Performance Analysis of One-to-Many Data Transformations

Paulo Carreira
Helena Galhardas
João Pereira
Fernando Martins
Mário J. Silva

DI–FCUL TR–06–25

December 2006

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Performance Analysis of One-to-Many Data Transformations

Paulo Carreira[1,2]
pjcarreira@xldb.di.fc.ul.pt

Helena Galhardas[2,3]
hig@inesc-id.pt

João Pereira[2,3]
joao@inesc-id.pt

Fernando Martins[1]
fmp.martins@gmail.com

Mário Silva[1]
mjs@di.fc.ul.pt

[1] Faculty of Sciences of the University of Lisbon, Portugal
[2] INESC-ID and [3] Instituto Superior Técnico, Tagus Park, Portugal

December 2006

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 of relational algebra. However, many useful data transformations cannot be expressed as SQL queries due to 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 report, we analyze alternatives to process one-to-many data transformations using Relational Database 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-tranform-load) processes supporting data warehousing, data cleaning processes and the integration of data from multiple sources [Lomet and Rundensteiner, 1999]. 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) [Codd, 1970].

Unfortunately, the limited expressive power of RA, hinders the use of SQL for specifying important classes of data transformations [Aho and Ullman, 1979]. A class of data transformations that may not be expressible in RA are the so called *one-to-many* data transformations [Carreira et al., 2006], 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* [Rahm and Do, 2000]. 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 [Melton and Simon, 2002] or table functions [Eisenberg et al., 2004].

In this report we explore 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., [Haas et al., 1999, Bernstein and Rahm, 2000]). Microsoft SQL Server and Oracle, already include additional software packages that provide specific support for ETL tasks. The main contributions of our work are the following:

- we arrange one-to-many data transformations into sub-classes 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 report is organized as follows: in Section 2 we further motivate the reader and introduce the two sub-classes of one-to-many transformations by example. In Section 3, we focus on the implementation possibilities of the distinct sub-classes of one-to-many data transformations. The experimental assessment is carried out in Section 4. Related work is reviewed in Section 5 and Section 6 gives conclusions of the report.

2 One-to-many data transformations

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

Example 2.1: *Consider a source table named* `LOANEVT`*, where each row represents an event that occurs with a loan. A loan event consists of a loan number, its type and several columns with amounts. Each event may apply one or more amounts. The type of event is encoded in the column* `EVTYPE`*. The following types of events may apply:* `OPEN`*, when the contract is established;* `PAY`*, meaning*

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 on the left and the corresponding target relation `EVENTS` on the right.

that a loan installment has been payed; `EARLY`*, when an early payment has been made;* `FULL`*, meaning that a full payment was made, or* `CLOSED` *meaning that the loan contract has been closed. The target table, named* `EVENTS`*, represents the same information using one row per event and per amount. Only positive amounts need to be considered.*

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 1. 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. 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`] *that stores the details of loans per account (see Figure 2). 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.*

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.

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. 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, like, for example, converting collection-valued attributes of SQL:1999 [Melton and Simon, 2002]. Each element of the collection must be mapped to a distinct row in the target table. One commonplace data transformation in the context of data-cleaning consists of converting a string attribute encoding a set 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 Implementing one-to-many data transformations

Bounded data transformations can be expressed as RA expressions. In turn, as we formally demonstrate elsewhere [Carreira et al., 2006], 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 data transformations have to be implemented resorting to *recursive queries* of SQL:1999 [Melton and Simon, 2002] or to table functions of SQL 2003 [Eisenberg et al., 2004]. 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

```
 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.

through the expression

$$\pi_{X_1,\ldots,X_n}\left(\sigma_{C_1}(r)\right) \cup \ldots \cup \pi_{X_1,\ldots,X_n}\left(\sigma_{C_k}(r)\right)$$

To illustrate the concept, in Figure 3 we illustrate 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 expressing one-to-many data transformations using RDBMS extensions, namely, recursive queries and table functions. Although these solutions enable expressing both bounded and unbounded transformations, we introduce them for expressing unbounded transformations due to lack of space.

3.2.1 Recursive Queries

The expressive power of RA can be considerably extended through the use of recursion [Aho and Ullman, 1979]. 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.

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 2–8). The second section, known as the *step*, is evaluated recursively on the result

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

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

set obtained so far (lines 10–18). The third section specifies through a query, the *output expression* responsible for returning the final result set (lines 19–20). 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 18) generate a new row with a new sequence number set to SEQNO $+ 1$ (line 14) and with remaining amount decreased by 100 (line 16). 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 Table Functions

Several RDBMSs support the concept of *user defined functions* (UDFs). This feature is primarily intended for storing business logic in the RDBMS for performance and reuse, or to perform operations on data that are not handled by SQL. Table functions, a special kind of UDF introduced in SQL 2003, return tables, extending the express power of SQL [Eisenberg et al., 2004]. Table functions allow recursion[1] and make it feasible to generate several output tuples for each input tuple.

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

```
 1: create function LOANSTOPAYMENTS return PAYMENTS_TABLE_TYPE pipelined is
 2:   ACCTVALUE LOANS.ACCT%TYPE;
 3:   AMVALUE LOANS.AM%TYPE;
 4:   REMAMNT INT;
 5:   SEQNUM INT;
 6:   cursor CLOANS is
 7:     select * from LOANS;
 8: begin
 9:     open CLOANS;
10:     loop
11:       fetch CLOANS into ACCTVALUE, AMVALUE;
12:       REMAMNT := AMVALUE;
13:       SEQNUM := 1;
14:       while REMAMNT > 100
15:         loop
16:           pipe row(PAYMENTS_ROW_TYPE(
17:             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'), 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.

