Extending Relational Algebra to express one-to-many data transformations

Paulo Carreira[1] **, Helena Galhardas**[2] **, Antónia Lopes**[1] **, João Pereira**[2]

[1] Faculty of Sciences of the University of Lisbon, C6 - Piso 3, 1700 Lisboa, Portugal

paulo.carreira@tagus.ist.utl.pt, mal@di.fc.ul.pt

[2]INESC-ID and Instituto Superior Técnico,
Avenida Prof. Cavaco Silva, Tagus Park, 2780-990 Porto Salvo, Portugal

hig@inesc-id.pt, joao@inesc-id.pt

Abstract. *Application scenarios such as* legacy-data migration, ETL processes, data cleaning *and* data-integration *require the transformation of input tuples into output tuples. Traditional approaches for implementing these data transformations enclose solutions as* Persistent Stored Modules (PSM) *executed by an RDBMS or transformation code using a commercial ETL tool. Neither of these solutions is easily maintainable or optimizable.*
To take advantage of the optimization capabilities of RDBMSs, data transformations are often expressed as relational queries. However, the limited expressive power of relational query languages like SQL hinder this approach. In particular, an important class of data transformations that produce several output tuples for a single input tuple cannot be expressed as a relational query.
In this paper, we present the formal definition of a new operator named data mapper operator *as an extension to the relational algebra to address this important class of data transformations. We demonstrate that relational algebra extended with the mapper operator is more expressive than standard relational algebra. Furthermore, we investigate several properties of the operator and supply a set of algebraic rewriting rules that enable the logical optimization of expressions that combine standard relational operators with mappers and present their proofs of correctness.*

1. Introduction

The setup of modern information systems comprises a number of activities that rely, to a great extent, in the employment of data transformations [Lomet and Rundensteiner, 1999]. Well known cases are the migrations of legacy-data, the ETL 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 specify 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

[Aho and Ullman, 1979], RA alone cannot used to specify many important data transformations [Lakshmanan et al., 1996].

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 this way are often difficult to maintain, and more importantly there is little possibility of optimization [Carreira et al., 2005]. We remark that only recently an optimization technique for ETL processes was proposed [Simitsis et al., 2005].

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 [Aho and Ullman, 1979]), 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 [Miller, 1998].

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 3.2), this class of data transformations cannot be expressed by standard RA.

In this paper, we propose an extension to RA to represent one-to-many data transformations. 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 [Chaudhuri, 1998]. 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. Following, 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 has been intentionally simplified for illustration purposes.

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

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

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 instalments with a maximum value of 100, such that, the sum of 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 instalment. This sequence number starts at* 1 *for each sequence of instalments 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. 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 formal study of the expressive power of the mapper operator;
3. a set of provably correct algebraic rewriting rules for expressions involving the mapper operator and relational operators, useful for optimization purposes.

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. Finally, related work is summarized in Section 5 and conclusions are presented in Section 6.

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* consists of a name R (the relation name) along with 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) of $R(A_1, ..., A_n)$ is 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 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 [Codd, 1970]. 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 definition of the new mapper operator and define other basic concepts. We assume two fixed relation schemas $S(X_1, ..., X_n)$ and $T(Y_1, ..., Y_m)$. We refer to S and T as the source and the target relation schemas, respectively.

A mapper is a unary operator μ_F that takes a relation instance of the source relation schema as input and produces a relation instance of the target relation schema as output. The mapper operator is parameterized by a set 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: *Let A be a non-empty list of distinct attributes in $Y_1, ..., Y_m$. An* $A-$mapper function *consists of a non-empty list of distinct attributes B in $X_1, ..., X_n$ and a computable function f_A:$Dom(B) \rightarrow \mathcal{P}(Dom(A))$.*

Let t be tuple of a relation instance of the source schema. We define $f_A(t)$ to be the application of the underlying function f_A to the tuple t, i.e., $f_A(t[B])$.

In this way, mapper functions describe how a specific part of the target data can be obtained from the source data. 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 the mapper function and the function 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 it. 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 according to *(ii)* the number of output attributes. Mapper functions that produce singleton sets, i.e., $\forall(t \in Dom(X)) \ |f_Y(t)| = 1$ 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 functions defined as $f_A : Dom(B) \rightarrow \mathcal{P}(Dom(A))$ s.t. $f_A(t) = \{t\}$. Notice that this is only possible when $Dom(B) = Dom(A)$.

