On Handling One-to-Many Transformations in Relational Systems

Paulo Carreira[1], Helena Galhardas[2], João D. Pereira[2], and Andrzej Wichert[2]

[1] Faculty of Sciences of the University of Lisbon, C6 - Piso 3, 1700 Lisboa, Portugal
paulo.carreira@xldb.di.fc.ul.pt

[2] Technical University of Lisbon and INESC-ID, Avenida Prof. Cavaco Silva, Tagus Park, 2780-990 Porto Salvo, Portugal
hig@inesc-id.pt, joao@inesc-id.pt, andreas.wichert@tagus.ist.utl.pt

Abstract. The optimization capabilities of RDBMSs make them attractive for executing data transformations that support ETL, data cleaning and integration activities. Despite the fact that many useful data transformations can be expressed as relational queries, an important class of data transformations that produces several output tuples for a single input tuple are not adequately supported by RDBMSs.
In this paper we address the issue of extending a RDBMS to include the mapper operator. In particular, we propose an SQL-like syntax together with several logical optimizations involving relational operators and the mapper. Finally, we experimentally compare the mapper operator with RDBMS implementations of one-to-many data transformations and validate the logical optimizations proposed.

Key words: Data Warehousing, Data Cleaning, Data Integration, ETL, Query optimization

1 Introduction

The setup of modern information systems comprises a number of activities that rely, to a great extent, in the use of data transformations [21]. Well known cases are legacy data migration, ETL (Extract, Transform, Load) processes that support data warehousing, data cleaning processes and the integration of data from multiple sources.

One natural way of expressing data transformations is to use a declarative query language and to specify the data transformations as queries (or views) over the source data. Because of the broad adoption of RDBMSs to store data underlying the above mentioned activities, such language is often SQL, a language based on Relational Algebra (RA). Unfortunately, due to its limited expressive power [1], RA alone cannot be used to specify some important classes of data transformations.

An important class of data transformations that may not be expressible in RA are the so called one-to-many data transformations [5], that are characterized by producing several output tuples for each input tuple. One-to-many data transformations occur normally due to the existence of *data heterogeneities*, i.e., due to the use of different data representations, in source and target schemas [25]. For instance, source data may consist of salaries aggregated by year, while the target data consists of salaries aggregated by month. Hence, each input row has to be converted into multiple output rows, one for each month. In this case, each input row corresponds to at most twelve output rows. However, expressing such data transformations as RA expressions is hampered by the fact that such bound cannot always be established a-priori. Consider the following example:

Relation `LOANS`

ACCT	AM
12	20.00
3456	140.00
901	250.00

Relation `PAYMENTS`

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

Fig. 1. Illustration of an unbounded data-transformation. *(a)* The source relation `LOANS` on the left, and *(b)* the target relation `PAYMENTS` on the right.

Example 1. Consider the source relation `LOANS[ACCT, AM]` (represented in Figure 1) that stores the details of loans per account. Suppose that `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 payment amounts greater than 100. The attribute `AMOUNT` is obtained by breaking down the value of `AM` 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. Furthermore, the target field `SEQNO` is a sequence number for the parcel. This sequence number starts at 1 for each sequence of parcels of a given account.

The implementation of data transformations similar to those requested for producing the target relation `PAYMENTS` of Example 1 is challenging, since the number of output rows, for each input row, is determined by the value of the source attribute `AM`. In this case, the upper bound on the number of output rows cannot be determined by analyzing the data transformation specification. We designate these data transformations as *unbounded* one-to-many data transformations. Other sources of unbounded data transformations exist like, for example, converting collection-valued attributes of SQL:1999 [22], where each element of the collection is mapped to a new row in the target table. In the context of data-cleaning, one commonplace transformation is converting a list of elements encoded as a string attribute into its atomic components.

Currently, one has to resort, either to a general purpose programming language, to some flavor of proprietary scripting supported by an ETL tool, or to an RDBMS using SQL:1999 *recursive queries* [22], or some sort of *Persistent Stored Modules* (PSMs) [15, Section 8.2] like stored procedures or *table functions* [11].

To address the problem of expressing one-to-many data transformations in a declarative and optimizable fashion, a specialized relational operator named *mapper* was recently proposed as an extension to RA [5]. Informally, a mapper is applied to an input relation and produces an output relation. It iterates over each input tuple and generates one or more output tuples, by applying a set of domain-specific functions. This way, it supports the dynamic creation of tuples based on a source tuple contents.

