On the performance of one-to-many data transformations

Paulo Carreira
University of Lisbon
paulo.carreira@xldb.di.fc.ul.pt

Helena Galhardas
Technical University of Lisbon
helena.galhardas@tagus.ist.utl.pt

João Pereira
Technical University of Lisbon
joao.serrenho@tagus.ist.utl.pt

Fernando Martins
University of Lisbon
fmp.martins@gmail.com

Mário J. Silva
University of Lisbon
mjs@di.fc.ul.pt

ABSTRACT

Relational Database Systems often support activities like data warehousing, cleaning and integration. All these activities require performing some sort of data transformations. Since data often resides on relational databases, data transformations are often specified using SQL, which is based on relational algebra. However, many useful data transformations cannot be expressed as SQL queries due to the limited expressive power of relational algebra. In particular, an important class of data transformations that produces several output tuples for a single input tuple cannot be expressed in that way. In this paper, we analyze alternatives to process one-to-many data transformations using Relational Database Management Systems, and compare them in terms of expressiveness, optimizability and performance.

1. INTRODUCTION

In modern information systems, an important number of activities rely, to a great extent, on the use of data transformations. Well known applications are the migration of legacy data, ETL (extract-transform-load) processes supporting data warehousing, data cleaning processes and the integration of data from multiple sources [25]. Declarative query languages propose a natural way of expressing data transformations as queries (or views) over the source data. Due to the broad adoption of RDBMSs, the language of choice is SQL, which is based on Relational Algebra (RA) [12].

Unfortunately, the limited expressive power of RA hinders the use of SQL for specifying important classes of data transformations [3]. A class of data transformations that may not be expressible in RA corresponds to the so called *one-to-many* data transformations [10], which are characterized by producing several output tuples for each input tuple. One-to-many data transformations are required for addressing certain types of *data heterogeneities* [33]. One familiar type of data heterogeneity arises when data is represented in the source and in the target using different aggregation levels. For instance, source data may consist of salaries aggregated by year, while the target data consists of salaries aggregated by month. In this case, the data transformation that takes place is frequently required to produce several tuples in the target relation to represent each tuple of the source relation.

Currently, one-to-many data transformations are implemented resorting to one of the following alternatives: *(i)* using a programming language, such as C or Java, *(ii)* using an ETL tool, which often requires the development of proprietary data transformation scripts; or *(iii)* using an RDBMS extension like recursive queries [26] or table functions [15].

In this paper we investigate the adequacy of RDBMSs for expressing and executing one-to-many data transformations. Implementing data transformations in this way is attractive since the data is usually stored in an RDBMS. Therefore, executing the data transformation inside the RDBMS appears to be the most efficient approach. The idea of adopting database systems as platforms for running data transformations is not revolutionary (see, e.g., [20, 5]). In fact, Microsoft SQL Server and Oracle, already include additional software packages that provide specific support for ETL tasks. But, as far as the authors are aware of, no experimental work to compare alternative RDBMS implementations of one-to-many data transformations has been undertaken.

The main contributions of our work are the following:

- we arrange one-to-many data transformations into subclasses using the expressive power of RA as dividing line;
- we study different possible implementations for each sub-class of one-to-many data transformations;
- we conduct an experimental comparison of alternative implementations, identifying relevant factors that influence the performance and optimization potential of each alternative.

The remainder of the paper is organized as follows: In Section 2 we further motivate the reader and introduce the two sub-classes of one-to-many data transformations by example. In Section 3, we focus on the implementation possibilities of the sub-classes of one-to-many data transformations.

VLDB '07, September 23-28, 2007, Vienna, Austria.

The experimental assessment is carried out in Section 4. Related work is reviewed in Section 5 and Section 6 presents the conclusions of the paper.

2. MOTIVATION

We now motivate the concept of one-to-many data transformations by introducing two examples based on real-world problems.

EXAMPLE 2.1: *Consider a relational table* LOANEVT *that, for each given loan, keeps the events that occur since the establishment of a loan contract until it is closed. A loan event consists of a loan number, a type and several columns with amounts. For each loan and event, one or more event amounts may apply. The field* EVTYPE *maintains the event type, which can be* OPEN *when the contract is established,* PAY *meaning that a loan installment has been payed,* EARLY *when an early payment has been made,* FULL *means that a full payment was made, or* CLOSED *meaning that the loan contract has been closed. In the target table named* EVENTS*, the same information is represented by adding one row per event with the corresponding amount. An event row is added only if the amount is greater than zero.*

Clearly, in the data transformation described in Example 2.1, each *input row* of the table LOANEVT corresponds to several *output rows* in the table EVENTS. See Figure 2. Moreover, for a given input row, the number of output rows depends on whether the contents of the columns CAPTL, TAX, EXPNS, BONUS are positive. Thus, each input row can result in at most four output rows. This means that there is a known *bound* on the number of output rows produced for each input row. We designate these data transformations as *bounded* one-to-many data transformations. However, in other one-to-many data transformations, such bound cannot always be established a-priori as shown in the following example:

EXAMPLE 2.2: *Consider the source relation* LOANS[ACCT, AM] *(represented in Figure 2) that stores the details of loans 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 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, initialized at one for each sequence of parcels of a given account.*

The implementation of data transformations like those for producing the target relation PAYMENTS of Example 2.2 is challenging, since the number of output rows, for each input row, is determined by the value of the attribute AM.

