# Execution of Data Mappers

Paulo Carreira
Oblog Consulting and FCUL
Rua da Barruncheira, 4
2795-477 Carnaxide, PORTUGAL
paulo.carreira@oblog.pt

Helena Galhardas
INESC-ID and IST
IST Taguspark, Av. Prof. Cavaco Silva
2780-990 Porto Salvo, PORTUGAL
hig@inesc-id.pt

## ABSTRACT

Data mappers are essential operators for implementing data transformations supporting schema mapping and integration scenarios such as *legacy data migration*, ETL processes for *data warehousing*, *data cleaning* activities, and *business integration* initiatives. Despite their widespread use, no formalization of this important operation has been proposed so far. In this paper we propose the data mapper operator as an extension to the relational algebra. We supply a set of algebraic rewriting rules for optimizing queries that combine standard relational operators with data mappers. Finally, we propose algorithms for their efficient physical execution.

## 1. INTRODUCTION

Today's business pace is bringing initiatives such as *business integration*, ETL processes for *data warehousing*, *data cleaning* activities, and *legacy data migration* into the center of concerns of a growing number of companies. When putting in place such initiatives, we often need to transform data from a source schema into a target schema. As a result, we often encounter two well-known classes of problems, *schema heterogeneities* and *data heterogeneities*. Schema heterogeneities refer to differences between schema elements representing the same data. This kind of heterogeneities occur because distinct schema modeling formalisms (e.g. Entity-Relationship (ER) *vs* Hierarchical Modeling) exist, different physical representations of the same logical concept (e.g. a logical entity *client* represented as one physical table *vs* many different tables are possible), or in the case of ER models, different degrees of normalization are required for optimization purposes. Data heterogeneities refer to different representations of the same data. These heterogeneities are caused by the following reasons: *(i)* different units of measurement, *(ii)* different abstraction levels (e.g. hourly *vs* daily), *(iii)* composition of data as attributes (e.g. a date attribute may represent day, month and year information), distinct representations of the same data domain (e.g. {true, false} *vs* {yes, no} for boolean values) or different data formats.

*Schema integration* and *schema transformation* techniques are used to solve the problem of schematic heterogeneities. *Data integration* and *data transformation* techniques are applied for handling data heterogeneities.

A natural form of performing schema and data transformations is to execute a sequence of SQL queries against source data. These queries can be executed either by the DBMSs if data resides in a database, or by some query engine able to connect to the source data (e.g. ASCII or binary files).

Query languages, in general, were not developed to represent complex transformations for solving schema and data heterogeneities. In particular, SQL cannot express many of the data and schema transformations necessary for transforming data among relational databases [7]. Nevertheless, due to the efficiency of the approach, solutions for handling complex data transformations like Clio [14, 9] and WOL [4] generate SQL queries. Since many interesting data and schema transformations cannot be expressed by SQL queries, collections of ad-hoc programs are developed to complement them. As pointed out by [8], this approach is not practical mainly because transformation programs cannot be easily optimized that way. Furthermore, managing and debugging data and schema transformations specified with two different formalisms brings an extra cost. These problems become acute as the number of transformations grows. Intuitively, the more expressive the query language is, the less ad-hoc programs need to be developed.

An important operator for schema and data transformation is the *data mapper* operator. Informally, a data mapper applies to an input relation and produces an output relation. It iterates over the input tuples, applies a set of specific-domain functions and generates one or more output tuples. This kind of operation appears implicitly in most languages aiming at implementing schema and data transformations, but has never been properly handled as a *first-class* operator, to the best of our knowledge. The mapper operation as it is presented here has been implemented in the Data Fusion tool [3].

This paper proposes to extend the relational algebra with the data mapper operator that significantly increases its expressive power by enabling, for instance, the dynamic creation of tuples based on a source tuple contents. Furthermore, expressions that combine the mapper operator with standard relational algebra operators can be optimized. Our contributions are the following:

- A formalization of the mapper operator;


*IQIS* 2004 Maison de la Chimie, Paris, France

Relation LOANS

| ACCT | AMT |
|---|---|
| 12 | 20,00 |
| 3456 | 140,00 |
| 901 | 250,00 |

Relation PAYMENTS

| ACCTNO | AMOUNT |
|---|---|
| 0012 | 20,00 |
| 3456 | 100,00 |
| 3456 | 40,00 |
| 0901 | 100,00 |
| 0901 | 100,00 |
| 0901 | 50,00 |

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

Relation CLIDATA

| CACCT | NAME | GDR | BTHDATE |
|---|---|---|---|
| 12 | Assato, Anna | F | 19750304 |
| 3890 | Lawrence, Louis | M | 19840203 |
| 4567 | Springs, Alice | F | 19830507 |
| 5923 | Smith, John | M | 19670809 |
| 0901 | Wilson, Peter | M | 19450506 |
| 3453 | York, Susan | F | 19760203 |

Relation CLIMKT

| CID | RANK |
|---|---|
| 3890 | C |
| 4567 | A |
| 5923 | D |
| 0901 | D |
| 3453 | B |

**Figure 2:** *(a)* **On the left, the CLIDATA relation and,** *(b)* **on the right, the relation CLIMKT.**

- A set of query rewriting rules for queries involving the mapper operator;
- A highlight of physical execution algorithms for the mapper operator.

Our paper is organized as follows: Section 2 gives the motivation for the use of the mapper operator. Some preliminary definitions are provided in Section 3. The formalization of the mapper is given in Section 4. Section 5 presents the algebraic rewriting rules for optimizing the mapper operator. A discussion of execution algorithms for the mapper operator is given in Section 6. Conclusions and plans for future work are presented in Section 7.

