One-to-many Data Transformations through Data Mappers

Paulo Carreira [a,b], Helena Galhardas [a], Antónia Lopes [b], João Pereira [a]

[a] *INESC-ID and Technical University of Lisbon, Avenida Prof. Cavaco Silva, Tagus Park, 2780-990, Porto-Salvo, Portugal*

[b] *University of Lisbon, C6 - Piso 3, 1749-016 Lisboa, Portugal*

Abstract

The optimization capabilities of RDBMSs are turning them attractive for executing data transformations. However, despite the fact that many useful data transformations can be expressed as relational queries, an important class of data transformations that produce several output tuples for a single input tuple cannot be expressed in that way.

To overcome this limitation, we propose to extend Relational Algebra with a new operator named *data mapper*. In this paper, we formalize the data mapper operator and investigate some of its properties. We then propose a set of algebraic rewriting rules that enable the logical optimization of expressions with mappers and prove their correctness. Finally, we validate experimentally the proposed optimizations and identify the key factors that influence the optimization gains.

Key words: Data Warehousing, ETL, Data Mapper operator, Query optimization, Relational algebra

1 Introduction

The setup of modern information systems comprises a number of activities that rely, to a great extent, in the use of data transformations [LR99]. Well known cases are the migrations of legacy-data, the Extract-Transform-Load (ETL)

Email addresses: `paulo.carreira@tagus.ist.utl.pt` (Paulo Carreira), `hig@inesc-id.pt` (Helena Galhardas), `mal@di.fc.ul.pt` (Antónia Lopes), `joao@inesc-id.pt` (João Pereira).

processes that support data warehousing, cleansing of data and the integration of data from multiple sources. This situation leads to the development of data transformation programs that must move data instances from a fixed source schema into a fixed target schema.

One natural way of expressing data transformations is using a declarative query language and specifying the data transformations as queries (or views) over the source data. Because of the broad adoption of RDBMSs, such language is often SQL, a language based on Relational Algebra (RA). Unfortunately, due to its limited expressive power [AU79], RA alone cannot be used to specify many important data transformations [LSS96].

To overcome these limitations, several alternatives have been adopted for implementing data transformations: *(i)* the implementation of data transformation programs using a programming language, such as C or Java, *(ii)* the use of an RDBMS proprietary language like Oracle PL/SQL; or *(iii)* the development of data transformation scripts using a commercial ETL tool. However, transformations expressed in this way are often difficult to maintain, and more importantly, there is little possibility of optimization [CGLP05]. We remark that only recently an optimization technique for ETL processes was proposed [SVS05].

The normalization theory underlying the relational model imposes the organization of data according to several relations in order to avoid redundancy and inconsistency of information. In Codd's original model, new relations are derived from the database by selecting, joining and unioning relations. Despite the fact that RA expressions denote transformations among relations, the notion that presided the design of RA (as noted by [AU79]) was that of retrieving data. This notion, however, is insufficient for reconciling the substantial differences in the representation of data that occur in a context of fixed source and target schemas [Mil98].

One such difference occurs when the source data is an aggregation of the target data. For example, source data may consist of salaries aggregated by year, while the target consists of salaries aggregated by month. The data transformation that has to take place needs to produce several tuples in the target relation to represent each tuple of the source relation. We designate these data transformations as *one-to-many data mappings.* As we will demonstrate latter (in Section 3.3), this class of data transformations cannot be expressed by standard RA, even if we use the generalized projection operator [SKS01].

Our experience with Ajax [GFSS00] data cleaning tool and with Data Fusion [CG04] legacy-data migration tool has shown that in the context of data transformation, there is a considerable amount of data transformations that require one-to-many mappings. In fact, as recognized in [Gal01], an important class of data transformations require the inverse operation of the SQL group by/aggregates primitive that, for each input tuple, has the ability to produce several output tuples.

In this paper, we propose an extension to RA to represent one-to-many data transformations. This extension is achieved through a new operator that, as the generalized projection operator, relies on the use of arbitrary, external functions.

There are two main reasons why we chose to extend RA. First, even though RA is not expressive enough to capture the semantics of one-to-many mappings, we want to make use of the provided expressiveness for the remaining data transformations. Second, we intend to take advantage of the optimization strategies that are implemented by relational database engines [Cha98]. Our decision of adopting database technology as a basis for data transformation is not completely revolutionary. Several RDBMSs, like Microsoft SQL Server, already include additional software packages specific for ETL tasks. However, to the best of our knowledge, none of these extensions is supported by the corresponding theoretical background in terms of existing database theory. Therefore, the capabilities of relational engines, for example, in terms of optimization opportunities are not fully exploited for ETL tasks.

In the remainder of this section, we first present a motivating example to illustrate the usefulness of one-to-many data transformations. Then, in Section 1.2, we highlight the contributions of this paper.

1.1 Motivating example

As already mentioned, there is a considerable amount of data transformations that require one-to-many mappings. Here, we present a simple example, based on a real-world data migration scenario, that was intentionally simplified for illustration purposes.

Example 1.1: *Consider the source relation* ***LOANS****[****ACCT****,* ***AM****] (represented in Figure 1) that stores the details of loans requested per account. Suppose* ***LOANS*** *data must be transformed into* ***PAYMENTS****[****ACCTNO****,* ***AMOUNT****,* ***SEQNO****], the target relation, according to the following requirements:*

1. *In the target relation, all the account numbers are left padded with zeroes. Thus, the attribute* ***ACCTNO*** *is obtained by (left) concatenating zeroes to the value of* ***ACCT****.*
2. *The target system does not support loan amounts superior to* 100*. The attribute* ***AMOUNT*** *is obtained by breaking down the value of* ***AM*** *into multiple installments with a maximum value of 100, such that, the sum of*

Relation LOANS

ACCT	AM
12	20.00
3456	140.00
901	250.00

Relation PAYMENTS

ACCTNO	AMOUNT	SEQNO
0012	20.00	1
3456	100.00	1
3456	40.00	2
0901	100.00	1
0901	100.00	2
0901	50.00	3

Fig. 1. *(a)* On the left, the LOANS relation and, *(b)* on the right, the PAYMENTS relation.

amounts for the same ACCTNO is equal to the source amount for the same account. Furthermore, the target field SEQNO is a sequence number for the installment. This sequence number starts at 1 *for each sequence of installments of a given account.*

The implementation of data transformations similar to those requested for producing the target relation PAYMENTS of Example 1.1 is challenging, since solutions to the problem involve the dynamic creation of tuples based on the value of attribute AM.

1.2 Major contributions

This paper proposes to extend RA with the *mapper* operator, which significantly increases its expressive power and, in particular, allows us to represent one-to-many data transformations. Informally, a mapper is applied to an input relation and produces an output relation. It iterates over the input tuples and generates zero, one or more output tuples per input tuple, by applying a set of domain-specific functions. In this way, it supports the dynamic creation of tuples based on a source tuple contents. This kind of operation appears implicitly in most languages aiming at implementing schema and data transformations but, as far as we know, it has never been properly handled as a first-class operator. New optimization opportunities arise when promoting the mapper to a relational operator. In fact, expressions that combine the mapper operator with standard RA operators can be optimized.

The main contributions of this paper are the following:

(1) the formalization of a new primitive operator, named *data mapper*, that allows to express one-to-many mappings;

(2) a set of provably correct algebraic rewriting rules for expressions involving the mapper operator and relational operators, useful for optimization purposes;
(3) the development of cost estimates for expressions involving filters and mappers;
(4) the identification of the main factors that influence the gains introduced by the proposed optimizations;
(5) experimental validation of the proposed optimizations and their corresponding cost formulas.

The rest of this paper is organized as follows. Preliminary definitions are provided in Section 2. The formalization of the mapper is presented in Section 3. Section 4 presents the algebraic rewriting rules that enable the logical optimization of several expressions involving the mapper operator. The cost estimates for expressions involving filters and mappers are developed in Section 5, and the validation experiments presented in Section 6. Finally, related work is summarized in Section 7 and conclusions are presented in Section 8.

2 Preliminaries

A domain D is a set of atomic values. We assume a set $\mathcal{D}$ of domains and a set $\mathcal{A}$ of names – attribute names – together with a function $Dom : \mathcal{A} \rightarrow \mathcal{D}$ that associates domains to attributes. We will also use Dom to denote the natural extension of this function to lists of attribute names: $Dom(A_1, ..., A_n) = Dom(A_1) \times ... \times Dom(A_n)$.

A *relation schema* R consists of a list $A = A_1, ..., A_n$ of distinct attribute names. We write $R(A_1, ..., A_n)$, or simply $R(A)$, and call n the degree of the relation schema. Its domain is defined by $Dom(A)$. A *relation instance* (or relation, for short) with schema $R(A_1, ..., A_n)$ has a finite set $r \subseteq Dom(A_1) \times ... \times Dom(A_n)$; we write $r(A_1, ..., A_n)$, or simply $r(A)$. Each element t of r is called a *tuple* or r*-tuple* and can be regarded as a function that associates a value of $Dom(A_i)$ with each A_i; we denote this value by $t[A_i]$. Given a list $B = B_1, ..., B_k$ of distinct attributes in $A_1, ..., A_n$, we denote by $t[B]$ the tuple $\langle t[B_1], ..., t[B_k] \rangle$ in $Dom(B)$.

We will use the term *relational algebra* to denote the standard notion as introduced by [Cod70]. The basic operations considered are *union*, *difference* and *Cartesian product* as well as *projection* (π_X, where X is a list of attributes), *selection* (σ_C, where C is the selection condition) and *renaming* ($\rho_{A \rightarrow B}$, where A and B are lists of attributes).

3 The mapper operator

In this section, we present the new relational operator and show how it allows to express one-to-many data transformations. We also analyze some of its properties and discuss the expressive power of the resulting setting.

