RDBMS support for one-to-many data transformations

Nuno G. Matos[1,2] (Thesis Student)
Helena Galhardas[1,2] (Thesis Supervisor)
João D. Pereira[1,2](Thesis Co-Supervisor)

[1] Instituto Superior Técnico
[2] INESC-ID

Resumo. As transformações de dados são essencias no contexto de problemas como a migração de dados, integração de dados e em "data warehousing". Neste tipo de problemas, a transformação de dados é a principal unidade de trabalho. As transformações um-para-muitos [6] constituém uma classe particular de transformação de dados em que cada tuplo de entrada, individualmente, é utilizado para gerar múltiplos tuplos de saída. Neste trabalho, descrevemos esta classe de transformações de dados, os contextos em que elas são úteis, e como são suportadas nos Sistemas de Gestão de Bases de Dados Relacionais (SGBD). Além disto, comparamos três SGBD em termos do seu desempenho e poder expressivo, neste contexto.

Abstract. Data transformations are essential in the context of data migration, data integration and data warehousing. A data transformation is the fundamental unit of work in each of these processes. One-to-many data transformations [6] constitute a particular class of data transformations that produce several output records for each input record. We describe this class of data transformations, the contexts in which they are useful, and their support by Relational technology. Furthermore, we compare three different Relational Database Management Systems (RDBMS) in regard to their performance and expressive power when faced with one-to-many data transformations.

1 Introduction

Data transformations consist in the conversion of input data into output data that satisfies the requirements of the most various challenges, in several Information Technology contexts. Namely, in data migration, data integration, and in data warehousing. Different technologies such as Relational Database Management Systems (RDBMS) (e.g. Oracle), extract-transform-load (ETL) and data migration tools (e.g. Data Fusion), data cleaning tools (e.g. AJAX), programming languages (e.g. Java), XQuery and XSLT, can be used to transform data [5, 13, 26, 2]. Each of these technologies has special affinities to specific domains of problems, for example RDBMS aim to support the secure management of large volumes of data, while allowing the efficient retrieval and update of data by business transactions and decision support systems.

In our work, we were interested in studying a seldom discussed class of data transformations, known as one-to-many (1-N). These data transformations are informally described for processing each input record into multiple output records.

1.1 Motivating Example

Social Networks are Web portals which grant, to their users, the access to various contents and services. For example, storing personal information (e. g. pictures, favorite sports, music tastes) and supporting relationships among users (e.g. favorite friends). We assume the existence of a fictional social network called Face5. This company stores its business information in a relational database that contains the relation `PROFILE=(ID, COUNTRY, FAVARTISTS, FAVF1, FAVF2, FAVF3, FAVF4)`, where `FAVFi:FK(ID)`. The attribute `ID` is the primary key of the relation, and the attributes `FAVF1` to `FAVF4` are foreign keys referencing `ID`. Table 1 illustrates the first two records of the `PROFILE` relation instance. This relation stores the personal information of each user (e.g.

the four favorite friends, favorite artists and country). In the following two scenarios, we describe two operational challenges of the Face5 company, which are used to introduce the two subclasses of one-to-many data transformations: *bounded* and *unbounded*.

ID	COUNTRY	FAVARTISTS	FAVF1	FAVF2	FAVF3	FAVF4
1	PT	Placebo, Billie Holiday, Amalia	2	3	5	
2	ENG	SIA, The Cure, KIN, Nine Inch Nails	1	4	10	3

Table 1. PROFILE relation instance, a subset of its data.

Scenario 1. In Face5, every user may promote to the title of favorite friend up to four of their friends. The limit restriction avoids a user's home page from being overwhelmed by a large number of top friends (that have to be displayed), thus reducing server load and improving page presentation. This year the directors of Face5 decided that it would be a good idea to give Christmas gifts to their most liked users. The user who is listed more often as a top friend is the user that is considered to be the most liked. Having defined a way to measure how much a person is liked, Face5 wants to query the database for the list of most liked people.

The relation PROFILE contains the information required for answering the desired query, namely in the attributes ID and FAVF1 to FAVF4. However, the data in this instance is not in an adequate format to easily allow the construction the query. Instead, if the relevant information in the PROFILE relation could be transposed into a target output relation with schema FAVFRIEND=(USRID, FAVF), then querying for the most liked users would be trivial (e.g. count users grouped by favorite friend). Table 2 illustrates the FAVFRIEND relation instance corresponding to the output of the data transformation of this scenario over input relation PROFILE of Table 1. Notice that each input record yields at most four output records, one for each favorite friend.