## 2. MOTIVATION

This section motivates the reader for the use of the mapper operator. We present simple examples based on a real-world data-migration scenario that has been intentionally simplified for illustration purposes.

EXAMPLE 2.1: Consider the source relation LOANS[ACCT, AMT] (represented in Figure 1) that stores the details of loans requested per account. Suppose LOANS data must be transformed into the target relation PAYMENTS[ACCTNO, AMOUNT] 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 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 AMT into multiple parcels with a maximum value of 100, in such a way that the sum of amounts for the same ACCTNO is equal to the source amount for the same account.

EXAMPLE 2.2: Consider the source relation CLIDATA[CACCT, NAME, GDR, BTHDATE] that contains information about clients (see Figure 2). This relation must be transformed into the relation CLIMKT[CID, RANK] to support the marketing features of a new client management system. The account numbers of the relation LOANS in Example 2.1, are the client accounts held by attribute CACCT of the relation CLIDATA. The transformation rules must implement the following requirements:

1. The target attribute CID is directly mapped from the source attribute CACCT.
2. The attribute RANK holds marketing rank information for each client. The value of this attribute is computed from the source attributes GDR and BDATE as follows. A female client is ranked A if it is less than 25 years old, else it is ranked B. A male client less than 25 years old is ranked C and D otherwise.
3. All clients with loans smaller than 100 units are not enrolled in the marketing application.

The implementation of data transformations similar to those requested for producing the target relations PAYMENTS and CLIMKT is challenging. Solutions to the problem involve the dynamic creation of tuples based on the value of attribute AMT. The second problem consists of composing relational algebra operators involving encoding complex decisions.

Traditional approaches for implementing the required transformations either consist of *(i)* a combination of SQL queries with *ad-hoc* transformation programs written in a general purpose language, *(ii)* writing a set of *Persistent Stored Modules* (PSMs) to be executed by the DBMS or *(iii)* using an ETL tool. Each of these approaches poses a number of drawbacks.

The use of a general purpose language is hindered by two factors. First, these languages have a procedural nature as opposed to the declarative nature of query languages. This characteristic turns data transformations difficult to understand and maintain. Second, apart from some static optimizations, transformation programs cannot be optimized. The use of PSMs has two disadvantages. First, PSM programs have a number of procedural constructs that are not amenable to optimization. Moreover, there are no elegant solutions for expressing dynamic creation of instances using PSMs. One needs to resort to intricate LOOP and INSERT INTO statements.

Using an ETL tool does not solve the problem either. To the best of our knowledge, no ETL tool provides native support for all the functionalities of the mapper operator we propose. Some ETL tools (e.g Sagent [11] and Datastage [1]), provide an extensive library of predefined functions. However, none of them allow expressing dynamic creation of records. This involves either *(i)* writing complex proprietary language scripts or *(ii)* coding external functions. Furthermore, they impose a clear separation between relational queries and data transformations, thus disabling any optimizations of sequences of data transformation operators and relational operators.

These three approaches for implementing the data mapper functionalities are commonly used in real world projects.

In research, two efforts, Potter's Wheel [10] and Ajax [5], have proposed operators for data transformation and cleaning purposes that are very similar to the data mapper. Potter's Wheel format operator also applies a function to each tuple of the source relation. However, Potter's Wheel generates exactly one output tuple for each input tuple, whereas our mapper operator can generate many. Ajax map operator introduces the idea of exceptions not handeled by our mapper operator. Ajax map operator also allows to specify a selection condition applied to each input tuple. In our case, we rely on composing the mapper operator with the relational *selection* operator.

# 3. PRELIMINARIES

A domain $D$ is a set of constants. The set $\mathcal{D}$ is a finite set of domains. The *cartesian product* (or simply *product*) of domains $D_1, ..., D_n$, written $D_1 \times ... \times D_n$, is the set of all elements of the form $(c_1, ..., c_n)$ where each $c_i$ is a constant of $D_i$. Let $S$ be a set. The set formed by all possible subsets of $S$, is denoted by $\mathcal{P}(S)$.

A relation $R$ is a finite subset of the product of one or more domains. Given a relation $R$, the number of domains that participate in the relation $R$ is called the *arity* of $R$. If $R \subseteq D_1 \times ... \times D_n$, then $arity(R) = n$. Each element of a relation is called a *tuple*. A tuple $(c_1, ..., c_n)$ of a relation $R$ is called an *R-tuple*. Each constant $c_i$ is said to be a *component* of the tuple. The name of a component is called an *attribute*.

We assume a set $\mathcal{A}$ of attributes. Attributes are associated to domains through a mapping $Dom : \mathcal{A} \rightarrow \mathcal{D}$ such that, given an attribute $A \in \mathcal{A}$, $D_A$ is a domain $D \in \mathcal{D}$ called *domain of A*. A set of attributes is called a *relation schema*. Given a relation $R \subseteq D_1 \times ... \times D_n$ and its relation schema $S = \{A_1, ..., A_n\}$ we write $R[A_1, ..., A_n]$ to indicate that the relation schema of $R$ is $S$. A subset $K \subseteq S$ of attributes that uniquely designate each tuple of $R$ is called a *key* of $R$.

We assume the usual relational algebra operators (see [13] or [6] for a detailed introduction): *Union*; *intersection*; *difference* and *cartesian product*; as well as *extended projection* represented as $\pi_L$, where $L$ is a set containing *(i)* attributes and *(ii)* expressions; *selection* represented as $\sigma_C$, where $C$ is the selection condition; *grouping* represented as $\gamma_{G,L}$, where $G$ is the set of *grouping attributes* and $L$ is the set of *aggregate functions*. Attributes to which aggregate functions are applied are called *aggregate attributes*.