A mapper is a unary operator μ_F that takes a relation instance of a given relation schema as input (source schema) and produces a relation instance of another relation schema as output (target schema). The mapper operator is parameterized by a list F of special functions, which we designate as *mapper functions.*

Roughly speaking, each mapper function allows one to express a part of the envisaged data transformation, focused on one or more attributes of the target schema. Although the idea is to apply mapper functions to tuples of a source relation instance, it may happen that some of the attributes of the source schema are irrelevant for the envisaged data transformation. The explicit identification of the attributes that are considered relevant is then an important part of a mapper function. Mapper functions are formally defined as follows.

DEFINITION 3.1: *A* mapper function *f_A is a triple $\langle A, B, f\rangle$ where A is a non-empty list of distinct attributes (it defines the output attributes), B is a list of distinct attributes (it identifies the relevant input attributes), and $f{:}Dom(B){\rightarrow}\mathcal{P}(Dom(A))$ is a computable function (if B is empty, then f is just a set). We say that f_A is an $A-$mapper function. Let t be a tuple of a relation instance $s(X_1, ..., X_n)$ s.t. all the attributes in B are also in $X_1, ..., X_n$. We define $f_A(t)$ to be the application of the underlying function f to the tuple t, i.e., $f(t[B])$.*

In this way, a mapper function describes how a specific part of the target data can be obtained from the source data, defining simultaneously part of the target schema. The intuition is that each mapper function establishes how the values of a group of attributes of the target schema can be obtained from the attributes of the source schema. The key point is that, when applied to a tuple, a mapper function produces a set of values, rather than a single value.

We shall freely use f_A to denote both a mapper function $\langle A, B, f\rangle$ and the function f itself, omitting the list B whenever its definition is clear from the context, and this shall not cause confusion. We shall also use $Dom(f_A)$ to refer to list B. This list should be regarded as the list of the source attributes declared to be relevant for the part of the data transformation encoded by the mapper function. Notice, however, that even if f_A is a constant function, f_A may be defined as being dependent on all the attributes of the source schema.

The relevance of the explicit identification of these attributes will be clarified in Section 4 when we present the algebraic optimization rules for projections.

Certain classes of mapper functions enjoy properties that enable the optimizations of algebraic expressions containing mappers (see also Section 4). Mapper functions can be classified according to *(i)* the number of output tuples they can produce, or *(ii)* the number of output attributes. Mapper functions that produce singleton sets, i.e., $\forall (t \in Dom(X))\ |f_A(t)| = 1$ are designated *single-valued mapper functions.* In contrast, mapper functions that produce multiple elements are said to be *multi-valued mapper functions.* Concerning the number of output attributes, mapper functions with one output attribute are called *single-attributed*, whereas functions with many output attributes are called *multi-attributed.*

We designate by *identity mapper functions* the single-valued mapper functions $\langle A, A, f\rangle$ s.t. $f(t) = \{t\}$. Also interesting is the class of the single-valued mapper functions $\langle A, B, f\rangle$ s.t. $Dom(B) = Dom(A)$ and $f(t) = \{t\}$. These are said to be *renaming mapper functions*, given that they only establish a transformation of the schema. Finally, a *constant mapper function* is a mapper function $\langle A, [], f\rangle$ s.t. $f(t) = c$, for every $t \in Dom(B)$ and some $c \in \mathcal{P}(Dom(A))$.

As mentioned before, a mapper operator is parameterized by a list of mapper functions. This list is proper for transforming the data from a given source schema if the attributes produced by these functions are all distinct.

Definition 3.2: *A list $F = f_{A_1}, ..., f_{A_k}$ of mapper functions is said to be* proper *for transforming the data of a relation $s(X_1, ..., X_n)$ iff the attributes included in the A_j lists, for $1 \leq j \leq k$, are all distinct.*

In other words, F is proper if it specifies, in a unique way, how the values of the schema $Y = A_1 \cdot ... \cdot A_k$ —the target schema— are produced.

The mapper operator μ_F puts together the data transformations of the input relation defined by the mapper functions in F. Given a tuple s of the input relation, $\mu_F(s)$ consists of the tuples t of $Dom(Y)$ that, to each list of attributes A_i, associate values in $f_{A_i}(s)$. Formally, the mapper operator is defined as follows.

Definition 3.3: *Given a relation $s(X_1, ..., X_n)$ and a proper list of mapper functions $F = f_{A_1}, ..., f_{A_k}$, the* mapper *of s with respect to F, denoted by $\mu_F(s)$, is the relation instance with schema $Y = A_1 \cdot ... \cdot A_k$ and the set of tuples defined by*

$$\mu_F(s) \stackrel{\text{def}}{=} \{t \in Dom(Y) \mid \exists u \in s\ \forall 1 \leq i \leq k\ \ t[A_i] \in f_{A_i}(u)\}$$

As mentioned before, this new operator relies on the use of arbitrary computable functions that are external to the resulting extension of the relational algebra. Notice also that mapper may be defined in terms of partial functions, i.e., the underlying functions do not have to be defined for all values of their source set. It follows from Definition 3.3 that if $f_{A_i}(t)$ is undefined for some $f_{A_i} \in F$ and $t \in s$, then so is $\mu_F(s)$.

The set of admissible functions can be further constrained, if required. As we will see in Section 3.2, for some specific classes of admissible functions, the integration of the mapper operator with existing query execution processors is easier.

In order to illustrate this new operator, we revisit Example 1.1.

Example 3.1: *The requirements presented in Example 1.1 can be described by the mapper $\mu_{acct,amt}$, where acct is an `[ACCTNO]`-mapper function with domain `ACCT` that returns a singleton with the account number `ACCT` properly left padded with zeroes and amt is the `[AMOUNT,SEQNO]`-mapper function with domain `AM` s.t., amt(am) is given by*

$$\{(100, i) \mid 1 \leq i \leq (am/100)\} \cup \{(am\%100, (am/100)+1) \mid am\%100 \neq 0\}$$

where we have used / and % to represent the integer division and modulus operations, respectively.

For instance, if t is the source tuple (`901`, `250.00`), the result of evaluating amt(t) is the set $\{(100, 1), (100, 2), (50, 3)\}$. Given a source relation s including t, the result of the expression $\mu_{acct,amt}(s)$ is a relation that contains the set of tuples $\{\langle$ `'0901'`, 100, 1$\rangle$, $\langle$ `'0901'`, 100, 2$\rangle$, $\langle$ `'0901'`, 50, 3$\rangle\}$.

In order to illustrate the full expressive power of mappers, we present an example of selective transformation of data.

Example 3.2: *Consider the conversion of yearly salary data into quarterly salary data. Let `EMPDATA`[`ESSN`, `ECAT`, `EYRSAL`] be the source relation that contains yearly salary information about employees. Suppose we need to generate a target relation with schema `EMPSAL`[`ENUM`, `QTNUM`, `QTSAL`], which maintains the quarterly salary for the employees with long-term contracts. In the source schema, we assume that the attribute `EYRSAL` maintains the yearly net salary. Furthermore, we consider that the attribute `ECAT` holds the employee category and that code `'S'` specifies a short-term contract whereas `'L'` specifies a long-term contract.*

This transformation can be specified through the mapper $\mu_{empnum,sal}$ where empnum is a `[ENUM]`-mapper function with domain `[ESSN,ECAT,EYRSAL]` *that makes up new employee numbers (i.e., a Skolem function [HY90]), and sal the* `[QTNUM,QTSAL]`*-mapper function*

$$sal_{\texttt{QTNUM, QTSAL}}(ecat, eyrsal)$$

with domain `[ECAT,EYRAL]` *that generates quarterly salary data, defined as:*

$$sal(ecat, eyrsal) = \begin{cases} \{(i, \frac{eyrsal}{4}) \mid 1 \leq i \leq 4\} & if\ ecat = \texttt{'L'} \\ \emptyset & if\ ecat = \texttt{'S'} \end{cases}$$

3.1 Properties of Mappers

We start to notice that the mapper operator admits a more intuitive definition in terms of the Cartesian product of the sets of tuples obtained by applying the underlying mapper functions to each tuple of the input relation. More concretely, the following proposition holds.

Proposition 1: *Given a relation $s(X_1, ..., X_n)$ and a proper list of mapper functions $F = f_{A_1}, ..., f_{A_k}$,*

$$\mu_F(s) = \bigcup_{u \in s} f_{A_1}(u) \times ... \times f_{A_k}(u).$$

Proof

$$\begin{aligned} \mu_F(s) &= \{t \in Dom(Y) \mid \exists u \in s\ \forall 1 \leq i \leq k\ t[A_i] \in f_{A_i}(u)\} \\ &= \bigcup_{u \in s} \{t \in Dom(Y) \mid \forall 1 \leq i \leq k\ t[A_i] \in f_{A_i}(u)\} \\ &= \bigcup_{u \in s} f_{A_1}(u) \times ... \times f_{A_k}(u) \end{aligned}$$

This alternative way of defining $\mu_F(s)$ is also important because of its operational flavor, equipping the mapper operator with a *tuple-at-a-time* semantics. When integrating the mapper operator with existing query execution processors, this property plays an important role because it means the mapper operator admits physical execution algorithms that favor pipelined execution [Gra93].

The task of devising an algorithm that computes data transformation through mappers becomes straightforward: it is just to compute the Cartesian product as stated by the proposition. Obviously, this algorithm relies on the computability of the underlying mapper functions and builds on concrete algorithms for computing them. Furthermore, the fact that the calculation of $\mu_F(s)$ can be carried out tuple by tuple clearly entails the monotonicity of the mapper operator.

Proposition 2: *The mapper operator is* monotonic, *i.e., for every pair of relations $s_1(X)$ and $s_2(X)$ s.t. $s_1 \subseteq s_2$, $\mu_F(s_1) \subseteq \mu_F(s_2)$.*

Proof

$$\begin{aligned}\mu_F(s_1) &= \{t \in Dom(Y) \mid \exists u \in s_1\ \forall 1 \leq i \leq k \quad \text{s.t. } t[A_i] \in f_{A_i}(u)\}\\ &\quad \text{by hypothesis } s_1 \subseteq s_2\\ &\subseteq \{t \in Dom(Y) \mid \exists u \in s_2\ \forall 1 \leq i \leq k \quad \text{s.t. } t[A_i] \in f_{A_i}(u)\}\\ &= \mu_F(s_2)\end{aligned}$$

3.2 Normal-forms

As defined in Definition 3.2, a list of mapper functions F is proper for transforming the data of a given relation only if the subset of attributes produced by any two different mapper functions in F do not overlap. It is not difficult to see that, in general, a data transformation can be achieved through different lists of functions.

