can be computed in polynomial time using linear programming. Please refer to [9] for details.

a quasilinear[2] algorithm to carry out the transformation on f-representations [1]. We now list the operators together with brief descriptions.

- the *push-up* $\psi_{\mathcal{B}}$ operator, or the *one-step normalisation operator*, factors out expressions common to all terms of a union. Consider the f-tree $\mathcal{T}$ in Figure 6. The nodes $\mathcal{A}$ and $\mathcal{B}$ do not have any attributes in common, and so we can push up the node $\mathcal{B}$ without violating the path constraint. To *normalise* an f-tree, we repeatedly apply the push-up operator to each node of the tree starting bottom up, until no more nodes can be pushed-up without violating the path constraint.

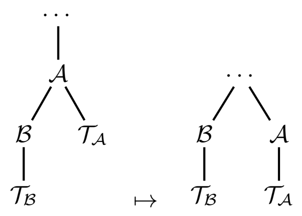

...
$\mathcal{A}$
$\mathcal{B}$
$\mathcal{T}_{\mathcal{A}}$
$\mathcal{T}_{\mathcal{B}}$
$\mapsto$
...
$\mathcal{B}$
$\mathcal{A}$
$\mathcal{T}_{\mathcal{B}}$
$\mathcal{T}_{\mathcal{A}}$

Figure 6: push-up $\psi_{\mathcal{B}}$

- the *swap* $\chi_{\mathcal{A},\mathcal{B}}$ operator exchanges a node with its parent maintaining normalisation. Consider the f-tree in Figure 7, where the subtrees $\mathcal{T}_{\mathcal{A}}$ and $\mathcal{T}_{\mathcal{B}}$ depend[3] only on the nodes $\mathcal{A}$ and $\mathcal{B}$, respectively. We can promote the node $\mathcal{B}$ to be the parent of $\mathcal{A}$, rearranging the subtrees so that normalisation is preserved.

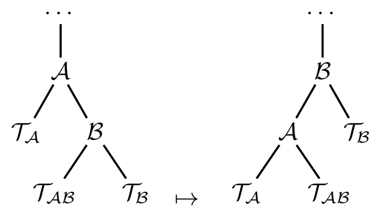
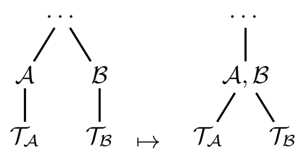

...
$\mathcal{A}$
$\mathcal{T}_{\mathcal{A}}$
$\mathcal{B}$
$\mathcal{T}_{\mathcal{AB}}$
$\mathcal{T}_{\mathcal{B}}$
$\mapsto$
...
$\mathcal{B}$
$\mathcal{A}$
$\mathcal{T}_{\mathcal{B}}$
$\mathcal{T}_{\mathcal{A}}$
$\mathcal{T}_{\mathcal{AB}}$

Figure 7: swap $\chi_{\mathcal{A},\mathcal{B}}$

[2]A quasilinear function of N is $O(N \log^k N)$ for some k.

[3]We say that two nodes are dependent if they have attributes in the same dependency set, with each relation R_i inducing a dependency set consisting of R_i's attributes

- the *merge* $\mu_{\mathcal{A},\mathcal{B}}$ operator, depicted in Figure 8, merges the sibling nodes $\mathcal{A}$ and B into a single node labelled by the attribute classes of $\mathcal{A}$ and $\mathcal{B}$. This operator also preserves normalisation.



...
$\mathcal{A}$
$\mathcal{B}$
$\mathcal{T}_{\mathcal{A}}$
$\mathcal{T}_{\mathcal{B}}$
$\mapsto$
...
$\mathcal{A},\mathcal{B}$
$\mathcal{T}_{\mathcal{A}}$
$\mathcal{T}_{\mathcal{B}}$

Figure 8: merge $\mu_{\mathcal{A},\mathcal{B}}$

- the *absorb* $\alpha_{\mathcal{A},\mathcal{B}}$ operator, depicted in Figure 9, absorbs a node $\mathcal{B}$ into its ancestor node $\mathcal{A}$, followed by a normalisation of the resulting f-tree, thus preserving normalisation. The node $\mathcal{A}$ is now labelled by the attribute classes of $\mathcal{A}$ and $\mathcal{B}$.

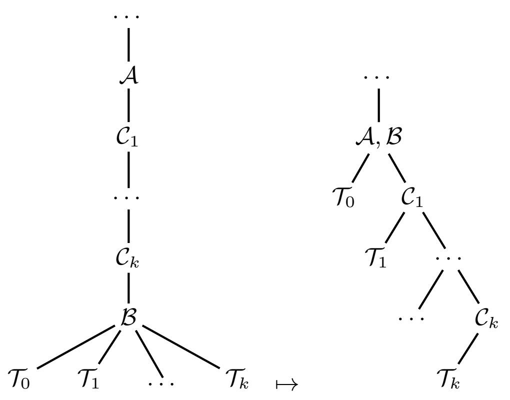


Figure 9: absorb $\alpha_{\mathcal{A},\mathcal{B}}$ (w/o full normalisation)

We further define the *cartesian product* operator on the f-trees $\mathcal{T}_1$ and $\mathcal{T}_2$, which results in their forest $\mathcal{T}_1 \times \mathcal{T}_2$. For an attribute set $\mathcal{P}$, the *projection* operator $\pi_{\mathcal{P}}$ preserves only those attributes in an $\mathcal{T}$ that are in $\mathcal{P}$ resulting in singletons of type $\langle B\!:\!b\rangle$, where $B \notin \mathcal{P}$, being replaced by the empty singleton $\langle\rangle$.

Any full conjunctive query Q can be evaluated by a sequential composition of the operators presented above. We call this sequential composition a *factorisation plan*, or an *f-plan*, for Q.

Example 8. Recall Q_1 = TeamColour $\bowtie_{\text{team}}$ TeamLoc $\bowtie_{\text{city}}$ LocArena. Figure 10 shows an f-plan for Q_1, starting with an f-tree forest over which we can represent the input relations. We begin by merging the f-tree for the relation CityArena with the f-tree for the relation TeamCity on the corresponding *city* nodes. To perform the second join we first need to restructure the factorisations so that in both trees the *team* node is the root. We can achieve this by swapping the *team* node with its parent, the *city* node. We are now ready to perform the second join operator. This results in the final f-tree, which is an f-tree for the query Q_1. □

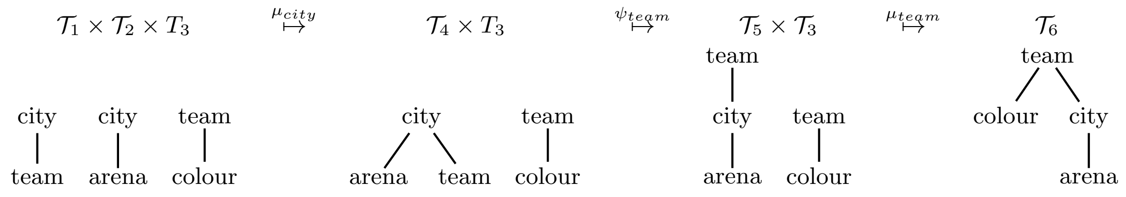

$\mathcal{T}_1 \times \mathcal{T}_2 \times T_3$
$\overset{\mu_{city}}{\mapsto}$
$\mathcal{T}_4 \times T_3$
$\overset{\psi_{team}}{\mapsto}$
$\mathcal{T}_5 \times \mathcal{T}_3$
$\overset{\mu_{team}}{\mapsto}$
$\mathcal{T}_6$
city
team
city
arena
team
colour
city
arena
team
team
colour
team
city
arena
team
colour
team
colour
city
arena

Figure 10: A factorisation plan for the query Q_1

3 Cost-based query optimisation

In the previous section, we have formally introduced the notion of factorised representations which provide a compact way of representing relational databases and query results over them. We have also listed the restructuring operator on f-trees which are used to evaluate queries on f-representations.