# 4. MAPPER OPERATOR

A mapper, is a unary operator $\mu_F$ that takes one relation $R[X_1, ..., X_n]$ as input and produces one relation $S[A_1, ..., A_k]$ as output. This operator is subscripted by a set $F$ of functions, designated as *mapper functions*. Each mapper function is subscripted with an attribute of $S$. We first introduce the notion of mapper function in Definition 4.1.

DEFINITION 4.1: Given an relation $R[X_1, ..., X_n]$ and an attribute $A$, such that $A$ is not an attribute of R, a *mapper function* from $R$-tuples into $A$ is defined as a function $f_A$:$D_{X_1} \times ... \times D_{X_n} \rightarrow \mathcal{P}(D_A)$. The set $D_{X_1} \times ... \times D_{X_n}$ is the *domain of the mapper function* $f_A$ and is represented by $D_{f_A}$. □

When applied to a tuple, a mapper function produces a set of values. Given an $R$-tuple $t$, $f_A(t)$ is a set of values contained in $D_A$.

Mapper functions can be classified according to the following criteria: A function that returns a single value can be seen as a particular case of a mapper function that returns a singleton set. A mapper function is called a *single-valued function* iff it produces a singleton as output for any tuple of the domain, i.e.,

$$\forall(t \in D_{f_A}) \mid f_A(t) \mid = 1$$

A mapper function whose outputs are not singletons is said to be *multi-valued*.

Consider a relation $R[X_1, ..., X_n]$. A mapper function $f_A(X_1, ..., X_n) = \{c\}$ that assigns a constant to an attribute is called a *constant assignment* of $c$ to $A$. A mapper function $f_A(X_1, ..., X_n) = \{X_i\}$ that returns the value of an attribute of the mapper input relation is called a *renaming* of $X_i$ to $A$. The *identity mapper function* $id_{R,X_i}$ is defined as a function $f_{X_i}(X_1, ..., X_n) = \{X_i\}$ that projects the attribute $X_i$. Whenever $R$ is understood from the context we write the identity mapper function simply as $id_{X_i}$.

For simplicity we often write mapper functions as subscripts of the mapper operator. For example, given a relation $R[X, Y, Z]$ we write $\mu_{X+Y \rightarrow A}(R)$ as a shorthand for $\mu_{f_A}(R)$, where $f_A(X, Y, Z) = \{X + Y\}$. Having presented the concepts underlying mapper functions, we are now able to define precisely the mapper operator.

DEFINITION 4.2: Let $F = \{f_{A_1}, ..., f_{A_n}\}$ be mapper functions and $R[X_1, ..., X_n]$ be a relation. Then, a *mapper* $\mu_F$ over $R$ represents a relation $S[A_1, ..., A_k]$ that contains all $k$-tuples obtained by combining, through a cartesian product, the values produced by each mapper function $f_{A_i}(t)$ applied to all tuples $t$ of relation $R$. Formally,

$$\mu_F(R) \stackrel{\text{def}}{=} \bigcup_{t \in R} f_{A_1}(t) \times ... \times f_{A_k}(t).$$

The *arity of a mapper* is given by the number of mapper functions in $F$. The relation $R$ is the *input* (or *source*) relation of the mapper, and the relation $S$ is the *output* ( or *target*) relation of the mapper. □

The target relation contains the union of all generated tuples for each tuple in the input relation. We note that the cardinality of the resulting relation can be greater than the cardinality of the input relation.

We are assuming that, no functional dependencies exist on the attributes of the target relation. Extending the mapper operator for specifying transformations that produce relations with functionally dependent attributes is work in progress.

EXAMPLE 4.1: The requirements presented in Example 2.1, can be implemented by the mapper $\mu_{acct,amt}$. The mapper function $acct$:$D_{\texttt{ACCT}} \times D_{\texttt{AMT}} \rightarrow$ACCTNO returns the account number ACCT properly left padded with zeroes. The mapper function $amt$ is defined as $amt$:$D_{\texttt{ACCT}} \times D_{\texttt{AMT}} \rightarrow$AMOUNT, and returns a set of parcels not greater than 100 and whose sum is AMT. Concretely, if $t$ is the tuple $t =$ (901, 250,00), the result of applying $amt(t)$ is the set $\{100, 100, 50\}$. Given tuple $t$, the corresponding transformed tuples are obtained from $acct(t) \times amt(t)$, i.e., $\{\texttt{'0901'}\} \times \{100, 100, 50\}$. A possible implementation for the $amt$ function previously described in Example 2.1 can be found in [2]. □

EXAMPLE 4.2: The first two requirements specified in Example 2.2 can be implemented by the mapper $\mu_{cid,rank}$. The mapper function *cid* returns the value of the attribute CACCT unmodified. The mapper function *rank* encapsulates: *(i)* the computation of the client age based on the attribute BTHDATE and *(ii)* the appropriate decisions to rank the client according to his age and gender. The last requirement involves a *join* operator with the LOANS relation to obtain the loan amount for this client, and a *selection* operator to verify that the amount is not smaller than 100. □

Definition 4.3 introduces to notion of monotonic mapper that will be useful later in Section 5.7.

DEFINITION 4.3: A mapper $\mu_F$ is *monotonic* iff given two relations $T$ and $U$ with the same schema, $T \subseteq U$ implies $\mu_F(T) \subseteq \mu_F(U)$.