Consider, for instance, the `[ACCTNO,AMOUNT,SEQNO]`-mapper function named *payments* with domain `[ACCT,AM]` that yields installment amounts jointly with the transformed account numbers. Clearly, the list of proper mapper functions $F = acct, amt$ defined in 3.1 is equivalent to the single element list $G = payments$, in what concerns the data transformation they specify. However, algebraic expressions containing μ_F are more prone to optimization than expressions containing μ_G. Compared to G, the list F can be regarded as being in a more reduced form. In a similar way, mapper functions may use dispensable input attributes. We can compare F with a list $H = acct', amt$ where $acct'$ only differs from $acct$ in the domain that, in the case of $acct'$, is `[ACCT,AM]`. Given that H includes one mapper function with a domain larger than it is required, F can be regarded as being in a more reduced form.

In fact, the list F is what we will call a normal form because it cannot be reduced in a sense that can be made precise as follows.

Definition 3.4: *Let $S(X_1, ..., X_n)$ be a fixed relation schema. The* reduction relationship *between lists of mapper functions proper for transforming the data of relations with schema $S(X_1, ..., X_n)$, represented as $\longrightarrow$, is the greatest transitive relationship satisfying the following constraints:*

(1) if $[f_1, ..., \langle A, B_f, f\rangle, ..., f_k] \longrightarrow [f_1, ..., \langle A, B_g, g\rangle, ..., f_k]$ then the list of attributes B_g is strictly a sublist of B_f and $f(t) = g(t)$, for every $t \in Dom(X)$.

(2) if $[f_1, ..., \langle A, B, f\rangle, ..., f_k] \longrightarrow [f_1, ..., \langle A_1, B_1, g_1\rangle, \langle A_2, B_2, g_2\rangle, ..., f_k]$ then B_1 and B_2 are sublists of B, and a permutation ϵ exists such that $A =$

$\epsilon(A_1 \cdot A_2)$ *and* $f(t) = \epsilon(g_1(t) \times g_2(t))$, *for every tuple* $t \in Dom(X)$.

Intuitively, a list of mapper functions can be reduced, if one of its mapper functions either includes superfluous attributes in its domain or defines a transformation of data that can be decomposed, that is, expressed as a Cartesian product of two functions.

Definition 3.5: *A mapper* μ_F *is a* normal-form *if it does not exist a list of mapper functions* G *s.t.* $F \longrightarrow G$, *i.e., if* F *cannot be reduced.*

From a practical point of view, a mapper that is not a normal-form presents a number of limitations. To start, the amalgamation of independent mapper functions in the same one, limits the choice of physical execution algorithms. For instance, consider using caching for the most expensive functions; if an expensive function is implemented together an inexpensive one, in one single a function, it may not be possible to apply this algorithm as it may no be feasible to decide in compile time which is the expensive function. Another important aspect is the number of optimization opportunities that may arise in expressions involving mappers: the opportunities for applying optimizations s.a. the ones we will present in Section 4 increase as the mapper operators involved are closer to normal forms.

From a software engineering point of view, trying to maintain an implementation where the logic of several functions condensed into fewer functions is also highly undesirable. It violates two highly desired properties of any software artifact: *high cohesion* and *low coupling*. The notion of normal-form establishes a principled way to tell if the specification of a mapper together with its functions abides by these principles or not.

3.3 Expressive power of mappers

Concerning the expressive power of the mapper operator, two important questions are addressed. First, we compare the expressive power of relational algebra (RA) with its extension by the set of mapper operators, henceforth designated as *M-relational algebra* or simply *MRA*. Second, we investigate which standard relational operators can be simulated by a mapper operator.

It is not difficult to recognize that MRA is more expressive than standard RA. It is obvious that part of the expressive power of mapper operators comes from the fact that they are allowed to use arbitrary computable functions. In fact, the class of mapper operators of the form μ_f, where f is a single-valued function, is computationally complete. This implies that MRA is computational complete and, hence, MRA is not a query language like standard RA.

The question that naturally arises is if MRA is more expressive than the relational algebra with a generalized projection operator π_L where the projection list L has elements of the form $Y_i \leftarrow f(A)$, where A is a list of attributes in $X_1, ..., X_n$ and f is a computable function.

With generalized projection, it becomes possible to define arbitrary computations to derive the values of new attributes. Still, there are MRA-expressions whose effect is not expressible in RA, even when equipped with the generalized projection operator. We shall use *RA-gp* to designate the extension of RA extended with generalized projection.

The additional expressive power results from the fact that mapper operators use functions that map values into sets of values and, thus, are able to produce a set of tuples from a single tuple. For some multi-valued functions, the number of tuples that are produced depends on the specific data values of the source tuples and does not even admit an upper-bound.

Consider for instance a database schema with relation schemas `S(NUM)` and `T(NUM, IND)`, s.t. the domain of `NUM` and `IND` is the set of natural numbers. Let s be a relation with schema S. The cardinality of the expression $\mu_{[f]}(s)$, where f is an `[NUM,IND]`-mapper function s.t. $f(n) = \{\langle n, i\rangle : 1 \leq i \leq n\}$, does not (strictly) depend on the cardinality of s. Instead, it depends on the values of the concrete $s-$tuples. For instance, if s is a relation with a single tuple $\{\langle x\rangle\}$, the cardinality of $\mu_{[f]}(s)$ depends on the value of x and does not have an upper bound.

This situation is particularly interesting because it cannot happen in RA-gp.

Proposition 3: *For every expression E of the relational algebra RA-gp, the cardinality of the set of tuples denoted by E admits an upper bound defined simply in terms of the cardinality of the atomic sub-expressions of E.*

Proof This can be proved in a straightforward way by structural induction in the structure of relational algebra expressions. Given a relational algebra expression E, we denote by $|E|$ the cardinality of E. For every non-atomic expression we have: $|E_1 \cup E_2| \leq |E_1| + |E_2|$; $|E_1 - E_2| \leq |E_1|$; $|E_1 \times E_2| \leq |E_1| \times |E_2|$; $|\pi_L(E)| \leq |E|$; $|\sigma_C(E)| \leq |E|$; $|\rho_{X_1,\dots,X_n \rightarrow Y_1,\dots,Y_n}(E)| \leq |E|$.

Hence, it follows that:

Proposition 4: *There are expressions of the M-relational algebra that are not expressible by the relational algebra RA-gp on the same database schema.*

Another aspect of the expressive power of mappers, that is interesting to address, concerns the ability of mappers for simulating other relational operators. In fact, we will show that projection, renaming and selection operators can be seen as special cases of mappers. That is to say, there exist three classes of mappers that are equivalent, respectively, to projection, renaming and selection. From this we can conclude that the restriction of MRA to the operators mapper, union, difference and Cartesian product is as expressive as MRA.

Projection can be obtained through mapper operators over identity mapper functions. One identity mapper function is included for each project attribute. The project attribute has to be an attribute of the source schema.

RULE 1: *Let $S(X_1, ..., X_n)$ be a relation schema and $Y_1, ..., Y_m$ a list of different attributes in $X_1, ..., X_n$. For every relation instance $s(X_1, ..., X_n)$, the term $\pi_{Y_1,...,Y_m}(s)$ is equivalent to $\mu_F(s)$, where $F = f_{Y_1}, ..., f_{Y_m}$ and f_{Y_i} is the identity mapper function, for every $1 \leq i \leq m$.*

PROOF

$$\begin{aligned}
\pi_{Y_1,...,Y_m}(s) &= \{t[Y_1, ..., Y_m] \mid t \in s\} \\
&= \{t \in \mathit{Dom}(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } u[Y_i] = t[Y_i]\} \\
&= \{t \in \mathit{Dom}(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } t[Y_i] \in \{u[Y_i]\}\} \\
&\text{because } f_{Y_i}(t) = \{t\}, \text{ for every } t \in \mathit{Dom}(Y_i) \\
&= \{t \in \mathit{Dom}(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } t[Y_i] \in f_{Y_i}(u)\} \\
&= \mu_{f_{Y_1},...,f_{Y_m}}(s)
\end{aligned}$$

Strictly speaking, a renaming *ren* is a bijective function among sets of attributes X and Y s.t. $\mathit{Dom}(X_i) = \mathit{Dom}(Y_i)$ and $\mathit{ren}(X_i) \neq X_i$, for every $X_i \in X$. This function is usually represented as $X_1, ..., X_n \rightarrow Y_1, ..., Y_n$. The relational renaming operator operator is a unary relational operator parameterized by a renaming function [AdA93, AHV95]. Renaming can also be expressed by a mapper parametrized by renaming mapper functions. We include one renaming function for mapping each source attribute to the corresponding target attribute.

RULE 2: *Let $S(X_1, ..., X_n)$ and $T(Y_1, ..., Y_n)$ be two relation schemas, such that, $\mathit{Dom}(X) = \mathit{Dom}(Y)$. For every relation instance $s(X_1, ..., X_n)$, the term $\rho_{X_1,...,X_n \rightarrow Y_1,...,Y_n}(s)$ is equivalent to $\mu_F(s)$ where $F = f_{Y_1}, ..., f_{Y_n}$ and, for every $1 \leq i \leq n$, f_{Y_i} is the renaming mapper function $\langle Y_i, X_i, id_{\mathit{Dom}(Y_i)} \rangle$.*

Proof

$$
\begin{aligned}
&\rho_{X_1,...,X_n \to Y_1,...,Y_n}(s)\\
&\ = \{t[Y_1, ..., Y_m] \mid t \in s\}\\
&\ = \{t \in Dom(Y) \mid \exists u \in s\ \forall 1 \le i \le m \ \text{ s.t. } u[Y_i] = t[Y_i]\}\\
&\ = \{t \in Dom(Y) \mid \exists u \in s\ \forall 1 \le i \le m \ \text{ s.t. } t[Y_i] \in \{u[Y_i]\}\}\\
&\text{because } f_{Y_i}(t) = \{t\}, \text{ for every } t \in Dom(Y_i)\\
&\ = \{t \in Dom(Y) \mid \exists u \in s\ \forall 1 \le i \le m \ \text{ s.t. } t[Y_i] \in f_{Y_i}(u)\}\\
&\ = \mu_{f_{Y_1},...,f_{Y_m}}(s)
\end{aligned}
$$

Since mapper functions may map input tuples into empty sets (i.e., no output values are created), they may act as filtering conditions which enable the mapper to behave not only as a tuple producer but also as a filter.