In this chapter, we further investigate the execution of queries on factorised databases and show how we can use size estimation techniques of query results to find query plans which are efficient to execute. We derive a size estimation method for factorised representations based on catalog information and introduce a query cost model based on this method. We also give details on two approaches to exploring the search space of possible f-plans for a given query — and exhaustive search that always returns an optimal plan but requires exponential time to find the solution, and a greedy search that dramatically improves the search time, but may return a sub-optimal result.

3.1 Assumptions

We make the following assumptions is our analysis, all of which are commonly used in query cost models [13].

1. Independence: Attribute values that are part of a join predicate are independent within each relation.

2. Uniformity: The distinct values of each attribute appear equi-frequently within each relation.

3. Containment: When joining two tables, the set of attribute values of the join attribute of the larger relation cardinality is a superset of the set of attribute values of the join attribute of the smaller relation cardinality

3.2 Useful equivalences

The following are a few relational algebra equivalences we use in our discussion. For a comprehensive list, please refer to [11]. Selections commute with projections as long as the selection operator attributes are contained within attributes retained by the projections.

$$\pi_{\mathcal{P}}\left(\sigma_{\varphi}\left(R\right)\right) \equiv \sigma_{\varphi}\left(\pi_{\mathcal{P}}\left(R\right)\right) \text{ when } \mathrm{attr}(\varphi) \subseteq \mathcal{P}$$

We can also commute projection with a cross product

$$\pi_{\mathcal{P}}\left(\mathcal{R} \times \mathcal{S}\right) \equiv \pi_{\mathcal{P}_1}\left(\mathcal{R}\right) \times \pi_{\mathcal{P}_2}\left(\mathcal{S}\right)$$

where $\mathcal{P}_1 \subseteq \mathcal{P}$ are attributes that appear in $\mathcal{R}$ and $\mathcal{P}_2 \subseteq \mathcal{P}$ are attributes that appear in $\mathcal{S}$. Additionally, we make use of two equivalences regarding selections. Cascading of selections

$$\sigma_{\varphi_1 \wedge \ldots \wedge \varphi_m}(R) \equiv \sigma_{\varphi_1}(\sigma_{\varphi_2}(\ldots(\sigma_{\varphi_m}(R))\ldots))$$

allows us to consider a selection condition involving multiple conjuncts looking at one conjunct at a time. The second important equivalence regarding the selection operator is commutativity of selections

$$\sigma_{\varphi_1}(\sigma_{\varphi_2}(R)) \equiv \sigma_{\varphi_2}(\sigma_{\varphi_1}(R))$$

3.3 Estimates

We now present typical estimates of the size of a query result used by query optimisers. The ideas originate from the System-R project, and are still relevant in modern query optimisation [12].

Let us consider a query of the form $Q = \pi_{\mathcal{P}}(\sigma_{\varphi}(R_1 \times \ldots \times R_n))$ where $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_m$ with each φ_k of the form $A_1 = A_2$ or $A_1 = k$, where A_1 and A_2 are attributes and k is a constant.

The upper bound on the size of the result of the query Q is the product of the cardinalities of the relations $R_1 \ldots R_n$. This size is, however, reduced by each clause in the conjunction φ. With each clause φ_k we associate a *selectivity factor* which is the ratio of the estimated result size to the input size, and we denote it as S_{φ_k}. We consider the following cases:

- Case 1: $\varphi_k \equiv A = k$: approximation for the selectivity factor is $\frac{1}{\pi_A(R)}$ (by the Uniformity assumption), where A is an attribute in R. If catalog information is not available, we approximate the selectivity factor to $\frac{1}{10}$.

- Case 2: $\varphi_k \equiv A_1 = A_2$: approximation for the selectivity factor is $\frac{1}{\max(\pi_{A_1}(R_1), \pi_{A_2}(R_2))}$, where A_1 is an attribute in R_1 and A_2 is an attribute in R_2. This follows from the

Uniformity assumption and also the Containment assumption, as we assume that each key value in the smaller size attribute projections has a matching value in the other projection. Again, if no catalog information is available, we approximate the selectivity factor to $\frac{1}{10}$. We do not distinguish the case when $R_1 = R_2$, and use the same selectivity estimate.

The selectivity of the conjunction $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_m$ is then $S_\varphi = S_{\varphi_1} \times \ldots \times S_{\varphi_k}$ (using the Independence assumption).

The size of an equi-join query $Q' = \sigma_\varphi(R_1 \times \ldots \times R_n)$ is then the product of the selectivity of the selection conjunct and the cardinalities of the relations $R_1, \ldots, R_n$, i.e.

$$|Q'| = S_\varphi \times |R_1| \times \ldots \times |R_2|$$

3.4 Size estimation of f-representations

We now introduce an approach to size estimation of the f-representation over a given f-tree $\mathcal{T}$ for a query Q and a database $\mathbf{D}$. We use the common assumptions about the input data listed above, and assume catalog information with cardinality estimates for $\mathbf{D}$ is available. Furthermore, we assume that the path constraint holds for the input f-tree.

Let Q be a conjunctive query, i.e. a query of the form $\pi_\mathcal{P}(\sigma_\varphi(R_1 \times \ldots \times R_n))$. The size of a node A in $\mathcal{T}$ is then the number of A-singletons in the f-representation of the query result over $\mathcal{T}$ and is given by

$$|A| = \left|Q_{\mathrm{path}(A)}(\mathbf{D})\right| \tag{1}$$

where $Q_{\mathrm{path}(A)} = \pi_{\mathrm{path}(A)}(Q)$. We now derive an approximation for the size of A:

$$\begin{aligned}
|A| &= \left|Q_{\mathrm{path}(A)}(\mathbf{D})\right| \\
&= \left|\pi_{\mathrm{path}(A)}(\sigma_\varphi(R_1 \times \ldots \times R_n))\right| \\
&\leq \left|\pi_{\mathrm{path}(A)}(\sigma_{\varphi\restriction\mathrm{path}(A)}((R_1 \times \ldots \times R_n) \restriction \mathrm{path}(A))\right| \\
&= \left|\sigma_{\varphi\restriction\mathrm{path}(A)}(\pi_{\mathrm{path}(A)\restriction R_1}(R_1 \restriction \mathrm{path}(A)) \times \ldots \times \pi_{\mathrm{path}(A)\restriction R_n}(R_n \restriction \mathrm{path}(A)))\right| \qquad (2) \\
&= \left|\sigma_{\varphi\restriction\mathrm{path}(A)}(\pi_{\mathrm{path}(A)\restriction R_1}(R_1) \times \ldots \times \pi_{\mathrm{path}(A)\restriction R_n}(R_n))\right| \\
&= S_{\sigma_{\varphi\restriction\mathrm{path}(A)}} \times \left|(\pi_{\mathrm{path}(A)\restriction R_1}(R_1) \times \ldots \times \pi_{\mathrm{path}(A)\restriction R_1}(R_1))\right| \\
&= S_{\sigma_{\varphi\restriction\mathrm{path}(A)}} \times \left|(\pi_{\mathrm{path}(A)\restriction R_1}(R_1)\right| \times \ldots \times \left|\pi_{\mathrm{path}(A)\restriction R_1}(R_1))\right|
\end{aligned}$$

where $\sigma_{\varphi \upharpoonright \text{path}(A)}$ is the selection σ_φ restricted to equalities in φ with attributes in the set path(A), and $R_n \upharpoonright$ path$(A))$ is the relation R_k restricted to the attributes in path(A). We also use the fact that

$$\pi_{\text{path}(A) \upharpoonright R_k}(R_k \upharpoonright path(A)) \equiv \pi_{\text{path}(A) \upharpoonright R_k}(R_k)$$

since if we restrict the projection $\pi_{\text{path}(A)}$ to attributes in R_k then it is unnecessary to restrict the attributes in R_k to the attributes in path(A).