One interesting aspect of table functions is that they are powerful enough to specify bounded as well as unbounded data transformations. In Figure 5, we present the implementation of the data transformation introduced in Example 2.2 as a PL/SQL table function. The table function has two sections: a declaration section and a body section. The first defines the set of working variables that are used in the procedure body and the cursor CLOANS (lines 6–7) that 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. The loop cycles until the **fetch** statement fails to retrieve more tuples from CLOANS (lines 10–11). The value contained in ACCTVALUE is loaded into the working variable REMAMNT (line 12). 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 14 and 22). An inner loop is used to form the parcels based on the value of REMAMNT (lines 14–21). A new parcel row is inserted in the target table PAYMENTS for each iteration of the inner loop. The tuple is returned through a **pipe row** statement that is also responsible for padding the value of ACCTVALUE with zeroes (lines 16–17 and 23–24). When the inner loop ends, a **pipe row** statement is issued to return the last parcel, which contains the remainder.

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 6: Implementation test plan for one-to-many data transformation using the selected RDBMSs.

4 Experiments

We now compare the performance the alternative implementations of the one-to-many data transformations introduced in Examples 2.1 and 2.2 using non-recursive relational queries, recursive queries and table functions. 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.

The alternative implementations were tested on two RDBMSs henceforth designated as DBX and OEX[2]. The entire set of planned implementations is shown in Figure 6. 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.

4.1 Setup

The tests were executed on a synthetic workload that consists of the input relations 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 sumarize 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

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

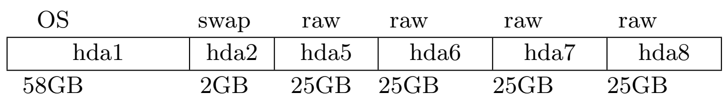


Figure 7: Hard-disk partitioning for the experiments

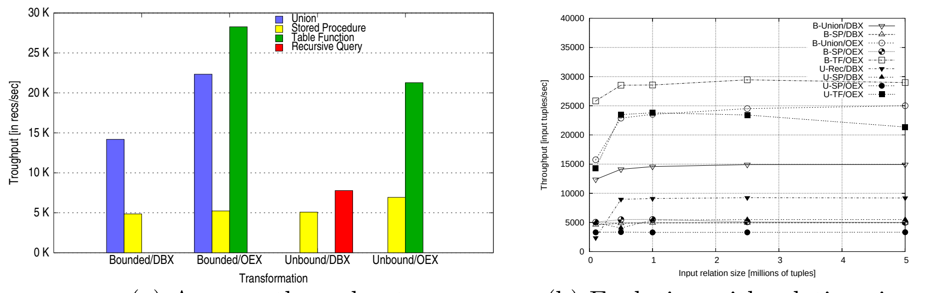
of the disk, I/O was forced through raw devices. The hard-disk is partitioned in cylinder boundaries as illustrated in Figure 7. The first partition is a primary partition formatted with Ext3 file system and journaling enabled and is used for the operating system and RDBMS installations as well as for 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 additon, the usage of kernel asynchronous I/O [Bhattacharya et al., 2003] 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* [Effelsberg and Haerder, 1984]. 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 [Gray et al., 1981, Mohan and Levine, 1992]. We attempt to minimize the logging activity by disabeling logging on both in DBX and OEX experiments. However, we note that logging cannot be disabeled in the case of stored procedured 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



(a) Average throughput (b) Evolution with relation size

Figure 8: 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.

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.

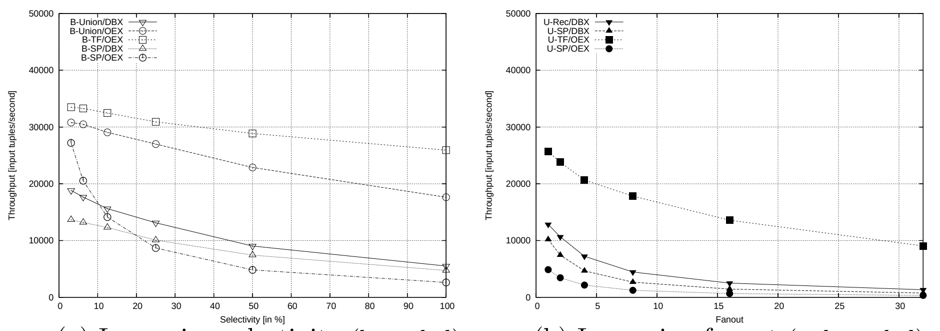
4.2 Throughput comparison

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 8. We observe that table functions are the most performant of the implementations. Then, implementations using unions and recursive queries are considerably more efficient than stored procedures. Figure 8b 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. The average number of output tuples produced for each input tuple is called *fanout* Similarly [Chaudhuri and Shim, 1993]. 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



(a) Increasing selectivity (bounded) (b) Increasing fanout (unbounded)

Figure 9: 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.