THEOREM 4.1: All mappers operators are monotonic.

PROOF. We want to show that, $T \subseteq U$ implies $\mu_F(T) \subseteq \mu_F(U)$. Given deterministic functions $F$, by the definition of mapper (Definition 4.2), it follows that $\forall(t \in T),\ \mu_F(\{t\}) \subseteq \mu_F(T)$. By hypothesis $t \in U$, and thus, $\forall(t \in T), \mu_F(\{t\}) \subseteq \mu_F(U)$. Hence, $\mu_F(T) \subseteq \mu_F(U)$. □

# 5. ALGEBRAIC RULES

This section presents a set of algebraic rewriting rules that enables algebraic optimizations of relational expressions extended with the mapper operator.

In general, algebraic rewriting rules aim at reducing I/O cost. A relational expression is transformed into an equivalent one that minimizes the amount of information passed from operator to operator. A well-known strategy consists of evaluating the operators that reduce the cardinality of the source relations as early as possible. This operation is achieved by *pushing selections* and *pushing grouping*. Another strategy consists of avoiding to propagate attributes that are not used by subsequent operators, known as *pushing projections*. On a distributed query processing environment we are also interested in executing the operators the *nearest* as possible of the data site to minimize transmission costs. In what concerns mappers, *mapper composition* and *mapper decomposition* are two classes of important rewriting rules useful for a distributed environment. Composition of mappers, allows mappers to be composed together and executed as a single mapper at a site. The decomposition of mappers allows a mapper to be splitted into composeable mappers that can be executed at distinct sites.

## 5.1 Pushing selections

In order to push a selection through a mapper we need to rewrite the selection condition. For example, consider the expression $\sigma_{Y>2}(\mu_{X^2\to Y}(R))$. The selection condition is expressed in terms of attributes of the mapper output relation. In order to push the selection condition $\sigma_{Y>2}$ over the mapper $\mu_{X^2\to Y}$, we need to rewrite it according to the attributes of the mapper input relation. In this simple case we obtain the equivalent expression $\mu_{X^2\to Y}(\sigma_{X<-\sqrt{2}\ \vee X>\sqrt{2}}(R))$. In general, to perform this kind of rewriting is an undecidable problem. However, by analyzing the problem closely, we can identify particular cases that make it possible to push selections through mappers. If a selection $\sigma_C$ is *independent* from the mapper then it can be safely pushed through the mapper.

RULE 5.1: Given a mapper $\mu_F$, and a selection $\sigma_C$, if $C$ does not contain an attribute generated by any mapper function in $F$, the following equivalence can be established:

$$\sigma_C(\mu_F(R)) = \mu_F(\sigma_C(R))$$

□

Often, mapper functions are renamings. Consider the renaming function $\rho$, induced by a mapper $\mu_F$. Technically, renamings can be identified by the static analysis of the mapper functions. For example, by renaming the attributes of the condition $A < B$ we may rewrite the expression $\sigma_{A<B}(\mu_{X\to A,Y\to B}(R))$ as $\mu_{X\to A,Y\to B}(\sigma_{X<Y}(R))$.

RULE 5.2: Given a mapper $\mu_F$ and a selection $\sigma_C$, in which all attributes of $C$ are mapped by renaming functions of $F$. Let $\rho(C)$ be a condition equivalent to $C$ expressed with the attributes of the mapper input relation. The following equivalence is obtained:

$$\sigma_C(\mu_F(R)) = \mu_F(\sigma_{\rho(C)}(R))$$

□

Another important observation is that many mapper functions are constant assignments of some constant $c$ to an attribute $A$. In these cases, we can replace the attributes of the condition $C$ with the constants of the constant mapper functions and produce an equivalent condition. This condition: *(i)* is faster to evaluate [1], and *(ii)* may reduce drastically the number of input tuples. For example, in the expression $\sigma_{A<B}(\mu_{X\to A,10\to B}(R))$ attribute $B$ replaced by the constant 10 to obtain the equivalent expression $\mu_{X\to A,10\to B}(\sigma_{X<10}(R))$.

RULE 5.3: Let $C[A \mapsto c]$ represent the new condition where the attribute $A$ is replaced by the constant $c$. Given a mapper $\mu_F$ containing a mapper function $f_A$, that assigns a constant $c$ to the attribute $A$, we can write the equivalence:

$$\sigma_C(\mu_F(R)) = \mu_F(\sigma_{C[A\mapsto c]}(R))$$

□

Rule 5.3 can be further generalized. Suppose that a selection condition $C$ maintains an attribute $A$ that is produced by a mapper function $f_A$. By replacing the attribute $A$ with the mapper function $f_A$ in condition $C$ we obtain an equivalent condition. Consider the expression $\sigma_{A<10}(\mu_{X+Y\to A}(R))$. Attribute $A$ can be expanded with the corresponding mapper function to obtain $\mu_{X+Y\to A}(\sigma_{X+Y<10}(R))$

RULE 5.4: Given a mapper $\mu_F$, where $F$ contains a single-valued mapper function $f_A$, we can write:

$$\sigma_C(\mu_F(R)) = \mu_F(\sigma_{C[A\mapsto f_A]}(R))$$

□

[1] Comparing with constants is, in principle, faster than comparing with memory locations holding values of attributes.