We can now estimate the total size of the factorisation $|\mathcal{T}(Q(\mathbf{D}))|$ by summing the expression derived above over all nodes in $\mathcal{T}$, which are by definition labelled by attributes in the projection list $\mathcal{P}$ of Q. We thus get the following:

$$|\mathcal{T}(Q(\mathbf{D}))| = \sum_{A \in \mathcal{T}(Q(\mathbf{D}))} |A| \tag{3}$$

When estimating the size of a node, we only consider selectivity estimates for query joins present in the current node-to-root path only. This is because we tailor our estimates to the typical setting in which factorised representations are used, that is data with many-to-many relationships in the query joins. Therefore, we assume that selectivities of joins in attribute classes labelling nodes further down the tree do not propagate to the upper nodes and, by extension, other branches. Consequently, while we expect our estimates to be accurate, we may sometimes over-estimate the actual f-representation size. This effect will be most prominent when the equalities not considered at the current node are very selective.

We assume that all the required catalog information for the input database D is available to use. In the case when we only have information on the sizes of projections on single attributes, we can use the following inequality to get an estimate for the projection of a relation R onto the attribute set $\mathcal{A} = \{A_1, \ldots, \mathcal{A}_n\}$:

$$|\pi_{\mathcal{A}}(R)| \leq |\pi_{\mathcal{A}_1}(R)| \times \ldots \times |\pi_{\mathcal{A}_n}(R)|$$

Example 9. Consider the query $Q_2 = \sigma_\varphi(R \times S \times T)$ with $\varphi = (A_R = A_S \wedge C_S = C_T)$, where R, S, and T are relations over schemas (A_R, B_R), (A_S, C_S), and (C_T, D_T) respectively. Figure 11

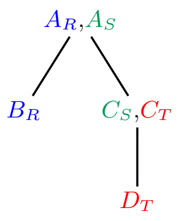


Figure 11: An f-tree $\mathcal{T}_6$ for the query Q_2

depicts an f-tree $\mathcal{T}_6$ for the query Q_2. Size estimates for the nodes of the tree from Figure 11 according to the formula (2) derived in this section are

$$|\mathcal{A}| = \sigma_{A_R=A_S} \times |\pi_{A_R}(R)| \times |\pi_{A_S}(S)|$$

$$|\mathcal{B}| = \sigma_{A_R=A_S} \times |\pi_{A_R,B_R}(R)| \times |\pi_{A_S}(S)|$$

$$|\mathcal{C}| = \sigma_{A_R=A_S \wedge C_S=C_T} \times |\pi_{A_S,C_S}(S)| \times |\pi_{C_T}(T)| \times |\pi_{A_R}(R)|$$

$$|\mathcal{D}| = \sigma_{A_R=A_S \wedge C_S=C_T} \times |\pi_{A_S,C_S}(S)| \times |\pi_{C_T,D_T}(T)| \times |\pi_{A_R}(R)|$$

and, using formula (3), the size estimate for the entire tree is then

$$|\mathcal{T}_6| = |\mathcal{A}| + |\mathcal{B}| + |\mathcal{C}| + |\mathcal{D}|$$

Now, let us take into account a database D with the following projection sizes:

$\lvert\pi_{A_R}(R)\rvert = 12$	$\lvert\pi_{A_S}(S)\rvert = 9$	$\lvert\pi_{C_T}(T)\rvert = 10$
$\lvert\pi_{B_R}(R)\rvert = 4$	$\lvert\pi_{C_S}(S)\rvert = 4$	$\lvert\pi_{D_T}(T)\rvert = 8$
$\lvert\pi_{A_R,B_R}(R)\rvert = 16$	$\lvert\pi_{A_S,C_S}(S)\rvert = 10$	$\lvert\pi_{C_T,D_T}(T)\rvert = 14$

Table 1: Projections sizes for D

Knowing the projection cardinalities, we can now compute the selectivity estimates:

$$\sigma_{A_R=A_S} = \frac{1}{\max(|\pi_{A_R}(R)|\,, |\pi_{A_S}(S)|)} = \frac{1}{12}$$

$$\sigma_{C_S=C_T} = \frac{1}{\max(|\pi_{C_S}(S)|\,, |\pi_{C_T}(T)|)} = \frac{1}{10}$$

$$\sigma_{A_R=A_S \wedge C_S=C_T} = \sigma_{A_R=A_S} \times \sigma_{C_S=C_T} = \frac{1}{120}$$

Which gives us

$$
\begin{aligned}
|\mathcal{A}| &= \frac{1}{12} \times 12 \times 9 = 9 \\
|\mathcal{B}| &= \frac{1}{12} \times 16 \times 9 = 12 \\
|\mathcal{C}| &= \frac{1}{120} \times 10 \times 10 \times 12 = 10 \\
|\mathcal{D}| &= \frac{1}{120} \times 10 \times 14 \times 12 = 14
\end{aligned}
$$

and so finally

$$|\mathcal{T}_6(\mathbf{D})| = 9 + 12 + 10 + 14 = 45$$

□

3.5 Query optimisation

As mentioned in § 2.5, queries on factorised data are evaluated by a sequence of transformations, and we call this sequence a *factorisation plan*, or an *f-plan* [1]. A given input query can be evaluated by a number of distinct f-plans which all give the same result but differ in cost. Therefore, our goal is to find plans which are cost-effective, that is, have low memory footprint and execute quickly. For a given query Q, we want to find an f-plan that not only gives a succinct f-representation of the result Q, but also minimizes the total size of the intermediate f-representations in the sequence of transformations.

We next define the notion of a cost of an f-plan for a given input query Q and initial tree $\mathcal{T}$ based on two alternative cost models and define optimality of an f-plan with respect to each model. We then describe an exhaustive search procedure over the search space of possible f-plans for a given query, as well as a search policy based on a greedy heuristic.

3.5.1 Cost of an f-plan

Let us consider an f-plan $\mathcal{F}$ consisting of k transformations $\omega_1, \ldots, \omega_k$ that map the input f-tree $\mathcal{T}_{\text{initial}}$ to the final f-tree $\mathcal{T}_{\text{final}}$:

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

where each of the operators $\omega_1, \ldots, \omega_k$ is one of the operators we defined in § 2.5, and $\mathcal{T}_{\text{initial}}$ is assumed to be normalised.

The optimality of an f-plan with respect to an order $<$ on f-plans is defined as follows:

Definition 5. *Let $\mathcal{F}_1$ be an f-plan for a query Q. $\mathcal{F}_1$ is* optimal *for Q if there is no f-plan $\mathcal{F}_2$ for Q with $F_2 < F_1$*

The current optimiser provided by FDB is based on asymptotic size bounds on factorisation sizes over f-trees. It tries to minimise the highest s($\mathcal{T}$) value for the intermediate f-trees in an f-plan. This is because the cost of the entire f-plan is bounded above by the cost of the most expensive step (the asymptotic size bound is exponential in s($\mathcal{T}$)). If there are multiple f-plans with most expensive operators having the same cost, the one with smaller value of s($\mathcal{T}_{\text{final}}$) is chosen [1].