Although mappers appear implicitly in systems supporting schema and data transformations underlying ETL processes, data cleaning and data warehousing [13, 26, 9, 2], as far as we know, their execution and optimization has never been properly studied. This paper addresses the issue of how to extend an RDBMS to include the mapper operator. There are several reasons to do so: First, implementing the mapper operator as a relational operator opens interesting optimization opportunities since expressions that combine the mapper operator with standard RA operators can be optimized. Second, many data transformations are naturally expressed as relational expressions, leveraging the optimization strategies already implemented by RDBMSs [7]. Third, such extension further equips RDBMSs for data transformation activities, broadening their applicability in a wider range of data management activities. We remark that our idea of using RDBMSs as data transformation engines is not revolutionary, see [18]. Furthermore, several RDBMSs like Microsoft SQL Server and Oracle already include additional software packages specific for ETL tasks. The contributions of the mapper are the following:

1. An SQL-like concrete syntax for the mapper operator accomplished by extending the select statement;
2. The study of several query rewriting possibilities to be incorporated in the query optimizer; and
3. An experimental validation the shows the usefulness of implementing the mapper operator by comparing its physical implementation with alternative RDBMS solutions.

The rest of the paper is organized as follows. Section 2 introduces the mapper operator and exposes its concrete syntax by example. Then, in Section 3 we discuss how to extend the query optimizer to handle mappers. In Section 4, we report on a series of experiments to ascertain the feasibility of implementing the mapper operator. Related work is reviewed in Section 5 and finally Section 6 presents the conclusions.

2 The mapper operator

The mapper operator is formalized as 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 operator is parameterized by a set F of functions, which we designate as *mapper functions*. The intuition is that each mapper function f_{A_i} expresses a part of the envisaged data transformation, focused on a non-empty set A_i of attributes of the target schema. A key insight is that, when applied to a tuple, a mapper function can produce a set of values in the domain of its target attributes $Dom(A_i)$, rather than a single value.

The mapper operator is formally defined as follows: Given a set of mapper functions $F = \{f_{A_1}, ..., f_{A_k}\}$, the *mapper* of a relation s with respect to F, denoted by $\mu_F(s)$, is the relation instance of the target relation schema defined by

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

The formal semantics of the mapper presented above can be emulated with a simple iterator-based execution model as follows: For each input tuple, perform the evaluation of each mapper function and then compute the Cartesian product of the results. The output relation is obtained by unioning all the tuples so obtained.

We can express the data transformation of Example 1 by means of a mapper $\mu_{acct,amt}$, comprising two mapper functions. The function *acct* is the mapper function that returns a singleton with the account number `ACCT` properly left padded with zeroes, while *amt* is a mapper function that produces the attributes `[AMOUNT,SEQNO]`, 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 % represents the modulus operation. For instance, if v is the source tuple $(\texttt{901}, \texttt{250.00})$, the result of evaluating $amt(v)$ is the set $\{(\texttt{100}, \texttt{1}), (\texttt{100}, \texttt{2}), (\texttt{50}, \texttt{3})\}$. Given a source relation s including v, the result of the expression $\mu_{acct,amt}(s)$ is another relation that contains the set of tuples $\{\langle \texttt{'0901'}, \texttt{100}, \texttt{1}\rangle, \ \langle \texttt{'0901'}, \texttt{100}, \texttt{2}\rangle, \ \langle \texttt{'0901'}, \texttt{50}, \texttt{3}\rangle\}$.

For some given input tuple t, the set of values returned by a mapper function can be empty. When this happens, the function is acting as a filter and no tuple corresponding to t will be reflected in the output. Different situations can cause a mapper function to return an empty set. First, the function may not be able to process correctly some ill-formed inputs. For example, an empty set is returned as a default after the occurrence of a division by zero exception. Second, the function may encode some constraint on input data, resulting in an explicit rejection encoded as an empty set. Consider, for instance, a function returning restaurant addresses corresponding to a Parisian zip code: this function will return an empty set if it is invoked with a zip code corresponding, say, to a government building.

select

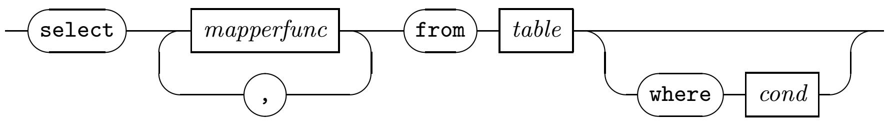


Fig. 2. Syntax diagram of a simplified version of the select statement.