Relation LOANEVT

LOANNO	EVTYP	CAPTL	TAX	EXPNS	BONUS
1234	OPEN	0.0	0.19	0.28	0.1
1234	PAY	1000.0	0.28	0.0	0.0
1234	PAY	1250.0	0.30	0.0	0.0
1234	EARLY	550.0	0.0	0.0	0.0
1234	FULL	5000.0	1.1	5.0	3.0
1234	CLOSED	0.0	0.1	0.0	0.0

Relation EVENTS

LOANNO	EVTYPE	AMTYP	AMT
1234	OPEN	TAX	0.19
1234	OPEN	EXPNS	0.28
1234	OPEN	BONUS	0.1
1234	PAY	CAPTL	1000
1234	PAY	TAX	0.28
1234	PAY	CAPTL	1250
1234	PAY	TAX	0.30
1234	EARLY	CAPTL	550
1234	FULL	CAPTL	5000
1234	FULL	TAX	1.1
1234	FULL	EXPNS	5.0
1234	FULL	BONUS	3.0
1234	CLOSED	EXPNS	0.1

Figure 1: Illustration of a bounded one-to-many data transformation: source relation LOANEVT for loan number 1234 (at the top) and the corresponding target relation EVENTS (at the bottom).

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 2: Illustration of an unbounded data-transformation: the source relation LOANS on the left for loan number 1234, and the corresponding target relation PAYMENTS on the right.

Unlike in Example 2.1, the upper bound on the number of output rows cannot be determined by analysis of the data transformation specification. We designate these data transformations as *unbounded* one-to-many data transformations. Other sources of unbounded data transformations exist: for example converting collection-valued attributes of SQL 1999 [26], where each element of the collection is mapped to a distinct row in the target table. A common data transformation in the context of data-cleaning consists of converting a set of values encoded as string attribute with a varying number of elements into rows. This data transformation is unbounded because the exact number of output rows can only be determined by analyzing the string.

3. IMPLEMENTATION

Bounded one-to-many data transformations can be expressed as RA expressions. In turn, as we formally demonstrate elsewhere [10], no relational expression is able to capture unbounded one-to-many data transformations. Therefore, bounded data transformations can be implemented as relational algebra expressions while unbounded one-to-many

```
 1: insert into EVENTS (LOANNO, EVTYP, AMTYP, AMT)
 2:   select LOANNO, EVTYP, 'CAPTL' as AMTYP, CAPTL
 3:     from LOANEVT
 4:     where CAPTL > 0
 5:   union all
 6:   select LOANNO, EVTYP, 'TAX' as AMTYP, TAX
 7:     from LOANEVT
 8:     where TAX > 0
 9:   union all
10:   select LOANNO, EVTYP, 'EXPNS' as AMTYP, EXPNS
11:     from LOANEVT
12:     where EXPNS > 0
13:   union all
14:   select LOANNO, EVTYP, 'BONUS' as AMTYP, BONUS
15:     from LOANEVT
16:    where BONUS > 0;
```

Figure 3: RDBMS implementation of Example 2.1 as an SQL union query.

data transformations have to be implemented resorting to SQL 1999 *recursive queries* [26] or to SQL 2003 *Persistent Stored Modules* (PSMs) [15]. We now examine these alternatives.

3.1 Relational Algebra

Bounded one-to-many data transformations can be expressed as relational expressions by combining projections, selections and unions at the expense of the query length. Consider k to be the maximum number of tuples generated by a one-to-many data transformation, and let the condition C_i encode the decision of whether the ith tuple, where $1 \leq i \leq k$, should be generated. In general, given a source relation s with schema $X_1, ..., X_n$, we can define a one-to-many data transformation over s that produces at most k tuples for each input tuple through the expression

$$\pi_{X_1,...,X_n}\big(\sigma_{C_1}(r)\big) \cup ... \cup \pi_{X_1,...,X_n}\big(\sigma_{C_k}(r)\big)$$

To illustrate the concept, in Figure 3 we present the SQL implementation of the bounded data transformation presented in Example 2.1 using multiple **union all** (lines 5, 9 and 13) statements. Each **select** statement (lines 2–4, 6–8, 10–12 and 14–16) encodes a separate condition and potentially contributes with an output tuple. The drawback of this solution is that the size of the query grows proportionally to the maximum number of output tuples k that has to be generated for each input tuple. If this bound value k is high, the query becomes too big. Expressing one-to-many data transformations in this way implies a lot of repetition, in particular if many columns are involved.

3.2 RDBMS Extensions

We now turn to express one-to-many data transformations using RDBMS extensions, namely, recursive queries and table functions. Although these solutions enable expressing both bounded and unbounded transformations, here we introduce them for expressing unbounded transformations.

3.2.1 Recursive Queries

The expressive power of RA can be considerably extended through the use of recursion [3]. Although the resulting setting is powerful enough to express many useful one-to-many data transformations, we argue that this alternative undergoes a number of drawbacks. Recursive queries are not broadly supported by RDBMSs, and they are difficult to optimize and hard to understand.

```
 1: with recpayments(digits(ACCTNO), AMOUNT, SEQNO,
 2:      REMAMNT) as
 3:    (select ACCT,
 4:          case when base.AM < 100 then base.AM
 5:            else 100 end,
 6:          1,
 7:          case when base.AM < 100 then 0
 8:            else base.AM - 100 end
 9:     from LOANS as base
10:   union all
11:     select ACCTNO,
12:          case when step.REMAMNT < 100 then
13:              step.REMAMNT
14:            else 100 end,
15:          SEQNO + 1,
16:          case when step.REMAMNT < 100 then 0
17:            else step.REMAMNT - 100 end,
18:       from recpayments as step
19:       where step.REMAMNT > 0)
20:   select ACCTNO, SEQNO, AMOUNT
21:   from recpayments as PAYMENTS
```

Figure 4: RDBMS implementation of Example 2.2 as a recursive query in SQL 1999.