Rule 3: *Let $S(X_1, ..., X_n)$ be a relation schema, C a condition over the attributes of this schema. There exists a set F of proper mapper functions for transforming $S(X)$ s.t., for every relation instance $s(X_1, ..., X_n)$, the term $\sigma_C(s)$ is equivalent to $\mu_F(s)$.*

Proof It suffices to show how F can be constructed from C and prove the equivalence of σ_C and μ_F. Let $F = f_{X_1}, ..., f_{X_n}$ where each mapper function f_{X_i} is the mapper function with domain X_i s.t.

$$
f_{X_i}(t) = \begin{cases} \{t[X_i]\} & \text{if } C(t) \\ \emptyset & \text{if } \neg C(t) \end{cases}
$$

We have,

$$
\begin{aligned}
\mu_F(s) &= \{t \in Dom(X) \mid \exists u \in s\ \forall 1 \le i \le n\ t[X_i] \in f_{X_i}(u)\}\\
&\quad \text{by the definition of } f_{X_i}\\
&= \{t \in Dom(X) \mid \exists u \in s \text{ s.t. } (\forall 1 \le i \le n\ t[X_i] \in \{u[X_i]\}) \text{ and } C(u)\}\\
&= \{t \in Dom(X) \mid \exists u \in s \text{ s.t. } (\forall 1 \le i \le n\ t[X_i] = u[X_i]) \text{ and } C(u)\}\\
&= \{t \in Dom(X) \mid \exists u \in s \text{ s.t. } t = u \text{ and } C(u)\}\\
&= \{t \in Dom(X) \mid t \in s \text{ and } C(t)\}\\
&= \sigma_C(s)
\end{aligned}
$$

4 Algebraic optimization rules

Algebraic rewriting rules are equations that specify the equivalence of two algebraic terms. Through algebraic rewriting rules, queries presented as relational expressions can be transformed into equivalent ones that are more efficient to evaluate. In this section we present a set of algebraic rewriting rules that enable the logical optimization of relational expressions extended with the mapper operator.

One commonly used strategy consists of minimizing the amount of information transferred from operator to operator. In this spirit, we adapt two classes of algebraic rewriting rules to the mapper operator. We start by presenting the rules for *pushing selections*, that attempt to reduce the cardinality of the source relations to be evaluated as early as possible and then we present rules *pushing projections*, that avoid propagating attributes that are not used by subsequent operators.

4.1 Pushing selections to mapper functions

When applying a selection to a mapper we can take advantage of the mapper semantics to introduce an important optimization. Given a selection $\sigma_{C_{A_i}}$ applied to a mapper $\mu_{f_{A_1},...,f_{A_k}}$, this optimization consists of pushing the selection $\sigma_{C_{A_i}}$, where C_{A_i} is a condition on the attributes produced by some mapper function f_{A_i}, directly to the output of the mapper function. Rule 4 formalizes this notion.

Rule 4: *Let $F = f_{A_1}, ..., f_{A_k}$ be a list of multi-valued mapper functions, proper for transforming relations with schema $S(X)$. Consider a condition C_{A_i} dependent of a set of attributes A_i for some $1 \leq i \leq k$. Then, for every relation instance $s(X)$,*

$$\sigma_{C_{A_i}}(\mu_F(s)) = \mu_{F \setminus \{f_{A_i}\} \cup \{\sigma_{C_{A_i}} \circ f_{A_i}\}}(s)$$

where

$$(\sigma_{C_{A_i}} \circ f_{A_i})(t) = \begin{cases} f_{A_i}(t) & \text{if } C(t) \\ \emptyset & \text{if } \neg C(t) \end{cases}$$

Proof Let $Y = A_1 \cdot ... \cdot A_k$.

$$\begin{aligned} \sigma_{C_{A_i}}(\mu_F(s)) &= \{t \in \mathit{Dom}(Y) \mid t \in \mu_F(s) \text{ and } C_{A_i}(t[A_i])\} \\ &= \{t \in \mathit{Dom}(Y) \mid \exists u \in s \\ &\qquad \forall 1 \leq j \leq k \ s.t. \ t[A_j] \in f_{A_j}(u) \text{ and } C_{A_i}(t[A_i])\} \\ &= \{t \in \mathit{Dom}(Y) \mid \exists u \in s \\ &\qquad \forall 1 \leq j \leq k, \ j \neq i \ s.t. \ t[A_j] \in f_{A_j}(u) \text{ and} \\ &\qquad t[A_i] \in f_{A_i}(u) \text{ and } C_{A_i}(t[A_i])\} \\ &= \{t \in \mathit{Dom}(Y) \mid \exists u \in s \\ &\qquad \forall 1 \leq j \leq k, \ j \neq i \ s.t. \ t[A_j] \in f_{A_j}(u) \text{ and} \\ &\qquad t[A_i] \in \sigma_{C_{A_i}}(f_{A_i}(u)\} \\ &= \mu_{F \setminus \{f_{A_i}\} \cup \{\sigma_{C_{A_i}} \circ f_{A_i}\}}(s) \end{aligned}$$

The benefits of Rule 4 are easier to understand when considering the alternative definition for the mapper semantics in terms of a Cartesian product presented in Section 3.1. Intuitively, if at least one of the mapper functions is multi-valued, it follows from Proposition 1, that the Cartesian product expansion generated by $f_{A_1}(u) \times ... \times f_{A_k}(u)$ can produce duplicate values for some set of attributes A_i, $1 \leq i \leq k$. To see how, please refer to Example 3.1. Hence, by pushing the condition C_{A_i} to the mapper function f_{A_i}, the condition will be evaluated fewer times, i.e., only once for each output value of $f_{A_i}(t)$ in opposition to once for each output tuple of $\mu_F(t)$. This is particularly important if we are speaking of expensive predicates, like those involving expensive functions or sub-queries (e.g., evaluating the SQL **exists** operator). See, e.g., [Hel98] for details on optimization of queries with expensive predicates.

Furthermore, note that when $C_{A_i}(t)$ does not hold, the evaluation of $(\sigma_{C_{A_i}} \circ f_{A_i})(t)$ returns the empty set. Considering the Cartesian product semantics of the mapper operator presented in Proposition 1, once a function returns the empty set, no output tuples will be generated. Thus, we may skip the evaluation of all mapper functions f_{A_j}, such that $j \neq i$. Physical execution algorithms for the mapper operator can take advantage of this optimization by evaluating f_{A_i} before any other mapper function.

Even in situations in which neither expensive functions nor expensive predicates are present, this optimization can be employed as it alleviates the average cost of the Cartesian product, which depends on the cardinalities of the sets of values produced by the mapper functions.

EXAMPLE 4.1: *Consider the relation* $\texttt{SMALLPAYMENTS}[\texttt{ACCTNO}, \texttt{AMOUNT}, \texttt{SEQNO}]$ *formed by all payments whose amount is smaller than* 5. *This relation can be obtained from the relation* $\texttt{PAYMENTS}$ *presented in Example 1.1 by composing a selection with a mapper. According to Example 3.1,* $\mu_{acct,amt}(\texttt{LOANS})$ *corresponds to the relation* $\texttt{PAYMENTS}$*, then the expression* $\sigma_{\texttt{AMOUNT}<5}(\mu_{acct,amt}(\texttt{LOANS}))$ *denotes the relation* $\texttt{SMALLPAYMENTS}$*. By applying Rule 4 to the above expression we obtain* $\mu_{acct,\sigma_{\texttt{AMOUNT}<5} \circ amt}(\texttt{LOANS})$*, which is faster to evaluate.*

4.2 Pushing selections through mappers

An alternative way of rewriting expressions of the form $\sigma_C(\mu_F(s))$ consists of replacing the attributes that occur in the condition C by the mapper functions that compute them. Suppose that, in the selection condition C, an attribute A is produced by the mapper function f_A. By replacing the attribute A with the mapper function f_A in condition C we obtain an equivalent condition.

In order to formalize this notion, we first need to introduce some notation. Let $F = f_{A_1}, ..., f_{A_k}$ be a list of mapper functions proper for transforming $S(X)$

and $Y = A_1 \cdot ... \cdot A_k$. The function resulting from the restriction of f_{A_i} to an attribute $Y_j \in A_i$ is denoted by $f_{A_i}|_{Y_j}$. Moreover, given an attribute $Y_j \in Y$, $F|_{Y_j}$ represents the function $f_{A_i}|_{Y_j}$ s.t. $Y_j \in A_i$. Note that, because F is a proper list of mapper functions, the function $F|_{Y_j}$ exists and is unique.

Rule 5: *Let $F = f_{A_1}, ..., f_{A_k}$ be a list of mapper functions, proper for transforming $S(X)$, $Y = A_1 \cdot ... \cdot A_k$ and $B = B_1, ..., B_m$ be a list of attributes in Y. If $H = F|_{B_1}, ..., F|_{B_m}$ is a list of single-valued functions then, for every relation instance s of $S(X)$,*

$$\sigma_{C_B}(\mu_F(s)) = \mu_F(\sigma_{C[B_1,...,B_m \leftarrow F|_{B_1},...,F|_{B_m}]}(s))$$

where C_B means that C depends on the attributes of B, and the condition that results from replacing every occurrence of each B_i by E_i is represented as $C[B_1, ..., B_m \leftarrow E_1, ..., E_m]$.