This rewriting rule adds an extra computation cost by evaluating the mapper function twice, once in the selection condition and another in the mapper. However, this rule plays an important role for optimizing mappers with many computation intensive mapper functions. Consider the case of a selection condition $C$ involving a mapper function $f_A$ and a mapper with many attributes generated by expensive mapper functions. Depending on the number of tuples filtered by the condition $C$, the cost of evaluating $f_A$ twice is neglectable when compared with the cost of producing a tuple (that will ultimately be discarded) evaluating all the mapper functions.

## 5.2 Pushing the grouping operator

The grouping operator $\gamma_{G,L}$ can be pushed through a mapper $\mu_F$ resulting in potentially huge reductions on the number of tuples passed to the mapper. Consider the expression $\gamma_{B,\texttt{COUNT(A)}}(\mu_{X\to A,Y\to B,f_C}(R))$, where $f_C$ is a potentially expensive mapper function. The grouping operator in this expression can only be computed after mapping all tuples of $R$. In contrast, the equivalent expression $\mu_{X\to A,Y\to B}(\gamma_{Y,\texttt{COUNT(X)}}(R))$, is more efficient to compute. The mapper is evaluated only once for each distinct value of $Y$.

Pushing the grouping operator over the mapper involves: *(i)* pushing the aggregate functions and *(ii)* pushing the grouping attributes. Intuitively, if the mapper functions correspond to renamings, the grouping operator can be safely pushed through the mapper by applying the renamings induced by the mapper functions. Rule 5.5 captures this idea.

RULE 5.5: Let $\gamma_{G,L}$ be a grouping operator and $\mu_{F\cup H}$ be a mapper such that, *(i)* all input attributes of the aggregate functions in $L$ and *(ii)* grouping attributes in $G$, are generated by renaming functions in $F$. Furthermore, consider $H$ to contain those mapping functions that do not produce aggregate or grouping attributes. Then,

$$\gamma_{G,L}(\mu_{F\cup H}(R)) = \mu_{H\cup id_G}(\gamma_{\rho(G),\rho(L)}(R))$$

□

Considering the *distributive* properties[2] of the SQL built-in aggregate functions `COUNT`, `SUM`, `AVG`, `MIN` and `MAX`, we can push grouping operators through mappers that map the attributes of aggregate functions through arithmetic expressions. . For example, we can write $\texttt{SUM}(2 + .25 \times A)$ as $k \times 2 + .25 \times \texttt{SUM}(A)$, where $k$ is the count of tuples (obtained, for example, as $k = \texttt{COUNT(A)}$).

These properties can be used to distribute aggregate functions over mapper functions. They enable a number of useful optimizations since many mapping functions consist of arithmetic expressions (for example to specify transformations of units of measurement).

EXAMPLE 5.1: Consider the following expression, that computes the sum of salaries by employee categories:

$$\gamma_{\texttt{CATEG},\texttt{SUM(SALARY)}}(\mu_{\texttt{ECAT}\to\texttt{CATEG},\texttt{SAL}\times.75+\texttt{BONUS}\to\texttt{SALARY}}(R)).$$

The expression can be re-written introducing two new attributes $X$ and $Y$:

$$\mu_{\texttt{ECAT}\to\texttt{CATEG},X\times.75+Y\to\texttt{SALARY}}(\gamma_{\texttt{ECAT},\texttt{SUM(SAL)}\to X,\texttt{SUM(BONUS)}\to Y}(R)).$$

[2] A detailed explanation of the distributive properties of the SQL built-in functions is beyond the scope of this paper.

□

RULE 5.6: Let $F_L$ be the set of mapper functions $f_A^L$ that result from distributing the aggregate functions of $L$ over each $f_A \in F_A$. Let $L'$ be the set of aggregate functions needed for computing the functions of the form $f_A^L$. Then,

$$\gamma_{G,L}(\mu_F(R)) = \mu_{F_L}(\gamma_{\rho(G),L'}(R))$$

□

Aggregate functions can apply to expressions as well as to attributes in some dialects of SQL. Furthermore, grouping attributes can also be expressions. These two generalizations lead to a straightforward extension of the grouping operator, henceforth denoted by $\widehat{\gamma}$. For example writing $\widehat{\gamma}_{3+Y,SUM(A/B)}(R)$ is legal. This extension allows further optimizations, since it enables pushing aggregate functions over mapping functions when the distribution of aggregate functions over a mapper function is not possible. Rule 5.7 formalizes this new equivalence.

RULE 5.7: Let $H_L$ be the set of all mapper functions $h_A^L$. Each $h_A^L$ results form replacing, by a new attribute, the expressions of each function $f_A \in F_A$, that do not admit distribution of an aggregate function. Let $L'$ be the set of aggregate functions needed for computing $H_L$, and further, let $H'$ be the set of grouping expressions needed for computing $H_L$. Then,

$$\gamma_{G,L}(\mu_F(R)) = \mu_{H_L}(\widehat{\gamma}_{G',L'}(R))$$

□

EXAMPLE 5.2: Consider the mapper functions for the attributes $X$ and $Y$ in $\gamma_{Y,SUM(X)}(\mu_{2\times C\to Y,2\times(A/B)\to X}(R))$. Using equivalency rules (5.5) and (5.6), not much can be done. However, by introducing attributes $V$ and $W$, the expression is equivalent to

$$\mu_{W\to Y,2\times V\to X}(\widehat{\gamma}_{2\times C\to W,SUM(A/B)\to V}(R))$$

□

## 5.3 Pushing projections

A projection applied to a mapper is an expression of the form $\pi_L(\mu_F(R))$. The attributes produced by the mapper functions may not be used by the projection. These attributes are said to be *projected out*. Mapper attributes that are projected out suggest an optimization. More concretely, projected-out mapper attributes whose corresponding mapper functions are single-valued, can be dropped from the mapper. Rule 5.8 formalizes this notion.