This entails a lexicographical order $<_{\max} \times <_{s(\mathcal{T})}$ on f-plans consisting of the following orders:

1. $\mathcal{F}_1 <_{\max} \mathcal{F}_2$ holds if $s(\mathcal{F}_1) < s(\mathcal{F}_2)$, and
2. $\mathcal{F}_1 <_{s(\mathcal{T})} \mathcal{F}_2$ holds if $s(\mathcal{T}_1) < s(\mathcal{T}_2)$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are the f-trees of the query result computed by $\mathcal{F}_1$ and $\mathcal{F}_2$ respectively.

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

Consequently, an f-plan for Q is optimal with respect to the cost model based on asymptotic bounds if it is optimal with respect to the order $<_{\max} \times <_{s(\mathcal{T})}$.

There is, however, a significant drawback to this optimisation approach — the resulting f-plans are only optimal in a static sense. That is, although they give asymptotically best solutions for any database $\mathbf{D}$, they are not exploiting information about particular data contexts.

Therefore, we introduce a new cost measure for f-plans based on the size estimates for f-representations described earlier in this chapter. When estimating the cost of an f-plan we need to consider not only the size of the final f-representation, but also the sizes of the intermediate representations. Therefore, we define the cost c($\mathcal{F}$) of an f-plan $\mathcal{F}$ for a query Q and an input database instance D as the sum of the cost estimates of the intermediate and final f-trees, giving

$$\begin{aligned} \mathrm{c}(\mathcal{F}) &= |\mathcal{T}_1(\mathbf{D})| + \cdots + |\mathcal{T}_k(\mathbf{D})| \\ &= \sum_{i=1}^{k} |\mathcal{T}_i(\mathbf{D})| \end{aligned} \tag{4}$$

We then define an order $<_c$ on f-plans as $\mathcal{F}_1 <_c \mathcal{F}_2$ if $\mathrm{c}(\mathcal{F}_2) < \mathrm{c}(\mathcal{F}_1)$. We say that an f-plan is optimal with respect to the estimates-based cost model if it is optimal with respect to the order $<_c$.

Let us now continue with the data from Example 9, and consider the query $Q_3 = \sigma_{B_R = D_T}(Q_2)$. The f-plan $\mathcal{F}_1$ depicted in Figure 12 evaluates the query.

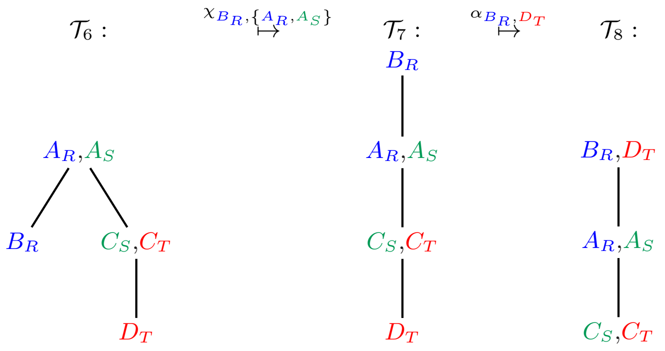


Figure 12: An f-plan $\mathcal{F}_1$ for the query Q_2

The size estimate of the final tree in this plan is $|\mathcal{T}_8(D)| = 18$. Furthermore, the size estimate of the intermediate tree is $|\mathcal{T}_7(D)| = 47$. Therefore, the total cost of this f-plan is 65.

An alternative f-plan $\mathcal{F}_2$ for the query Q_3 is depicted in Figure 13.

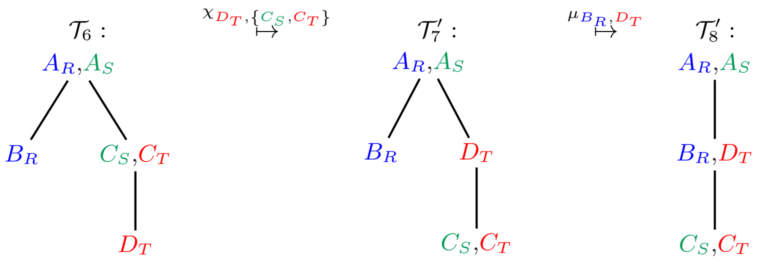


Figure 13: An f-plan $\mathcal{F}_2$ for the query Q_2

In this case, the size of the final tree is $|\mathcal{T}_8'(D)| = 23$, so this f-representation is slightly less succinct that the one over $\mathcal{T}_8$. Moreover, the size of the intermediate tree in this f-plan is $|\mathcal{T}_7'(D)| = 107$ giving a total f-plan cost 130. Therefore, we prefer the first f-plan as it is twice cheaper to execute. In fact, the f-plan $\mathcal{F}_1$ is the optimal plan for this query with respect to our cost function.

In contrast, the asymptotic size bound on the output f-trees is 2 in both cases. This makes these two f-representations indistinguishable with respect to the asymptotic cost model, although we know that result of the first f-plan is more succinct. Furthermore, the optimiser based on the asymptotic cost model would return $\mathcal{F}_2$ as the optimal f-plan for Q_3. This is because the cost of the intermediate f-tree in $\mathcal{F}_1$ is higher ($s(\mathcal{T}_7) = 3$, whereas $s(\mathcal{T}_7') = 2$), and so this f-plan would be considered sub-optimal.

This example clearly indicates the deficiencies of the cost model based on asymptotic bounds. Catalog information allows the instance-based optimiser not only to distinguish between f-representations with equal size bounds, but also to find f-plans that are efficient to execute even if their cost is statically sub-optimal.

3.5.2 Full search

When searching for an optimal f-plan for a given input query Q, we explore a search space of normalised f-trees which are on some path from the input f-tree $\mathcal{T}_{\text{initial}}$ to some final f-tree $\mathcal{T}_{\text{final}}$. A final f-tree in the search is one whose attribute equivalence classes are the classes in $\mathcal{T}_{\text{initial}}$ joined by the equalities in Q.

The search space is defined not only by the order in which we consider the equalities in Q, but also by the shapes of the intermediate and final f-trees. Consequently, the size of the search space, is $\mathrm{O}(k^{2k}n^n)$, for a query with k equalities and an initial f-tree with n nodes [1]. The n^n term follows from the fact that for any f-tree $\mathcal{T}$ with n nodes, there are at most n^n f-trees which can be obtained from $\mathcal{T}$ by rearranging its nodes using the swap operator. For a given query with k equalities, we perform up to k merges out of k^2 potential merge pairs, giving the k^{2k} term.

The objective of the search is to find an f-plan for Q which has minimal cost according to a given cost function. As we aim to find a plan which results in most succinct representations of the intermediate and final result, we use our cost($\mathcal{F}$) function to guide the search. We use Dijkstra's algorithm to find an optimal f-plan, where the distance between two nodes $\mathcal{T}_i$ and $\mathcal{T}_j$ in the search space is defined to be the size estimate of the resulting f-tree.

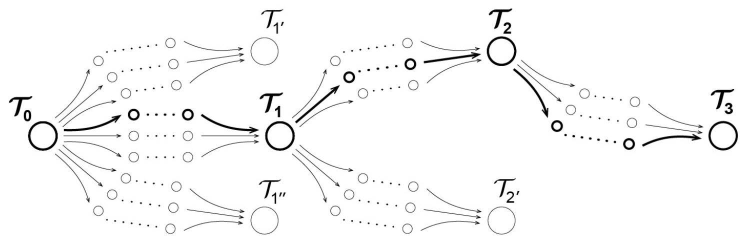


Figure 14: An example of search space exploration by the greedy search, for an initial f-tree $\mathcal{T}_0$ and an input query Q with 3 equalities. The bold lines indicate the f-plan returned from the search.