This rule replaces each attribute B_i in the condition C by the expression that describes how its values are obtained. In practice, this rule is of broad application as the attributes used in the condition of a selection are often generated either by single-valued functions like *(i)* identity mapper functions, *(ii)* constant mapper functions or *(iii)* arithmetic expressions. Cases *(i)* and *(ii)* draw from attribute renamings and value assignments. As an illustration, consider the condition C to be $A < B$. We may rewrite the expression $\sigma_{A<B}(\mu_{X\to A,2\to B,f_C}(s))$ as $\mu_{X\to A,2\to B,f_C}(\sigma_{X<2}(s))$. Concerning case *(iii)*, a new condition is produced by expanding attributes with arithmetic expressions. In this case, although the expression is evaluated twice —once in the condition and once in the mapper—, the number of tuples that have to be handled by the mapper operator can be drastically reduced. These tradeoffs are analyzed in detail in Section 5.

Proof In order to prove Rule 5 we proceed in two steps. We start by expanding both expressions into their corresponding sets of tuples. Then we establish the equivalence of these sets. So, on the one hand we have that,

$$\begin{aligned}\sigma_{C_B}(\mu_F(s)) &= \{t \in Dom(Y) \mid t \in \mu_F(s) \text{ and } C_B(t)\} \\ &= \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } t[A_i] \in f_{A_i}(u) \text{ and } C_B(t), \qquad (1) \\ &\qquad \forall 1 \le i \le k\}\end{aligned}$$

On the other hand we have that,

$$\begin{aligned}
&\mu_F(\sigma_{C[B_1,...,B_m \leftarrow F|_{B_1},...,F|_{B_m}]}(s)) \\
&\quad = \{t \in Dom(Y) \mid \exists u \in \sigma_{C[B_1,...,B_m \leftarrow F|_{B_1},...,F|_{B_m}]} \text{ s.t. } t[A_i] \in f_{A_i}(u), \\
&\qquad \forall 1 \leq i \leq k\} \\
&\quad = \Big\{t \in Dom(Y) \mid \exists u \in \{v \in Dom(X) \mid v \in s \text{ and} \qquad (2) \\
&\qquad C[B_1,...,B_m \leftarrow F|_{B_1},...,F|_{B_m}](v)\} \text{ s.t. } t[A_i] \in f_{A_i}(u), \forall 1 \leq i \leq k\Big\} \\
&\quad = \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } t[A_i] \in f_{A_i}(u) \\
&\qquad\quad \text{and } C[B_1,...,B_m \leftarrow F|_{B_1},...,F|_{B_m}](u), \forall 1 \leq i \leq k\}
\end{aligned}$$

It now remains to prove that, if $t[A_i] \in f_{A_i}(u)$, for every $1 \leq i \leq k$ then

$$C[B_1,...,B_m \leftarrow F|_{B_1},...,F|_{B_m}](u) \text{ iff } C_B(t)$$

This trivially follows by the definition of $F|_{B_i}$ and the fact that all functions in H are single-valued.

4.3 Pushing projections

A projection applied to a mapper is an expression of the form $\pi_Z(\mu_F(s))$. If $F = f_{A_1},...,f_{A_k}$ is a list of mapper functions, proper for transforming $S(X)$, then an attribute Y_i in $Y = A_1 \cdot ... \cdot A_k$ such that $Y_i \notin Z$, (i.e., that is not projected by π_Z) is said to be *projected-away.* Attributes that are projected-away are prone to optimizations. Since they are not required for subsequent operations, the mapper functions that generate them do not need to be evaluated. Hence, in some situations, they can be forgotten. More concretely, a mapper function can be forgotten if the attributes that it generates are projected-away. Rule 6 makes this idea precise.

Rule 6: *Let $F = f_{A_1},...,f_{A_k}$ be a list of mapper functions, proper for transforming $S(X)$ and $Y = A_1 \cdot ... \cdot A_k$. Let Z and Z' be lists of attributes in Y. For every relation instance s of $S(X)$, $\pi_Z(\mu_F(s)) = \pi_Z(\mu_{F'}(s))$, where $F' = \{f_{A_i} \in F \mid \ A_i$ contains at least one attribute in $Z\}$.*

Proof In what follows, we use $A_i \cap Z \neq \emptyset$ to represent that "*at least one attribute of A_i is in the list Z*". Thus,

$$\begin{aligned}
\pi_Z(\mu_F(s)) &= \{t[Z] \mid t \in Dom(Y) \text{ and } t \in \mu_F(s)\} \\
&= \{t[Z] \mid t \in Dom(Y) \text{ and } \exists u \in s\ \forall f_{A_i} \in F \text{ s.t. } \ t[A_i] \in f_{A_i}(u)\} \\
&\quad \text{because only attributes in } A_i \cap Z \text{ are projected} \\
&\quad \text{and, by hypothesis, } A_i \cap Z \neq \emptyset \Leftrightarrow f_{A_i} \in F' \\
&= \{t[Z] \mid t \in Dom(Y) \text{ and } \exists u \in s\ \forall f_{A_i} \in F' \text{ s.t. } t[A_i] \in f_{A_i}(u), \} \\
&= \pi_Z(\mu_{F'}(s))
\end{aligned}$$

Concerning Rule 6, it should be noted that if $Z = A_1 \cdot ... \cdot A_k$ (i.e, all attributes are projected), then $F' = F$ (i.e., no mapper function can be forgotten).

Example 4.2: *Consider the mapper $\mu_{acct,amt}$ defined in Example 3.1. The expression $\pi_{\texttt{AMOUNT}}(\mu_{acct,amt}(\texttt{LOANS}))$ is equivalent to $\pi_{\texttt{AMOUNT}}(\mu_{amt}(\texttt{LOANS}))$. The acct mapper function is forgotten because the **ACCOUNT** attribute was projected-away. Conversely, neither of the mapper functions can be forgotten in the expression $\pi_{\texttt{ACCTNO,SEQNO}}(\mu_{acct,amt}(\texttt{LOANS}))$.*

In what concerns optimization, another important observation is that attributes that are not used as input of any mapper function need not be retrieved from the mapper input relation. Thus, we may introduce a projection that retrieves only those attributes that are relevant for the functions in F'.

Rule 7: *Let $F = f_{A_1}, ..., f_{A_k}$ be a list of mapper functions, proper for transforming $S(X)$ and $Y = A_1 \cdot ... \cdot A_k$. For every relation instance s of $S(X)$, $\mu_F(s) = \mu_F(\pi_N(s))$, where N is a list of attributes in X, that only includes the attributes in $Dom(f_{A_i})$, for every mapper function f_{A_i} in F.*

Proof

$$\mu_F(s) = \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \le i \le k \text{ s.t. } t[A_i] \in f_{A_i}(u)\}$$

by the definition of mapper function,

$$f_{A_i}(u) = f_{A_i}(u[B]) = f_{A_i}(u[N])$$
$$= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \le i \le k \text{ s.t. } t[A_i] \in f_{A_i}(u[N])\}$$
$$= \{t \in Dom(Y) \mid \exists u \in \pi_N(s) \ \forall 1 \le i \le k \text{ s.t. } t[A_i] \in f_{A_i}(u)\}$$
$$= \mu_F(\pi_N(s))$$

Example 4.3: *Consider the relation $\texttt{LOANS}[\texttt{ACCT}, \texttt{AM}]$ of Example 1.1. The attribute **AM** is an input attribute of the mapper function amt defined in Example 3.1. Thus, the expression $\mu_{amt}(\texttt{LOANS})$ is equivalent to $\mu_{amt}(\pi_{\texttt{AM}}(\texttt{LOANS}))$.*

5 Estimating the cost of filter expressions

In this section, we present the cost estimation framework for expressions that combine selections and mappers. First, the cost of the mapper operator is analyzed in isolation. Then, we proceed to estimate the cost of applying a selection to a mapper. Finally, we elaborate on the cost estimates for optimized expressions obtained by applying Rule 4 and Rule 5, giving particular attention to the gains obtained with the proposed optimizations.

We have identified the *predicate selectivity* [SAC+79], the *mapper function fanout* and the *mapper function evaluation* cost as the primary factors that affect the gain obtained when applying the proposed optimizations. Similarly to [CS93], we designate the average cardinality of the output values produced by a mapper function as the *function fanout*. Analogously, the *mapper fanout* is defined as the average number of tuples produced by the mapper for each input tuple.

5.1 Estimating the cost of a mapper operator

Since the evaluation of a mapper can be performed in a tuple by tuple basis, the cost of evaluating the mapper operator expression $\mu_F(r)$ can be estimated by adding up the per-tuple cost of transforming each tuple of the input relation r. For each tuple $t \in r$, the cost of producing the output tuples can be defined as the sum of the cost of evaluating all mapper functions and the cost of performing the Cartesian product of the function outputs. We will not take the I/O cost into account, since the proposed algebraic optimization rules are unable to improve it.

Let C_f be the estimated per-tuple cost of a mapper function f. Then, C_F is the estimated per-tuple cost of evaluating all the mapper functions $f \in F$, given by $C_F = \sum_{f \in F} C_f$.

The cost of computing a Cartesian product is linear in the size of its inputs, i.e., given two sets of elements A and B, the Cartesian product $A \times B$ can be computed in time linear to $|A| \cdot |B|$.

For a given tuple t, when evaluating an expression of the form $\mu_F(t)$, the input of the Cartesian product consists of the sets returned by the mapper functions in F. In this way, if $F = f_{A_1}, ..., f_{A_m}$, then the cost of computing the Cartesian product algorithm, is $k \cdot |f_{A_1}(t)| \cdot ... \cdot |f_{A_m}(t)| + m \cdot k_0$, where k is an adjustment factor, m is the number of functions in F and the constant k_0 represents the overhead incurred by the algorithm for checking the emptiness of the input sets. In practice, when $|f_{A_i}(t)| = 0$, the cost of the Cartesian product algorithm is not zero but a small amount captured by $m \cdot k_0$.