2.1 Concrete syntax

The mapper operator can be easily embedded into the SQL syntax by incorporating mapper functions as expressions into the *select* block. The main change consists of replacing the standard list of columns and expressions that follow the **select** keyword by a list of mapper functions as illustrated in Figure 2. The relation to be used as input to the mapper operator is defined through the *table* expression that comes after the **from** keyword. Coarsely speaking, such expression denotes a relation and consists of relation names and sub-select statements combined through relational operators such as joins, unions, among others, applied to table names or sub-selects. Optionally, a filtering condition *cond* can be specified after the **where** keyword. The input schema of the mapper is the schema of the relation denoted by the *table* expression. The resulting schema of the mapper is obtained by concatenating the columns of the mapper functions. For clarity of presentation, aspects such as sorting, controlling duplicates, or grouping and aggregation are not considered.

As illustrated in Figure 3, a mapper function can be a column name, an expression, a function call or an inline mapper function definition. The existence of an input column in the select clause denotes an identity mapper function. Alternatively, a new name for the column in the output schema can be specified.

A mapper function call is identified by the **map** keyword followed by the function name. In order to avoid clashing of the output column names of the mapper function with the ones produced by other functions, the mapper function call can be followed by the specification of new column names. These mapper function must have been previously declared. The function is either a system built-in function or a function defined by the user. We do not propose a specific syntax for declaring *user defined mapper functions*. User defined mapper functions can be defined in mostly any language as long as it provides some mechanism for returning multiple values. One example of such mechanism is the **pipe row** statement of Oracle's PL/SQL [12]. In this way, the mapper function is specified outside the select statement using a more appropriate programming language. This usage of mapper functions is aligned with the SQL syntax for the computation of aggregates in the sense that aggregate functions like `COUNT` or `SUM` are implemented elsewhere and then embedded in the select statement as parameters of the aggregate operator.

Another way to define a mapper function consists of specifying inline an anonymous function. This function is specified through the **map** keyword with

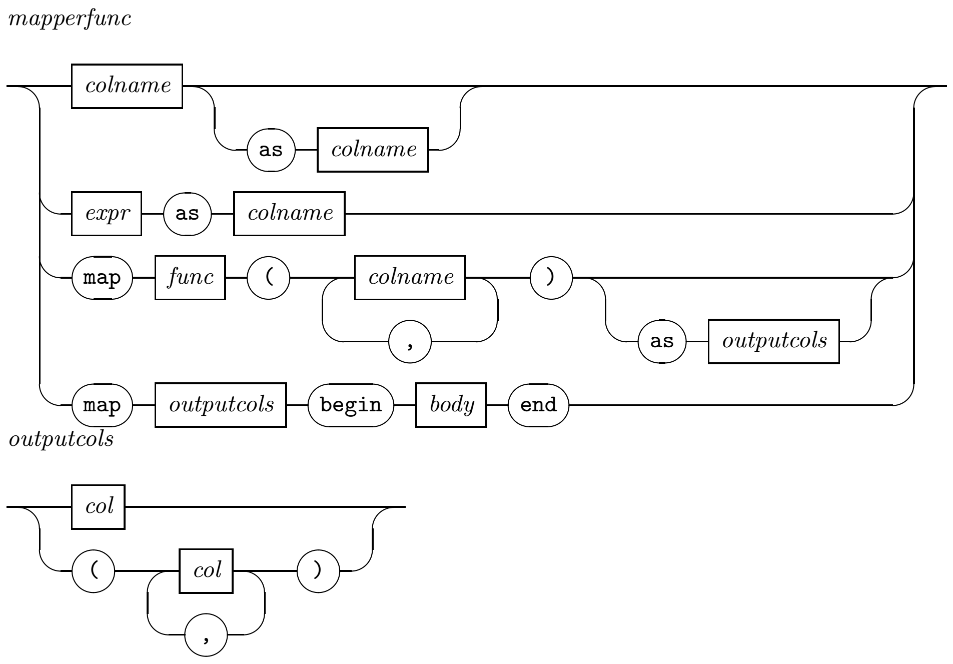


Fig. 3. Syntax diagram of a mapper function specification.

the output column names that will contribute to the output schema, followed by an inline specification of the function *body* within the **begin**...**end** block. In the case of inline function specifications, the input columns do not need to be specified. Instead, they are implicitly defined when the function implementation body accesses the columns of the input relation.

2.2 Specifying Filters

Filters are specified through the boolean expression of the **where** clause. Two kinds of filters can be specified, *(i) a-priori filters*, that apply to each tuple of the input relation, which are evaluated before the mapper and *(ii) a-posteriori filters* that are evaluated on the output of the mapper and are used to limit the mapper results. They are identified by sub-expressions defined over particular sets of columns. Sub-expressions that are defined only over the columns of the input relation define a-priori filters, while sub-expressions that are defined over columns generated by the mapper functions define a-posteriori filters.