In Figure 4 we present a solution for Example 2.2 written in SQL 1999. A recursive query written in SQL 1999 is divided in three sections. The first section is the *base* of the recursion that creates the initial result set (lines 3–9). The second section, known as the *step*, is evaluated recursively on the result set obtained so far (lines 11–19). The third section specifies through a query, the *output expression* responsible for returning the final result set (lines 20–21). In the base step, the first parcel of each loan is created and extended with the column `REMAMNT` whose purpose is to track the remaining amount. Then, at each step we enlarge the set of resulting rows. All rows without `REMAMNT` constitute already a valid parcel and are not expanded by recursion. Those rows with `REMAMNT` > 0 (line 19) generate a new row with a new sequence number set to `SEQNO` $+ 1$ (line 15) and with remaining amount decreased by 100 (line 17). Finally, the `PAYMENTS` table is generated by projecting away the extra `REMAMNT` column.

Clearly, when using recursive queries to express data transformations, the logic of the data transformation becomes hard to grasp, specially if several functions are used. Even in simple examples like Example 2.2, it becomes difficult to understand how the cardinality of the output tuples depends on each input tuple. Furthermore, a great deal of ingenuity is often needed for developing recursive queries.

3.2.2 Persistent Stored Modules

Several RDBMSs support some form of procedural construct for specifying complex computations. This feature is primarily intended for storing business logic in the RDBMS for performance reasons or to perform operations on data that cannot be handled by SQL. Several database systems support their own procedural languages, like SQL-PL in the case of DB2 [22], TransactSQL in the case of Microsoft SQL Server and Sybase [24], or PL/SQL in the case of Oracle [16]. These extensions, designated as *Persistent Stored Modules* (PSMs), were introduced in the SQL 1999 standard [18, Section 8.2]. A module of a PSM can be, among others, a procedure, usually known as *stored procedure* (SP), or a function, known as a *user defined function* (UDF).

```
 1: create function LOANSTOPAYMENTS
 2:   return PAYMENTS_TABLE_TYPE pipelined is
 3:   ACCTVALUE LOANS.ACCT%TYPE;
 4:   AMVALUE LOANS.AM%TYPE;
 5:   REMAMNT INT;
 6:   SEQNUM INT;
 7:   cursor CLOANS is
 8:     select * from LOANS;
 9: begin
10:     open CLOANS;
11:     loop
12:       fetch CLOANS into ACCTVALUE, AMVALUE;
13:       REMAMNT := AMVALUE;
14:       SEQNUM := 1;
15:       while REMAMNT > 100
16:         loop
17:           pipe row(PAYMENTS_ROW_TYPE(
18:             LPAD(ACCTVALUE, 4, '0'), 100.00, SEQNUM));
19:           REMAMNT := REMAMNT - 100;
20:           SEQNUM := SEQNUM + 1;
21:         end loop
22:       if REMAMNT > 0 then
23:         pipe row(PAYMENTS_ROW_TYPE(
24:           values (LPAD(ACCTVALUE, 4, '0'),
25:             REMAMNT, SEQNUM));
26:       end if
27:   end loop
28: end LOANSTOPAYMENTS
```

Figure 5: Possible RDBMS implementation of Example 2.2 as a table function using Oracle PL/SQL. The details concerning the creation of the supporting row and table types PAYMENTS_ROW_TYPE and PAYMENTS_TABLE_TYPE are not shown.

```
1: select *
2: from (LOANEVT
3:   unpivot AMT for
4:     AMTYPE in ('LOANNO', 'EVTYP', 'TAX', 'EXPNS', BONUS'))
5: where AMT > 0
```

Figure 6: Implementation of Example 2.1 using an unpivot operation on SQL Server 2005.

Table functions extend the expressive power of SQL because they may return a relation. Table functions allow recursion[1] and make it feasible to generate several output tuples for each input tuple. The advantages are mainly enhanced performance and re-use [33]. Moreover, complex data transformations can be expressed by nesting UDFs within SQL statements [33]. However, table functions are often implemented using procedural constructs that hamper the possibilities of undergoing the dynamic optimizations familiar to relational queries.

Besides table functions, other kinds of UDFS exist, like *user defined scalar functions* (UDSFs), and *user defined aggregate functions* (UDAFs) [21]. Still, SQL extended with UDSFs and UDAFs may not be enough for expressing one-to-many data transformations. First, calls to UDSFs need to be embedded in an extended projection operator, which, as discussed in Section 3.1, is not powerful enough for expressing one-to-many transformations. Second, UDAFs must be embedded in aggregation operations, which can only represent many-to-one data transformations.

[1]Recursive calls of table functions are constrained in some RDBMSs, like DB2.