We now need to have an estimate for $|f(t)|$, for every $f \in F$. The estimated value of $|f(t)|$ is given by the expected fanout of a mapper function f, designated as O_f. The mapper fanout is represented as O_F. We assume that the functions outputs are not correlated, thus the value of O_F can be approximated by $\prod_{f \in F} O_f$. Therefore, if we consider F to have m mapper functions, the per-tuple cost of executing the Cartesian product is estimated as $C_{prd} = k \cdot O_F + m \cdot k_0$. Finally, for an input relation r with cardinality n, the estimated cost of $\mu_F(r)$ is given by $n \cdot (C_{prd} + C_F) = n \cdot (k \cdot \prod_{f \in F} O_f + m \cdot k_0 + \sum_{f \in F} C_f)$.

5.2 Estimating the cost of a filter applied to a mapper

The cost of the expression $\sigma_{C_A}(\mu_F(r))$ can be estimated to be the cost of evaluating the mapper plus the cost of evaluating the selection condition on each tuple produced by the mapper. In the sequel, we will refer to this expression as the non-optimized expression.

Consider C_{sel} to be the average per-tuple cost of evaluating the selection condition C_A and let α be its corresponding selectivity, with $0 \leq \alpha \leq 1$. The cost of the non-optimized expression is:

$$n \cdot (C_{prd} + C_F) + n \cdot O_F \cdot C_{sel} \tag{3}$$

where $n \cdot (C_{prd} + C_F)$ is the expected cost of μ_F. Multiplying n by the fanout of the mapper O_F, we get the expected number of output tuples for the mapper operator. Since the selection condition is evaluated once for each tuple returned by the mapper, $n \cdot O_F \cdot C_{sel}$ represents the total cost of evaluating the selection condition.

5.3 Estimating the cost of an expression optimized with Rule 4

We now consider the cost of the optimized expression $\mu_{F \backslash g_{A_j} \cup \{\sigma_{C_{A_j}} \circ g_{A_j}\}}(r)$ obtained through Rule 4. Assuming that $g_{A_j} \in F$ is the mapper function onto which the condition is pushed, we consider a list of functions where the mapper function $\sigma_{C_{A_j}} \circ g_{A_j}$ replaces g_{A_j}.

The per-tuple cost of evaluating $\sigma_{C_{A_j}} \circ g_{A_j}$ is estimated to be $C_{g_{A_j}} + O_{g_{A_j}} \cdot C_{sel}$, i.e., the cost of evaluating the mapper function g_{A_j} plus the cost of evaluating the selection condition C_{A_j} for each element produced by the function.

Obviously, the cost of the Cartesian product for the optimized expression is not the same as the cost of the Cartesian product for the non-optimized expression, since $\sigma_{C_{A_j}} \circ g_{A_j}$ and g_{A_j} have different fanouts. More precisely, since α represents the probability that C_{A_j} holds, the fanout of $\sigma_{C_{A_j}} \circ g_{A_j}$ is given by $\alpha \cdot O_{g_{A_j}}$. This means that the cost of the Cartesian product for the optimized expression, represented as $C_{prd'}$ is given by $k \cdot O_{F \backslash g_{A_j}} \cdot \alpha \cdot O_{g_{A_j}} + m \cdot k_0$, which is equivalent to $k \cdot \alpha \cdot O_F + m \cdot k_0$.

The cost of the mapper corresponding to the optimized expression is estimated to be:

$$n \cdot (C_{prd'} + C_{F \backslash g_{A_j}} + C_{g_{A_j}} + O_{g_{A_j}} \cdot C_{sel}) \tag{4}$$

which corresponds to the cost of the Cartesian product plus the cost of computing all functions except g_{A_j}, plus the cost of computing $\sigma_{C_{A_j}} \circ g_{A_j}$. This can be simplified to

$$n \cdot (C_{prd'} + C_F + O_{g_{A_j}} \cdot C_{sel}) \tag{5}$$

The expected gain for this optimization represented as Δ_{G_4} is now computed as the difference between (3) and (5), which becomes:

$$\Delta_{G_4} = n \cdot (C_{prd} + C_F) + n \cdot O_F \cdot C_{sel} - n \cdot (C_{prd'} + C_F + O_{g_{A_j}} \cdot C_{sel}) \tag{6}$$

Since $C_{prd'} = k \cdot \alpha \cdot O_F + m \cdot k_0$, developing and simplifying (6) we obtain:

$$\Delta_{G_4} = n \cdot k \cdot O_F \cdot (1 - \alpha) + n \cdot C_{sel} \cdot (O_F - O_{g_{A_j}}) \tag{7}$$

We remark that high gains are obtained for small selectivities. In contrast, as the selectivity α approaches 100%, the factor $n \cdot k \cdot O_F \cdot (1 - \alpha)$ in (7) tends to zero, thus decreasing the gain. Concerning the influence of the mapper function fanout $O_{g_{A_j}}$, we conclude from (7) that the greater is the difference between O_F and $O_{g_{A_j}}$, the higher is the gain. It is interesting to observe that if $O_{g_{A_j}} > O_F$, when the selectivity is near 100%, Δ_{G_4} will be negative. However, for this to be possible, since $g_{A_j} \in F$, some other function in F should have a fanout much smaller than 1. If the fanout $O_{g_{A_j}}$ is smaller than the mapper fanout, i.e., $O_{g_{A_j}} < O_F$, the gain will always be positive. In this situation, the higher is the value of C_{sel}, the higher is the gain Δ_{G_4} obtained.

5.4 Estimating the cost of an expression optimized with Rule 5

As presented in Section 4.2, the optimized expression obtained by applying Rule 5 takes the form $\mu_F(\sigma_{C[B_1,...,B_l \leftarrow F|_{B_1},...,F|_{B_l}]}(s))$, where $H = F|_{B_1}, ..., F|_{B_l}$ is the set of mapper functions that are propagated into the selection condition.

The cost of the optimized expression is given by summing *(i)* the cost of evaluating the new selection condition $C[B_1, ..., B_l \leftarrow F|_{B_1}, ..., F|_{B_l}]$, with *(ii)* the cost of evaluating the mapper μ_F for every tuple that is not filtered by the condition. Since the new condition is obtained by inlining the mapper functions of H in the condition C, the per-tuple cost of evaluating the new condition is estimated as $C_{sel} + C_H$, that is, the cost of evaluating the initial selection plus the cost of evaluating the propagated functions. Therefore, when applying this rule, the Cartesian product and the rest of the mapper functions are only evaluated when $\sigma_{C[B_1,...,B_l \leftarrow F|_{B_1},...,F|_{B_l}]}$ holds. Thus, we estimate the cost of the optimized expression as:

$$n \cdot (C_{sel} + C_H) + n \cdot \alpha \cdot (C_{prd} + C_F) \tag{8}$$

where $n \cdot (C_{sel} + C_H)$ represents the cost of evaluating the condition and $n \cdot \alpha \cdot (C_{prd} + C_F)$ represents the cost of evaluating the mapper for the tuples that are not filtered by the condition. Note that, since only single-valued functions can be pushed into the condition, the mapper functions in H have fanout equal to one.

The gain of this optimization is obtained as the difference between (3) and (8). Hence,

$$\Delta_{G_5} = n \cdot (C_{prd} + C_F) + n \cdot O_F \cdot C_{sel} - n \cdot (C_{sel} + C_H) - n \cdot \alpha \cdot (C_{prd} + C_F) \quad (9)$$

which becomes

$$\Delta_{G_5} = n \cdot (1 - \alpha) \cdot (C_{prd} + C_F) + n \cdot C_{sel} \cdot (O_F - 1) - n \cdot C_H \quad (10)$$

Looking at the gain formula (10), we observe that smaller selectivities α result in higher gains. The gain Δ_{G_5} increases with the fanout of the mapper O_F, with the evaluation cost of the selection condition C_{sel} and with the evaluation cost of all mapper functions C_F. Pushing fewer functions or cheaper functions to the selection condition means lower values of C_H, which also results in higher gains.

5.5 Selecting the best optimization

In some situations, only one of the rewritting rules apply. Rule 5 can only be applied when the attributes of the condition are produced by single-valued functions, while Rule 4 can be employed when optimizing selections whose conditions involve attributes mapped by multi-valued or single-valued functions. Additionally, Rule 4 can only be applied when the attributes of the selection condition are produced by only one function, while Rule 5 can be applied when conditions involve multiple attributes that are produced by multiple functions.

If the attributes of the selection condition are produced by only one mapper function and, furthermore, if this mapper function is single-valued, then both rules can be applied. In this case, we need to identify the rule that brings the higest gain. This is determined by comparing the gains obtained by both rules. It is more advantageous to use Rule 4 instead of Rule 5 when $\Delta_{G_4} - \Delta_{G_5} > 0$, which is the same as

$$n \cdot C_H + n \cdot C_{sel} \cdot (O_{g_{A_j}} - 1) - n \cdot (1 - \alpha) \cdot (C_F + m \cdot k_0) > 0 \quad (11)$$

We develop (11) taking into account that, since g_{A_j} is single-valued, the fanout $O_{g_{A_j}}$ is 1, and we get

$$C_H > (1 - \alpha) \cdot (C_F + m \cdot k_0) \quad (12)$$

As the selectivity α approaches 100%, we see that $(1-\alpha)\cdot(C_F + m\cdot k_0)$ gets smaller. This indicates that for higher selectivities, Rule 5 is more likely to perform better than Rule 4. Since C_H and C_F are fixed, we note that there always exists a selectivity α_0 for which Rule 5 is better than Rule 4. Moreover, the higher is the difference between the C_F and C_H, the lower is α_0.

6 Experimental Validation

In this section, we seek to validate the optimizations, the cost formulas proposed and to compare the different optimization rules. To achieve this desiderate, we implemented the mapper operator and conducted a number of experiments that compare expressions combining selections with mappers to their optimized equivalents. The experiments address the influence of predicate selectivity, the mapper function fanout and the mapper function cost on the optimizations proposed in Rule 4 and in Rule 5.