USRID	FAVF
1	2
1	3
1	5
2	1
2	4
2	10
2	3

Table 2. FAVFRIEND relation instance, used to determine the most liked friend.

Scenario 2. Face5 has started covering news about artists from all over the world, since users in general like music and want to know what their favorite artists are doing. With this in mind, Face5 directors considered that some users might appreciate being automatically notified with news concerning their favorite artists. A reporting feature is being implemented for this purpose and users will be able to choose whether they want it active. Face5 will determine if a user is interested in a news article by querying the user's favorite artists list. When an artist mentioned in an article is part of the user's favorite artist list, then the user is dimmed as interested in the article.

Once more, the relation PROFILE contains the information required for answering the desired query, namely in the attributes ID and FAVARTISTS. Decoupling the information nested in the FAVARTISTS field would make this query much simpler. We want to transpose the relevant information in the source relation PROFILE into a target output relation with schema FAVARTIST=(USRID, ANAME), where each record pairs one user to one favorite artist. Table 3 illustrates the FAVARTIST relation instance corresponding to the output of the data transformation of this scenario over input relation PROFILE of Table 1. Notice that each input record yields an unknown number of output records, one for each favorite artist.

USRID	FAVARTIST
1	Placebo
1	Billie Holiday
1	Amalia
2	SIA
2	The Cure
2	KIN
2	Nine Inch Nails

Table 3. `FAVARTIST` relation instance, used to determine the artists that users like.

1.2 Data transformations in the different technologies

The two scenarios described in the previous section introduced two similar data transformations, which are classified as one-to-many. In the first scenario, we illustrated a one-to-many data transformation where each input record was independently processed into at most four output records, one for each favorite friend specified by the user. In the second scenario, each input recorded yielded as many output records as the favorite artists of the user.

What determines the types of data transformations supported by the different data transformation technologies is the expressive power of their data transformation languages. Programming languages are general purpose Turing Complete tools. They can process any computable data transformation, by means of ad-hoc procedural solutions, which are hard to optimize, and can take long to implement. ETL and data cleaning tools are specialised in transforming data, but there are many tools in widespread use, each with its own data transformation language, and they suffer from the lack of standards, which makes it hard to evaluate their expressive power [25]. RDBMS have a common data model and support a declarative, standard query language (SQL), which can process many types of data transformations with simplicity and efficiency [19, 20]. Namely, one-to-one, one-to-zero, many-to-one, Cartesian products, and transitive closures data transformations are all properly supported by the projection, selection, group by, join and recursive query operators [3, 17].

We focused our study on the support of the support of one-to-many data transformations in RDBMS, since these systems are the main repositories of enterprise data; they have the means to transform the data they are responsible for; they adhere to standards, with their common operators being highly efficient having been improved with the years; and these transformations are useful to solve specific database problems, such as restructuring relations suffering from inadequate data schemas, as we will discuss in Section 2.

1.3 Contributions

With this work we tried to achieve the following goals. First, to assess the different implementation methods supported by three commercial RDBMS that can be used to implement one-to-many data transformations, and their expressive power in this context. Second, to describe how each method can be used to implement solutions for these problems, and their code complexity. Third, to analyse the computational resources consumed by each method. Fourth, to study how they are affected by changes of each individual *critical parameter*, and to compare the performance of the best implementation methods of the three database systems with one-another.

1.4 Outline

This document is comprised by six additional sections. Section 2 identifies the two subclasses of one-to-many transformations, introduces the fundamental concepts of *fanout* and *critical parameters*, and enumerates domains of use for these data transformations. Section 3 summarily describes theoretic operators that were proposed as extensions to the relation algebra that could be used to implement subsets, or fully, one-to-many data transformations. Section 4 enumerates the different implementation methods that can be used to implement one-to-many data transformations in modern RDBMS. Section 5 describes the computation resource consumption profile of each implementation method. Section 6 resumes the benchmark results of the several experiments undertaken in each RDBMS, comparing the performance of the best implementation method of each RDBMS in each of the experiments. Section 7 presents the conclusions that resulted from our work and suggests possible future work to be undertaken in this domain of study.

2 One-to-many data transformations

Previously, we informally introduced the concept of one-to-many data transformations and accompanied it with two practical examples of their application. In this section, we formalize their definition; introduce their subclasses (*bounded* and *unbounded*); present the concepts of *fanout* and *critical parameters*; and identify three domains of their use.

2.1 Concepts: one-to-many data transformations, their subclasses, *fanout* and *critical parameters*

A data transformation is considered *one-to-many* whenever each tuple in the output relation is associated to a single tuple in the input relation, and at least one input tuple has yielded two or more output records. The data transformations described in Scenario 1 and 2 are both one-to-many data transformations.