An interesting aspect of PSMs is that they are powerful enough to specify bounded as well as unbounded data transformations. Figure 5 presents the implementation of the data transformation introduced in Example 2.2 as a *user defined table function* (TF), as proposed by the SQL 2003 [15]. The table function implementation written in PL/SQL has two sections: a declaration section and a body section. The first one defines the set of working variables that are used in the procedure body and the cursor `CLOANS` (lines 7–8), which will be used for iterating through the `LOANS` table. The body section starts by opening the cursor. Then, a **loop** and a **fetch** statement are used for iterating over `CLOANS` (lines 11–12). The loop cycles until the **fetch** statement fails to retrieve more tuples from `CLOANS`. The value contained in `ACCTVALUE` is loaded into the working variable `REMAMNT` (line 13). The value of this variable will be later decreased in parcels of 100 (line 19). The number of parcels is controlled by the guarding condition `REMAMNT>0` (lines 15 and 22). An inner loop is used to form the parcels based on the value of `REMAMNT` (lines 15–21). A new parcel row is inserted in the target table `PAYMENTS` for each iteration of the inner loop. The tuple is generated through a **pipe row** statement that is also responsible for padding the value of `ACCTVALUE` with zeroes (lines 17–18 and 24–25). When the inner loop ends, a last **pipe row** statement is issued to insert the parcel that contains the remainder. The details concerning the creation of the row and table types `PAYMENTS_ROW_TYPE` and `PAYMENTS_TABLE_TYPE` are not presented.

The main drawback of PSMs is that they use a number of procedural constructs that are not amenable to optimization. Moreover, there are no elegant solutions for expressing the dynamic creation of tuples using PSMs. One needs to resort to intricate **loop** and **pipe row** statements (or **insert into** statements in the case of a stored procedure) as shown in Figure 5. From the description of Example 2.2, it is clear that a separate logic is used to compute each of the attributes. Nevertheless, in the PL/SQL code, the computation of `ACCTNO` is coupled with the computation of `AMOUNT`. Thus, the logic to calculate `ACCTNO` is duplicated in the code. This makes the code maintenance difficult and the code itself hard to optimize.

3.2.3 Pivoting operations

The *pivot* and *unpivot* operators constitute an important extension to RA, which were first natively supported by SQL Server 2005 [27]. The pivot operation collapses similar rows into a single wider row adding new columns on-the-fly [13]. In a sense, this operator collapses rows to columns. Thus, it can be seen as expressing a many-to-one data transformation. Its dual, the unpivot operator transposes columns into rows. Henceforth, the discussion focuses on the unpivot operator, since this operator can be used for expressing bounded one-to-many data transformations.

In what concerns expressiveness, the unpivot operator does not increase the expressive power of RA, since, as [13] admit, the unpivot operator can be implemented with multiple unions. Its semantics can be emulated by employing multiple union operations as proposed above for expressing bounded one-to-many data transformations through RA (Section 3.1).

Nevertheless, expressing one-to-many data transformations using the unpivot operator brings two main benefits comparatively to using multiple unions. First, the syntax is more

compact. Figure 6 shows how the unpivot operator can be employed to express the bounded one-to-many data transformation of Example 2.1. Second, data transformations expressed using the unpivot operator are more readily optimizable using the logical and physical optimizations proposed in [13].

4. EXPERIMENTS

We now compare the performance of the alternative implementations of the one-to-many data transformations introduced in Examples 2.1 and 2.2 using relational queries, recursive queries, table functions and stored procedures. We start by comparing the performance of each alternative to address bounded and unbounded transformations. Then, we investigate how the different solutions react to two intrinsic factors of one-to-many data transformations. Finally, we analyze the optimization possibilities of each solution.

We have tested the alternative implementations of one-to-many data transformations on two RDBMSs henceforth designated as DBX and OEX[2]. The entire set of planned implementations is shown in Figure 7. Unbounded data transformations cannot be implemented as relational queries. Furthermore, the class of recursive queries supported by the OEX system is not powerful enough for expressing unbounded data transformations. Additionally, due to limitations of the DBX system, table functions could not be implemented. Thus, to test another implementation across both systems, bounded an unbounded data transformations were implemented also as stored procedures. Finally unpivoting operations were not considered because both DBX and OEX do not support them.

4.1 Setup

The tests were executed on a synthetic workload that consists of input relations whose schemas are based on those used in Examples 2.1 and 2.2, for bounded and unbounded data transformations, respectively. Since the representation of data types may not be the same across all RDBMS, special attention must be given to record length. To equalize the sizes of the input rows of bounded and unbounded data transformations, a dummy column was added to the table `LOANS` so that its record size matches the record size of the table `LOANEVT`. We computed the average record size of each input table after its load. Both `LOANS` and `LOANEVT` have approximately 29 bytes in all experiments.

In addition, several parameters of both RDBMSs were carefully aligned. Below, we summarize the main issues that received our attention.

I/O conditions An important aspect regarding I/O is that all experiments use the same region of the hard-disk. To induce the use of the same area of the disk, I/O was forced through raw devices. The hard-disk is partitioned in cylinder boundaries as illustrated in Figure 8. The first partition is a primary partition formatted with Ext3 file system and journaling enabled and is used for the operating system and RDBMS installations as well as for the database control files. The second partition is used as swap space. The remaining partitions are the logical partitions accessed as raw devices. These partitions handle data and log files. Each RDBMSs accesses tablespaces created in distinct raw devices. The first logical partition (`/dev/hda5`) handles the tablespace named `RAWSRC` for input data; the second logical partition (`/dev/hda6`) handles the tablespace named `RAWTGT` for output data. The partition (`/dev/hda7`) is used for raw logging and finally (`/dev/hda8`) is used as the temporary tablespace. To minimize the I/O overhead, both input and output tables were created with `PCTFREE` set to 0. In addition, the usage of kernel asynchronous I/O [6] was turned off.