3.5.3 Greedy search

If we restrict the search to choosing a join with the minimum cost from the remaining joins at each step, we can reduce the search space to be polynomial in the size k of the input query. Furthermore, if we only consider applying restructuring operators no nodes that are subject to a selection condition in the query, we can reduce the search space to be polynomial in n as well.

To choose the join to be performed next, we first find an optimal sequence of restructuring steps for each of the remaining joins, and perform the one with the the minimum cost before repeating the procedure again.

For each join condition involving two nodes labelled by attribute classes $\mathcal{A}$ and $\mathcal{B}$, we have the following three possible restructuring schemes:

- bring both nodes up using the *swap* operator until they become siblings, and then merge them using the *merge* operator
- bring the node $\mathcal{A}$ up until it becomes the ancestor of $\mathcal{B}$ using the *swap* operator, and then merge them using the *absorb* operator
- bring the node $\mathcal{B}$ up until it becomes the ancestor of $\mathcal{A}$ using the *swap* operator, and then merge them using the *absorb* operator

Figure 14 depicts how the greedy heuristic affects the way in which the search space is explored.

The improvement in performance comes at a cost, since the best f-plan found in this greedy procedure may in fact be sub-optimal. On the other hand, the exhaustive search, by definition, always finds an optimal f-plan with respect to the cost function. We evaluate the two search algorithms experimentally in §5.3.

4 Design and Implementation

This chapter documents the development of the query optimisation framework for f-plans built on top of the FDB engine. We provide support for the exhaustive search as well as the greedy heuristic, and implement the size-estimation based cost model introduced in the previous section. We present the design choices made during development and outline the code structure and interfaces by which it is composed. We then give a more detailed analysis of the implementation and describe tools used to aid the development process, as well as the approach to testing.

4.1 Design overview

A high-level overview of the system is presented in Figure 15.

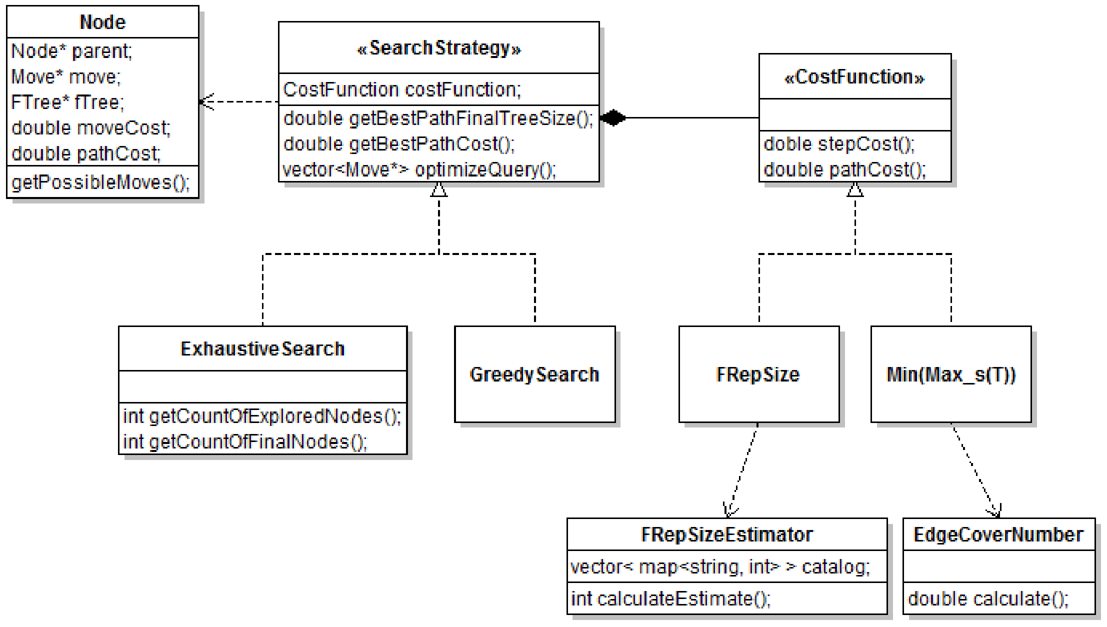


Figure 15: class diagram

The SearchStrategy abstract class encapsulates the notion of a search for an f-plan. The optimizeQuery method returns a complete f-plan for a given initial f-tree and a query. The SearchStrategy class is composed of a CostFunction. This allows us to decouple the cost estimation function from the search algorithm itself, making it easy to switch between concrete implementations of the cost function (such us the original cost function based on the parameter $s(\mathcal{T})$) at design-time or at run-time. This design is known as the strategy pattern, recognised by defining a collection of encapsulated and interchangeable algorithms, which can vary independently from clients using them [5].

We provide two concrete implementations of SearchStrategy — the ExhaustiveSearch class, which in an implementation of the full search based on Dijkstra's algorithm, described in § 3.5.2, and the GreedySearch class, which is based on the greedy heuristic described in § 3.5.3.

The CostFunction interface encapsulates the meaning of cost in our search. The first concrete implementation of this interface, SizeFRep, is based on our size estimation function and is implemented in the FRepSizeEstimator class. The constructor of this class requires a reference to catalog information for the input relations, which is then used by the FRepSizeEstimator. We also include an alternative implementation of the cost function based on the one present in the original FDB optimiser, Min(Max_s(T)), where a cost of an f-tree transformation step is based on the value $s(\mathcal{T})$ of the resulting f-tree.

The Node class represent a node in the search space. It includes a reference to the parent node, the current f-tree shape, and a reference to the transformation that, when applied to the f-tree of the parent node, resulted in the f-tree in this node. It also stores the step cost of the transformation, and a total running cost of the path from this node to the root node in the search. The getPossibleMoves method returns all possible restructuring operations (represented as a list of Move objects) which can be applied to this node's f-tree, and the isFinal method returns true if and only if the current node is a goal node in the search (a node is final if its f-tree's equivalence classes are the classes of the initial f-tree joined by the query equalities).

4.2 Implementation of size estimation function

4.2.1 Algorithm outline

The FRepSizeEstimator class contains an implementation of the size estimation algorithm. The algorithm is outlined in a pseudo-code form in Figure 16.

We calculate the size estimate by summing the sizes of all the nodes in an input f-tree $\mathcal{T}$, with the nodes of processed using a recursion on the shape of $\mathcal{T}$ in a pre-order traversal fashion. The first parameter of the function, Node n, stores a reference to the currently examined node. The remaining three parameters are an accumulated list of nodes and set of relations in the

```
 1: estimate(Node $n$, List nodes$_{path}$, Set rels$_{path}$, double $sel$)
 2:     nodes$_{path}$ $\leftarrow$ nodes$_{path}$ ++ n
 3:     rels$_{path}$ $\leftarrow$ rels$_{path}$ $\cup$ relations(n)
 4:     $sel \leftarrow sel \times$ selectivity(n)
 5:     $card \leftarrow 1$
 6:     foreach relation r in rels$_{path}$ do
 7:         attrs$_r$ $\leftarrow$ attributes(nodes$_{path}$) $\restriction$ attributes(r)
 8:         $card \leftarrow card \times |\pi_{\text{attrs}_r}(r)|$
 9:     size $\leftarrow$ $select \times card$
10:     foreach node $m$ in children($n$) do
11:         size $\leftarrow$ size + estimate($m$, nodes$_{path}$, rels$_{path}$, $sel$)
12:     return size
```

Figure 16: Algorithm for f-representation size estimation.

current node-to-root path, respectively, and a total selectivity factor for the path. The value returned from the function is the size of the subtree rooted at the current node.