RULE 5.8: Given a mapper $\mu_F$ and a list of attributes that have been projected-out. Let $F' \subseteq F$ contain mapper functions $f_A$ such that either: *(i)* $A$ is an attribute used in the projection $\pi_L$ or *(ii)* the mapper function $f_A$ is multi-valued. Then,

$$\pi_L(\mu_F(R)) = \pi_L(\mu_{F'}(R))$$

□

EXAMPLE 5.3: The expression $\pi_{\texttt{AMOUNT}}(\mu_{acct,amt}(\texttt{LOANS}))$ is equivalent to the expression $\pi_{\texttt{AMOUNT}}(\mu_{amt}(\texttt{LOANS}))$. The *acct*

mapper function is dropped because the ACCOUNT attribute was projected out and the *acc* function is single-valued. However, in the expression $\pi_{\text{ACCTNO}}(\mu_{acct,amt}(\text{LOANS}))$, the *amt* mapper function cannot be dropped because, according to Example 2.1, this function may produce multiple output values.

Attributes that are input of any mapper function need not be retrieved from the mapper input relation $R$.

EXAMPLE 5.4: Since ACCT is not an input attribute of the mapper function *amt*, the expression $\mu_{amt}(\text{LOANS})$ is equivalent to $\mu_{amt}(\pi_{\text{AMT}}(\text{LOANS}))$. □

RULE 5.9: Given a mapper $\mu_F$ applied to a relation $R$. Let $\pi_N$ be the projection that projects only the attributes required for computing any mapper function of $F$. Then,

$$\mu_F(R) = \mu_F(\pi_N(R))$$

□

Finally, Rule 5.10 generalizes rules (5.8) and (5.9) for pushing projections.

RULE 5.10: Let $\mu_F$ be a mapper and $\pi_L$ a projection over the attributes of the mapper output relation. Let $F'$ be a set of mapper functions with the properties stated in Rule 5.8. Let $N$ be the set of attributes needed for computing the functions in $F'$. Then,

$$\pi_L(\mu_F(R)) = \pi_L(\mu_{F'}(\pi_N(R)))$$

□

## 5.4 Composing mappers

Mappers represent data transformations, so we expect them to be composeable. Intuitively, mapper composition is obtained by composing the underlying mapper functions. Given mapper functions $f$ and $g$, $\mu_f(\mu_g(R))$ should be the same as $\mu_{f \circ g}(R)$. However, since mapper functions may return sets of values, writing $f \circ g$ constitutes an abuse of notation. Thus, a more complete definition for composing mapper functions is required.

DEFINITION 5.1: Let $A$ and $X_1, ..., X_n$ be attributes with corresponding domains $D_A$ and $D_{X_1}, ..., D_{X_n}$. Let $f_A$ be a mapper function such that $D_{f_A} \subseteq D_{X_1} \times ... \times D_{X_n}$, and $g_{X_1}, ..., g_{X_n}$ be mapper functions where for each $g_{X_i}$, $D_{g_{X_i}} \subseteq D_{A_1} \times ... \times D_{A_k}$. Consider $A_1, ..., A_k$ to be attributes of a mapper input relation $R[A_1, ..., A_k]$. We define *composition of $f_A$ with $g_{X_1}, ..., g_{X_n}$*, written

$$f_A(g_{X_1}, ..., g_{X_n})(R)$$

as the set of values

$$\{f_A(t') \mid t' \in g_{X_1}(t) \times ... \times g_{X_n}(t), t \in R\}$$

□

Based on Definition 5.1 we can now define the composition two mappers. This rewriting rule has great practical interest since it enables to save pipelining stages in query execution plans.

RULE 5.11: Given two mappers $\mu_F$ and $\mu_G$, the expression $\mu_F \circ \mu_G$ represents the composition of $\mu_F$ with $\mu_G$ which is itself a mapper. Given sets of mapper functions $F = \{f_1, ..., f_k\}$ and $G = \{g_1, ..., g_n\}$, then

$$(\mu_F \circ \mu_G)(R) = \mu_{f_1(g_1,...,g_n),...,f_k(g_1,...,g_n)}(R)$$

□

## 5.5 Promoting projections to mappers

There cases where the query optimizer has to handle sequences of projections and mappers, for which no more rewriting apply. In these cases, one more optimization is possible promoting the projections to mappers. An expression used in a projection can be seen as a mapper function. Thus, every projection operator $\pi_L$ can be emulated by a mapper $\mu_L$. Executing a single mapper is usually to be faster than performing the projection after the mapper.

RULE 5.12: Given a mapper $\mu_F$, and projection $\pi_L$. Let $\mu_L$ be the mapper that emulates the projection $\pi_L$. Then,

$$\pi_L(\mu_F(R)) = (\mu_L \circ \mu_F)(R)$$

□

## 5.6 Decomposing mappers

A major improvement in distributed environments, is to be able to execute query operators at different sites. For queries expressions complex mappers it is advantageous to decompose them. We can break complex mappers into simpler mappers and distribute its processing.