3 Optimization

While parsing a *select* block, as soon as a mapper function is found, the parser knows that a mapper operator is present. In this case, upon parsing the query

```
select map acct(ACCT) as ACCTNO,
       map amt(AM) as AMOUNT, SEQNO
from LOANS, ACCOUNTS
where ACCOUNTS.ACCTN = LOANS.ACCT
  and ACCOUNTS.STATUS = 'O'
  and AMOUNT < 50
```

Fig. 4. A query that selects small payments of open accounts by implementing a mapper together with a-priori and a-posteri filters.

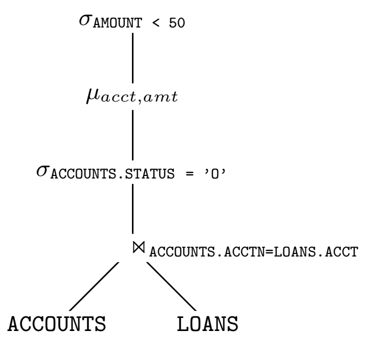
successfully, the parser identifies all the mapper functions being used and computes the output schema of the mapper. The input schema of the mapper is determined by the schema of input relations. Once the input schema is known, the input columns specified for each mapper function can be validated. The following step consists in rewriting the filter condition into the conjunctive normal form and validating it considering the input and output schemas. Then, each conjunct is analyzed to decide whether it constitutes a candidate to an a-priori or to an a-posteriori filter. In the query presented in Figure 4, the sub-expression `ACCOUNTS.STATUS = 'O'` defines an a-priori filter while `AMOUNT < 50` defines an a-posteriori filter.

We note also that in some situations it is not possible to clearly separate these two kinds of filters. For example, if the condition is dependent on both input and output columns of the mapper like e.g., `AMOUNT < ACCOUNTS.WDRAWLIMIT`, where `AMOUNT` is an output attribute produced by a mapper function and `ACCOUNTS.WDRAWLIMIT` is an attribute of the input relation. In these cases, the predicate can only be evaluated after all the mapper functions, i.e., a-posteriori.

The specification of a-posteriori filters in the **where** clause opens an interesting possibility of defining the condition using mapper functions. The sets of values returned by mapper functions can be tested with set operators like **in** or **exists**.

Moreover, whenever the input relation is defined through join operations, some of the conjuncts can be immediately pushed down into the appropriate join operators. Generically, the query plan that results from this process applies an a-posteriori filter to a mapper operator. This mapper operator, in turn, is evaluated over the input relation resulting from applying an a-priori filter to a query sub-plan that represents the input relation.

This concept is illustrated in Figure 3, by applying the filter $\sigma_{\texttt{AMOUNT < 50}}$ to the mapper $\mu_{acct,amt}$ which takes as input $\texttt{ACCOUNTS} \bowtie_{\texttt{ACCOUNTS.ACCTN=LOANS.ACCT}} \texttt{LOANS}$ that are not filtered by $\sigma_{\texttt{ACCOUNT.STATUS = 'O'}}$. The plans so obtained are then handed to the query optimizer where they undergo a sequence of rewritings that turn them into equivalent ones that are more efficient to evaluate. Besides the usual rewritings implemented by RDBMSs, others, specific to mappers can be introduced. Some of these rewritings are interesting because they take direct advantage of the mapper semantics. Herein, we briefly sketch the main ideas. Please refer to [5] for further details about rewriting rules and their corresponding proofs of correctness.



$\sigma_{\texttt{AMOUNT < 50}}$
$\mu_{acct,amt}$
$\sigma_{\texttt{ACCOUNTS.STATUS = 'O'}}$
$\bowtie_{\texttt{ACCOUNTS.ACCTN=LOANS.ACCT}}$
ACCOUNTS
LOANS

Fig. 5. Query plan representation for the query presented in Figure 4.

Projections A projection applied to a mapper is an expression of the form $\pi_Z(\mu_F(s))$. Those mapper functions whose output attributes are not contained in the list Z can be dropped from the mapper since the values that they produce will not be available for subsequent operations. Thus,

$$\pi_Z(\mu_F(s)) = \pi_Z(\mu_{F'}(s)) \tag{2}$$

where $F' = \{f_{A_i} \in F \mid \ A_i \text{ contains at least one attribute in } Z\}$.