Block sizes In our experiments, tables are accessed through full-table scans. Since there are no updates and no indexed-scans, different block sizes have virtually no influence in performance. The block size parameters are set to the same value of 8KB. Since full table scans use multi-block reads, we configure the amount of data transferred in a multi-block read to 64K.

Buffers To improve performance, RDBMSs cache frequently accessed pages in independent memory areas. One such area is the which caches disk pages *buffer pool* [14]. The configuration of buffer pools in DBX differs from that of the OEX system. For our purposes, the main difference lies in the fact that, in DBX, individual buffer pools can be assigned to each tablespace, while OEX uses one global buffer pool for all tablespaces. In DBX, we assign a buffer pool of 4MB to the `RAWSRC` tablespace, which contains the source data. In OEX we set the size of the cache to 4MB.

Logging Both DBX and OEX use *write-ahead* logging mechanisms that produce undo and redo log [19, 28]. We attempt to minimize the logging activity by disabling logging on both in DBX and OEX experiments. However, we note that logging cannot be disabled in the case of stored procedures because **insert into** statements executed within stored procedures always generate log.

We measured the *throughput*, i.e., the amount of work done per second, of the considered implementations of one-to-many data transformations. Throughput is expressed as the number of source records transformed per second and is computed by measuring the *response time* for a data transformation applied to an input table. The response time is measured as the time interval that mediates the submission of the data transformation implementation from the command line prompt and its conclusion. The interval that mediates the submission of the request and the execution by the system, known as *reaction time*, is considered neglectable. The hardware used was a single CPU machine (running at 3.4 GHz) with 1GB of RAM and Linux (kernel version 2.4.2) installed.

4.2 Throughput comparison

To compare the throughput of the evaluated alternatives, we executed their implementations on input relations with increasing sizes. The results for both bounded and unbounded implementations, are shown in Figure 9. We observe that table functions are the most performant of the

[2]Due to the restrictions imposed by DBMS licensing agreements, the actual names of the systems used for this evaluation will not be revealed.

Implementations of one-to-many data transformations						
	Bounded			Unbounded		
	Relational Query	Table Function	Stored Procedure	Recursive Query	Table Function	Stored Procedure
DBX	yes	no	yes	yes	no	yes
OEX	yes	yes	yes	no	yes	yes

Figure 7: Different mechanisms used for implementing the one-to-many data transformations developed for the experiments.

OS	swap	raw	raw	raw	raw
hda1	hda2	hda5	hda6	hda7	hda8
58GB	2GB	25GB	25GB	25GB	25GB

Figure 8: Hard-disk partitioning for the experiments

implementations. Then, implementations using unions and recursive queries are considerably more efficient than stored procedures. Figure 9b shows that the throughput is mostly constant as the input relation size increases.

The low throughput observed in stored procedures is mainly due to the huge amounts of redo logging activity incurred during their execution. Unlike the remaining solutions, it is not possible to disable logging for stored procedures. In particular, the logging overhead monitored for stored procedures is ≈ 118.9 blocks per second in the case of DBX and ≈ 189.2 blocks per second in the case of OEX. We may conclude that, if logging was disabled, stored procedures would execute with a comparable performance to table functions.

4.3 Influence of selectivity and fanout

In one-to-many data-transformations, each input tuple may correspond to zero, one, several output tuples. The ratio of input tuples for which at least one output tuple is produced is known as the *selectivity* of the data transformation. Similarly to [11], the average number of output tuples produced for each input tuple is called *fanout* Different data sets generating data transformations with different selectivities and fanouts have been used in our working examples. These data sets produce predefined average selectivities and fanouts. A set of experiments varying the selectivity and fanout factors was put in place, to help understand the effect of selectivity and fanout on data transformations. The results are depicted in Figure 10.

Concerning selectivity, we observe on Figure 10a that higher throughputs are obtained for smaller selectivities. This stems from having less output tuples created when the selectivity is smaller. The degradation observed is explained having more output tuples produced and materialized at higher selectivities. Stored procedures degrade faster due to an increase in the log generation.

With respect to the fanout factor, greater fanout factors imply generating more output tuples for each input tuple and hence I/O activity is directly influenced. To observe the impact of this parameter, we increase the fanout factor from 1 to 32. Figure 10b illustrates the evolution of the throughput for unbounded transformations. The throughput of all implementations decreases because more time is spent writing the output tuples. In the case of recursive queries, more I/O is incurred because higher fanouts increase the size of the intermediate relations used for evaluating the recursive query. Finally, for stored procedures, the more tuples are written, the more log is generated.