Any one-to-many data transformation can be classified as either *bounded* or *unbounded*. In *bounded* data transformations a limit k for the maximum *fanout*, which is the number of output records yielded by each input record, is known, such as in Scenario 1 . In *unbounded* data transformations the maximum *fanout* is unknown, such as in Scenario 2 . This distinction is relevant, because *bounded* one-to-many transformations can be accomplished using only standard relational algebra operators [6]. The *unbounded* can only be accomplished through extensions to the RA.

We denote the data transformations of the Scenario 1 and 2 by $E2$ and $E3$, respectively. In our thesis these two expressions are defined mathematically using the *mapper* notation proposed in [5], since there is no standard notation to express these data transformations in RDBMS.

Three *critical parameters* affect the performance of one-to-many data transformations: *(i)* the number of records of the input relation (*ntups*); *(ii)* the *selectivity* of predicate clauses over the input records; and *(iii)* the average *fanout* of a data transformation [5]. By increasing *ntups*, a data transformation has to process more data. Thus, the overall cost of the data transformation should increase. Likewise, increasing *selectivity* means that more input records satisfy the predicate clauses of a query. Consequently, more records of have to be processed. The *ntups* and *selectivity* parameters are similar, they determine the work batch of an implementation method processing a data transformation. Finally, the *fanout* of a data transformation should influence the amount of time the algorithm takes to process each individual record. The larger the *fanout*, the longer it should take to produce the output records.

2.2 Domains of use of one-to-many data transformations

In terms of their domain of application, we denote three types of relations over which these data transformations may prove most useful. First, *relations with semantically equivalent attributes*. These relations contain multiple attributes for storing data values with equivalent semantic meaning (e.g. various telephone number or favorite friend attributes). It may be necessary to export such data into a target output relation where only one attribute exists associated to the corresponding semantic domain, such as in Scenario 1 . Second, *relations with multivalued data*. These relations contain attributes whose data values can be perceived as sets of distinct data elements (e.g. a string with a list of artists). These relations may have to be restructured through one-to-many data transformations in order to decouple the data contained in each tuple, such as in Scenario 2 . Third[3], *relations with aggregated data*. These relations contain attributes whose data values can be perceived as the result of an SQL `GROUP BY` operation (e.g. converting the annual earnings of an enterprise into the respective trimester earnings). Since `GROUP BY` operations are many-to-one data transformations, their inverse is obviously one-to-many. However, inverting these operations can only be approximated, since we cannot determined the original values that generated an aggregated value.

[3] In the Chapter 4 of the master thesis, we present a third scenario to illustrate one-to-many data transformations used in this context. In this article, we discuss a simplified motivating example.

3 Theoretic operators supporting one-to-many data transformations

Investigation in the field of database theory has many times been focused in proposals of extensions to the relational data model and algebra, for the so called *nested database* models [15, 1]. One way or another, these theoretic database models are meant to support non-first-normal-form (N1NF) relations, by allowing attribute values to be relation instances (non atomic). Incidentally, such relations are very similar to the *relations with multivalued data* which we defined in Section 2.2. The difference being that from a relational database perspective a string is just an atomic value, whereas in a nested database the several elements of a relation valued attribute are distinguishable. Per chance, these new database models normally propose new operators (e.g. the *unnest*) that can transform N1NF relations into the first-normal-form (1NF) [15]. The resulting data transformations are quite identical to that our $E3$ transformations of Scenario 2 . Their theoretic application in the context of one-to-many data transformations is limited to unnesting data from relation valued attributes.

Presently, the only theoretic operator proposed with the intention of supporting this class of data transformations is the *mapper* [5]. Naturally, it applies to the standard relational data model and it fully supports *bounded* and *unbounded* data transformations. Evidentially, it can address problems in any of the three domains of use for these data transformations described in Section 2.2.

4 Summary of the RDBMS implementation methods for one-to-many data transformations

In order to implement one-to-many data transformations in a commercial RDBMS, we have to make due with the operators they do support. We have identified that the following implementation methods can be of use for this purpose: the *selection-projection-union* (*SPU*), *unpivot*, *unnest*, *table function*, *recursive query* and *model*. Table 4 summarizes the expressive power and code complexity (CC) of these implementation methods, and the RDBMS that support them.

The *SPU* can process the *bounded* one-to-many and is simple to implement [5]. We implement the data transformation as a set of projections and selections (as many as the maximum *fanout* of the data transformation), and unite the results yielded by each of them[4].