EXAMPLE 5.5: Consider a mapper $\mu_F$ to be processed at a site $s_1$ and a relation $R[U, V, W, X, Y, Z]$ stored at site $s_2$. Computing $\mu_F(R)$ implies to transfer tuples of $R$ from $s_2$ to $s_1$. Suppose that $\mu_F$ produces few attributes but uses many attributes of $R$ (e.g. $\mu_F = \mu_{U+V+W+X+Y \to A, 2 \times Z \to B}$). We can write the expression $\mu_F(R)$ as $\mu_G(\mu_H(R))$, where $\mu_H = \mu_{id_A, 2 \times Z \to B}$ and $\mu_G = \mu_{U+V+W+X+Y \to A, id_Z}$. As a result, the optimizer may schedule $\mu_H$ to run at $s_1$ and $\mu_G$ at $s_2$, thus transferring much smaller tuples from $s_2$ to $s_1$. □

DEFINITION 5.2: Given a mapper $\mu_F$, and two sets of mapper functions $G$ and $H$, such that each of the sets $G$ and $H$ contain at least one mapper function of $F$ and $G \cap H = \emptyset$. A *decomposition* of $\mu_F$, is a pair of mappers $\mu_G$ and $\mu_H$ such that, $\mu_F = \mu_G(\mu_H(R))$. A mapper $\mu_F$ is *decomposable* if it admits a decomposition. □

THEOREM 5.1: Every mapper with more than one mapper function is decomposable.

PROOF. Given a mapper $\mu_F$. It suffices to find a suitable partition for the set of mapper functions. Without loss of generality, consider $F = \{f_A, f_{X_1}, ..., f_{X_n}\}$. Furthermore, consider a partition of $F$ formed by two sets $F_1 = \{f_A\}$ and $F_2 = \{f_{X_1}, ..., f_{X_n}\}$. Now, let $\mu_G$ and $\mu_H$ be two mappers, where $G = F_1 \cup \{id_{X_i} \mid X_i \text{ is generated by } f_{X_i} \in F_2\}$ and $H = \{id_A\} \cup F_2$. Then,

$$\mu_G(\mu_H(R)) = \mu_{f_A, id_{X_1}, ..., id_{X_n}}(\mu_{id_A, f_{X_1}, ..., f_{X_n}}(R)) = \mu_F(R)$$

□

From Theorem 5.6 we can devise a systematic method for decomposing mappers. However, we need a criterion for establishing appropriate mapper function partitions. Let us consider a predicate $Cr$ to be such a criterion. For example, a criterion based on the number of source attributes could have been used for obtaining the appropriate partition used in Example 5.5. Criteria based on some measure of computational complexity can be used to partition the mapper functions such that most expensive mappers (in terms of computational cost) get processed on a site with highest computing power.

RULE 5.13: Given a mapper $\mu_F$ and a criterion defined by a predicate $Cr$. Consider the set $F = \{f_{X_1}, ..., f_{X_n}\}$ with more than one mapper function. Let

$$G = \{f_{X_i} \mid Cr(f_{X_i}),\ f_{X_i} \in F\}$$
$$\cup\ \{id_{X_i} \mid \neg Cr(f_{X_i}),\ f_{X_i} \in F\}$$

and

$$H = \{f_{X_j} \mid \neg Cr(f_{X_i}),\ f_{X_j} \in F\}$$
$$\cup\ \{id_{X_j} \mid Cr(f_{X_j}),\ f_{X_j} \in F\}$$

such that $\mu_F$ and $\mu_H$ form a decomposition of $\mu_F$. Let $\mathbf{Cr}(F)$ represent $G$ and $\overline{\mathbf{Cr}}(F)$ to represent $H$. Then, for a fixed $\mathbf{Cr}$ we have,

$$\mu_F = \mu_{\mathbf{Cr}(F)}(\mu_{\overline{\mathbf{Cr}}(F)}(R))$$

□

## 5.7 Mappers and other binary operators

Unary operators often enjoy interesting distribution laws over binary operators. In fact, since mappers are monotonic operators, a number of straightforward equivalences can be established.

RULE 5.14: Let $\mu_F$ be a mapper. Given relations $R$ and $S$, the following equivalencies hold:

$$\mu_F(R \cup S) = \mu_F(R) \cup \mu_F(S) \quad (1)$$
$$\mu_F(R \cap S) = \mu_F(R) \cap \mu_F(S) \quad (2)$$
$$\mu_F(R - S) = \mu_F(R) - \mu_F(S) \quad (3)$$

□

In general, the equivalence $\mu_F(R \times S) = \mu_F(R) \times \mu_F(S)$ does not hold. However, if the mapper is decomposable, we can partition partition the set of mapper functions $F$ into two disjoint subsets. Each of the subsets contains mapper functions that depend only on attributes of either $R$ or $S$.

RULE 5.15: Consider a mapper $\mu_F$. Let $F_R \subseteq F$ and $F_S \subseteq F$ be sets of mapper functions, such that each function $f_r \in F_R$ (respectively $f_s \in S$) has only source attributes of the relation $R$ (respectively, of relation $S$). We can write,

$$\mu_F(R \times S) = \mu_{F_R}(R) \times \mu_{F_S}(S)$$

□

# 6. PHYSICAL EXECUTION

The mapper operator admits a simple *tuple-at-a-time* semantics. This suggests an iterator-based algorithm for executing the mapper that consists of three steps. First, a tuple $t$ is fetched from the source. Second, the mapper functions are executed taking $t$ as source. Third, the result of each mapper function is combined to produce the output tuples of $t$. These three steps are repeated until all the source tuples are processed.