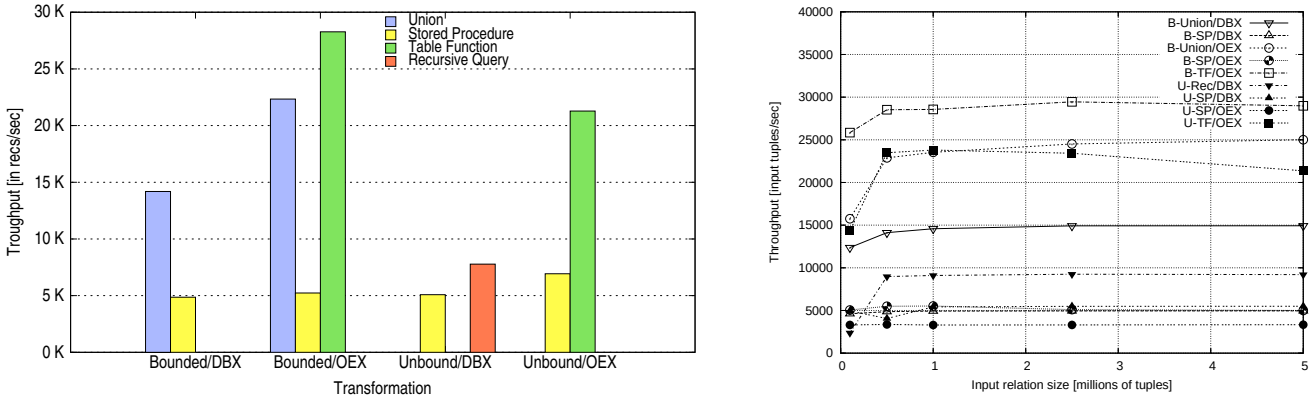
4.4 Query optimization issues

The analysis of the query plans of the different implementations shows that the RDBMSs used in this evaluation are not always capable of optimizing queries involving one-to-many data transformations.

To validate this hypothesis, we contrasted the execution of a simple selection applied to a one-to-many transformation, represented as $\sigma_{\texttt{ACCTNO}>p}(T(s))$, with its corresponding optimized equivalent, represented as $T(\sigma_{\texttt{ACCT}>p}(s))$, where T represents the data transformation specified in Example 2.2, except that the column `LOANS` is directly mapped, and p is a constant used only to induce a specific selectivity. We stress that the optimized versions are obtained manually, by pushing down the selection condition. Figure 11a presents the response times of the original and optimized versions implemented as recursive queries and as table functions. We observe that the optimized versions are considerably more efficient that their corresponding originals.

We conjecture that the optimization handicap of RDBMSs for processing one-to-many data transformations has to do with the intrinsic difficulties of optimizing queries using recursive functions and table functions. In fact, the optimization of recursive queries is far from being a closed subject [30]. In turn, table functions are implemented using procedural constructs that hamper optimizability. Once the table function makes use of procedural constructs, it is not possible to perform the kind of optimizations that relational queries undergo. We have found that bounded one-to-many data transformations take advantage of the logical optimizations built into the RDBMS when they are implemented through a union statement. Applying a filter to a union is readily optimized. The response time for of the experiment was included in Figure 11a for comparison.

Another type of optimization that RDBMSs can apply in one-to-many data transformations is the use of cache. This factor is important to optimize the execution of queries that use multiple union statements and therefore need to scan the input relation multiple times. Likewise, recursive queries perform multiple joins with intermediate relations. This happens because the physical execution of a recursive query involves performing one full select to seed the recursion and then a series of successive union and join operations to unfold the recursion. As a result, these operations are likely to



(a) Average throughput

(b) Evolution with relation size

Figure 9: Throughput of data transformation implementations with different relation sizes. Fanout is fixed to 2.0, selectivity fixed to 0.5, and cache size set to 4MB.

be influenced by the buffer cache size.

To evaluate the impact of the buffer pool cache size on one-to-many transformations, we executed a set of experiments varying the buffer pool size. The results, depicted in Figure 11b, show that a larger buffer pool cache is most beneficial for bounded data transformations implemented as unions. This is explained by larger buffer pool caches reduce the number of physical reads that required when scanning the input relations multiple times. We also remark a distinct behavior of the RDBMSs used in the evaluation as cache size increases. The throughput in OEX increases smoothly while in DBX there is sharp increase. This has to do with the differences in cache the replacement policies of these systems while performing table scans [14]. DBX uses the *least recently used* (LRU) [29] to select the next page to be replaced from the cache while the OEX system, according to its documentation, uses a *most recently used* (MRU) replacement policy. The LRU replacement policy performs quite poorly on sequential scans if the cache smaller than the input relation. The LRU replacement policy purges the cache when full table scans are involved and the size of the buffer pool is smaller than the size of the table [23]. We conclude that for small input tables using multiple unions is the most advantageous alternative for bounded one-to-many data transformations. However, in the presence of large input relations, table functions are the best alternative since they are invariant to cache size. This is due to the fact that input relation being scanned only once. Stored procedure implementations also scan the input relation only once but are less performant due to logging.

According to [13], the pivot operator processes the input relation only once. As a result, it is not likely to be influenced by buffer cache size, unlike the chaining of multiple unions we present in Section 3.1