Lines 2-4 update the path information with this node's information, where:

- relations(n) returns the set of all relation symbols in the equivalence class labelling the the node n
- selectivity(n) returns a selectivity estimate for the joins we infer from the attributes in the equivalence class labelling the node n

The code in lines 6-8 calculates the cardinality estimates for the node n. For each relation R in the path, we first read the size of the projection of R on attributes in the path from catalog information (line 7, where the call attributes(n) returns the attributes that label node n). We then update the variable *card*, which holds the product of cardinalities (line 8).

In line 9 we calculate the size estimate for this node, which is a product of the accumulated selectivity factor and the product of cardinalities.

Lines 10-11 handle the recursive call of the estimate function for all children of the current node, and updates the size variable accordingly.

We finally return the total size of the subtree rooted at node n in line 12.

We calculate a size estimate for a given f-tree $\mathcal{T}$ by invoking the **estimate** function with the root of $\mathcal{T}$, an empty list, an empty set, and value 1.0 as parameters, respectively.

4.2.2 Time complexity analysis

We now present a time complexity analysis of the size estimation function.

Let $\mathcal{T}$ be an f-tree over which we represent the result of a query Q over a schema with attributes A and relations R. Let $\mathcal{A}$ be the attribute class labelling the currently considered node.

- line 2: a constant time append operation;
- line 3: we use the $\texttt{std::set}$ data structure, giving a constant time set union operation; we store a map of attributes and relations they belong to, giving a total on $|\mathcal{A}|$ operations to iterate over attributes in $\mathcal{A}$;
- line 4: multiplication is a constant time operation; the selectivity(Node n) subroutine requires time proportional to $|\mathcal{A}| \log |\mathcal{A}|$, since for each attribute in $\mathcal{A}$ we need to look up its cardinality;
- line 5: a constant time assignment operation
- line 6-8: we calculate the attribute restriction in time proportional to $|\text{attributes}(\text{nodes}_{\text{path}})| \leq |A|$; we look up the projection cardinality in the catalog in time proportional to $|\text{attrs}_r| \log |\text{attrs}_r| \leq |A| \log |\mathcal{A}|$ (we need to sort the attribute names first); we repeat the loop at most $|R|$ times, giving total time within the order of growth of $|R|\,|A| \log |\mathcal{A}|$;
- line 9: a constant time multiplication and assignment operations
- line 10-11: recursive calls to child nodes

Because we have a total of up to $|A|$ attributes in the equivalence classes labelling the nodes in the tree and there are at most $|R| \leq |A|$ relations on any node-to-root path in the tree, the total time required for the procedure is $\mathcal{O}(|A|^3 \log |A|)$.

We benchmark the performance of the approximation function, as well as the quality of its results over a series of experiments in the next chapter (§5).

4.3 Tools

I have implemented the framework using the C++ programming language, mainly because the existing implementation of the FDB system is also implemented in C++. I used the Eclipse integrated development environment for developing the code together with its incorporated debugger to aid bug fixing. I used the g++ 4.6 compiler on Ubuntu 11.10 to compile my code.

Furthermore, I extensively used FogBugz — an issue tracking and project management system, which includes an evidence based scheduler. This helped me with keeping the development pace at a suitable level to finish the project on time, and aided keeping record of issues and bugs found in the code.

I used the Subversion revision control system to maintain historical versions of files in case I needed to retrieve them at a later point.

4.4 Testing

During the development of the program I have been regularly testing the code to make sure it behaves as expected. Due to the nature of the program, it was difficult to develop a set of unit tests for the size estimation function, because there is no one 'right' answer to an estimate, especially as the method was being refined during the course of the project. Therefore, my primary method of testing was through extensive use of printf statements spread throughout the code. These statements were used to print the state of the program (values of variables and computations) during its runtime, which I then checked for correctness. As I use a #define directive to define the DEBUG flag, the optimiser removes the extra code when the flag is set 0, and, hence, the debugging code has no negative impact on the performance of the program.

I have also prepared a few test datasets which I then used to check if the results produced by the search algorithms were sensible. This was done by inspecting the resulting f-plans, and confirming that the plan returned by the full search was in fact optimal. This has proved to be an effective approach, and allowed me to catch many bugs early on in the development.

5 Experimental Evaluation

In this chapter, we evaluate the cost-based query optimisation technique we introduced in the previous sections in a series of experiments. We use the current asymptotic-based optimiser as a benchmark. The results indicate that our size estimation method is providing accurate estimates when the input data resembles uniform distribution and that our instance-based query optimiser is producing convincingly better f-plans than the default FDB optimiser. In particular:

- The f-representation size estimation function provides very accurate estimates when the input data is uniformly distributed; the quality of estimates degrades for input data generated with a zipf distribution (Experiments 1a and 1b).
- The f-plans returned by our optimiser produce more succinct query result representations compared to the default optimiser. The gap is most significant for queries with up to 2 join conditions (Experiment 2b).
- Our optimiser computes an f-plan for the input query in less time than the default optimiser. This result holds in the full-search case as well as the greedy-search case, although in both cases the size of the explored search space dominates the execution time (Experiment 2a). The time taken by the greedy variant of our optimiser is in the order of milliseconds for up to 5 equalities in the input query. Nevertheless, it still produces f-plans close to optimal (Experiment 2c).

5.1 Setup

All experiments were performed on a machine running a 64-bit Ubuntu 11.10 operating system in a VMWare Virtual Machine with a 64-bit Intel Core i5 2.50Ghz processor and 8GB of RAM.

For each experiment we generate random schemas, data and queries subject to input parameters, which are:

- **R**: the number of generated relations
- **A**: the number of generated attributes, which are distributed uniformly over the relations
- **K**: the number of equalities in the input f-tree

- **L**: the number of equalities in the query
- **data distribution**: for each relation, the tuples are generated either using a uniform distribution or using a zipf distribution; each value in a tuple is generated in the range from 1 to **N**.

5.2 Experiment 1: Size estimation

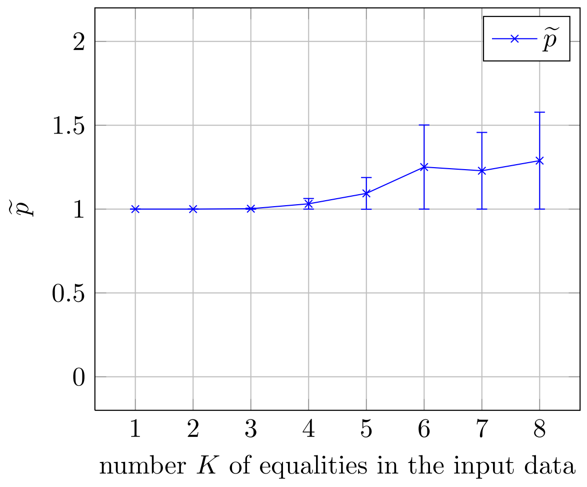


Figure 17: **Experiment 1a:** Proportion of estimated size to actual size of an f-representation for K equalities in the input data on R=4 relations and A=12 attributes; data uniformly distributed with $16^{\text{arity}(R)}$ tuples per relation and each value generated in the range [0..100)

For Experiment 1, we do the following. First, we generate $R = 4$ relations with a total of $A = 12$ attributes, giving 3 attributes per relation. Second, for each value of $K = 1..8$, we generate 40 f-trees for random queries with K transitively independent attribute equalities, giving a total of 240 input f-trees. We then repeat 10 times the following procedure:

