Computing Queries on Linear Constraint Databases

Stéphane Grumbach*
University of Toronto
and INRIA

Zoé Lacroix†
L.R.I.- Université de Paris Sud
and INRIA

Abstract

We consider rational linear constraint databases, and study the problem of evaluating efficiently first-order queries with linear constraints. We show that although queries admit low data complexity (NC), their naive evaluation is rather inefficient. The computation paradigm is based on relational algebra and constraint solving. We focus on the former, and show that the query processing should differ drastically from classical relational databases. In particular, optimization strategies used for relational queries may be inadequate. We describe the problem and propose some preliminary optimization principles.

1 Introduction

Until recently, databases were considered to be finite collections of data items. New applications such as those involving temporal and spatial data (e.g. geographical databases) lead very naturally to more general data models allowing infinite collections of items to be stored in the database. This is possible under the assumption that the databases admit a finite representation. In their seminal paper, Kanellakis, Kuper and Revesz [KKR90], introduced *constraint query languages*. The basic idea is to generalize the relations of the relational model [Cod70] by defining generalized tuples as conjunctions of constraints. For instance, $x \leqslant y \wedge x \leqslant 0$ defines a binary generalized tuple. A generalized, or finitely representable, relation is then a finite set of such tuples.

We focus on linear constraints over the rational numbers, and databases defined by means of such constraints. We consider first-order queries with linear constraints, over the language $\mathcal{L} = \{\leqslant, +\} \cup \mathbb{Q}$. First-order queries can be evaluated bottom-up in closed form, i.e., the output of any query on a input consisting of a set of generalized tuples is a set of generalized tuples. This follows from the fact that the theory of dense order without endpoints and with addition admits quantifier elimination. The data complexity, defined with respect to the size of the formulas representing the input relations, has been shown in [KKR90] to be in NC (the class of problems that can be solved on

*I.N.R.I.A. Rocquencourt BP 105, 78153 Le Chesnay, France stephane.grumbach@inria.fr
Work supported in part by Esprit Project BRA AMUSING, and an NSERC fellowship in Canada.
†L.R.I., Université de Paris Sud, 91405 Orsay, France – zoe@lri.fr – Work done in the Verso Project at INRIA.

boolean circuits with a polynomial number of processors and polylog depth). A lower data complexity upper-bound, AC^0 (the class of problems that can be solved on boolean circuits (with arbitrary fan-in) with a polynomial number of processors and constant depth) was obtained over restricted classes of inputs with essentially a bounded number of slopes in the inputs [GST94]. These results show that first-order queries with linear constraints have very low data complexity.

Nevertheless, there are fundamental difficulties in the implementation of constraint databases and query languages. First of all, efficient methods to access the data on secondary storage are needed. Indexing for constraint databases has been studied in [KRVV93]. Another issue is the ability to optimize queries, and this is the one we address here.

In order to study the evaluation of first-order queries and define optimization strategies, we use the algebra introduced in [GST94] which is equivalent to first-order with linear constraints. It is very similar to the classical relational algebra and consists of the following operations: Cartesian product, selection, projection, set operations (union, intersection and set difference) and rename. The evaluation of a query consists of evaluating purely algebraic operations and solving systems of linear equations and inequalities. The main computational difference between linear constraint databases and finite relational databases, is that the size of the input may vary dramatically during the computation. Moreover, the effect of algebraic operations is drastically different. For instance, the projection, can increase the size of an input in a quadratic way, while it always reduces the size in relational databases. Relations of unpredictable size are costly to manipulate due in particular to complex memory management [Ull88].

The size of the relations can be controlled by simple geometric transformation. Relations of arity (dimension) k, can be seen as finite unions of polyhedra of at most $k+1$ facets. A polyhedron of $k+1$ facets can be represented by $k+1$ constraints over k variables. We define a normal form of relations of arity k based on the fact that each tuple represents a single polyhedron of at most $k+1$ facets, and tuples are disjoint (duplicate free). There is a trade-off between normalizing relations during computation and manipulating non normal form relations. Relations in normal form are extremely convenient for space management on disc since tuples have a fixed size. Moreover, algebraic operations are easier to perform on normalized relations. On the other hand, normalization is a costly operation, and in addition it may result in an increase of the number of tuples in the relation.

We define a list of primitive operations used in the evaluation of queries. Most of them are similar to classical techniques for constraint solving [PS85, DHS88]. We then study the complexity of evaluating algebraic operations in terms of the primitive operations involved. Finally, we study various query evaluation schemes (execution order of the algebraic operations intertwined with constraint solving). The query modification rules differ sometimes from those of the classical relational algebra.

The paper is organized as follows. In the next section, we present an exam-

Cities		
Name	Area	Population
Paris	$3x_O - 4x_A \leqslant 2$ $x_O \leqslant 10$ $x_O - x_A \geqslant -3$ $x_O + 3x_A \leqslant 37$ $x_O \geqslant 6$ $3x_O + 4x_A \geqslant 46$	$2,000,000$
Rocquencourt	$x_A \geqslant 0$ $x_O \geqslant 8$ $2x_O + 2x_A \leqslant 17$	$3,877$
Orsay	$2x_O - 2x_A \geqslant -13$ $2x_O + 2x_A \leqslant 17$ $2x_O \geqslant 1$	$15,000$

Forests
$2x_O \leqslant 19$ $-3x_O - 4x_A \leqslant 2$ $2x_O - 6x_A \geqslant -17$
$2x_O - 6x_A \geqslant -17$ $2x_A \geqslant 11$ $2x_O - 4x_A \leqslant 5$
$2x_A \leqslant 15$ $4x_O - 2x_A \geqslant -13$ $4x_O + 2x_A \geqslant 13$
$x_O \geqslant 9$ $9x_O - x_A \leqslant 85$ $x_O - x_A \leqslant 10$

Figure 1: A Database Instance, I

ple of a linear constraint database with some natural queries and give formal definitions. Section 3 is devoted to the algebra, and the effect of algebraic operators on the representation of databases. In Section 4, we introduce normal forms of tuples and relations, and the list of primitive operations used in query evaluation. Finally, in Section 5, we analyze the query modification techniques of the relational model in the context of linear constraint databases.

2 Linear Constraint Databases

Let us start with a simple example of a linear constraint database over the schema $\sigma = \{C, F\}$ consisting of a relation C (Cities) of arity 4, and a relation F (Forests) of arity 2, respectively over the attributes (N, A, O, P) and (A, O), where N stands for name, A for abscissa (longitude), O for ordinate (latitude), and P for population. N and P are of *alphanumeric type*, while A and O are of *rational type*, $\mathbb{Q}$.

The relations C and F are sets of *Generalized tuples* which are conjunctions of constraints. The first tuple in relation "Cities" for instance contains the area of Paris, an infinite set of points over $\mathbb{Q}^2$, defined by linear constraints (inside the tuple box on Figure 1). Relations are infinite, but represented in a finite way using constraints.

We next consider some simple queries expressed in the relational calculus with quantification over $\mathbb{Q}$.

$\mathcal{Q}_1$: "*Which are the cities with more that 15 000 inhabitants?*"
$\mathcal{Q}_1$ is a non-spatial query expressible in the relational calculus by:

$$\{x_N \mid \exists x_A \exists x_O \exists x_P (C(x_N, x_A, x_O, x_P) \wedge 15000 \leqslant x_P)\}.$$

$\mathcal{Q}_1(I) = \{Paris, Orsay\}$. Consider now the spatial query:

$\mathcal{Q}_2$: *"Which are the cities that are adjacent to a forest?"*
expressed by: $\{x_N \mid \exists x_A \exists x_O \exists x_P (C(x_N, x_A, x_O, x_P) \wedge F(x_A, x_O))\}$. The output, computed by constraint solving is $\mathcal{Q}_2(I) = \{Paris, Orsay\}$. Finally, consider the query:

$\mathcal{Q}_3$: *"Which are the longitudes of parts of cities different from Paris that are in a forest?"*
expressed by: $\{x_A | \exists x_N \exists x_O \exists x_P (C(x_N, x_A, x_O, x_P) \wedge F(x_A, x_O) \wedge x_N \neq Paris)\}$. $\mathcal{Q}_3$ has a non-alphanumeric output: $\mathcal{Q}_3(I) = \{[7 \leqslant x_A \wedge 2x_A \leqslant 15]\}$.

We next define formally linear constraint databases. Consider the first-order language $\mathcal{L} = \{\leqslant, +\} \cup \mathbb{Q}$ and the structure $\mathbb{Q} = \langle \mathbb{Q}, \leqslant, +, (q)_{q \in \mathbb{Q}} \rangle$, of the linearly ordered set of the rational numbers with addition and all rational constants. Therefore, constraints will be composed of linear equations or inequalities of the form:

$$\textstyle\sum_{i=1}^{p} a_i\, x_i = a_0, \qquad \sum_{i=1}^{p} a_i\, x_i \leqslant a_0$$

where the x_i's denote variables and the a_i's are integer constants (note that rational constants can always be avoided in linear equations and inequalities).

Let $\sigma = \{R_1, \ldots, R_n\}$ be a signature (or a database schema) such that $\mathcal{L} \cap \sigma = \emptyset$, where $R_1, \ldots, R_n$ are relation symbols. We distinguish between *logical predicates* (e.g., $=, \leqslant$) in $\mathcal{L}$ and *relations* in σ. We next introduce a restricted definition of *finitely representable structures* [GS94]. We consider expansions of $\mathbb{Q}$ to σ. Intuitively, the relations in σ constitute a database *in the context* of $\mathbb{Q}$.

Definition 1 Let $S \subseteq D^k$ be some k-ary relation. The relation S is *finitely representable in $\mathcal{L}$ over $\mathbb{Q}$* (*$\mathcal{L}$-representable* for short) if there exists a quantifier free formula $\varphi(x_1, \ldots, x_k)$ in $\mathcal{L}$ with k distinct free variables $x_1, \ldots, x_k$ such that:

$$\mathbb{Q} \models \forall x_1 \cdots x_k (S(x_1, \ldots, x_k) \leftrightarrow \varphi(x_1, \ldots, x_k)).$$

Let $\mathcal{A}$ be an expansion of $\mathbb{Q}$ to σ. The structure $\mathcal{A}$ is finitely representable (over $\mathbb{Q}$) if for every relation symbol R in σ, $R^{\mathcal{A}}$ is $\mathcal{L}$-representable (over $\mathbb{Q}$).