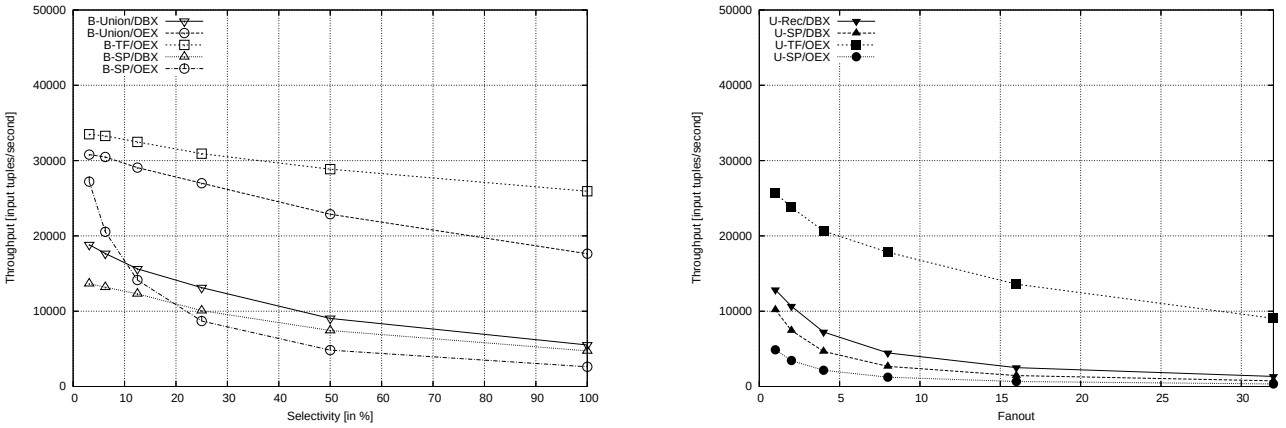
5. RELATED WORK

In Codd's original model [12], RA expressions denote transformations among relations. In the following years, the idea of using a queries for specifying data transformations would be pursued by two prototypes, Convert and Express [36, 37], shortly followed by results on expressivity limitations of RA by [3, 31]. Many useful data transformations can be appropriately defined in terms of relational expressions, if we consider relational algebra equipped with a generalized projection operator [38, p. 104]. However, this extension is still weak to express unbounded one-to-many data transformations.

To support the growing range of RDBMS applications, several extensions to RA have been proposed in the form of new declarative operators and also through the introduction of language extensions to be executed by the RDBMS. One such extension, interesting for one-to-many transformations, is the *pivot operator* [13], which is not influenced by buffer cache size. However, the pivot operator cannot express unbounded one-to-many data transformations and, as far as we know it is only implemented by SQL Sever 2005.

Recursive query processing was early addressed by [3], and then by several works about recursive query optimization, like, for example [35, 39]. There are also proposals for extending SQL to handle particular forms of recursion [2], like the Alpha Operator [1]. Despite being relatively well understood at the time, recursive query processing was not supported by SQL-92. By the time the SQL 1999 [26] was introduced, some of the leading RDBMSs (e.g., Oracle, DB2 or POSTGRES) were in the process of supporting recursive queries. As a result, these systems ended up supporting different subsets of recursive queries with different syntaxes. Presently, the broad support of recursion still constitutes a subject of debate [32].

The problem of specifying one-to-many data transformations has also been addressed in the context of data cleaning and transformations by tools like Potter's Wheel [34], Ajax [17] and Data Fusion [7]. These tools have proposed operators for expressing one-to-many data transformations. Pot-



(a) Increasing selectivity (bounded)

(b) Increasing fanout (unbounded)

Figure 10: Evolution of throughput for varying selectivities and fanouts over input relations with 1M tuples and 4MB of cache: (a) shows the evolution for bounded transformations with increasing selectivity (fanout set to 2.0) and (b) show the evolution for unbounded transformations with increasing fanout and (selectivity fixed to 0.5). The corresponding unbounded and bounded variants display identical trends.

ter's Wheel fold operator addresses bounded one-to-many transformations, while Ajax and Data Fusion also implement operators for addressing also unbounded data transformations.

Building on the above contributions, we recently proposed the extension of RA with a specialized operator named *data mapper*, which addresses one-to-many transformations [8, 9, 10]. The interesting aspect of this solution lies in that mappers are declarative specifications of a one-to-many data transformation, which can then be logically and physically optimized.

6. CONCLUSIONS

We organized our discussion of one-to-many data transformations into two groups representing bounded and unbounded data transformations. There is no general solution for expressing one-to-many data transformations using RDBMSs. We have seen that although bounded data transformations can be expressed by combining unions and selections, unbounded data transformations require advanced constructs such as recursive queries of SQL:1999 [26] and table functions introduced in the SQL 2003 standard [15]. However, these are not yet supported by many RDBMSs.

We then conducted an experimental assessment of how RDBMSs handle the execution of one-to-many data transformations. Our main finding was that RDBMSs cannot, in general, optimize the execution of queries that comprise one-to-many data transformations. One-to-many data transformations expressed both as unions or as recursive queries incur in unnecessary consumptions of resources, involving multiple scans over the input relation and the generation of intermediate relations, which makes them sensible to buffer cache size. Table functions are acceptably efficient since their implementation emulates an iterator that scans the input relation only once. However, their procedural nature blends logical and physical aspects, hampering dynamic optimization.

An additional outcome of the experiments was the identification of selectivity and fanout, two important factors of one-to-many data transformations, that influence their cost. Together with the input relation size, these factors can be used to predict the cost of one-to-many data transformations. This information can be exploited to take advantage when the cost-based optimizer chooses among alternative execution plans involving one-to-many data transformations.

In fact, we believe that one-to-many data transformations can be logically and physically optimized when expressed through a specialized relational operator like the one we propose in [9, 10]. As future work, we plan to extend the Derby [4] open source RDBMS to execute and optimize one-to-many data transformations expressed as queries that incorporate this operator. In this way, we equip an RDBMSs to be used not only as data store but also as data transformation engine.