Selections When applying a selection to a mapper, we can take advantage of the fact that many attributes are mapped by arithmetic expressions. Some expressions are simple identity functions. A selection $\sigma_{C_{A_i}}$ where C_{A_i} is a condition on the attributes produced by some mapper function $f_{A_i} \in F$, can be pushed through a mapper. Hence,

$$\sigma_{C_{A_i}}(\mu_F(s)) = \mu_F(\sigma_{C'_{A_i}}(s)) \tag{3}$$

where C'_{A_i} is a rewritten condition that uses the attributes of the input relation schema.

Joins It is often the case that mappers are applied to joins resulting in the expression $\mu_F(r \bowtie s)$. Depending of the type of join being performed, the output of the relation $r \bowtie s$ can be very large. In these cases, whenever the join is being performed in attributes mapped by identity mapper functions, it is possible to use the rule

$$\mu_F(r \bowtie s) = \mu_F(r) \bowtie \mu_F(s) \tag{4}$$

where the mapper functions in F do not produce duplicate values.

3.1 Plan selection

The choice of a particular plan is governed by the minimization of a cost metric. The cost of a mapper operator depends fundamentally on: *(i)* the costs of evaluation each mapper function and *(ii)* on the cardinality of the input relation.

In order to estimate the cost of subsequent operators whose input is produced by mappers, the cardinality of the output relation produced by a mapper also needs to be estimated. Since mappers can generate variably multiple output tuples for each input tuple, this estimation is an interesting problem in itself. One way to approach it consists of estimating the average *mapper fanout* factor[1]. If a mapper is being executed for the first time, an initial estimate for its fanout needs to be computed. This can be done by combining the estimated fanout factors of the mapper functions involved in the mapper operator. Another interesting observation is that when mapper functions return empty sets, no output tuples are produced. Thus, the mapper in some situations may act as a filter, which turns the *selectivity* of the mapper into another relevant factor. Like fanout, the initial mapper selectivity can also be estimated from the selectivities of the mapper functions. For more details about the cost model for the mapper operator, we refer the reader to [5].

4 Experiments

In this section, we analyze the performance of the mapper operator and consider the gains obtained with the proposed logical optimizations. Our results indicate that one-to-many data transformations can be evaluated substantially faster than traditional database solutions like table functions or recursive queries. Moreover, we shall see that the optimizations defined for mappers impart performance gains that are not matched by traditional RDBMS solutions.

To that aim, we contrast implementations of the data transformation proposed in Example 1 using the mapper operator with alternative implementations developed as table functions and recursive queries using two leading commercial RDBMSs. For more details on how to implement one-to-many transformations using RDBMSs, please refer to [6]. The mapper operator was implemented on top of the XXL DBMS library [30] which provides database query processing and optimization functionalities.

The database implementations were tested on two systems henceforth designated as DBX and OEX[2]. The parameters of both RDBMSs were carefully aligned and the same I/O conditions were enforced through the usage of the same raw devices. The hardware used was a single CPU machine (running at 3.4 GHz), with 1GB main memory RAM, and Linux (kernel version 2.4.2) installed. Concerning workload, a synthetic version of the input relation `LOANS` used in Example 1 was employed. To equalize the record length on XXL, DBX and OEX, a dummy column was added to the input table.

[1] Similarly to [8], we designate the average cardinality of the output produced for each input tuple by mappers and mapper functions as *fanout*.

[2] Due to the restrictions imposed by the license agreements, the true names of the systems under test cannot be revealed.