1. generate data using a uniform (Experiment 1a) and a zipf (Experiment 1b) distribution with values in the range [1..100) and relation sizes equal to $16^{\text{arity}(R)}$
2. run a catalog generation program that computes, for each relation R, the cardinalities $|\pi_{\mathcal{A}}|$, where $\mathcal{A} \in \mathcal{P}(\text{attrs}(R))$, i.e. the projection sizes for of all subsets of attributes of the relation R

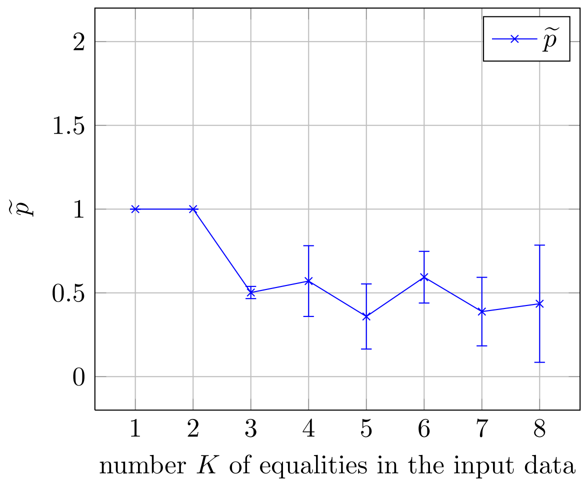


Figure 18: **Experiment 1b:** Proportion of estimated size to actual size of an f-representation for K equalities in the input data on R=4 relations and A=12 attributes; zipf-distributed data with $16^{\text{arity}(R)}$ tuples per relation and each value generated in the range [0..100).

3. execute our size estimation method on the generated data for each f-tree

We then collect the data and report, for each value of K, the median $\widetilde{p}$ of the proportion $p =$ size estimate/actual result size; we plot the results in Figure 17, where error bars indicate the difference between the median first and third quartile values of the results.

Because our size estimation function was devised based on the Uniformity assumption (§ 3.1), the quality of the estimates is very high for uniformly distributed data. The estimates are very precise for $K < 5$. For values of $K \geq 5$ greater that five, the results start to over-estimate the actual result. The error bars start to grow for $K \geq 5$ as well, although the results are quite consistent for all the possible values of K.

In the case of zipf distribution (Experiment 1b) the quality of the estimates is lower. This, however, is *expected* since zipf distribution is far from uniform. In fact, it is highly skewed, which renders our estimation technique fairly inaccurate. Nevertheless, the results indicate that even in the case of zipf distribution our estimates are within an order of magnitude of the actual result size. The estimates are very accurate for values of $K < 3$ and consistent for values of $K < 4$. The graph also indicates that the estimated size is smaller than the actual size. This is because, for zipf distribution, the Independence assumption does not hold either causing our selectivity estimates to be lower than the actual selectivities.

A closer inspection of the results indicated that the accuracy suffers most when multiple equalities are present on a single path. This is due to the fact that any errors in the estimates are growing exponentially with every join present in the path, as we are multiplying the selectivity factors together. The accuracy is better when the joins are evenly distributed among distinct branches in the input f-tree.

5.3 Experiment 2: Query optimisation

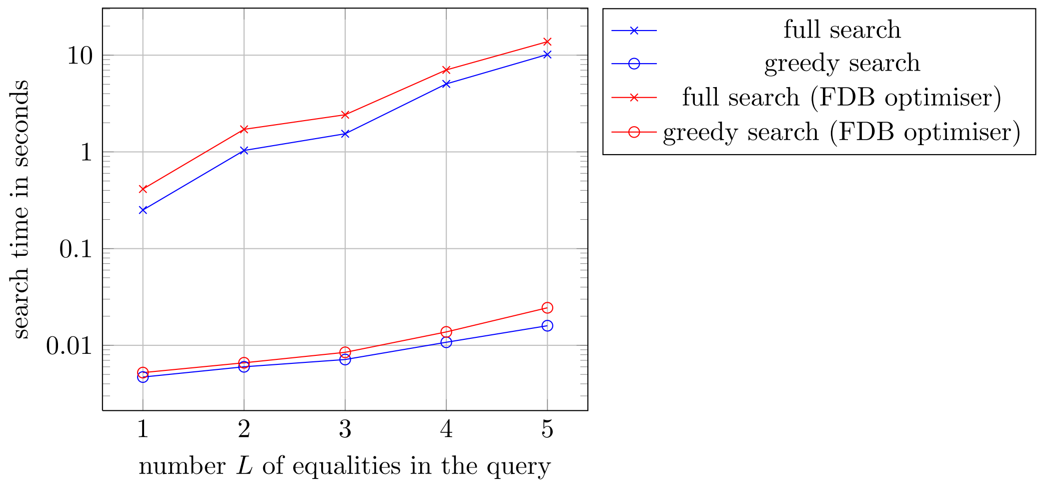


Figure 19: Experiment 2a: Search time for optimal query plan according to full search and greedy heuristic

In this experiment, we test our query optimisation techniques, using both the full search strategy and the greedy heuristic. We set R to 4 and A to 10. We vary the K parameter in the range [1..3], and for each value for K we run 20 tests per possible values for L in the range [1..5]. The data is generated with a uniform distribution, with $24^{\text{arity}(R)}$ tuples per relation, and with each value generated in the range [0..100) (i.e. the N parameter is set to 100). When reporting times we show averaged wall-clock results over five runs. In Figure 19 (Experiment 2a) we report average search times for both search techniques together with search times of the standard FDB optimiser (please note the logarithmic scale for the y-axis). The results indicate that our novel query optimisation technique is performing the search quicker both in the full search case and

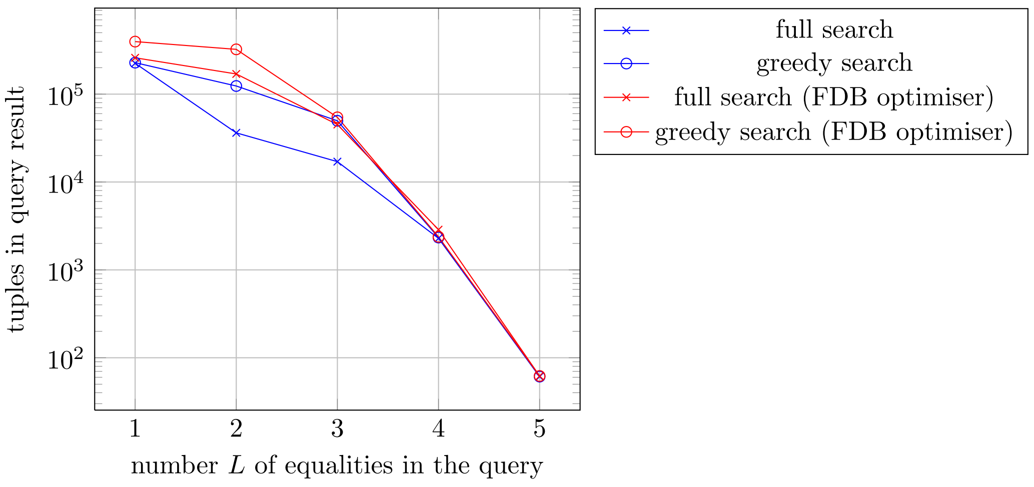


Figure 20: Experiment 2b: Final query result size

when using the greedy heuristic. This difference comes from the fact that the complexity of the size estimation function is lower than the complexity of the linear program used by the default optimiser. However, the order of growth of the plots for the corresponding optimisers is similar because the complexity of the search is dominated by the size of the explored search space. The gap between the corresponding optimisers is slightly larger in the greedy case, because the exact nodes which are explored later on in the search depend on the earlier decisions.