To ensure the same conditions for both rules, our setup is as follows. A base expression $\sigma_{p_i}(\mu_{f_1,f_2,f_3,f_4}(r))$ is compared with the optimized variants $\mu_{f_1,\sigma_{p_i}\circ f_2,f_3,f_4}(r)$ for Rule 4, and $\mu_{f_1,f_2,f_3,f_4}(\sigma_{p_i[f_2]}(r))$ for Rule 5. The mapper function f_1, unless otherwise stated, has a fanout of 2.0, f_2 always has a fanout of 1.0 and the remaining functions, f_3 and f_4, have a fanout of 2.0. The input relation r is an input relation with synthetic data. The predicate p_i corresponds to a condition with a predefined selectivity. Furthermore, the predicate $p_i[f_2]$ represents a new predicate that results from expanding the function f_2 in the condition corresponding to p_i, as presented in Section 4.2.

For the sake of accuracy, we applied predicates that guarantee predefined selectivity values. Likewise, when we needed to vary the fanout or the cost factors of a function, we employed functions specifically designed to guarantee predefined fanout factors and per-tuple costs. Each experiment measured *total work*, i.e., the sum of the time taken to read the input tuples, plus the time taken to compute the output tuples, plus the time taken to write them.

The implementation of the mapper operator was developed on top of the XXL library [vdBDS00, vdBDK$^+$01], which provides database query processing functionalities through a set of relational operators.

All experiments were conducted on a standard PC with an AMD Athlon processor at 1.8Ghz, 1GB of RAM, Linux kernel version 2.4.27 and Sun's JRE 1.4.2. The tests were performed with the machine physically disconnected from the network and only the essential operating system processes were running. Time measurements were performed using stopwatch objects implemented with the Java time utility classes.

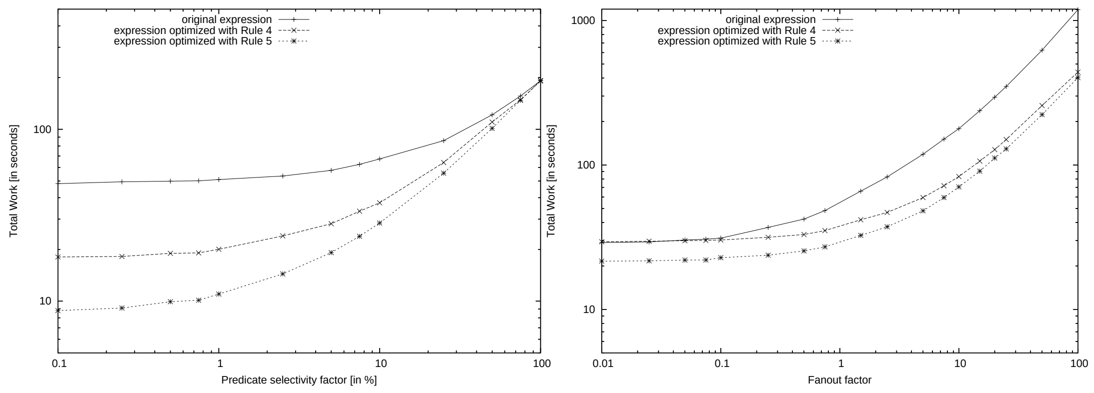

Fig. 2. Evolution of total work for the original and optimized expressions with one million tuples using cheap functions. *(a)* On the left, the effect of applying predicates p_i with increasing selectivities. *(b)* On the right, the effect of increasing the fanout of the mapper function f_1, maintaining the predicate selectivity fixed to 2.5%.

To ascertain that the differences in performance were caused by improvements brought by one optimized expression over the original, we verified that the amount of I/O performed on both expressions was the same and, furthermore, that it was performed in the same regions of the disk. To that end, raw devices were used instead of regular files.

6.1 The influence of the predicate selectivity

Seeking to validate the effect of the predicate selectivity, a set of experiments was carried out using a different p_i predicate with selectivities ranging from 0.1% to 100%. The tests were executed over an input relation with 1 million input tuples. Figure 2a shows the evolution of the total work for different selectivities, using cheap functions with their default fanouts.

As expected, for both rules, the highest gains brought by the optimization were obtained for small selectivities. For Rule 4, more concretely, for a selectivity of 0.1%, the optimized expression was 2.7 times faster than the original one. For Rule 5, with the same selectivity, the optimized expression was 5.5 times faster.

In what concerns Rule 4, as the selectivity decreases, more results are filtered out from function f_2 by the predicate p_i and, therefore, the cost of computing the Cartesian product involved in the mapper is smaller. As the selectivity tends to 100%, the gain drops since the number of tuples filtered out from f_2 tend to zero. These results validate the gain formula (7). This rule also reduces the number of times the condition is evaluated. Even for a selectivity of 100%, the non-optimized expression evaluates the condition more often that the optimized expression. However, since in these experiments the predicate evaluation is very cheap, the small gain obtained is not visible in the figure.

In what concerns Rule 5, as a direct effect of pushing the condition through the mapper, the mapper is evaluated over fewer tuples and thus, many Cartesian product computations and function evaluations are saved. As the selectivity of the condition tends to 100%, the number of tuples fed into the mapper grows. Therefore, the cost of the non-optimized expression is approximately the same as the cost of the optimized expression.

6.2 The influence of the function fanout

In order to experimentally check how the function fanout affects the proposed optimizations, we tracked the evolution of total work for the original and optimized expressions when the fanout factor varies. Function f_1 was replaced by a function that guarantees a predefined fanout factor ranging from 0.01 (unusually small) to 100. To isolate the effect of the fanout, the selectivity of the predicate was kept constant at 2.5%. The results are depicted in Figure 2b.

For small values of the fanout, Rule 4 presents a slight degradation of $\approx 1\%$ in performance with respect to the performance of the original expression, while Rule 5, displays an improvement of $\approx 35\%$. The modest improvement brought by Rule 5 is explained by the fact that, for small values of the fanout, the Cartesian product is rarely performed, so no gain is introduced. Additionally, in the case of Rule 4, for small values of the mapper fanout, the expression $O_F - O_{g_{A_j}}$ is negative. As a consequence, according to the gain formula (7), the gain is also negative.

As explained in Section 5.1, the cost of the Cartesian product increases with the fanout, since the higher the fanout, the more tuples have to be produced by the Cartesian product for each input tuple. For high values of fanout, the cost of performing the Cartesian product becomes the dominant factor. Thus, the gain obtained by both rules increases with the fanout since both optimizations reduce the cost of the Cartesian product. For a fanout of 100, we observed that Rule 4 was 2.7 times faster than the original and Rule 5 was 2.95 times faster. See Figure 2a and Figure 2b.

In this experiment, Rule 5 is consistently cheaper than Rule 4. Since the selectivity for this experiment is 2.5%, according to (12), we know that Rule 5 is cheaper than Rule 4 whenever $C_{f_2} < 97.5\% \cdot (C_F + m \cdot k_0)$. Trivially, this inequality holds because the cost of all functions in F is the same.

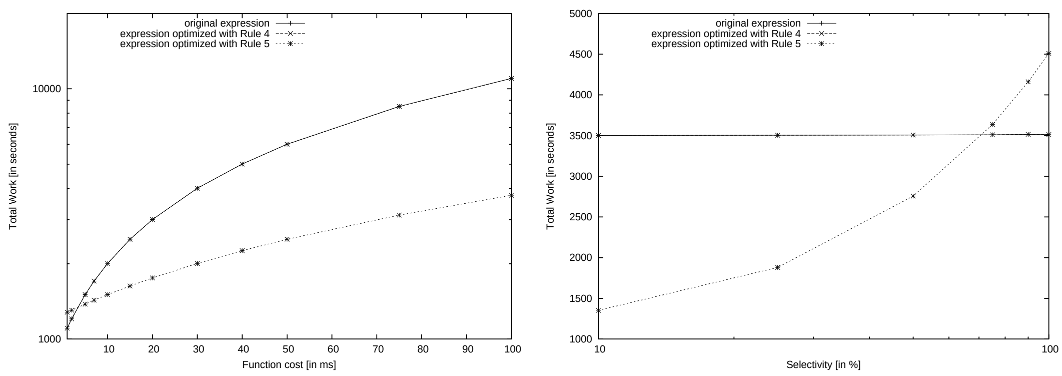

Fig. 3. Evolution of total work for the original and optimized expressions in the presence of expensive functions while processing 100K tuples with the cost of f_2 set to 10ms per call. *(a)* On the left, the effect of increasing the cost of f_3 with a constant selectivity factor of 2.5% for p_i. *(b)* On the right, the effect of increasing the selectivity factor of p_i, maintaining the cost of f_3 constant at 25ms per call.

6.3 The influence of the function evaluation cost

To validate how the function cost influences the optimization gains, two sets of experiments were put in place. The first set of experiments increased the cost of an expensive function, while the second set of experiments varied the selectivity of the condition in the presence of expensive functions. We considered f_3 to be the expensive mapper function. In the first set of experiments, shown in Figure 3a, the cost of f_3 varied from 1ms per call to 100ms per call. In the second set of experiments, shown in Figure 3b, the cost of f_3 is fixed to 25ms. In both sets of experiments the function being optimized, which is f_2, had a fixed cost of 10ms per call.

In Section 5.3, we remarked that the gain for Rule 4 is independent of the mapper function cost. Although there is a gain resulting from savings in the Cartesian product computation, as show by gain formula (7), this gain is very small by comparison with the mapper execution cost when we are in the presence of expensive functions. The outcome of the experiments is aligned with the cost estimates. Notice that in Figure 3a and Figure 3b, the line plots of the optimized expressions for Rule 4 overlap the line for the original expression.

In what concerns Rule 5, we observe that the cost of the mapper functions and the predicate selectivity directly influence the gain. Our observations validate the gain formula (10), in that small selectivities and a high function cost result in high gains.