A *(database) instance (of σ)* is a mapping which associates with each k-ary relation symbol R in σ a quantifier-free formula in disjunctive normal form with k distinct variables. Clearly, each instance of σ corresponds to the restriction of a finitely representable structure to σ. In practice, we assume that the databases contain the formula defining their relations. Instances will be denoted by the letters I, J, etc. Note that the class $\mathcal{K}$ of σ-instances is effectively enumerable, and that instances are recursive. $\mathcal{K}$ has interesting closure properties. It is closed under finite union and intersection and moreover under complementation.

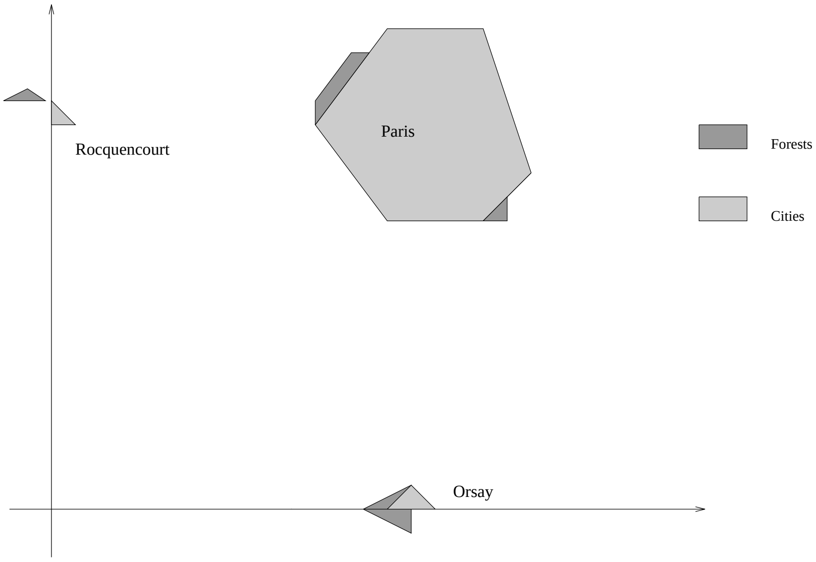

Figure 2: Representation of the relations Cities and Forests in I

3 The Algebra

We recall in this section the algebra $\mathrm{ALG}_{\mathcal{L}}$ for linear constraint databases, which it equivalent to $\mathrm{FO}_{\mathcal{L}}$ [GST94]. This algebra is similar to Codd's algebra for finite relations [Cod70], but the operators apply to the finite representations of possibly infinite sets. The algebra consists of the following operations: Cartesian product, $\times$, joins, $\bowtie$, selections (σ_F), projection, π, set operations (union, $\cup$, intersection, $\cap$, and set difference, $-$), and rename, ρ. The algebra operations are performed on sets of generalized tuples, i.e. on quantifier-free formulas in *disjunctive normal form.* Unlike the algebra of Kanellakis and Goldin [KG94], no special encoding for relations and generalized tuples is assumed. $\mathrm{ALG}_{\mathcal{L}}$ can also be viewed as a simplified sub-language of the algebra of Paredaens, Van den Bussche and Van Gucht [PVV94] (which also includes multiplication). Suppose R is an n-ary relation represented by a quantifier-free formula, φ, of the form:

$$\varphi \equiv \bigvee_{i=1}^{k} \bigwedge_{j=1}^{\ell_i} \varphi_{i,j}$$

where the $\varphi_{i,j}$'s are atomic formulas. Then, we also denote the representation φ as a collection of generalized tuples t_i in the set notation:

$$\left\{ t_i \;\middle|\; 1 \leqslant i \leqslant k,\; t_i = \bigwedge_{j=1}^{\ell_i} \varphi_{i,j} \right\}$$

Furthermore, if I is an instance over signature σ and $R \in \sigma$, we consider the relation $I(R)$ as a set of generalized tuples as above. We also assume that attributes (columns) of relations have names and for each attribute name A, there is a distinct variable x_A associated with it. Attribute names are usually denoted by $A, B, C, \ldots$ (and possibly with subscripts). When the context is clear, we may blur the distinction between variables and attribute names. The family of *algebraic expressions* over a schema σ is defined as usual.

We now describe the *semantics* of the algebra. (Note that the operators work directly on generalized tuples, so the semantics is given with respect to generalized tuples.) Suppose that I is an instance of σ, and e is an expression over σ. The *result* of e on I, denoted by $e(I)$, is defined inductively as follows:

1. (a) If $e = (R)$, $e(I) = I(R)$ (a set of generalized tuples).
   (b) If $e = (A : \mathbb{Q})$, $e(I) = \{x_A = x_A\}$, where x_A is the variable corresponding to the attribute A.

2. If $e = (e_1 \times e_2)$, then $e(I) = \{t_1 \wedge t_2 \mid t_1 \in e_1(I), t_2 \in e_2(I)\}$.

3. If $e = (e_1 \bowtie e_2)$, then $e(I) = \{t_1 \wedge t_2 \mid t_1 \in e_1(I), t_2 \in e_2(I)\}$.

4. If $e = (\sigma_F\ e_1)$, where F is a *selection formula*, that is an atomic formula of one of the following three forms: $t_1 = t_2$, $t_1 \leqslant t_2$, $t_1 + t_2 = t_3$, where t_1, t_2, t_3 are attribute names or constants (in $\mathbb{Q}$), $e(I) = \{t \wedge F \mid t \in e_1(I)\}$, where each attribute name A in F is replaced with the corresponding variable x_A.

5. If $e = \pi_{B_1,\ldots,B_k}\ e_1$, then $e(I)$ is obtained from $e_1(I)$ by "eliminating" the variables which do not correspond to attributes B_1 through B_k. One proceeds as follows. Suppose $e_1(I) = \{t_1, \ldots, t_m\}$ and has attributes $A_1, \ldots, A_n$ and $\{C_1, \ldots, C_{n-k}\} = \{A_1, \ldots, A_n\} - \{B_1, \ldots, B_k\}$. We apply the well-known Fourier-Motzkin Elimination method [Sch86] (see below) to eliminate one by one all existentially quantified variables $x_{C_1}, \cdots, x_{C_{n-k}}$ in each of the formulas $\exists x_{C_1} \cdots \exists x_{C_{n-k}} t_i$. Each tuple t_i then results in t'_i. Finally, $e(I) = \{t'_1, \ldots, t'_m\}$.

6. (a) If $e = (e_1 \cup e_2)$, then $e(I) = e_1(I) \cup e_2(I)$.
   (b) If $e = (e_1 \cap e_2)$, then $e(I) = \{t_1 \wedge t_2 \mid t_1 \in e_1(I), t_2 \in e_2(I)\}$.
   (c) If $e = (e_1 - e_2)$, then $e(I) = \{t_1 \wedge t_2 \mid t_1 \in e_1(I), t_2 \in (e_2(I))^c\}$, where R^c is the complement of R obtained as follows. Suppose $R = \{t_1, \ldots, t_n\}$ is a set of generalized tuples and for each i, $t_i = \bigwedge_j \varphi_{i,j}$. Then R^c is the formula in disjunctive normal form which is equivalent to $\bigwedge_i \bigvee_j \neg\varphi_{i,j}$.

7. If $e = \rho_{A \rightarrow B}\ e_1$, then $e(I) = e_1(I)[x_A/x_B]$ (all occurrences of x_A are replaced by x_B).

The Fourier-Motzkin elimination method (see for instance [Sch86], pp. 155–157) works as follows. Consider a generalized tuple t which defines a polyhedron $P(\overline{x}, y) \subseteq \mathbb{Q}^{n+1}$ described by the inequalities (once the coefficients of y have been normalized):

$$\begin{cases} a^\ell \overline{x} + y \leqslant a_0^\ell & \text{for } \ell = 1, \ldots, L \\ b^k \overline{x} - y \leqslant b_0^k & \text{for } k = 1, \ldots, K \\ c^i \overline{x} \leqslant c_0^i & \text{for } i = 1, \ldots, I \end{cases}$$

where $\overline{x}$ ranges over $\mathbb{Q}^n$, and y over $\mathbb{Q}$. One can show that after the "elimination" of y (i.e. after P has been projected on its first n coordinates), the relation over $\overline{x}$ is exactly:

$$\left\{ \; \overline{x} \in \mathbb{Q}^n \;\middle|\; b^k \overline{x} - b_0^k \leqslant a_0^\ell - a^\ell \overline{x} \;\text{ for all } \ell \text{ and } k,\; c^i \overline{x} \leqslant c_0^i \;\text{ for all } i \; \right\}.$$

Therefore:

$$\pi_{\overline{x}}\, t = \bigwedge_{1 \leqslant k \leqslant K, 1 \leqslant \ell \leqslant L} b^k \overline{x} - b_0^k \leqslant a_0^\ell - a^\ell \overline{x} \;\wedge \bigwedge_{1 \leqslant i \leqslant I} c^i \overline{x} \leqslant c_0^i.$$

Note that the expressions for $\bowtie$, $\cap$ and $\times$ are similar. The attributes are assumed to be identical (pairwise distinct) in case of the intersection (Cartesian product), while no assumption is made for the join operator which generalizes them all. We chose to offer all these operators as in the relational algebra, where the join operator can also simulate Cartesian product and intersection.

It was shown in [GST94] that the algebra $\mathrm{ALG}_{\mathcal{L}}$ is equivalent to first-order logic over the class of linear constraint databases. The proof is quite similar to that of the equivalence of the classical relational algebra and calculus over finite structures. The combination of selections and Cartesian products can yield complicated form of selections. The following example illustrates the equivalence.

Example 1 Consider again the queries $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$ of Section 2. The equivalent algebra queries are:
$\mathcal{Q}_1 = \pi_N \sigma_{15000 \leqslant P}(C^I)$,
$\mathcal{Q}_2 = \pi_N(C^I \bowtie F^I)$, and
$\mathcal{Q}_3 = \pi_A((\pi_{AO} \sigma_{N \neq Paris} C^I) \cap F^I)$.

The data complexity of linear constraint queries has first been investigated in [KKR90], where it was shown more generally that polynomial constraint queries had NC data complexity. It has been shown in [GST94] that the data complexity of linear constraint queries restricted to k-bounded classes of linear constraint inputs is in AC^0 (a class of linear constraint relations is k-bounded if each individual linear constraint appearing in a tuple of a relation contains at most k occurrences of the addition symbol, and only integer parameters).