As mentioned before, a mapper operator is parameterized by a set of mapper functions. This set is proper for transforming the data from the source to the target schemas if it specifies, in a unique way, how the values of every attribute of the target schema are produced.

DEFINITION 3.2: *A set $F = \{f_{A_1}, ..., f_{A_k}\}$ of mapper functions is said to be* proper *(for transforming the data of S into T) iff every attribute Y_i of the target relation schema is an element of exactly one of the A_j lists, for $1 \leq j \leq k$.*

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 a list values in $f_{A_i}(s)$. Formally, the mapper operator is defined as follows.

DEFINITION 3.3: *Given a relation $s(X)$ and a proper set 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 of the target relation schema defined by*

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

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 `ACCT`-mapper function that returns a singleton with the account number `ACCT` properly left padded with zeroes and amt is the `[AMOUNT,SEQNO]`-mapper function 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 $(\texttt{901}, \texttt{250.00})$, the result of evaluating $amt(t)$ is the set $\{(\texttt{100}, \texttt{1}), (\texttt{100}, \texttt{2}), (\texttt{50}, \texttt{3})\}$. 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 $\{\langle \texttt{'0901'}, \texttt{100}, \texttt{1}\rangle, \ \langle \texttt{'0901'}, \texttt{100}, \texttt{2}\rangle, \ \langle \texttt{'0901'}, \texttt{50}, \texttt{3}\rangle\}$.

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 data into quarterly 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 that makes up new employee numbers (i.e., a Skolem function [Hull and Yoshikawa, 1990]), and sal the `[QTNUM, QTSAL]`-mapper function

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

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 in some situations, 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* $s(X)$ *and a proper set of mapper functions* $F = \{f_{A_1}, ..., f_{A_k}\}$ *s.t.* $A_1 \cdot ... \cdot A_k = Y$,

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

PROOF

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

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 [Graefe, 1993].

The task of devising an algorithm that computes data transformation through mappers becomes straightforward: if every underlying mapper function only involves adjacent attributes of the target relation schema and in the same order (i.e., $A_1 \cdot ... \cdot A_k = Y$), then it is just to compute the Cartesian product as stated by the proposition; in the other situations, after calculating the Cartesian product, it is necessary to exchange the elements of each tuple so they become in the correct order. 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. The proof of this proposition can be found in [Carreira et al., 2005].

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

3.2. 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 $\mu_{\{f\}}$, 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: $|E_1 \cup E_2| \leq |E_1| + |E_2|; |E_1 - E_2| \leq |E_1|; |E_1 \times E_2| \leq |E_1| \times |E_2|; |\pi_L(E)| \leq |E|; |\sigma_C(E)| \leq |E|$.

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 renaming, projection 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 renaming, projection 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.

Renaming and projection can be obtained through mapper operators over identity mapper functions.

RULE 1**:** *Let $S(X_1, ..., X_n)$ and $T(Y_1, ..., Y_m)$ be two relation schemas s.t. Y is a sublist of X and let s be a relation instance of $S(X)$. The term $\pi_{Y_1,...,Y_m}(s)$ is equivalent to $\mu_F(s)$ where $F = \{f_{Y_1}, ..., f_{Y_m}\}$ and, for every $1 \leq i \leq m$, f_{Y_i} is the identity function over $Dom(Y_i)$.*

PROOF

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

RULE 2**:** *Let $S(X_1, ..., X_n)$ and $T(Y_1, ..., Y_n)$ be two relation schemas, such that, $Dom(X) = Dom(Y)$ and let s be a relation instance of $S(X)$. Then, the term $\rho_{X_1,...,X_n \to Y_1,...,Y_n}(s)$ 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 identity mapper function over $Dom(Y_i)$.*

PROOF

$$\begin{aligned}
&\rho_{X_1,...,X_n \to Y_1,...,Y_n}(s) \\
&= \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } u[X_i] = t[Y_i], \forall 1 \leq i \leq n\} \\
&= \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } u[X_i] \in \{t[Y_i]\}, \forall 1 \leq i \leq n\} \\
&\text{because } f_{Y_i} \text{ is the identity on } Dom(Y_i) \\
&= \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } u[X_i] \in f_{Y_i}(t), \forall 1 \leq i \leq n\} \\
&= \mu_{f_{Y_1},...,f_{Y_n}}(s)
\end{aligned}$$