The *unpivot* can process a subset of *bounded* data transformations, namely those involving the restructuring of *relations with semantically equivalent attributes*. This operator transforms a set of specified columns into rows. Consequently, each input record is transformed into as many output records as the columns that were suppressed. The operator does not extend the expressive power of the standard relational algebra, but performs this type of operation efficiently [8]. The code complexity of this method is simple.

The *unnest* can process *bounded* data transformations. This method transforms an array of values into tables [11]. Since arrays require a maximum array size definition, or an explicit enumeration of its elements, we must know the maximum *fanout* of a data transformation to use it (even if the array is being populated and returned by a procedural function in the RDBMS). The code complexity is simple and familiar to programmers.

The *model* can process *bounded* and *unbounded* data transformations. This method transforms the input relation into multi dimensional arrays, and afterwards transforms the arrays back into a relation instance. In this process a set of specified attributes is used as the index dimensions of the arrays, and another as the values held by the arrays. The data transformation is processed by a set of rules which specify the index positions of the arrays whose values we wish to affect. These rules also allow the assignment of values to non existent index positions of an array, which results in the creation of new output records. To process the *unbounded* data transformations with this method, we define rules that recursively compute values that are assigned to index positions created dynamically. The code complexity is very high, piratically incomprehensible without a large time investment in learning the operator, its complex syntax and the non evident solution.

[4] Each of the implementation methods mentioned in this article is explained in detail and mathematically in the master thesis. There, we also present coded solutions for the schema mapping expression $E2$ and $E3$ described here.

The *table function* can process *bounded* and *unbounded* data transformations. This method allows complex record-by-record processing. For each input record, the *table function* enters a processing cycle, where each iteration produces one output record. In Oracle, the table function can be invoked as the inner table of a join, receiving the relevant attribute values for its input variables from the input record of the outer table. Thus, it does not need a cursor over a table to perform record-by-record processing. In MSQLS, the *table function* has to open cursor over the input table to perform its record-by-record processing, rendering the solution purely procedural. The code complexity is simple, and familiar to any programmer.

The *recursive query* can process *bounded* and *unbounded* data transformations. This mechanism resembles the *SPU* since the solution results in the execution of multiple one-to-one data transformations whose results are united.A recursive query is executed in several iterations. Each iteration applies a one-to-one data transformation over the results produced in the previous iteration, and stops when the results produced are an empty relation instance. This should only occur in the iteration max $fanout+1$ due to a predicate that becomes unsatisfiable. The solutions are complex, requiring significant learning time of the operator and methodology.

IMP. METHOD	ORACLE	MSQLS	DB2	CC	Legend	
SPU	B	B	B	S	B	*bounded*
unpivot	$\bar{\text{B}}$	$\bar{\text{B}}$		S	$\bar{\text{B}}$	subset of *bounded*
unnest	B		B	S	U	*unbounded*
model	BU			V	S	simple
table function	BU	BU		S	C	complex
recursive query		BU	BU	C	V	very complex

Table 4. RDBMS implementation methods expressive power and code complexity (CC).

5 Computational resources consumption profiles

One of our concerns was to have an understanding of the computational resources consumption associated to the physical execution algorithm of each implementation method. Table 1 resumes our evaluation of the resource consumption profiles of the different implementation methods. We evaluated the *cpu time* costs of each method, based on the average *cpu time* spent to process the 1 Million output records for the data transformation $E2$. This data transformation was chosen because its processing cost per input record was the smallest of the two data transformations $E2$ and $E3$. It consisted simply in the rotation of its `FAVFi` data values of each input record. Therefore, this data transformation maximizes the overall *cpu time* weight of the individual operators composing the query execution plan of each method (e.g. table scans), in the final *cpu time*.

The *SPU* has a cheap query execution plan in terms of its *cpu time* cost, in Oracle and MSQLS. However, it is IO inefficient since it has to logically read the source table a number of times equal to the maximum *fanout* of the data transformation. For each projection coded in an *SPU* implementation, the RDBMS performed one table scan of the input relation.

The *unpivot* resource consumption profile was perfect, with low *cpu time* cost, and only reading the source table once. Its only limitation, as explained, is its expressive power.

The *unnest* had low *cpu time* cost, which was borderline high (much higher than the *SPU* and *unpivot*). In terms of IO, its resource consumption profile was perfect.

The Oracle *table function* had high *cpu time* cost, but its IO resource consumption profile was perfect. The MSQLS *table function* had a distinct behaviour. Its *cpu time* cost was very high, and its IO profile involved reading source table once, and materializing the *table function* output results into auxiliary memory and fully logging the materialization operations.