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 9.

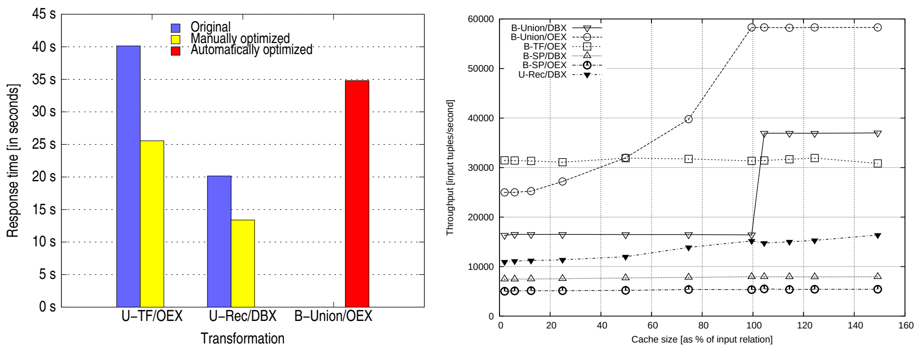
Concerning selectivity, we observe on Figure 9a that higher throughputs are obtained for smaller selectivities. This stems from having less output tuples created when the selectivity is smaller. The degradation observed is explained having more output tuples produced and materialized at higher selectivities. 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 9b 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 and execution

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,



(a) Original vs. optimized (b) Evolution with cache size

Figure 10: Sensibility of data transformation implementations with one 1M tuples: (a) to manual 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.

by pushing down the selection condition. Figure 10a 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 [Ordonez, 2005]. 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 10a 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 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 10b, 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 [Effelsberg and Haerder, 1984]. DBX uses the *least recently used* (LRU) [O'Neil et al., 1993] 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 [Jiang and Zhuang, 2002]. 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.

5 Related work

In Codd's original model [Codd, 1970], 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 [Shu et al., 1975, Shu et al., 1977], shortly followed by results on expressivity limitations of RA by [Aho and Ullman, 1979, Paredaens, 1978]. Many useful data transformations can be appropriately defined in terms of relational expressions, if we consider relational algebra equipped with a generalized projection operator [Silberschatz et al., 2005, 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* [Cunningham et al., 2004]. The performance of the pivot operator is not influenced by buffer cache size, unlike the chaining of multiple unions we present in Section 3.1. 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 [Aho and Ullman, 1979], and then by several works about recursive query optimization, like, for example [Shan and Neimat, 1991, Valduriez and Boral, 1986]. There are also proposals for extending SQL to handle particular forms of recursion [Ahad and Yao, 1993], like the Alpha Operator [Agrawal, 1988]. Despite being relatively well understood at the time, recursive query processing was not supported by SQL-92. By the time the SQL:1999 [Melton and Simon, 2002] 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 [Pieciukiewicz et al., 2005].

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 [Raman and Hellerstein, 2001], Ajax [Galhardas et al., 2001] and Data Fusion [Carreira and Galhardas, 2004a]. These tools have proposed operators for expressing one-to-many data transformations. Potter'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 [Carreira and Galhardas, 2004b, Carreira et al., 2005, Carreira et al., 2006]. 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 organize 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 [Melton and Simon, 2002] and table functions introduced in SQL 2003 standard [Eisenberg et al., 2004]. 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 input relations size, these factors can be used to predict the cost of one-to-many data transformations. This information can be exploited to 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 [Carreira et al., 2005, Carreira et al., 2006]. As future work, we plan to extend the Derby [Apache, 2006] 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.

References

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

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

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

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

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

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

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

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

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

[Carreira et al., 2006] Carreira, P., Galhardas, H., Lopes, A., and Pereira, J. (2006). One-to-many transformation through data mappers. *Data and Knowledge Engineering Journal.*

[Chaudhuri and Shim, 1993] Chaudhuri, S. and Shim, K. (1993). 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.

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

[Cunningham et al., 2004] Cunningham, C., Graefe, G., and Galindo-Legaria, C. A. (2004). 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.

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

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

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

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

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

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

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

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

[Mohan and Levine, 1992] Mohan, C. and Levine, F. (1992). 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.

[O'Neil et al., 1993] O'Neil, E. J., O'Neil, P. E., and Weikum, G. (1993). 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.

[Ordonez, 2005] Ordonez, C. (2005). 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.

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

[Pieciukiewicz et al., 2005] Pieciukiewicz, T., Stencel, K., and Subieta, K. (2005). 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.

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

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

[Shan and Neimat, 1991] Shan, M.-C. and Neimat, M.-A. (1991). 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.

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

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

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

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