4 Query Evaluation and Normal Form

In this section, we introduce canonical representations of linear constraint database relations, and define the basic primitive operations on which the evaluation of the algebraic operations rely. The cost of evaluating individual operations (in terms of primitives operations) is presented in the appendix. First remark that the behavior of the operations differ from the classical finite relational case.

The projection on O of the tuple (with $n = 7$ constraints) on the right, results in a tuple with 12 constraints many of them being redundant. We say that a constraint C is *redundant* in a tuple $t = t' \wedge C$ if $t \sim t'$. The blow up is quadratic (in the order of $(\frac{n}{2})^2$).

$3x_O$	$-8x_A$	$\leqslant$	-38
$3x_O$	$+2x_A$	$\geqslant$	33
x_O		$\leqslant$	10
x_O	$+x_A$	$\geqslant$	13
x_O	$-x_A$	$\geqslant$	8
x_O		$\geqslant$	6
x_O	$+3x_A$	$\leqslant$	46

It follows that the projection of linear constraint database relations may drastically increase the number of constraints per tuple and so the size of the relation. This is unlike the relational model.

The intersection may also result in numerous redundant constraints and empty tuples also with a quadratic blow-up. The selection doesn't reduce the size of the relation either. A computing/cleaning step is required after one of these operations has been executed, but it is not always clear when it should be done. This question has similarities with the duplicate elimination problem in SQL.

The result of a query over linear constraint relations may contain an important amount of redundancies. (i) Tuples may be *over-defined*, if they contain redundant constraints (such as $x \leqslant 3$ and $x \leqslant 4$) or *unsatisfiable*, if they contain contradictory constraints (such as $x \leqslant 3$ and $x \geqslant 4$). A precise description of all classes of redundant constraints can be found in [LHM89]. (ii) Relations, may contain duplicates, if distinct tuples have non empty intersection.

We introduce a *normal form* for generalized tuples with respect to k variables. It is related to the number of constraints in each tuple. Note that only $(k+1)$ constraints are necessary to define polyhedra with at most $(k+1)$ facets (polygons).

Definition 2 A generalized tuple is in *normal form* with respect to k variables if it contains at most $(k+1)$ constraints and they are not redundant.

It is easy to observe [GST94] that any finitely representable k-ary relation can be represented as a finite union of pairwise disjoint polyhedra with $k+1$ facets (triangles in dimension 2).

Definition 3 A relation over k variables is in *normal form* if each tuple is in normal form with respect to k variables, and tuples are pairwise disjoint.

The normalization is an important step in the evaluation process. Since the number of constraints per tuple is bounded, it results in an interesting gain in terms of complexity.

Tuple Normalization: The algorithm which normalizes k-tuples performs the following: (i) verify if the input k-tuple is the definition of a non empty polyhedron, (ii) remove redundant constraints, and finally (iii) verify if the input k-tuple has at most $(k+1)$ constraints which are not redundant; otherwise normalize it (i.e. split the polyhedron with $n > k+1$ facets into polyhedra of at most $(k+1)$ facets).

Relation Normalization: The algorithm which normalizes relations works on a set of normalized k-tuples as follows: for each pair of tuples in the relation, check if the intersection is empty; if not, remove the intersection from one of the two tuples and normalize it.

The primitive operations that are used to achieve normalization and to implement the algebraic operations are the following.

The EMPTINESS-REDUNDANCY algorithm[1] tests if the input k-tuple is the definition of a non empty polyhedron, and remove redundant constraints. The cost of the EMPTINESS-REDUNDANCY test is $ER(n)$ (where n the number of constraints in the input).

The 1-TUPLE-NORMALIZATION algorithm takes as input a tuple and outputs an equivalent generalized relation in normal form. It divides tuples that represent polyhedron of two many facets. A Greedy Triangulation algorithm which is a particular case of 1-TUPLE-NORMALIZATION in dimension 2 is described in [PS85]. The cost of the 1-TUPLE-NORMALIZATION algorithm is $TN(n)$ (where n the number of constraints in the input).

The RELATION-NORMALIZATION algorithm takes a relation as input (set of generalized tuples) and outputs an equivalent relation in normal form. The cost of the RELATION-NORMALIZATION algorithm is $RN(p,n)$ (where p is the number of tuples and n the maximum number of constraints per tuple in the input).

The FOURIER-MOTZKIN algorithm takes as input a relation and an attribute name, and outputs a relation from which the attribute name has been eliminated. The cost of the FOURIER-MOTZKIN algorithm is $FM(p,n)$ (where p is the number of tuples and n the maximum number of constraints per tuple in the input).

The SET-COMPLEMENT algorithm takes as input a relation and outputs a relation in normal form that is equivalent to the complement of the input. The cost of the SET-COMPLEMENT algorithm is $SC(p,n)$ (where p is the number of tuples and n the maximum number of constraints per tuple in the input).

[1] This algorithm gives an output similar to CHIP's algorithm [DHS88].

5 Towards Optimization Strategies

The optimization of algebraic expressions is based on algorithms to perform query modification [Mai83, Ull88]. We analyze in this section the classical equivalence rules of algebraic expressions in the relational algebra. We distinguish three steps in the evaluation process:

- *syntactic computation*: purely what the algebra does.
- *semantic computation*: remove empty tuples and redundant constraints.
- *normalization* of the output assuming the input is normalized.

The evaluation of a query in constraint databases implies several syntactic computation steps and, as it is shown in Example 2, at least, one semantic computation step. The *naive evaluation* consists in doing successively all syntactic computations and a final semantic computation at the end. Note that the semantic computation we consider only consists in removing empty tuples and redundant constraints with the EMPTINESS-REDUNDANCY algorithm applied on tuples. The *normalization* of the output removes redundant tuples.

We first study the cost of algebraic operations in the case of the *naive evaluation*. We then propose an algorithm based on equivalent expressions and the *naive evaluation*. Note that this algorithm, *Naive Strategy*, does not guarantee optimality. It follows a worst case analysis. Then, we use respectively the *semantic computation* and the *normalization* as tools to further optimize. Optimization of the evaluation of constraint queries supposes intertwining algebraic steps with constraint solving steps (EMPTINESS-REDUNDANCY or NORMALIZATION). This constitutes of course a difference with the relational model. We propose two other algorithms *Semantical Strategy* and *Normalization Strategy* which are based on these properties. The algorithms we introduce take as input an algebraic expression, and output an evaluation plan.

Example 2 Consider again the query $\mathcal{Q}_1$ and the database instance I of Section 2.

$\mathcal{Q}_1 = \pi_N \sigma_{15000 \leqslant P} \pi_{N,P}(C^I)$

The successive steps of the syntactic computation are described in Figure 3. After the syntactic computation, we can see that if no semantic computation is done, then the output of $\mathcal{Q}_1(I) = \pi_N \sigma_{15000 \leqslant P} \pi_{N,P}(C^I)$ would be $\{\text{Paris}, \text{Rocquencourt}, \text{Orsay}\}$ since no syntactic step can witness that $15000 \leqslant 3877$ is an unsatisfiable constraint. This example shows the need of a semantic computation step after the syntactic computation.

We describe in Appendix A.1 the cost of algebraic operations. It depends upon the *dimension* or arity of the input relations, the *size* of the relations (number of tuples) and the number of constraints per tuple. We distinguish two types of costly operations:

1. those for which the syntactic computation itself is a costly operation,

$\pi_{N,P}(C^I)$		
$x_N =$ Paris	$\wedge$	$x_P = 2,000,000$
$x_N =$ Rocquencourt	$\wedge$	$x_P = 3,877$
$x_N =$ Orsay	$\wedge$	$x_P = 15,000$

$\sigma_{15000 \leqslant P}(\pi_{N,P}(C^I))$				
$x_N =$ Paris	$\wedge$	$x_P = 2,000,000$	$\wedge$	$15,000 \leqslant x_P$
$x_N =$ Rocquencourt	$\wedge$	$x_P = 3,877$	$\wedge$	$15,000 \leqslant x_P$
$x_N =$ Orsay	$\wedge$	$x_P = 15,000$	$\wedge$	$15,000 \leqslant x_P$

$\pi_N(\sigma_{15000 \leqslant P}(\pi_{N,P}(C^I)))$		
$x_N =$ Paris	$\wedge$	$15,000 \leqslant 2,000,000$
$x_N =$ Rocquencourt	$\wedge$	$15,000 \leqslant 3,877$
$x_N =$ Orsay	$\wedge$	$15,000 \leqslant 15,000$

Figure 3: Syntactic computation of $\mathcal{Q}_1(I)$

2. those whose computation increases the number of tuples and/or the number of constraints per tuple.

All syntactic computations, except the projection and the complementation (which is used to compute the set-difference), are linear in the number of tuples.Projection and complementation use respectively the FOURIER-MOTZKIN, FM and SET-COMPLEMENT, SC algorithms. FM depends mainly upon the arity of the relation and the number of constraints per tuple. SC depends on the number of tuples and the number of constraints per tuple, in the relation. Note that SC and FM are used only in the presence of multi-variable linear constraints. It follows that an optimization strategy has to decrease as much as possible these parameters before applying those algorithms. Note that in contrast to the classical relational algebra, the projection operator shall not be used at the beginning of the computation, if possible, in order to decrease first the number of constraints per tuple.

The second class of costly operations includes selection, projection, intersection, join and set-difference. These operations may generate redundant tuples (apart from projection), redundant constraints per tuples and empty tuples. It follows that a *semantic computation* or a *normalization* may decrease a lot the different parameters.

Remark: In contrast to the relational algebra, no syntactic operation decreases at least one of the parameters (the number of tuples, the number of constraints and the arity) except the projection which decreases the arity but increases quadratically the number of constraints per tuple. Only the *semantical computation* decreases, as much as possible, the parameters. It follows that the *Naive Strategy* is quite different from the *Semantical* and *Normalization* strategies.

We analyze now the classical equivalence rules of algebraic expressions in the relational algebra. For each rule, we compare the cost of the syntactic computation of both expression. The cost model is very simple. It relies only on the cost of the primitive operations introduced in the previous section. Moreover, it is always a worst case analysis. The strategy suggests an evaluation order on the algebraic operations of a query and the normalization operations (linear constraint solving) which is different from the algorithm described in [BJM93]. The details of the cost computation can be found in the Appendix.

Let e_i ($i \in \mathbb{N}$) be an algebra expression of arity k_i containing p_i tuples of at most n_i constraints. Let F_i be a selection formula, and A_i an attribute name. The following rules are commonly used:

(1) Commutativity and associativity for products: $e_1 \times e_2 = e_2 \times e_1$; $(e_1 \times e_2) \times e_3 = e_1 \times (e_2 \times e_3)$.