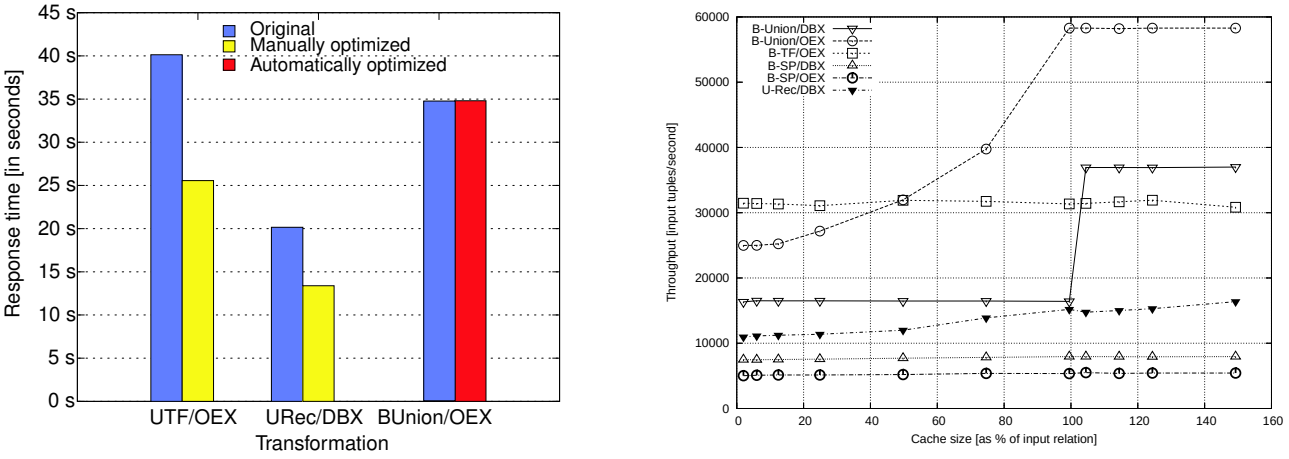
7. REFERENCES

[1] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, 1988.

[2] R. Ahad and S. B. Yao. Rql: A recursive query language. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):451–461, June 1993.

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

[4] Apache. Derby homepage. http://db.apache.org/derby, 2006.



(a) Original vs. optimized

(b) Evolution with cache size

Figure 11: Sensibility of data transformation implementations with one 1M tuples: (a) to optimization, with cache size fixed to 4MB and, (b) to cache size variations. Selectivity is fixed to 0.5 and fanout set to 2.0.

[5] P. A. Bernstein and E. Rahm. Data wharehouse scenarios for model management. In *International Conference on Conceptual Modeling / The Entity Relationship Approach*, pages 1–15, 2000.

[6] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan. Asynchronous I/O Support in Linux 2.5. In *Proc. of the Linux Symposium*, 2003.

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

[8] P. Carreira and H. Galhardas. Execution of Data Mappers. In *Int'l Workshop on Information Quality in Information Systems (IQIS)*. ACM, June 2004.

[9] P. Carreira, H. Galhardas, A. Lopes, and J. Pereira. Extending relational algebra to express one-to-many data transformations. In *20th Brasillian Symposium on Databases SBBD'05*, Oct. 2005.

[10] P. Carreira, H. Galhardas, A. Lopes, and J. Pereira. One-to-many transformation through data mappers. *Data and Knowledge Engineering Journal*, 62(3):483–503, September 2007.

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

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

[13] C. Cunningham, G. Graefe, and C. A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'04)*, pages 998–1009. Morgan Kaufmann, 2004.

[14] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)*, 9(4):560–595, 1984.

[15] A. Eisenberg, J. Melton, K. K. J.-E. Michels, and F. Zemke. SQL:2003 has been published. *ACM SIGMOD Record*, 33(1):119–126, 2004.

[16] S. Feuerstein and B. Pribyl. *Oracle PL/SQL Programming*. O'Reilly & Associates, 4 edition, 2005.

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

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

[19] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, 1981.

[20] L. Haas, R. Miller, B. Niswonger, M. T. Roth, P. Scwarz, and E. L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *Special Issue on Data Transformations. IEEE Data Engineering Bulletin*, 22(1), 1999.

[21] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational DBMS. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 379–389. ACM Press, 1998.

[22] Z. Janmohamed, C. Liu, D. Bradstock, R. chong, M. G. abd F. McArthur, and P. Yip. *DB2 SQL PL. Essential Guide for DB2 UDB*. Prentice-Hall, 2005.

[23] S. Jiang and X. Zhuang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. of SIGMETRICS 2002*, 2002.

[24] K. Kline, L. Gould, and A. Zanevsky. *TransactSQL Programming*. O'Reilly & Associates, 1 edition, 1999.

[25] D. Lomet and E. A. Rundensteiner, editors. *Special Issue on Data Transformations*, volume 22. IEEE Data Engineering Bulletin, 1999.

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

[27] Microsoft. Sql server home page. http://www.microsoft.com/sql/, 2005.

[28] C. Mohan and F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 371–380. ACM Press, 1992.

[29] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD Int'l Conf. on the Managment of Data*, pages 297–306. ACM Press, 1993.

[30] C. Ordonez. Optimizing recursive queries in SQL. In *SIGMOD '05: Proc. of the 2005 ACM SIGMOD International Conference on Management of data*, pages 834–839. ACM Press, 2005.

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

[32] T. Pieciukiewicz, K. Stencel, and K. Subieta. Usable recursive queries. In *Proc. of the 9th East European Conference, Advances in Databases and Information Systems (ADBIS)*, volume 3631 of *Lecture Notes in Computer Science*, pages 17–28. Springer-Verlag, 2005.

[33] E. Rahm and H.-H. Do. Data Cleaning: Problems and current approaches. *IEEE Bulletin of the Technical Comittee on Data Engineering*, 24(4), 2000.

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

[35] M.-C. Shan and M.-A. Neimat. Optimization of relational algebra expressions containing recursion operators. In *CSC '91: Proc. of the 19th annual conference on Computer Science*, pages 332–341. ACM Press, 1991.

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

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

[38] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. MacGraw-Hill, 5th edition, 2005.

[39] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *1st Int'l Conference of Expert Databases*, pages 271–293, 1986.