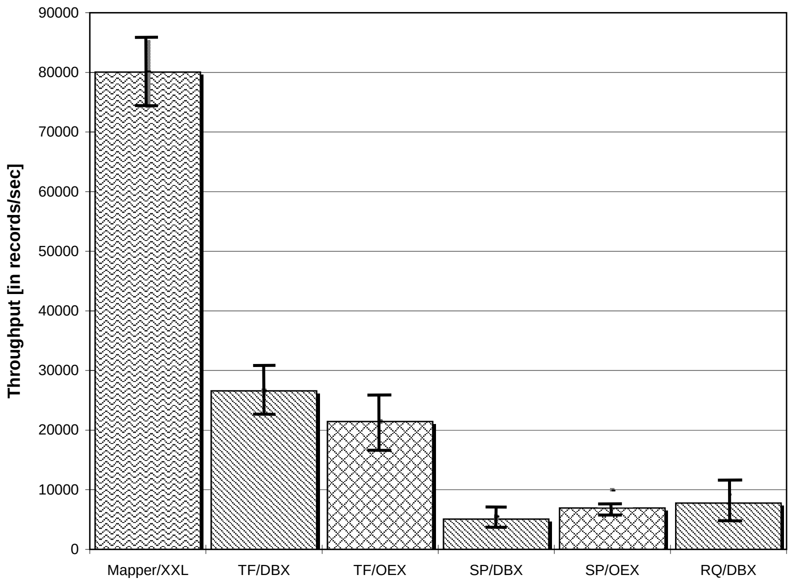


Fig. 6. Average throughput of the different implementations of Example 1 tested with input relation sizes varying from 100K to 5M rows graphed with standard deviation. No query results are reported for recursive queries on the OEX system since the subset of recursive queries supported by OEX is not powerful enough for expressing one-to-many transformations.

4.1 Results

We compare *throughput*, i.e., the amount of work done per second, of the distinct implementations of one-to-many data transformations. Throughput is expressed as the ratio of source records transformed per second and it is computed by measuring the *response time* of a data transformation that consists of reading the input table, transforming it and materializing the output table. All the timings reported were obtained with logging disabled.

In the first experiment, we intended to test the raw performance of the mapper operator for the three distinct implementations. The results depicted in Figure 6 show that one-to-many data transformations implemented with the mapper operator are more than two times better than table functions over DBX, which is the best alternative using RDBMSs. Since the amount of I/O incurred by all the systems is similar, even considering the overhead of the RDBMSs by comparison with XXL, we conjecture that one-to-many data transformations implemented as mappers running inside the RDBMS are very efficient. We also considered the implementations using stored procedures. However, it turns out that the performance is quite poor because logging cannot be disabled during their execution.

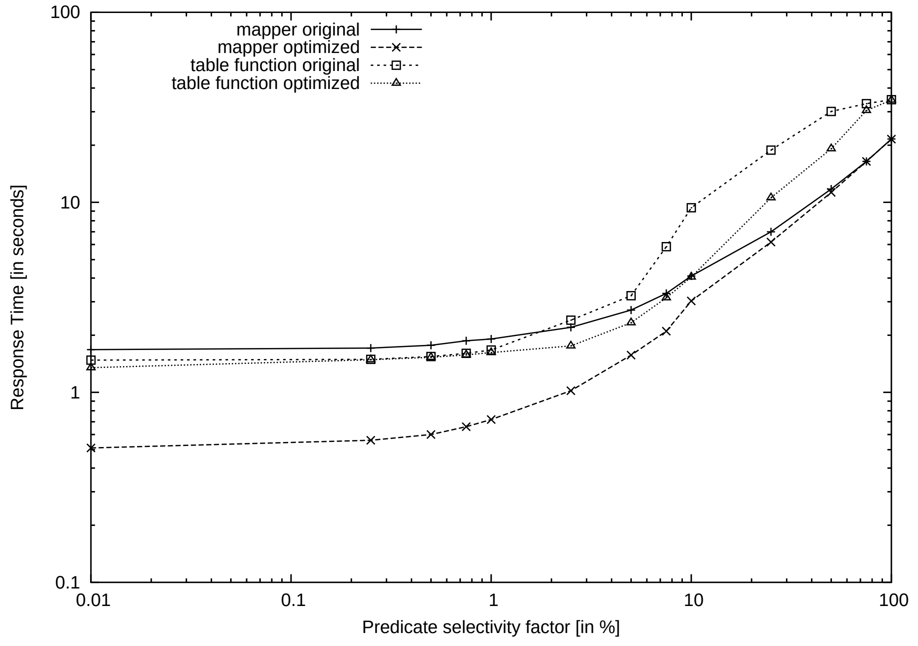


Fig. 7. Evolution of response time of applying selections to one-to-many transformations for increasing selectivity factors plotted in logarithmic scale. The mapper's original expression refers to $\sigma_{\texttt{ACCTNO}>p}(\mu_{acct,amt}(s))$, the mapper optimized expression refers to $\mu_{acct,amt}(\sigma_{\texttt{ACCT}>p}(s))$, the table function original expression refers to $\sigma_{\texttt{ACCTNO}>p}(TFacct, amt(s))$ and the table function optimized expression refers to $TFacct, amt(\sigma_{\texttt{ACCT}>p}(s))$. The size of the input relation s is fixed to 1M tuples.