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 and $s(X)$ a relation. There exists a set F of proper mapper functions for transforming $S(X)$ into $S(X)$ s.t. 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 be $\{f_{X_1}, ..., f_{X_n}\}$ where each mapper function f_{X_i} is defined by the function with signature $Dom(X) \to \mathcal{P}(Dom(X_i))$ 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,

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

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 in query rewriting aims at reducing I/O cost by transforming relational expressions into equivalent ones that, from an operational point of view, minimize the amount of information passed from operator to operator. Most algebraic rewriting rules for query optimization proposed in literature fall into one of the following two categories. Either they push the operators that reduce the cardinality of the source relations to be evaluated as early as possible (this is the case of the rules for *pushing selections*) or they avoid propagating attributes that are not used by subsequent operators (this the case of rules for *pushing projections*).

In the following, we adapt these classes of algebraic rewriting rules to the mapper operator.

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 set of multi-valued mapper functions, proper for transforming $S(X)$ into $T(Y)$. Consider a condition C_{A_i} dependent on a set of attributes A_i such that $f_{A_i} \in F$. Then,*

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

where $(\sigma_{C_{A_i}} \circ f_{A_i})(t) = \{f_{A_i}(t) \mid C_{A_i}(t)\}$.

PROOF

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

The benefits of Rule 4 can be better understood at the light of the alternative definition for the mapper semantics in terms of a Cartesian product presented in Section 3.1. Intuitively, 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, $1 \leq i \leq k$. Hence, by pushing the condition C_{A_i} to the mapper function f_{A_i}, the condition will be evaluated fewer times. This is especially 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., [Hellerstein, 1998] for details.

Furthermore, note that when $C_{A_i}(t)$ does not hold, the function $\sigma_{C_{A_i}}(f_{A_i})(t)$ 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[1]. Even in situations where neither expensive functions or 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.

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 with the mapper functions that compute them. Suppose that, in the selection condition C, 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 set of mapper functions proper for transforming $S(X)$ into $T(Y)$. The function resulting from the restriction of f_{A_i} to an attribute $Y_j \in A_i$ is denoted by $f_{A_i} \downarrow Y_j$. Moreover, given an attribute $Y_j \in Y$, $F \downarrow Y_j$ represents the function $f_{A_i} \downarrow Y_j$ s.t. $Y_j \in A_i$. Note that, because F is a proper set of mapper functions, the function $F \downarrow Y_j$ exists and is unique.

[1]The reader may have remarked that this optimization can be generalized to first evaluate those functions with higher probability of yielding an empty set. This issue is fundamentally the same as the problem of *optimal predicate ordering* addressed in [Hellerstein, 1998].

RULE 5**:** *Let $F = \{f_{A_1}, ..., f_{A_k}\}$ be a set of single-valued mapper functions, proper for transforming $S(X)$ into $T(Y)$. Let $B = B_1 \cdot ... \cdot B_k$ be a list of attributes in Y and s a relation instance of $S(X)$. Then,*

$$\sigma_{C_B}(\mu_F(s)) = \mu_F(\sigma_{C[B_1,...,B_k \leftarrow F\downarrow B_1,...,F\downarrow B_k]}(s))$$

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 $C[B_1, ..., B_n \leftarrow E_1, ..., E_n]$.

This rule replaces each attribute B_i in the condition C by the expression that describes how its values are obtained. Often, the attributes used in the condition of a selection are generated either by *(i)* identity mapper functions or *(ii)* constant mapper functions. Please refer to [Carreira et al., 2005] for the proof of this rule.

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 set of mapper functions, proper for transforming $S(X)$ into $T(Y)$, then an attribute Y_i of Y 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 suggest an optimization. Since they are not required for subsequent operations, the mapper functions that generate them do not need to be evaluated. Hence they can, in some situations, 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. The proofs of rules 6 and 7 are presented in [Carreira et al., 2005].