(2) Cascade of projections:
$\pi_{A_{i_1},\ldots,A_{i_m}}(\pi_{A_1,\ldots,A_n}(e_1)) = \pi_{A_{i_1},\ldots,A_{i_m}}(e_1)$ if $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$.

(3) Cascade of selections: $\sigma_{F_1}(\sigma_{F_2}(e_1)) = \sigma_{F_2}(\sigma_{F_1}(e_1)) = \sigma_{F_1 \wedge F_2}(e_1)$.

In Contrast to the relational case, $\sigma_{F_1 \wedge F_2}(e_1)$ is the most efficient.

(4) Commuting selection with Cartesian product:
(4a) $\sigma_F(e_1 \times e_2) = \sigma_F(e_1) \times e_2$ if all attributes mentioned in F are of e_1

$\sigma_F(e_1) \times e_2$ is the most efficient.

(4b) $\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2)) = \sigma_{F_1}(e_1) \times \sigma_{F_2}(e_2)$ if F_1 only involves attributes of e_1 and F_2 only involves attributes of e_2

$\sigma_{F_1}(e_1) \times \sigma_{F_2}(e_2)$ is the most efficient.

(4c) $\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2)) = \sigma_{F_1}(\sigma_{F_2}(e_1) \times e_2) = \sigma_{F_1 \wedge F_2}(e_1 \times e_2)$ if all attributes mentioned in F_2 are of e_1

$\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$ is the most efficient.

(5) Commuting selection with a union: $\sigma_F(e_1 \cup e_2) = \sigma_F(e_1) \cup \sigma_F(e_2)$.

The two expressions have similar costs.

(6) Commuting selection with an intersection: $\sigma_F(e_1 \cap e_2) = \sigma_F(e_1) \cap \sigma_F(e_2) = \sigma_F(e_1) \cap e_2$.

$\sigma_F(e_1) \cap e_2$ is the most efficient. Note that in contrast to the relational case were $\sigma_F(e_1) \cap \sigma_F(e_2)$ is more efficient, in constraint databases this later expression is not only the most costly one, but it also generates redundant constraints in each tuple.

(7) Commuting selection with a join:
(7a) $\sigma_F(e_1 \bowtie e_2) = \sigma_F(e_1) \bowtie e_2$ if all attributes mentioned in F are of e_1

$\sigma_F(e_1) \bowtie e_2$ is the most efficient.

(7b) $\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2)) = \sigma_{F_1}(e_1) \bowtie \sigma_{F_2}(e_2)$ if F_1 only involves attributes of e_1 and F_2 only involves attributes of e_2

$\sigma_{F_1}(e_1) \bowtie \sigma_{F_2}(e_2)$ is the most efficient.
(7c) $\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2)) = \sigma_{F_1}(\sigma_{F_2}(e_1) \bowtie e_2) = \sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$ if all attributes mentioned in F_2 are of e_1

$\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$ is the most efficient.

(8) Commuting selection with a set difference: $\sigma_F(e_1 - e_2) = \sigma_F(e_1) - \sigma_F(e_2) = \sigma_F(e_1) - e_2$.

$\sigma_F(e_1) - e_2$ is the most efficient.

(9) Commuting selections and projections:
(9a) $\pi_{A_1,\ldots,A_n}(\sigma_F(e_1)) = \sigma_F(\pi_{A_1,\ldots,A_n}(e_1))$ if the formula F only involves attributes among $A_1, \ldots, A_n$.

$\sigma_F(\pi_{A_1,\ldots,A_n}(e_1))$ is the most efficient.
(9b) $\pi_{A_1,\ldots,A_n}(\sigma_F(e_1)) = \pi_{A_1,\ldots,A_n}(\sigma_F(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(e_1)))$, if all attributes involved in F that are not in $A_1, \ldots, A_n$ are $B_1, \ldots, B_m$ (F may be a conjunction of constraints).

$\pi_{A_1,\ldots,A_n}(\sigma_F(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(e_1)))$ is the most efficient.

(10) Commuting a projection with Cartesian product:
$\pi_{A_1,\ldots,A_n}(e_1 \times e_2) = \pi_{B_1,\ldots,B_m}(e_1) \times \pi_{C_1,\ldots,C_p}(e_2)$ where $A_1, \ldots, A_n$ is a list of attributes of which $B_1, \ldots, B_m$ are from e_1 and the remaining attributes $C_1, \ldots, C_p$ are from e_2.

$\pi_{B_1,\ldots,B_m}(e_1) \times \pi_{C_1,\ldots,C_p}(e_2)$ is the most efficient.

(11) Commuting a projection with a union:
$\pi_{A_1,\ldots,A_n}(e_1 \cup e_2) = \pi_{A_1,\ldots,A_n}(e_1) \cup \pi_{A_1,\ldots,A_n}(e_2)$.

$\pi_{A_1,\ldots,A_n}(e_1 \cup e_2)$ and $\pi_{A_1,\ldots,A_n}(e_1) \cup \pi_{A_1,\ldots,A_n}(e_2)$ are as efficient, since the union doesn't increase the number of constraints per tuple.

New optimization principles are then needed in constraint databases. Following the principles, we propose an algorithm for optimizing the naive evaluation of algebraic expressions.
Algorithm for the *Naive Strategy*: Perform each of the following steps in order.
Step 1. For each subexpression, use rules *(2)* and *(9b)* to decrease the number of projections.
Step 2. For each selection, use rules *(4a), (4b), (5), (6), (7a), (7b), (8)* to push down the selection as far down the tree as possible.
Step 3. For each projection, use rules *(9a), (10), (11)* to push down the projection as far down the tree as possible.
Step 4. For each cascade of selections, use rule *(3)* to group them.
Step 5. For each subexpression, use rule *(9b)* to increase the number of projections.
Step 6. Partition the interior nodes of the resulting tree into groups, as in the relational algorithm (see [Mai83, Ull88]).
Step 7. Produce a program.

Example 3 Consider the query

$$\mathcal{Q} = \pi_N \sigma_{15000 \leqslant P} \pi_{N,P} \sigma_{5 \leqslant A} \sigma_{1 \leqslant O} (C^I \bowtie F^I)$$

and the database instance I of Section 2. The expected answer of $\mathcal{Q}$ on I is $\{Orsay\}$.

We first evaluate the query as it is defined. We consider n the maximum number of constraints per tuple, and p the number of tuples resulting of the computation step.

	C^I	F^I	$C^I \bowtie F^I$	$\sigma_{1\leqslant O}$	$\sigma_{5\leqslant A}$	$\pi_{N,P}$	$\sigma_{15000\leqslant P}$	π_N	ER
n	8	3	11	12	13	39	40	40	1
p	3	4	12	12	12	12	12	12	1

The cost of the syntactic computation is linear in $48 + 12(FM(13,4) + FM(40,2))$. Applying the *Naive Strategy*, the query is equivalent to

$$\pi_N((\sigma_{15000 \leqslant P \wedge 1 \leqslant O \wedge 5 \leqslant A} C^I) \bowtie F^I).$$

The computation corresponding to the equivalent expression is described by the following table.

	C^I	$\sigma_{15000\leqslant P\wedge 1\leqslant O\wedge 5\leqslant A} C^I$	F^I	$\bowtie$	π_N	ER
n	8	11	3	14	40	1
p	3	3	4	12	12	1

The cost of the syntactic computation resulting of the application of the *Naive Strategy* is then linear in $16 + 12\ FM(14,4)$.

We now use *semantic computations* in order to reduce the number of tuples and the number of constraint per tuples after costly operations that is after intersection, projection and selection.

The *Semantical strategy* consists then in intertwining algebraic steps with constraint solving steps in order to optimize the evaluation.

Algorithm for the *Semantical Strategy*: Perform each of the following steps in order.

Step 1. For each subexpression, use rules *(2)* and *(9b)* to decrease the number of projections.

Step 2. For each selection, use rules *(4a), (4b), (5), (6), (7a), (7b), (8), (9a)* to push down the selection as far down the tree as possible.

Step 3. For each projection, use rules *(10), (11)* to push down the projection as far down the tree as possible and rule *(2)* to decrease the number of projections.

Step 4. For each cascade of selections, use rule *(3)* to group them.

Step 5. For each subexpression, use rule *(9b)* to decrease the number of projections.

Step 6. Use the ER algorithm after each intersection, selection and each use of FM.

Step 7. Partition the interior nodes of the resulting tree into groups, as in the

relational algorithm (see [Mai83, Ull88]).
Step 8. Produce a program.

Note that the resulting computations of both strategies are *semantically* equivalent but not *syntactically* equivalent, that is the resulting sets of constraints are different but they define the same polyhedrons.

Example 4 Consider again the query $\mathcal{Q}$ of Example 3 and the database instance I of Section 2. Applying the *Semantical Strategy*, the query is equivalent to

$$ER[\pi_N ER[ER[\sigma_{15000 \leqslant P \wedge 1 \leqslant O \wedge 5 \leqslant A} C^I] \bowtie F^I]].$$

The computation corresponding to the equivalent expression is described by the following table.

	C^I	$\sigma_{15000 \leqslant P \wedge 1 \leqslant O \wedge 5 \leqslant A} C^I$	ER	F^I	$\bowtie$	ER	π_N	ER
n	8	11	8	3	11	5	6	1
p	3	3	2	4	8	3	3	1

The cost of the evaluation of $\mathcal{Q}$ resulting of the application of the *Semantical Strategy* is then linear in $11 + 11\ ER(11) + 3\ ER(6) + 3\ FM(5,4)$.

We now introduce another strategy which uses the normalization tool. A normalized relation is such that each of its tuple has a bounded number of constraints. More precisely, the number of constraints is bounded by the arity (the number of attributes) plus one. From this property follows a different strategy to optimize the evaluation of queries. We first describe the principles for algebraic operations in the *Normalization Strategy*.
1- Use rule *(9b)* to decrease the arity (applying projections) before applying selections. Note that the rule *(9b)* is used in the opposite way as in the *Naive Strategy*.
2- Perform projection and selection as early as possible (rules *(4a), (4b), (5), (6), (7), (8), (9), (10)*).
3- Perform cascade of selections in one step with rule *(3)*.
4- Perform tuple normalization, TN, after intersection, set-difference, and selection.
5- Perform relation normalization, RN, at the end.