In the second experiment, we analyzed the potential gains of logical optimizations like those suggested in Section 3. To that aim, we considered the evaluation of the expression $\sigma_{\texttt{ACCTNO}>p}(\mu_{acct,amt}(s))$ together with its optimized equivalent $\mu_{acct,amt}(\sigma_{\texttt{ACCT}>p}(s))$ obtained by pushing down the selection. The constant p was used to induce different selectivities. Moreover, we consider that the function *acct* performs a direct mapping, i.e., is an identity function. In Figure 7, we depict the performance of the original and the optimized expressions with varying selectivities. We observe that smaller selectivities correspond to the highest gains of the optimized expression over the original. For comparison, we draw the evolution of the selection applied to one-to-many transformations implemented using table functions on the OEX system, represented as $\sigma_{\texttt{ACCTNO}>p}(TFacct, amt(s))$. The performance of this solution is similar to the unoptimized version of mapper and only for small selectivities. This is due to the fact that the whole relation is read but few output tuples are generated.

Surprisingly, by analyzing the query plans generated by RDBMSs, we came across the fact that, whenever table functions or recursive queries are used to encode one-to-many data transformations, neither DBX nor OEX are capable of pushing down a selection on a directly mapped attribute. Hence, for comparison, we also tested the corresponding optimized expression $TFacct, amt(\sigma_{\texttt{ACCT}>p}(s))$ obtained manually. We observed that the manually optimized expression of the table functions brings higher gains specially on relatively high selectivities. For high selectivities, the response time of the original (non-optimized) RDBMS solution increases sharply. We conjecture that this behavior has to do with idiosyncrasies of OEX related with the pipelining of the tuples resulting from the table function into the selection operator.

5 Related Work

In order to support a growing range of applications of RDBMSs, several extensions to RA have been proposed since its inception, mainly in the form of new operators (like e.g., aggregates [20], recursivity [27], or OLAP operators [16]).

Recently, two leading commercial relational database systems have introduced operators aiming at Business Intelligence applications. Both these operators are capable of expressing one-to-many data transformations. The SQL Server 2005 unpivot operator transposes columns into rows and can be used for expressing one-to-many data transformations [10]. Oracle introduced the partitioned outer join, which can be employed for expressing data densification operations [17]. However, in both unpivot and partitioned outer join alternatives, the number of output tuples is bound to the number of columns of the input relation. In contrast, the mapper operator may generate an arbitrary number of output tuples based on each input tuple's contents.

To address the problem of efficiently extracting and loading data among heterogeneous databases, [2] proposed that data undergoes a series of transformations expressed through RA operations extended with a grouping operator and a map operator. A similar map operator is considered by [24] to model expensive function calls for the purpose of optimizing queries with expensive predicates. The main difference to the mapper operator is that the map operator only performs one-to-one tuple transformations.

Three frameworks that utilize data transformations, namely, Potter's Wheel [26], Ajax [14] and Data Fusion [4], have also proposed operators for addressing one-to-many data transformations. 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. The fold operator is similar to the unpivot referred above: the number of output tuples generated for each input tuple is bounded. Both the Ajax map operator and Data Fusion's mapper are powerful enough to express one-to-many data transformations that generate a number of output tuples that is dependent on each input tuple content.

Clio [23] 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. As we demonstrate in [5], these queries are not powerful enough for addressing the class of one-to-many data transformations.

One-to-many data transformations also arise in the context of ETL processes. To the best of our knowledge, in most ETL tools, to express one-to-many data-transformations, the user has to resort to some form of ad-hoc scripting. Furthermore, the optimization of ETL data transformations is a recent effort. Recently, [28] has proposed the optimization of ETL workflows as a global state-space search problem. In our approach, we use local optimization, since an ETL transformation program must be represented by a set of extended relational algebra expressions to be optimized one at a time.

6 Conclusions

In this paper, we focused on the feasibility of incorporating a specialized operator for handling one-to-many data transformations in RDBMSs. This extension is attractive, not only because one-to-many data transformations cannot be expressed using relational algebra but also because data usually resides in a RDBMS. We outlined the concrete syntax for this operator and then examined how a query optimizer can be extended to consider more advantageous execution plans in the presence of mappers. To test our ideas we analyzed experimentally different implementations of one-to-many data transformations using mappers and contrasted them with traditional implementations using table functions and recursive queries using two industry-leading RDBMSs.

The experiments showed that a native implementation of the mapper operator outperformed the best RDBMS solution by almost three times. We have also observed that RDBMSs do not in general perform very simple but highly valuable optimizations when table functions and recursive queries are used. Thus, we posit that one-to-many data transformations expressed by combining standard relational operators and mappers constitute a valid alternative.