## 6.1 Mapper mixed evaluation

Many data sources often reside on an RDBMSs. DBMSs are often capable of processing extended projection operators that use expressions. Since many mapper functions can be represented as expressions, we can push these functions to be processed as projections. The idea is that, mappers be decomposed in order to avoid transferring very wide tuples over the network. Using the decomposition algebraic rewriting rules of Section 5.6, we can produce a mapper that is representable as a projection $\pi_L$. For this purpose we use a criterion $Cr$ such that

$$Cr(f_a) \equiv f_a \textit{ is equivalent to an expression of the projection statement on the source data.}$$

After obtaining the projection $\pi_L$ we perform a *push-down* to the DBMS where the source relation resides. Substantial savings of transmission of attribute data are obtained if the arity of the source relation is high but the arity of the mapper is low.

Performing a mapper push-down brings a further benefit. Part of the mapper is optimized by the DBMS where the source data resides.

## 6.2 Further optimizations

In the following we present a optimization strategies that can be used to optimize mapper executions.

### 6.2.1 *Cached computation*

The execution of mappers that involve expensive computations motivates the use of caching techniques. The idea is to assign a cache to every expensive function in order to lower its cost. Before computing the value of $f_A(t)$ the cache is looked up. If the value exists it is retrieved (instead of being computed), else the result of $f_A(t)$ is computed and stored in the cache.

Not all mapper functions are good candidates for caching. To be a candidate for caching, a mapper function must meet two criteria. First, the average expected cost of storage and lookup must not exceed the average cost of computation. Clearly, a constant mapper function is not a good candidate for caching. Second, the selectivity of the input attributes used to compute $f_A$ must be low enough so that, the probability of a cache miss also remains low.

### 6.2.2 *Mapper pre-computation*

Data mappers are often executed over large amounts of data that is not updated. For mappers that involve expensive computations or costly I/O we can take advantage of pre-computing the mapper result.

If we can guarantee that between two consecutive executions of a mapper $\mu_F$, a tuple with key $k$ has not been updated. Then *(i)* we can use the locally stored result, *(ii)*

otherwise, we download the tuple, compute the mapper result and store it in the pre-computation structure.

Consider a relation $R$ with a key $K$. We can hash the result of computing $\mu_F(t)$ using the key $k$ and store it in an auxiliary structure. As a consequence we need not to transfer all the tuples from the source relation $R$ the second time the mapper is executed. Only the keys need to be transferred. Instead of activating all the mapper functions to compute the $\mu_F(t)$, this result can be immediately retrieved from the auxiliary structure using the key $k$.

### 6.2.3 Local attribute storage

Another interesting strategy consists of storing attributes of the mapper input relation locally [3]. The underlying idea is that for subsequent executions of the mapper only transfer *(i)* attributes that form the key of the input relation and *(ii)* attributes that are not stored locally.

Different criteria can be used for selecting attributes to be stored locally. A first criterion is to store attributes that present greater I/O cost. A second criterion consists of storing attributes that are less frequently updated. Thus, upon mapper execution, only those that are most frequently updated are transferred.

### 6.2.4 Parallel execution

Mappers with complex functions are good candidates for parallelization. Using a mapper decomposition criterion based on the expected computational cost of complex mapper functions, the system can decide which mapper functions are good candidates for parallel execution. The algorithm supporting this strategy would then compose the output tuples as the result of each function execution become available.

## 7. CONCLUSIONS

This paper presents the formalization of an operator widely used in schema and data transformations. The data mapper operator is presented as a generalization of the relational extended projection operator. We have identified both semantic and physical optimization strategies to obtain fast executions of data mappers.

The data mapper operator subsumes previous data transformation operators such as the AJAX Map and Potter's Wheel Format operators while elegantly solving problems not addressed by the former tools.

Many useful data transformations can not be tackled with relational algebra. An important class of transformations are those that require to process the entire input relation. Our mapper operator is based on a tuple semantics and does not aim at specifiying such transformations. However, we believe that composing mappers with other operators proposed in literature such as *generalized groupping* [12] we can successfully specify many useful global transformations.

We believe that current relational database technology enhanced with the mapper operator would become not only a data staging repository, but also a more powerful transformation engine. This work requires experimental validation. Some of the optimizations presented in this paper are being deployed as part of the DATA FUSION platform [2].

## 8. REFERENCES


[1] Ascential. http://www.ardent.com.

[2] P. Carreira and H. Galhardas. Efficient development of data migration transformations. Demo paper. In *Proc. of the Semantic Integration Workshop (The Second Int'l Semantic Web Conf.)*, Sanibel Island, Florida, USA, October 2003.

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

[4] Susan B. Davidson and Anthony Kosky. Wol: A language for database transformations and constraints. In Alex Gray and Per-Åke Larson, editors, *Proceedings of the Thirteenth Int'l Conf. on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 55–65. IEEE Computer Society, 1997.

[5] 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)*, Rome, Italy, September 2001.

[6] J. Widom H. Garcia-Mollina, J. D. Ullman. *Database Systems – The Complete Book*. Prentice-Hall, 2002.

[7] 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, Bombay, India, September 1996.

[8] R. J. Miller, L. M. Haas, and M. Hernandéz. Schema Mapping as Query Discovery. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'00)*, pages 77–78, Cairo, Egypt, September 2000.

[9] L. Popa, Y. Velegrakis, R. Miller, and M. A. Hernáandez adn R. Fagin. Translating web data. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'02)*, Hong-Kong, August 2002.

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

[11] Sagent. http://www.sagent.com.

[12] E. Schallehn, K. Sattler, and G. Saake. Advanced grouping and aggregation for data integration. In *Proc. 10th Int'l Conf. on Information and Knowledge Management, CIKM'01*, Atlanta, GA, USA, 2001.

[13] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press. New York., 1988.

[14] L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *Proc. of ACM SIGMOD Int'l Conf. on the Managment of Data*, May 2001.


[3] Where the mapper is being executed