Note that this strategy uses TN in intermediate steps, and RN at the end only. RN is very costly since it increases the number of tuples in the relations. Following these principles, we propose an algorithm for optimizing the evaluation of algebraic expressions.
Algorithm for the *Normalization Strategy*: Perform each of the following steps in order.
Step 1. For each subexpression, use *(9b)* to decrease the arity (number of attributes).
Step 2. For each selection, use rules *(4a), (4b), (4c), (5), (6), (7a), (7b), (8), (9a)* to push down the selection as far down the tree as possible.

Step 3. For each projection, use rules *(9a), (10), (11)* to push down the projection as far down the tree as possible and rule *(2)* to decrease the number of projections.
Step 4. For each cascade of selections, use rule *(3)* to group them.
Step 5. For each subexpression, use *(9b)* to decrease the arity (number of attributes).
Step 6. Perform normalization TN after selection, set-difference, intersection and each use of FM.
Step 7. Perform normalization RN at the end.
Step 8. Partition the interior nodes of the resulting tree into groups, as in the relational algorithm (see [Mai83, Ull88]).
Step 9. Produce a program.

In the next example, we illustrate the application of the previous strategy.

Example 5 Consider again the query $\mathcal{Q}$ and the database instance I of Example 4. Applying the *Normalization Strategy*, the query is equivalent to

$$RN[\pi_N\ TN[\pi_{NOA}\ TN[\sigma_{15000 \leqslant P \wedge 1 \leqslant O \wedge 5 \leqslant A} C^I] \bowtie F^I]].$$

The computation corresponding to the equivalent expression is described by the following table.

	C^I	$\sigma_{15000 \leqslant P \wedge 1 \leqslant O \wedge 5 \leqslant A}$	TN	π_{NOA}	F^I	$\bowtie$	TN	π_N	RN
k	4	4	4	3	2	3	3	1	1
n	5	8	5	4	3	7	4	2	1
p	6	6	4	4	4	16	1	1	1

The cost of the evaluation of $\mathcal{Q}$ resulting of the application of the *Semantical Strategy* is then linear in $22 + 4\ FM(5,4) +\ FM(4,3) + 6\ TN(8) + 16\ TN(7) +\ RN(1,1)$.

6 Conclusion

This work constitutes a preliminary step in the development of query processing methods for queries over linear constraint databases. We intend to refine the analysis in two directions. First, by considering a more sophisticated cost model. The cost model used here is very simple, since it is based on the cost of the primitive operations (which are quite high level). It is sufficient for the purposes of the present paper though. Second, by considering average case analysis instead of worst case analysis.

The main question that we address here concerns the evaluation strategy for queries involving both relational algebra operators and constraint solving. We have shown that the classical query modification rules of relational algebra do not apply directly to linear constraint databases. We proposed some evaluation principles, leading to an evaluation strategy that intertwine algebraic steps with constraint solving steps (normalization). Some of the results, based on geometry, are not trivial. We are thinking of implementing an evaluation algorithm based on these results.

Acknowledgments : The authors wish to thank Michel Scholl, Jianwen Su and Christophe Tollu for fruitful discussions on the algebra evaluation, and S. Sudarshan for pointing our attention to the advantage of the normalization for secondary memory management.

References

[BJM93] A. Brodsky, J. Jaffar, M. Maher. Towards Practical Constraint Databases. In *Very Large DataBases* Conference, Dublin, Ireland, 567–580, 1993.

[Cod70] E.F. Codd. A relational model of data for large shared data banks. *Communications of ACM*, 13:6:377–387, 1970.

[Col75] G. E. Collins. Quantifier elimination for real closed fields by cylindric decompositions. In *Proc. 2nd GI Conf. Automata Theory and FOrmal Languages*, volume 35 of *Lecture Notes in Computer Science*, pages 134–83. Springer-Verlag, 1975.

[DHS88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proc. Fifth Generation Computer Systems*, Tokyo, 1988.

[GS94] S. Grumbach and J. Su. Finitely representable databases. In *13th ACM Symp. on Principles of Database Systems*, pages 289–300, Minneapolis, May 1994.

[GST94] S. Grumbach, J. Su, and C. Tollu. Linear constraint databases. In D. Leivant, editor, *Logic and Computational Complexity Workshop*, Indianapolis, 1994. Springer Verlag. LNCS 960.

[KG94] P. Kanellakis and D. Goldin. Constraint programming and database query languages. In *Manuscript*, 1994.

[KKR90] P. Kanellakis, G Kuper, and P. Revesz. Constraint query languages. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 299–313, Nashville, 1990.

[KRVV93] P. Kanellakis, S. Ramaswamy, D. Vengroff, and J. Vitter. Indexing for data models with constraints and classes. In *Proc. 12th ACM Symp. on Principles of Database Systems*, pages 233–243, 1993.

[LHM89] J-L. Lassez, T. Huynh, K. McAloon. Simplification and Elimination of Redundant Linear Arithmetic Constraints. In *Proc. North American Conf. on Logic Programming*, Cleveland, pages 37–51, 1989.

[Mai83] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[PS85] F. Preparata and M. Shamos. *Computational Geometry: An Introduction.* Springer Verlag, 1985.

[PVV94] J. Paredaens, J. Van den Bussche, and D. Van Gucht. Towards a theory of spatial database queries. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pages 279–288, 1994.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[Ull88] J.D. Ullman. *Database and Knowledge Base Systems.* Computer Science Press, 1988.

Appendix

A.1 Cost of Algebraic Operations

For each algebraic operation, we analyze its evaluation cost, and the additional cost of normalizing the output, as a function of the cost of the primitive operations. It depends upon (i) the *dimension* or arity of the input relations (which determines the number of constraints per tuples if they are in normal form), and (ii) the *size* of the relations (number of tuples, and values of the parameters). We also study the variation of the number of tuples in the relation, and the number of constraints in each tuple.

The parameters used are the following:
Input: relations R_i $(i \in \{1, 2\})$ of arity k_i with p_i tuples of at most n_i constraints. For simplicity, we ignore the size of the integers in the constraints.
Output of the *semantic computation*: relation with p tuples of at most n constraints.
Output of the *normalization*: relation of arity k' with p' tuples of at most $k' + 1$ constraints.

Selection :
The selection is defined by: $\sigma_F\ R_1 = \{t \wedge F \mid t \in R_1\}$. The *syntactic computation* is linear in p_1. It increases the number of constraints in each tuple of the relation by 1. The *semantic computation* of the selection requires the use of the *Emptiness-Redundancy test* on each tuple of the relation. After the semantic computation, $p \leqslant p_1$, $n \leqslant n_1 + 1$. The cost is $p_1 \times ER(n_1 + 1)$.

The *normalization* of the selection only requires an operation at the level of the tuples. The number of constraints per tuple in $\sigma_F\ R_1$ is bounded by $n_1 + 1$. A simple geometrical argument shows that each individual tuple in $\sigma_F\ R_1$ may lead to zero, one, or two normal form tuples (after *1-Tuple-Normalization*). The cost of *normalization* of $\sigma_F\ R_1$ is then in the worst case $p_1 \times TN(n_1 + 1)$, and $p' \leqslant 2p_1$.

Cartesian product :
The Cartesian product is defined by: $(R_1 \times R_2) = \{t_1 \wedge t_2 \mid t_1 \in R_1, t_2 \in R_2\}$. The *syntactic computation* is linear in $p_1 p_2$, and produces directly the output of the *semantic computation*, with $p = p_1 p_2$, and $n = n_1 + n_2$.

The *normalization* of the Cartesian product only requires an operation at the level of the tuples. The number of constraints per tuple is bounded by $k_1 + k_2 + 2$. The cost of *normalization* is then in the worst case $p_1 p_2 \times TN(n_1 + n_2)$, and $p' \leqslant 2p_1 p_2$.

Join :
The join is defined by: $(R_1 \bowtie R_2) = \{t_1 \wedge t_2 \mid t_1 \in R_1, t_2 \in R_2\}$. The *syntactic computation* is linear in $p_1 p_2$. The cost of the *semantic computation* is bounded by $p_1 p_2 \times ER(n_1 + n_2)$.

The *normalization* only requires the normalization at the level of the tuples. The cost is $p_1 p_2 \times TN(n_1 + n_2)$.

Projection :
$\pi_{\overline{x}} R_1(\overline{x}, y)$ is computed by the Fourier-Motzkin algorithm. The *syntactic computation* takes time $p_1 \times FM(n_1, k_1)$, and outputs p_1 tuples of at most $(\frac{n_1}{2})^2$ constraints over $k_1 - 1$ variables.

The *semantic computation* of the projection is in time $p_1 \times ER((\frac{n_1}{2})^2)$. Notice that after the semantic computation, the number of constraints per tuple is bounded by n_1 (the polyhedron resulting of a projection from a polyhedron with k facets cannot have more than k facets).

The cost of normalization is in the worst case $RN(p_1, k_1 + 1)$.

Union :
The union is defined by $R_1 \cup R_2$. The *syntactic computation* is linear in $p_1 + p_2$ and produces directly the output of the *semantic computation*.

The *normalization* requires an operation at the level of the relation. It may happen that there is a non-empty intersection between tuples of R_1 and tuples of R_2. The cost of the *normalization* is $(p_1 + p_2) \times RN(p_1 + p_2, k_1 + 1)$.

Intersection :
The intersection is defined by: $(R_1 \cap R_2) = \{t_1 \wedge t_2 \mid t_1 \in R_1, t_2 \in R_2\}$.

The *syntactic computation* is linear in $p_1 p_2$.

The cost of the *semantic computation* is bounded by $p_1 p_2 \times ER(n_1 + n_2)$.

The *normalization* only requires the normalization at the level of the tuples. The cost is $p_1 p_2 \times TN(n_1 + n_2)$.

Set difference :
The set difference is defined by: $(R_1 - R_2) = \{t_1 \wedge t_2 \mid t_1 \in R_1, t_2 \in (R_2)^c\}$.

The *syntactic computation* consists in first computing the complement of R_2, $(R_2)^c$, and then its intersection with R_1. The cost of the syntactic computation is bounded by $SC(p_2, n_2) + p_1 \times \mathcal{P}(p_2 n_2)$ (where $\mathcal{P}(p_2 n_2)$ is a polynomial giving an upper-bound on the number of cells [Col75], and therefore tuples, in $(R_2)^c$).

The *semantic computation* costs $p_1 \times \mathcal{P}(p_2 n_2) \times ER(n_1 + p_2)$.