As expected, the execution time of the full-search optimiser grows exponentially with the number of input queries due to the exponential growth in search space. On the other hand, the greedy optimiser exhibits a polynomial growth in execution time as the value for L increases.

Figure 20 (Experiment 2b) reports the average result size produced by executing the f-plans found in Experiment 1a. The graph indicates that the size estimation function based search results in more succinct f-representations. This is an important improvement, as it means that our program will use less memory to process the query. Furthermore, any new queries executed on top of the result f-representations will also likely be more efficient.

The greedy version of our optimiser gives results which are very close to the ones produced by the full search. Furthermore, the quality of the f-plans found by our greedy optimiser matches the one of the f-plans found by the full search variant of the default FDB query optimiser.

As expected, the size of the result is converging towards 0 as the number of equalities in the

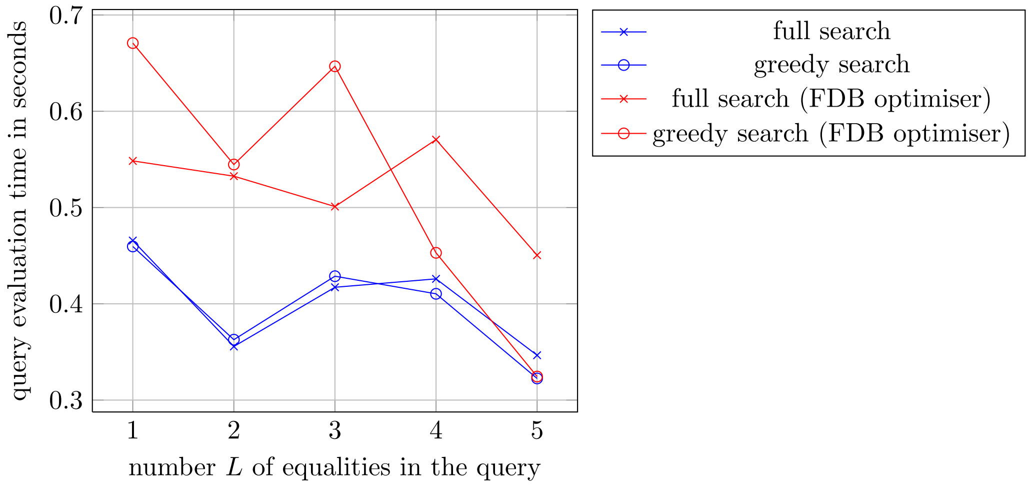


Figure 21: Experiment 2c: F-plan execution time

input query increases.

Interestingly, the asymptotic cost of the final f-tree in f-plans found by the estimates-based search was asymptotically optimal in every single case. This confirms that out optimiser is able to distinguish between f-trees which have equal asymptotic cost, and use catalog information to guide the search towards most cost efficient query plans.

Finally, Figure 21 (Experiment 2c) shows a plot of the time taken by FDB to execute the f-plans found in Experiment 1a. Again, f-plans found by our cost-based query optimisation algorithm are showing better results compared to the f-plans found by the original FDB optimiser. Interestingly, f-plans found by the greedy search are sometimes executed more efficiently compared to f-plans found by the full search. A manual inspection of the results revealed that this is because the greedy heuristic sometimes finds slightly more expensive plans which are, however, made up of fewer transformation steps. Shorter plans are, thus, better handled by the FDB engine, and indicate that our cost-based optimiser would benefit from a heuristic which incorporates the length of the generated f-plan as well as the sizes off the intermediate f-representations.

6 Summary

6.1 Conclusions

For this project, I have successfully applied estimation techniques known from relational databases to estimating result sizes of queries on factorised databases. I first derived a size estimation technique based on catalog information and common assumptions about the input data, and then used the estimate as a cost function in query optimisation. I have also implemented a loosely coupled, modular f-plan search framework on top of the FDB engine, which can easily be extended with new search strategies and definitions of the cost function. The design choices made appear to be successful, as I have extended the system with a cost function that mimics the original FDB optimiser without any modifications to the search code.

I have performed careful experimental evaluation to compare the performance of my query optimiser with the one currently provided by FDB. The results indicate that my optimiser finds f-plans which give a convincing improvement in query evaluation time of across a wide range of input data, and result in more succinct representations of the result. Furthermore, the execution time of the search itself is an improvement over the original optimiser. Importantly, the gains hold not only for the exhaustive search optimiser, but are also sustained when we compare the optimisers based on greedy heuristic.

Of course, because of the assumptions we made when deriving our size estimation method the quality of the estimates varies depending on the distribution of the input data. This has been verified by the experiments, which indicated a decline in accuracy for zipf (and so highly non-uniform) distributed data.

6.2 Future work

There are a number of directions in which the project can be extended:

- We have developed the estimation method based on the uniformity and independence assumptions, which often do not hold for real data. There has been a lot of research devoted to relaxing these assumptions for relational databases [6, 10], and these techniques can also be adapted to factorised databases. Furthermore, due to the fact that real data such as word frequencies in languages or people's names often follows the zipf distribution, it would be beneficial to adjust our selectivity estimates to work better with the zipf distribution [4];

- Sampling is another common technique of query-cost estimation in relational databases. We could extend our f-representations size estimation system with a sampling-based approach, or, perhaps, introduce a hybrid method which uses catalog information when available and falls back on sampling techniques when no such information is present;

- Our experiments indicate that sometimes the full search returns an f-plan which is suboptimal in terms of execution time. It would be worthwhile to investigate heuristics which would combine multiple cost functions into a single cost model, aiming to minimise for multiple objectives at once. One such heuristic would first find all f-plans which require fewest transformation steps to evaluate a query, and then choose the cheapest one out of those plans with respect to the cost function based on estimates as the final output.

6.3 Acknowledgements

I would like to thank my project supervisor Dr Dan Olteanu for introducing me to the topic of factorised representations and supervising my work. I would also like to thank Jakub Závodný for his helpful comments and assistance throughout this project. Finally, I would like to thank Professor Luke Ong for being my tutor and advisor throughout the four years of my degree.

References

[1] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodnỳ. Fdb: A query engine for factorised relational databases. *PVLDB*, 2012.

[2] M.L. Brodie, J. Mylopoulos, and J.W. Schmidt. *On conceptual modelling: perspectives from artificial intelligence, databases, and programming languages.* Topics in information systems. Springer-Verlag, 1984.

[3] Latha S. Colby. A recursive algebra and query optimization for nested relations. *SIGMOD Rec.*, 18(2):273–283, June 1989.

[4] Christos Faloutsos and H. V. Jagadish. On b-tree indices for skewed distributions. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 363–374, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[5] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First design patterns.* O'Reilly, Sebastopol, CA, 2004.

[6] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. *SIGMOD Rec.*, 20(2):268–277, April 1991.

[7] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, June 1984.

[8] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical profile estimation in database systems. *ACM Comput. Surv.*, 20(3):191–221, September 1988.

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

[10] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[11] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000.

[12] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[13] Arun Swami and K. Bernhard Schiefer. On the estimation of join result sizes. In *Proceedings of the 4th international conference on extending database technology: Advances in database technology*, EDBT '94, pages 287–300, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

A Source Code

The entire code repository for the project is available at `svn://edison.cs.ox.ac.uk/olteanu/fdb/cost`

A selection of classes from the project follows.