In Figure 3a, the cost of the optimized expression for Rule 5 is initially higher than the cost of the original expression. This happens because for lower function costs, the mapper function f_2, which is the remaining function pushed into the selection condition, is more expensive than the function f_3. This means that, in the gain formula (10), $n \cdot C_H$ is higher than the other factors of the formula, which results in a negative gain. As f_3 gets more expensive, the value of C_F grows. This causes $n \cdot (1-\alpha) \cdot (C_{prd}+C_F)$ to increase, eventually leading to a positive gain.

In Figure 3b, we observe that the cost of the optimized expression for Rule 5 eventually becomes more expensive than the cost of the original expression. In fact, as the selectivity factor α increases, $n \cdot (1-\alpha) \cdot (C_{prd}+C_F)$ decreases, and since C_H is high, the gain eventually becomes negative.

These two experiments highlight the limitation of Rule 4. This rule does not optimize the cost of evaluating the functions. Thus, when the cost of evaluating the mapper functions is increases, both the original and the optimized expressions are more expensive. By contrast, Rule 5 reports important gains.

Nevertheless, Rule 4 is quite successful if the cost of applying the predicate is high. In the optimized version for Rule 4, the predicate is applied for each output value of the mapper function. In the non-optimized version, the predicate is applied for each tuple of the result Cartesian product. The number of tuples produced by the Cartesian product, for each input tuple, is given by multiplying the fanout factors of *all* mapper functions. In the presence of expensive predicates, for functions with high fanout, high gains can be achieved.

7 Related work

To support the growing span of applications of RDBMSs, several extensions to RA have been proposed since its inception (like e.g., aggregates [Klu82]), mainly in the form of new operators. Applications requiring data transformations bring a new requirement to RA as their focus is no more limited to the initial idea of deriving information, [Par78, AU79] but also involves the production of new data items.

Data transformation is an old problem and the idea of using a query language to specify such transformations has been proposed back in the 1970's with two prototypes, Convert [SHL75] and Express [SHT$^+$77], both aiming at data conversion.

More recently, three efforts, Potter's Wheel [RH01], Ajax [GFS$^+$01] and Data Fusion [CG04], have proposed operators for data transformation and cleaning purposes. Potter's Wheel fold operator is capable of producing several output tuples for each input tuple. The main difference w.r.t. the mapper operator lies in the number of output tuples generated. In the case of the fold operator, the number of output tuples is bound to the number of columns of the input relation, while the mapper operator may generate an arbitrary number of output tuples.

The semantics of the Ajax map operator represents exactly a one-to-many mapping. Unlike our data mapper, the Ajax operator allows the specification of a selection condition applied to each input tuple. The semantics of the Ajax map operator represents exactly a one-to-many mapping, but it has not been proposed as an extension of the relational algebra. Consequently, the issue of semantic optimization, as we propose in this paper, has not been addressed for the Ajax map.

Data Fusion tool [CG04] implements the semantics of the mapper operator as it is presented here, to express one-to-many data transformations in the context of legacy-data migrations. The employment of this tool in large commercial legacy-data migration projects corroborated the need of supporting data transformations that require one-to-many mappings. However, the current version of Data Fusion is not supported by an extended relational algebra as we propose.

Solutions for restructuring semi-structured data [Suc98] like WOL [DK97], YAT [CS97], and TransScm [MZ98] aim at transforming both schema and data. These systems use Datalog-style rules in their specification languages. Their expressiveness is restricted to avoid potentially dangerous specifications (that may result in diverging computations). As a result, they cannot express the dynamic creation of tuples.

Data transformations are also required in ETL processes. To the best of our knowledge, in most ETL tools, to express one-to-many data-transformations, the user has to resort to some form of ad-hoc scripting. Furthermore, the optimization of ETL data transformations is a recent effort [SVS05].

Clio [MHH$^+$01] is a tool aiming at the discovery and specification of schema mappings. It has the ability to generate SQL queries for data transformations from schema mappings. However, the class of data transformations supported by Clio is induced by *select-project-join* queries. Recent work on Clio [FKMP03] proposed to perform the transformation of data instances from a source schema into a target schema based on source-to-target schema dependencies, but their semantics of *universal solutions* is not powerful enough to entail the class of one-to-many transformations we propose to tackle in this document.

8 Conclusions

This paper addresses the problem of expressing one-to-many data transformations that frequently arise in legacy-data migrations, ETL processes, data cleaning and data integration scenarios. Since these transformations cannot be expressed as RA expressions, we have proposed a new operator named data mapper that is powerful enough to express them.

We then presented a simple semantics for the mapper operator and proved that RA extended with the mapper operator is more expressive than standard RA. Interesting properties of mappers were described. We showed that mappers admit a tuple-at-a-time semantics and that they can subsume standard relational operators like projection, renaming and selection. Then, a set of standard algebraic optimization rules for pushing projections and selections through mappers, that enable the logical optimization of a subset of relational queries extended with mappers, were proposed together with their corresponding proofs of correctness.

To better understand the issues that arise in the optimization of algebraic expressions involving our new operator, we developed and experimentally validated the cost estimation formulas for expressions that combine relational filters with mappers. This effort led us to identify the main factors that influence the optimization gains. This is an essential step towards designing heuristics for a cost-based optimizer.

We strongly believe that current relational database technology enhanced with the mapper operator will provide a powerful data transformation engine. We aim at providing both logical and physical optimization strategies to the query optimizer specially tailored for data transformations. To that end, we are developing and experimenting different physical execution algorithms for the mapper operator. Furthermore, we are extending the work presented with the algebraic optimization rules for other operators of the RA and maturing the cost formulas presented towards cost-based optimization.

From an application point-of-view, we also plan to incorporate this technology in the newer versions of the Ajax [GFS$^+$01] data cleaning tool and in Data Fusion [CG04] legacy-data migration tool.

References

[AdA93] P. Atzeni and V. de Antonellis. *Relational Database Theory.* The Benjamin/Cummings Publishing Company, Inc., 1993.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Database Systems*. Addison-Wesley, 1995.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc. of the 6th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, pages 110–119. ACM Press, 1979.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *ACM SIGMOD Int'l Conf. on the Managment of Data*, pages 177–188, 1998.

[CG04] P. Carreira and H. Galhardas. Efficient development of data migration transformations. In *ACM SIGMOD Int'l Conf. on the Managment of Data*, 2004.

[CGLP05] P. Carreira, H. Galhardas, A. Lopes, and J. Pereira. Extending the relational algebra with the Mapper operator. DI/FCUL TR 05–2, Department of Informatics, University of Lisbon, January 2005. URL `http://www.di.fc.ul.pt/tech-reports`.

[Cha98] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 34–43. ACM Press, 1998.

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

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'93)*, pages 529–542, 1993.

[CS97] S. Cluet and J. Siméon. Data integration based on data conversion and restructuring. Extended version of [CDSS98], 1997.

[DK97] S. B. Davidson and A. Kosky. Wol: A language for database transformations and constraints. In Alex Gray and Per-Åke Larson, editors, *Proc. of the 13th Int'l Conf. on Data Engineering*, pages 55–65. IEEE Computer Society, 1997.

[FKMP03] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *Proc. 8th Int'l Conf. on Database Theory (ICDT)*. IEEE Computer Society, 2003.

[Gal01] H. Galhardas. *Data Cleaning: Model, declarative language and algorithms*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2001.

[GFS$^+$01] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. A. Saita. Declarative data cleaning: Language, model, and algorithms. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*, 2001.

[GFSS00] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Ajax: An extensible data cleaning tool. *ACM SIGMOD Int'l Conf. on Management of Data*, 2(29), 2000.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases.

ACM Computing Surveys, 2(25), 1993.

[Hel98] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 22(2):113–157, 1998.

[HY90] R. Hull and M. Yoshikawa. Ilog: Declarative creation and manipulation of object identifiers. In *Proc. Int'l Conf. on Very Large Databases (VLDB'90)*, pages 455–468, 1990.

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[LR99] D. Lomet and E. A. Rundensteiner, editors. *Special Issue on Data Transformations.* IEEE Data Engineering Bulletin, 1999.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL - a Language for Querying and Restructuring Database Systems. In *Proc. Int'l Conf. on Very Large Databases (VLDB'96)*, pages 239–250, 1996.

[MHH$^+$01] R. J. Miller, L. M. Haas, M. Hernandéz, C. T. H. Ho, R. Fagin, and L. Popa. The Clio Project: Managing Heterogeneity. *SIGMOD Record*, 1(30), 2001.

[Mil98] R. J. Miller. Using Schematically Heterogeneous Structures. *Proc. of ACM SIGMOD Int'l Conf. on the Managment of Data*, 2(22):189–200, 1998.

[MZ98] T. Milo and S. Zhoar. Using schema matching to simplify heterogeneous data translation. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'98)*, 1998.

[Par78] J. Paredaens. On the expressive power of the relational algebra. *Inf. Processing Letters*, 7(2):107–111, 1978.

[RH01] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*, 2001.

[SAC$^+$79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM SIGMOD Int'l Conf. on the Managment of Data*, pages 23–34, 1979.

[SHL75] N. C. Shu, B. C. Housel, and V. Y. Lum. CONVERT: A High Level Translation Definition Language for Data Conversion. *Communications of the ACM*, 18(10):557–567, 1975.

[SHT$^+$77] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum. EXPRESS: A Data EXtraction, Processing and REStructuring System. *ACM Transactions on Database Systems*, 2(2):134–174, 1977.

[SKS01] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts.* MacGraw-Hill, 4th edition, 2001.

[Suc98] D. Suciu. An overview of semistructured data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and*

Computability Theory), 29(4):28–38, 1998.

[SVS05] A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL processes in data warehouses. In *Proc. of the 21st Int'l Conf. on Data Engineering (ICDE)*, 2005.

[vdBDK$^+$01] J. van den Bercken, J. P. Dittrich, J. Kräamer, T. Schäafer, M. Schneider, and B. Seeger. XXL a library approach to supporting eficient implementations of advanced database queries. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*, 2001.

[vdBDS00] J. van den Bercken, J. P. Dittrich, and B. Seeger. XXL: A prototype for a library of query processing algorithms. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*. ACM Press, 2000.