The *normalization* requires both TUPLE-NORMALIZATION and RELATION-NORMALIZATION. The costs of normalization is less than $p_1 \times \mathcal{P}(p_2 n_2) \times RN(n_1 + n_2)$.

A.2 Analysis of the Query Modification Rules

For each equivalence rule, we precise the most efficient computation based on the syntactic computation cost. The first table describes the cost resulting of the *Naive Evaluation*, that is when all *syntactic computations* are successively done followed by a final *semantical computation* or *normalization* step. The second (third) table, studies the cost when *semantical* (*normalization*) steps are applied after each *syntactic computation*. Then we justify the best place for the *semantical computation* or *normalization*.

First we analyze the behavior of selection. A selection during the syntactic computation increases by one the number of constraints per tuple.

(3) Cascade of selections: $\sigma_{F_1}(\sigma_{F_2}(e_1)) = \sigma_{F_1 \wedge F_2}(e_1)$.

	$\sigma_{F_1}(\sigma_{F_2}(e_1))$	$\sigma_{F_1 \wedge F_2}(e_1)$
Syntactic	$2p_1$	p_1
Semantical	$p_1\ ER(n_1 + 2)$	$p_1\ ER(n_1 + 2)$
Normalization	$p_1\ TN(n_1 + 2)$	$p_1\ TN(n_1 + 2)$

$\sigma_{F_1 \wedge F_2}(e_1)$ is then the most efficient.

Consider now the cost in the case a semantical computation is applied after each step. That is after each algebraic operation, we execute the ER algorithm.

	$\sigma_{F_1}(\sigma_{F_2}(e_1))$	$\sigma_{F_1 \wedge F_2}(e_1)$
Syntactic	$2p_1$	p_1
Semantical	$p_1\ ER(n_1 + 1) + p_1\ ER(n_1 + 2)$	$p_1\ ER(n_1 + 2)$

$\sigma_{F_1 \wedge F_2}(e_1)$ is still the most efficient.

Consider now the cost in the case normalization is applied after each step. Note that in case the input is in normal form, the maximum number of constraints per tuple is $k + 1$ where k is the arity of the relation.

	$\sigma_{F_1}(\sigma_{F_2}(e_1))$	$\sigma_{F_1 \wedge F_2}(e_1)$
Syntactic	$3p_1$	p_1
Normalization	$3p_1\ TN(k_1 + 2)$	$p_1\ TN(k_1 + 3)$

It results that it is more efficient to group the selections together.

The following rules use the particular property of the Cartesian product. The syntactic computation of $R_1 \times R_2$ produces directly the output the semantic computation.

(4) Commuting selection with Cartesian product:

(4a) $\sigma_F(e_1 \times e_2) = \sigma_F(e_1) \times e_2$ if all attributes mentioned in F are of e_1.

	$\sigma_F(e_1 \times e_2)$	$\sigma_F(e_1) \times e_2$
Syntactic	$2p_1p_2$	$p_1 + p_1p_2$
Semantical	$p_1p_2\ ER(n_1+n_2+1)$	$p_1p_2\ ER(n_1+n_2+1)$
Normalization	$p_1p_2\ TN(n_1+n_2+1)$	$p_1p_2\ TN(n_1+n_2+1)$

$\sigma_F(e_1) \times e_2$ is the most efficient.

Consider now the cost in the case the semantical computation is applied after each syntactic computation step. Note that we do not apply a semantical computation after a Cartesian product.

	$\sigma_F(e_1 \times e_2)$	$\sigma_F(e_1) \times e_2$
Syntactic	$2p_1p_2$	$p_1 + p_1p_2$
Semantical	$p_1p_2\ ER(n_1+n_2+1)$	$p_1\ ER(n_1+1)$

$\sigma_F(e_1) \times e_2$ is the most efficient, but it is better to apply the semantical computation between the selection and the Cartesian product, since the semantical computation costs $p_1\ ER(n_1+1)$ instead of $p_1p_2\ ER(n_1+n_2+1)$.

Consider now the cost in the case normalization is applied after each step.

	$\sigma_F(e_1 \times e_2)$	$\sigma_F(e_1) \times e_2$
Syntactic	$3p_1p_2$	$p_1 + 2p_1p_2$
Norm.	$3p_1p_2TN(k_1+k_2+2)$	$p_1TN(k_1+2) + 2p_1p_2TN(k_1+k_2+2)$

It results that $\sigma_F(e_1) \times e_2$ is the most efficient. It is better to normalize after a selection, .

(4b) $\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2)) = \sigma_{F_1 \wedge F_2}(e_1 \times e_2) = \sigma_{F_1}(e_1) \times \sigma_{F_2}(e_2)$ if F_1 only involves attributes of e_1 and F_2 only involves attributes of e_2.

	$\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2))$	$\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$	$\sigma_{F_1}(e_1) \times \sigma_{F_2}(e_2)$
Syntactic	$3p_1p_2$	$2p_1p_2$	$p_1 + p_2 + p_1p_2$

In all cases, the *semantical computation* (*normalization*) is bounded by $p_1p_2ER(n_1+n_2+2)$ $(p_1p_2TN(n_1+n_2+2))$.

$\sigma_{F_1}(e_1) \times \sigma_{F_2}(e_2)$ is the most efficient.

Consider now the cost in the case a semantical computation is applied after each step.

	$\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2))$	$\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$
Synt.	$3p_1p_2$	$2p_1p_2$
Sem.	$p_1p_2(ER(n_1+n_2+1) + ER(n_1+n_2+2))$	$p_1p_2ER(n_1+n_2+2)$

	$\sigma_{F_1}(e_1) \times \sigma_{F_2}(e_2)$
Syntactic	$p_1 + p_2 + p_1p_2$
Semantical	$p_1ER(n_1+1) + p_2ER(n_2+1)$

$\sigma_{F_1}(e_1) \times \sigma_{F_2}(e_2)$ is the most efficient, and a *semantical computation* should apply after a selection.

Consider now the cost in the case normalization is applied after each step.

	$\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2))$	$\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$
Synt.	$7p_1p_2$	$3p_1p_2$
Norm.	$7p_1p_2TN(k_1+k_2+2)$	$p_1p_2(TN(k_1+k_2+2)+2TN(k_1+k_2+3))$

	$\sigma_{F_1}(e_1) \times \sigma_{F_2}(e_2)$
Syntactic	$p_1+p_2+4p_1p_2$
Normalization	$p_1TN(k_1+2)+p_2TN(k_2+2)+4p_1p_2TN(k_1+k_2+2))$

(4c) $\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2)) = \sigma_{F_1}(\sigma_{F_2}(e_1) \times e_2) = \sigma_{F_1 \wedge F_2}(e_1 \times e_2)$ if all attributes mentioned in F_2 are of e_1.

	$\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2))$	$\sigma_{F_1}(\sigma_{F_2}(e_1) \times e_2)$	$\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$
Syntactic	$3p_1p_2$	$p_1+2p_1p_2$	$2p_1p_2$

In all cases, the *semantical computation* (*normalization*) is bounded by $p_1p_2ER(n_1+n_2+2)$ $(p_1p_2TN(n_1+n_2+2))$.

$\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$ is the most efficient.

Consider now the cost in the case a semantical computation is applied after each step. Note that since a selection may generate many empty tuples, then the use of the ER algorithm after each selection may decrease the number of tuples per relation.

	Syntactic	Semantical
$\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2))$	$3p_1p_2$	$p_1p_2(ER(n_1+n_2+1)+ER(n_1+n_2+2))$
$\sigma_{F_1}(\sigma_{F_2}(e_1) \times e_2)$	$p_1+2p_1p_2$	$p_1ER(n_1+1)+p_1p_2ER(n_1+n_2+2)$
$\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$	$2p_1p_2$	$p_1p_2ER(n_1+n_2+2)$

In the worst case, $\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$ is the most efficient, but it is reasonable to use $\sigma_{F_1}(\sigma_{F_2}(e_1) \times e_2)$ when ER is applied after each selection.

Consider now the cost in the case normalization is applied after each step.

	Syntactic	Normalization
$\sigma_{F_1}(\sigma_{F_2}(e_1 \times e_2))$	$7p_1p_2$	$7p_1p_2TN(k_1+k_2+2)$
$\sigma_{F_1}(\sigma_{F_2}(e_1) \times e_2)$	$p_1+6p_1p_2$	$p_1TN(k_1+2)+6p_1p_2TN(k_1+k_2+2)$
$\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$	$3p_1p_2$	$p_1p_2(TN(k_1+k_2+2)+2TN(k_1+k_2+3))$

In the worst case, $\sigma_{F_1 \wedge F_2}(e_1 \times e_2)$ is the most efficient, but it is reasonable to use $\sigma_{F_1}(\sigma_{F_2}(e_1) \times e_2)$ when TN is applied after each selection.

(5) Commuting selection with a union: $\sigma_F(e_1 \cup e_2) = \sigma_F(e_1) \cup \sigma_F(e_2)$.

The cost of the *syntactic computation* is $2(p_1+p_2)$, the cost of the *semantic computation* is $p_1 \times ER(n_1+1) + p_2 \times ER(n_2+1)$ and the *normalization* cost is $RN(p_1+p_2, n+1)$, where n is $Max\{n_1, n_2\}$, in both cases.

The costs are the same in the case a semantical computation is applied after each step. Note that a *semantical computation* should not apply after a union.

Consider now the cost in the case a tuple-normalization is applied after each step. Note that a tuple-normalization should not apply after a union. Let n

be $Max\{n_1, n_2\}$. The syntactic cost is $2(p_1+p_2)$ and the normalization cost is $(p_1+p_2)TN(k_1+2)$ is both cases. But since a selection should generates many empty tuples, it is reasonable to prefer $\sigma_F(e_1) \cup \sigma_F(e_2)$ to $\sigma_F(e_1 \cup e_2)$.

(6) Commuting selection with an intersection: $\sigma_F(e_1 \cap e_2) = \sigma_F(e_1) \cap \sigma_F(e_2) = \sigma_F(e_1) \cap e_2$.

	$\sigma_F(e_1 \cap e_2)$	$\sigma_F(e_1) \cap \sigma_F(e_2)$	$\sigma_F(e_1) \cap e_2$
Synt.	$2p_1p_2$	$p_1 + p_2 + p_1p_2$	$p_1 + p_1p_2$
Sem.	$p_1p_2ER(n_1+n_2+1)$	$p_1p_2ER(n_1+n_2+2)$	$p_1p_2ER(n_1+n_2+1)$
Norm.	$p_1p_2TN(n_1+n_2+1)$	$p_1p_2TN(n_1+n_2+2)$	$p_1p_2TN(n_1+n_2+1)$

$\sigma_F(e_1) \cap e_2$ is the most efficient.