The simple iterator-based semantics of the mapper operator enables efficient executions of one-to-many data transformations and favors an easy integration into the query processor of a database system. Towards physical optimization, we are developing different execution algorithms for the mapper operator. These algorithms take advantage of input duplicate values by employing caching techniques and hybrid-hashing proposed by [19]. Additionally, we consider incorporating the mapper operator in Apache Derby open source RDBMS [3].

One limitation of our work is that, despite the effort to configure the different systems so that they run in similar conditions, the alignment of these configurations lacks quantification. To address this shortcoming, we consider running TPC-H [29] loads on the different systems in order to obtain a metric for comparing their corresponding configurations.

References

1. A. V. Aho and J. D. Ullman. Universality of Data Retrieval Languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–119. ACM Press, 1979.
2. S. Amer-Yahia and S. Cluet. A Declarative Approach to Optimize Bulk Loading into Databases. *ACM Transactions of Database Systems*, 29(2):233–281, 2004.
3. Apache. Derby homepage. http://db.apache.org/derby, 2005.
4. P. Carreira and H. Galhardas. Efficient Development of Data Migration Transformations. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 2004.
5. P. Carreira, H. Galhardas, A. Lopes, and J. Pereira. One-to-many Transformation Through Data Mappers. *Data and Knowledge Engineering Journal (DKE)*, Elsevier Science, 2006.
6. P. Carreira, H. Galhardas, J. Pereira, F. Martins, and M. J. Silva. On the Performance of One-to-many Data Transformations. In *Proc. of the 5th International Workshop on Quality in Databases at VLDB (QDB'2007)*, 2007.
7. S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98)*, pages 34–43. ACM Press, 1998.
8. S. Chaudhuri and K. Shim. Query Optimization in the Presence of Foreign Functions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'93)*, pages 529–542, 1993.
9. Y. Cui and J. Widom. Lineage Tracing for General Data Warehouse Transformations. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01)*, 2001.
10. C. Cunningham, G. Graefe, and C. A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'04)*, pages 998–1009. Morgan Kaufmann, 2004.
11. A. Eisenberg, J. Melton, K. Kulkarni J.-E. Michels, and F. Zemke. SQL:2003 has been published. *Proceedings of the ACM SIGMOD Record*, 33(1):119–126, 2004.
12. S. Feuerstein and B. Pribyl. *Oracle PL/SQL Programming.* O'Reilly & Associates, 4th edition, 2005.
13. H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: An Extensible Data Cleaning Tool. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2(29), 2000.
14. H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. A. Saita. Declarative Data Cleaning: Language, Model, and Algorithms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01)*, 2001.
15. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems – The Complete Book.* Prentice-Hall, 2002.
16. J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.
17. A. Gupta, S. Subramanian, S. Bellamkonda, T. Bozkaya, N. Folkert, L. Sheng, and A. Witkowski. Data Densification in a Relational Database System. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*, pages 855–859. ACM, 2004.

18. L. M. Haas, R. J. Miller, B. Niswonger, M. T. Roth, P. M. Schwarz, and E. L. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.
19. J. M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *ACM Transactions on Database Systems*, 22(2):113–157, 1998.
20. A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 29(3):699–717, 1982.
21. D. Lomet and E. A. Rundensteiner, editors. *Special Issue on Data Transformations*, volume 22. IEEE Data Engineering Bulletin, 1999.
22. J. Melton and A. R. Simon. *SQL:1999 Understanding Relational Language Components*. Morgan Kaufmann Publishers, Inc., 2002.
23. R. J. Miller, L. M. Haas, M. Hernandéz, C. T. H. Ho, R. Fagin, and L. Popa. The Clio Project: Managing Heterogeneity. *SIGMOD Record*, 1(30), 2001.
24. T. Neumann, S. Helmer, and G. Moerkotte. On the Optimal Ordering of Maps, Selections, and Joins under Factorization, 2005.
25. E. Rahm and H.-H. Do. Data Cleaning: Problems and Current Approaches. *IEEE Bulletin of the Technical Committee on Data Engineering*, 24(4), 2000.
26. V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01)*, 2001.
27. M.-C. Shan and M.-A. Neimat. Optimization of Relational Algebra Expressions Containing Recursion Operators. In *Proceedings of the 19th Annual Conference on Computer Science (CSC'91)*, pages 332–341. ACM, 1991.
28. A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL Processes in Data Warehouses. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, 2005.
29. TPC. Benchmark H Standard Specification. http://www.tpc.org, 1999.
30. J. van den Bercken, J. P. Dittrich, J. Kräamer, T. Schäafer, M. Schneider, and B. Seeger. XXL – A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01)*, 2001.