RULE 6**:** *Let $F = \{f_{A_1}, ..., f_{A_k}\}$ be a set of mapper functions, proper for transforming $S(X)$ into $T(Y)$. Let Z and Z' be lists of attributes in Y and let s be a relation instance of $S(X)$. Then, $\pi_Z(\mu_F(s)) = \pi_Z(\mu_{F'}(s))$, where $F' = \{f_{A_i} \in F \mid A_i$ contains at least one attribute in $Z\}$.*

EXAMPLE 4.1**:** *Consider the mapper $\mu_{acct,amt}$ defined in Example 3.1. The expression $\pi_{\mathsf{AMOUNT}}(\mu_{acct,amt}(\mathsf{LOANS}))$ is equivalent to $\pi_{\mathsf{AMOUNT}}(\mu_{amt}(\mathsf{LOANS}))$. The $acct$ mapper function is forgotten because the $\mathsf{ACCOUNT}$ attribute was projected-away. Conversely, neither of the mapper functions can be forgotten in the expression $\pi_{\mathsf{ACCTNO,\ SEQNO}}(\mu_{acct,amt}(\mathsf{LOANS}))$.*

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).

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 set of mapper functions, proper for transforming $S(X)$ into $T(Y)$. Let s be a relation instance of $S(X)$. Then, $\mu_F(s) = \mu_F(\pi_N(s))$, 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.*

RULE 8**:** *Let $F = \{f_{A_1}, ..., f_{A_k}\}$ be a set of mapper functions proper for transforming $SR(X, Y)$ into $T(Z)$. Let s and r be relation instances with schemas $S(X)$ and $R(Y)$ respectively. If there exist Z_R and Z_S, such that, $Z_R \cdot Z_S = Z$, and two disjoint subsets $F_R \subseteq F$ and $F_S \subseteq F$ of mapper functions, proper for transforming, respectively, $S(X)$ into $T_R(Z_R)$ and $R(Y)$ into $T_S(Z_S)$ then $\mu_F(s \times r) = \mu_{F_S}(s) \times \mu_{F_R}(r)$.*

5. Related work

Over the years, several extensions of RA, (like e.g., aggregates [Klug, 1982] for data consolidation, or controlled recursion for solving problems like the classical *bills-of-material* [Melton and Simon, 2002]) have been introduced, enhancing the expressive power of RA to support new applications.

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 [Shu et al., 1975] and Express [Shu et al., 1977], both aiming at data conversion.

More recently, three efforts, Potter's Wheel [Raman and Hellerstein, 2001], Ajax [Galhardas et al., 2001] and Data Fusion [Carreira and Galhardas, 2004], 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 implements the semantics of the mapper operator as it is presented here. However, the current version of Data Fusion is not supported by an extended relational algebra as we propose.

Solutions for restructuring semi-structured data [Suciu, 1998] like WOL [Davidson and Kosky, 1997], YAT [Cluet and Siméon, 1997], and TransScm [Milo and Zhoar, 1998] 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.

Clio [Miller et al., 2001] 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 [Fagin et al., 2003] 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.

6. Conclusions and future work

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 powerful than standard RA. Interesting properties of mappers were described. We showed that mappers admit a tuple-a-time semantics, which indicates that non-blocking physical execution algorithms for this operator can be implemented. We also showed that mappers 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 relational queries extended with mappers were proposed together with their corresponding proofs of correctness.

We strongly believe that current relational database technology enhanced with the mapper operator will provide a powerful data transformation engine. We have been developing and experimenting different physical execution algorithms for the mapper operator. We aim at providing both logical and physical optimization strategies to the query optimizer specially tailored for data transformations. We plan to incorporate this technology in the newer versions of the Ajax data cleaning tool and in Data Fusion, a legacy-data migration tool that has been used commercially in two large-scale data migration projects.

References

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

Carreira, P. and Galhardas, H. (2004). Efficient development of data migration transformations. In *ACM SIGMOD Int'l Conf. on the Managment of Data*, Paris, France.

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

Chaudhuri, S. (1998). 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.

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

Cluet, S. and Siméon, J. (1997). Data integration based on data conversion and restructuring. Extended version of [Cluet et al., 1998].

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

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

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

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

Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Computing Surveys*, 2(25).

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

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

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

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

Lomet, D. and Rundensteiner, E. A., editors (1999). *Special Issue on Data Transformations*. IEEE Data Engineering Bulletin.

Melton, J. and Simon, A. R. (2002). *SQL:1999 Understanding Relational Language Components*. Morgan Kaufmann Publishers, Inc.

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

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

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

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

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

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

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

Suciu, D. (1998). An overview of semistructured data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29(4):28–38.