Consider now the cost in the case a *semantical computation* is applied after each step.

	$\sigma_F(e_1 \cap e_2)$
Syntactic	$2p_1p_2$
Semantical	$p_1p_2ER(n_1+n_2) + p_1p_2ER(n_1+n_2+1)$

	$\sigma_F(e_1) \cap \sigma_F(e_2)$
Syntactic	$p_1 + p_2 + p_1p_2$
Semantical	$p_1ER(n_1+1) + p_2ER(n_2+1) + p_1p_2ER(n_1+n_2+1)$

	$\sigma_F(e_1) \cap e_2$
Syntactic	$p_1 + p_1p_2$
Semantical	$p_1ER(n_1+1) + p_1p_2ER(n_1+n_2+1)$

$\sigma_F(e_1) \cap e_2$ is the most efficient.

Consider now the cost in the case a normalization is applied after each step. Since the intersection is possible, the attributes of e_1 coincide with the attributes of e_2 and the two expressions have the same arity k_1. Let p'_1, p'_2 and p' be respectively the number of tuples resulting of the normalization of $\sigma_F(e_1)$, $\sigma_F(e_2)$ and $(e_1 \cap e_2)$. Note that, if p (p') is the number of tuples before (after) normalization, $p \leqslant p'$ and moreover, often $2p \leqslant p'$.

	$\sigma_F(e_1 \cap e_2)$
Syntactic	$p_1p_2 + p'$
Normalization	$p_1p_2TN(2k_1+2) + p'TN(k_1+2)$

	$\sigma_F(e_1) \cap \sigma_F(e_2)$
Syntactic	$p_1 + p_2 + p'_1p'_2$
Normalization	$(p_1+p_2)TN(k_1+2) + p'_1p'_2TN(2k_1+2)$

	$\sigma_F(e_1) \cap e_2$
Syntactic	$p_1 + p'_1p_2$
Normalization	$p_1TN(k_1+2) + p'_1p_2TN(2k_1+2)$

$\sigma_F(e_1) \cap e_2$ is the most efficient.

(7) Commuting selection with join:

(7a) $\sigma_F(e_1 \bowtie e_2) = \sigma_F(e_1) \bowtie e_2$ if all attributes mentioned in F are of e_1.

	$\sigma_F(e_1 \bowtie e_2)$	$\sigma_F(e_1) \bowtie e_2$
Syntactic	$2p_1p_2$	$p_1 + p_1p_2$
Semantical	$p_1p_2ER(n_1+n_2+1)$	$p_1p_2ER(n_1+n_2+1)$
Normalization	$p_1p_2TN(n_1+n_2+1)$	$p_1p_2TN(n_1+n_2+1)$

$\sigma_F(e_1) \bowtie e_2$ is the most efficient.

Consider now the cost in the case the semantical computation is applied after each syntactic computation step.

	Syntactic	Semantical
$\sigma_F(e_1 \bowtie e_2)$	$2p_1p_2$	$p_1p_2ER(n_1+n_2) + p_1p_2ER(n_1+n_2+1)$
$\sigma_F(e_1) \bowtie e_2$	$p_1 + p_1p_2$	$p_1ER(n_1+1) + p_1p_2ER(n_1+n_2+1)$

Consider now the cost in the case normalization is applied after each step. Let p' be the number of tuples resulting of the normalization of $e_1 \bowtie e_2$.

	$\sigma_F(e_1 \bowtie e_2)$	$\sigma_F(e_1) \bowtie e_2$
Syntactic	$p_1p_2 + p'$	$p_1 + 2p_1p_2$
Norm.	$(p_1p_2 + p')TN(k_1+k_2+2)$	$p_1TN(k_1+2) + 2p_1p_2TN(k_1+k_2+2)$

It results that $\sigma_F(e_1) \bowtie e_2$ is the most efficient, but it is better to use ER or TN after a selection.

(7b) $\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2)) = \sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2) = \sigma_{F_1}(e_1) \bowtie \sigma_{F_2}(e_2)$ if F_1 only involves attributes of e_1 and F_2 only involves attributes of e_2.

	$\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2))$	$\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$	$\sigma_{F_1}(e_1) \bowtie \sigma_{F_2}(e_2)$
Syntactic	$3p_1p_2$	$2p_1p_2$	$p_1 + p_2 + p_1p_2$

In all cases, the *semantical computation* (*normalization*) is bounded by $p_1p_2ER(n_1+n_2+2)$ $(p_1p_2TN(n_1+n_2+2))$.

$\sigma_{F_1}(e_1) \bowtie \sigma_{F_2}(e_2)$ is the most efficient.

Consider now the cost in the case a semantical computation is applied after each step.

	$\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2))$
Syntactic	$3p_1p_2$
Semantical	$p_1p_2(ER(n_1+n_2) + ER(n_1+n_2+1) + ER(n_1+n_2+2))$

	$\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$
Syntactic	$2p_1p_2$
Semantical	$p_1p_2ER(n_1+n_2) + p_1p_2ER(n_1+n_2+2)$

	$\sigma_{F_1}(e_1) \bowtie \sigma_{F_2}(e_2)$
Syntactic	$p_1 + p_2 + p_1p_2$
Semantical	$p_1ER(n_1+1) + p_2ER(n_2+1) + p_1p_2ER(n_1+n_2+2)$

Consider now the cost in the case normalization is applied after each step. Let p' be the number of tuples resulting of the normalization of $e_1 \bowtie e_2$.

	$\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2))$	$\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$
Synt.	$p_1p_2 + 2p'$	$p_1p_2 + p'$
Norm.	$(p_1p_2 + 2p')TN(k_1 + k_2 + 2)$	$p_1p_2TN(k_1 + k_2 + 2)$ $+2p'TN(k_1 + k_2 + 3)$

	$\sigma_{F_1}(e_1) \bowtie \sigma_{F_2}(e_2)$
Syntactic	$p_1 + p_2 + 4p_1p_2$
Normalization	$p_1TN(k_1 + 2) + p_2TN(k_2 + 2) + 4p_1p_2TN(k_1 + k_2 + 2))$

(7c) $\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2)) = \sigma_{F_1}(\sigma_{F_2}(e_1) \bowtie e_2) = \sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$ if all attributes mentioned in F_2 are of e_1.

	$\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2))$	$\sigma_{F_1}(\sigma_{F_2}(e_1) \bowtie e_2)$	$\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$
Syntactic	$3p_1p_2$	$p_1 + 2p_1p_2$	$2p_1p_2$

In all cases, the *semantical computation* (*normalization*) is bounded by $p_1p_2ER(n_1 + n_2 + 2)$ $(p_1p_2TN(n_1 + n_2 + 2))$.

$\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$ is the most efficient.

Note that since a selection may generate many empty tuples, then the use of the ER algorithm after each selection may decrease the number of tuples per relation.

Consider now the cost in the case a semantical computation is applied after each step.

	Syntactic	Semantical
$\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2))$	$3p_1p_2$	$p_1p_2(ER(n_1 + n_2) + ER(n_1 + n_2 + 1)$ $+ER(n_1 + n_2 + 2))$
$\sigma_{F_1}(\sigma_{F_2}(e_1) \bowtie e_2)$	$p_1(1 + 2p_2)$	$p_1ER(n_1 + 1) + p_1p_2ER(n_1 + n_2 + 1)$ $+p_1p_2ER(n_1 + n_2 + 2)$
$\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$	$2p_1p_2$	$p_1p_2ER(n_1 + n_2) + p_1p_2ER(n_1 + n_2 + 2)$

In the worst case, $\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$ is the most efficient, but it is reasonable to use $\sigma_{F_1}(\sigma_{F_2}(e_1) \bowtie e_2)$ when ER is applied after each selection.

Consider now the cost in the case normalization is applied after each step. Let p' be the number of tuples resulting of the normalization of $e_1 \bowtie e_2$.

	Syntactic	Normalization
$\sigma_{F_1}(\sigma_{F_2}(e_1 \bowtie e_2))$	$p_1p_2 + 2p'$	$(p_1p_2 + 2p')TN(k_1 + k_2 + 2)$
$\sigma_{F_1}(\sigma_{F_2}(e_1) \bowtie e_2)$	$p_1 + p_1p_2 + p'$	$p_1TN(k_1 + 2)$ $+(p_1p_2 + p')TN(k_1 + k_2 + 2)$
$\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$	$p_1p_2 + p'$	$p_1p_2TN(k_1 + k_2 + 2)$ $+p'TN(k_1 + k_2 + 3))$

In the worst case, $\sigma_{F_1 \wedge F_2}(e_1 \bowtie e_2)$ is the most efficient, but it is reasonable to use $\sigma_{F_1}(\sigma_{F_2}(e_1) \bowtie e_2)$ when TN is applied after each selection.

(8) Commuting selection with a set difference: $\sigma_F(e_1 - e_2) = \sigma_F(e_1) - \sigma_F(e_2) = \sigma_F(e_1) - e_2$.

	$\sigma_F(e_1 - e_2)$	$\sigma_F(e_1) - \sigma_F(e_2)$
Syntactic	$SC(p_2, n_2) + 2p_1n_2$	$p_1 + p_2 + SC(p_2, n_2 + 1) + p_1(n_2 + 1)$
Semantical	$p_1n_2ER(n_1 + p_2 + 1)$	$p_1(n_2 + 1)ER(n_1 + p_2 + 1)$
Norm.	$p_1n_2TN(n_1 + p_2 + 1)$	$p_1(n_2 + 1)TN(n_1 + p_2 + 1)$

	$\sigma_F(e_1) - e_2$
Syntactic	$p_1 + SC(p_2, n_2) + p_1n_2$
Semantical	$p_1n_2ER(n_1 + p_2 + 1)$
Normalization	$p_1n_2TN(n_1 + p_2 + 1)$

$\sigma_F(e_1) - e_2$ is the most efficient.

Consider now the cost in the case a *semantical computation* is applied after each step. Note that a selection may generate many empty tuples which are removed by the use of the ER algorithm.

	$\sigma_F(e_1 - e_2)$
Syntactic	$SC(p_2, n_2) + 2p_1n_2$
Semantical	$p_1n_2ER(n_1 + p_2) + p_1n_2ER(n_1 + p_2 + 1)$

	$\sigma_F(e_1) - \sigma_F(e_2)$
Syntactic	$p_1 + p_2 + SC(p_2, n_2 + 1) + p_1(n_2 + 1)$
Semantical	$p_1ER(n_1 + 1) + p_2ER(n_2 + 1) + p_1(n_2 + 1)ER(n_1 + p_2 + 1)$

	$\sigma_F(e_1) - e_2$
Syntactic	$p_1 + SC(p_2, n_2) + p_1n_2$
Semantical	$p_1ER(n_1 + 1) + p_1n_2ER(n_1 + p_2 + 1)$

$\sigma_F(e_1) - e_2$ is the most efficient.

Consider now the cost in the case a tuple-normalization is applied after each step. Let p'_1, p'_2 and p' be respectively the number of tuples resulting of the normalization of $\sigma_F(e_1)$, $\sigma_F(e_2)$ and $(e_1 - e_2)$.

	$\sigma_F(e_1 - e_2)$
Syntactic	$SC(n_2, p_2) + n_2p_1 + p'$
Normalization	$n_2p_1TN(k_1 + 1 + p_2) + p'TN(2k_1 + 2)$

	$\sigma_F(e_1) - \sigma_F(e_2)$
Syntactic	$p_1 + p_2 + SC(p_2, k_1 + 2) + (k_1 + 1)p'_1$
Normalization	$(p_1 + p_2)TN(k_1 + 2) + (k_1 + 1)p'_1TN(k_1 + 1 + p'_2)$

	$\sigma_F(e_1) - e_2$
Syntactic	$p_1 + SC(p_2, k_1 + 1) + (k_1 + 1)p'_1$
Normalization	$p_1TN(k_1 + 2) + (k_1 + 1)p'_1TN(k_1 + 1 + p_2)$

The worst case shows that $\sigma_F(e_1) - e_2$ the most efficient. But the use of TN after each selection may decrease the number of tuples (since it removes empty tuples) with a bounding of the number of constraints per tuple. It follows that $\sigma_F(e_1) - \sigma_F(e_2)$ is preferable when TN is applied after each selection.

The following rules analyze the behavior of projection. The projection may increase dramatically the number of constraint per tuple. Let R be a relation of arity k with p tuples and a maximum of n constraints per tuple. The syntactic computation of $\pi_{A_1,\ldots,A_n}(R)$ is the result of the $k - m$ successive application of the FM algorithm. The first application of FM may generate in the worst case $(\frac{n}{2})^2$ constraints. Let $N_i(n)$ be the number of constraints resulting of the $(i-1)$th application of FM on tuple with n constraints. Since the Beaujolais nouveau has arrived, it is easy to compute $N_i(n) = \frac{n^{2^i}}{2^{(2^i-1)}}$. Let $N_0(n) = n$.

(9) Commuting selections and projections:

(9a) $\pi_{A_1,\ldots,A_n}(\sigma_F(e_1)) = \sigma_F(\pi_{A_1,\ldots,A_n}(e_1))$ if the formula F only involves attributes $A_1, \ldots, A_n$.

	$\pi_{A_1,\ldots,A_n}(\sigma_F(e_1))$	$\sigma_F(\pi_{A_1,\ldots,A_n}(e_1))$
Synt.	$p_1 + \Sigma_{i=0}^{i=k_1-n-1} FM(p_1, N_i(n_1+1))$	$\Sigma_{i=0}^{i=k_1-n-1} FM(p_1, N_i(n_1)) + p_1$
Sem.	$p_1 ER(N_{k_1-n}(n_1+1))$	$p_1 ER(N_{k_1-n}(n_1)+1)$
Norm.	$RN(p_1, N_{k_1-n}(n_1+1))$	$RN(p_1, N_{k_1-n}(n_1)+1)$

$\sigma_F(\pi_{A_1,\ldots,A_n}(e_1))$ is the most efficient.

Consider now the cost in the case a *semantical computation* is applied after each step. The FM algorithm generates many redondant constraints. The application of the ER algorithm between each projection then optimizes.

	$\pi_{A_1,\ldots,A_n}(\sigma_F(e_1))$	$\sigma_F(\pi_{A_1,\ldots,A_n}(e_1))$
Synt.	$p_1 + \Sigma_{i=0}^{i=k_1-n-1} FM(p_1, N_i(n_1+1))$	$\Sigma_{i=0}^{i=k_1-n-1} FM(p_1, N_i(n_1)) + p_1$
Sem.	$p_1 \Sigma_{i=0}^{i=k_1-n} ER(N_i(n_1+1))$	$p_1 \Sigma_{i=0}^{i=k_1-n} ER(N_i(n_1))$

The application of ER at each step of the computation may reduce a lot the number of tuples. Then $\pi_{A_1,\ldots,A_n}(\sigma_F(e_1))$ is more efficient.

Consider now the cost in the case a tuple-normalization is applied after each step. Let q_i and q'_j $(1 \leqslant i, j \leqslant k_1 - n)$ be the number of tuples respectively resulting of the tuple-normalization of $\pi_{A_1,\ldots,A_n}(\pi_{B_{k_1-n},\ldots,B_i}(\sigma_F(e_1)))$ and $\pi_{A_1,\ldots,A_n}(\pi_{B_{k_1-n},\ldots,B_j}(e_1))$. Remark that $q_1 = 2p_1$ and $q'_1 = p_1$.

	$\pi_{A_1,\ldots,A_n}(\sigma_F(e_1))$	$\sigma_F(\pi_{A_1,\ldots,A_n}(e_1))$
Syntactic	$p_1 + \Sigma_{i=1}^{i=k_1-n} FM(q_i, k_1+1)$	$\Sigma_{i=1}^{i=k_1-n} FM(q'_i, k_1+1) + q_{k_1-n}$
Norm.	$p_1 TN(k_1+2)$ $+\Sigma_{i=1}^{i=k_1-n} q_i TN((\frac{k_1+1}{2})^2)$	$\Sigma_{i=1}^{i=k_1-n} q'_i TN((\frac{k_1+1}{2})^2)$ $+q_{k_1-n} TN(k_1+2)$

The application of TN at each step of the computation may reduce the number of tuples with a bounded number of constraints per tuple. Then $\pi_{A_1,\ldots,A_n}(\sigma_F(e_1))$ is more efficient.

(9b) $\pi_{A_1,\ldots,A_n}(\sigma_F(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(e_1))) = \pi_{A_1,\ldots,A_n}(\sigma_F(e_1))$ if all attributes involved in F that are not in $A_1,\ldots,A_n$ are $B_1,\ldots,B_m$.

	$\pi_{A_1,\ldots,A_n}(\sigma_F(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(e_1)))$	$\pi_{A_1,\ldots,A_n}(\sigma_F(e_1))$
Sy.	$\Sigma_{i=0}^{i=k_1-(n+m)-1} FM(p_1, N_i(n_1)) + p_1$ $+\Sigma_{j=0}^{j=m} FM(p_1, N_j(N_{k_1-(n+m)}(n_1)+1))$	p_1 $\Sigma_{i=0}^{i=k_1-n} FM(p_1, N_i(n_1+1))$
Se.	$p_1 ER(N_m(N_{k_1-(n+m)}(n_1)+1))$	$p_1 ER(N_{k_1-n}(n_1+1))$
N.	$RN(p_1, N_m(N_{k_1-(n+m)}(n_1)+1))$	$RN(p_1, N_{k_1-n}(n_1+1)$

$\pi_{A_1,\ldots,A_n}(\sigma_F(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(e_1)))$ is the most efficient.

The computation made for rule *(9a)* already showed that in case ER or TN is applied between two successive use of FM and after each selection, $\pi_{A_1,\ldots,A_n}(\sigma_F(e_1))$ is the most efficient.

(10) Commuting a projection with Cartesian product: $\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(e_1 \times e_2) = \pi_{A_1,\ldots,A_n}(e_1) \times \pi_{B_1,\ldots,B_m}(e_2)$ if e_1 is of attributes $A_1,\ldots,A_n$ and e_2 of attributes $B_1,\ldots,B_m$.

	$\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(e_1 \times e_2)$	$\pi_{A_1,\ldots,A_n}(e_1) \times \pi_{B_1,\ldots,B_m}(e_2)$
Synt.	$p_1 p_2$ $+\Sigma_{i=0}^{i=o-1} FM(p_1, N_i(n_1+n_2))$	$\Sigma_{i=0}^{i=k_1-n-1} FM(p_1, N_i(n_1))$ $+\Sigma_{j=0}^{i=k_2-m-1} FM(p_2, N_j(n_2))$
Sem.	$p_1 p_2 ER(N_o(n_1+n_2))$	$p_1 ER(N_{k_1-n}(n_1))$ $+p_2 ER(N_{k_2-m}(n_2))$
Norm.	$RN(p_1 p_2, N_o(n_1+n_2))$	$RN(p_1 p_2, N)$

Where $o = k_1 + k_2 - (n+m)$ and $N = Max\{N_{k_1-n}(n_1), N_{k_2-m}(n_2)\}$.

$\pi_{A_1,\ldots,A_n}(e_1) \times \pi_{B_1,\ldots,B_m}(e_2)$ is the most efficient one, and moreover efficient when applying ER or TN after each successive use of FM.

(11) Commuting a projection with a union: $\pi_{A_1,\ldots,A_n}(e_1) \cup \pi_{A_1,\ldots,A_n}(e_2) = \pi_{A_1,\ldots,A_n}(e_1 \cup e_2)$.

The naive computation of the two expressions is the same.

Syntactic	$p_1 + p_2 + \Sigma_{i=0}^{i=k_1-(n+1)} FM(p_1 + N_i(n_1)) +$ $\Sigma_{i=0}^{i=k_1-(n+1)} FM(p_2, N_i(n_2))$
Semantical	$p_1 ER(N_{k_1-n}(n_1) + p_2 ER(N_{k_2-n}(n_2)$
Normalization	$RN(p_1 + p_2, N)$

Where $N = Max\{N_{k_1-n}(n_1), N_{k_2-n}(n_2\}$.

Both $\pi_{A_1,\ldots,A_n}(e_1 \cup e_2)$ and $\pi_{A_1,\ldots,A_n}(e_1) \cup \pi_{A_1,\ldots,A_n}(e_2)$ are efficient. Since $\pi_{A_1,\ldots,A_n}(e_1) \cup \pi_{A_1,\ldots,A_n}(e_2)$ enables to push further down the projection, it is chosen. It is more efficient to use ER or TN after each successive use of FM.