The *model* had very high *cpu time* cost, and it has to convert the input relation, which it only reads once, into a temporary multi dimensional array structure which it materializes using auxiliary memory space. Once the model finishes its processing steps, the multi dimensional array is converted back into a relational table object.

The *recursive query* read the source table once while materializing its output results into auxiliary memory space. This method is associated to a complex query execution plan, which is identical in the two RDBMS, and which does not constitute a tree of operators since it has cycles. It involves

IMP. METHOD	CPU COST	SRC IO	TEMP IO	LOG IO
SPU	low	*fanout* reads	negligible	negligible
SPU (DB2)	high	*fanout* reads	negligible	negligible
unpivot	low	1 read	negligible	negligible
unnest	low	1 read	negligible	negligible
table function (Oracle)	high	1 read	negligible	negligible
table function (MSQLS)	very high	1 read	materialized	logged
model	very high	1 read	special	negligible
recursive query	NA	1 read	materialized	negligible

LEGEND:

CPU COST `low` if the *cpu time* slope of the $E2$ varying *ntups* experiment was less than 1.5 sec per million of output records; `high` between 1.5 and 7 seconds; and `very high` more than 7 seconds. `NA` not applicable to *recursive queries*, as they were not used to implement $E2$.

SRC IO Describes the number of times the source table is logically read by the implementation method.

TEMP IO `negligible` if the operator does not perform any significant IO in the temporary tablespace; `materialized` if the output results of the operator may be written into the temporary tablespace; `special` the *model* method may write into the temporary tablespace the input that it reconverts into a multi-dimensional array before processing a query. Notice the use of the word "may" in the previous statements, which implies that full - or partial - output materialization of an operator only occurs if the database cache memory is insufficient to hold it.

LOG IO Identifies if the implementation method does not enforce logging.

Fig. 1. Summary of the different profiles of computational resources consumption by the RDBMS implementation methods. We only specify the RDBMS supporting the implementation method, if two or more RDBMS support the same method with significantly different resource consumption profiles (e.g. Oracle and MSQLS *table function* methods).

a processing step where results are written to an output temporary file, which in turn also serves as input of this processing step.

6 RDBMS benchmarks

We developed a benchmark for one-to-many data transformations, in order to evaluate: *(i)* the individual effect of each *critical parameter* on the performance of the various implementation methods that support them; and *(ii)* to compare three commercial RDBMS on their performance of these data transformations, namely Oracle 11gR2, SQL Server 2008 and DB2 UDB for LUW 9.7. The benchmark was comprised by several experiments. Each experiment targeted a single data transformation (e.g: $E2$ and $E3$[5]) and measured the effect of only one critical parameter on the performance of each implementation method supported by the RDBMS under evaluation. Each individual experiment was comprised by four tests, where the critical parameter whose effect was being tested was increased exponentially from test-to-test, while the other two critical parameters remained constant. In turn, each test was executed five times. The statistical values used in our analysis and charts correspond to the average of the values they assumed in each of the five runs. The details on the generation of input data for each test are covered in the thesis.

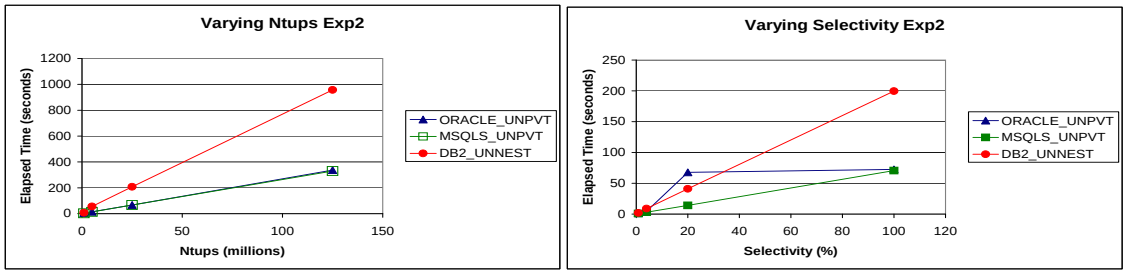
To ensure the fairness of our experiments, we equalized as much as possible the configuration of all RDBMS. Summarily, we controlled details such as: the database *block size*; *buffer cache size*; use of *raw partitions* to hold common data objects to all RDBMS in identical physical locations, and imbue the databases with responsibilities over file fragmentation and buffered data access; and reduced database *logging* for bulk operations to the minimum possible amounts.

In this section, we compare the performance demonstrated by the best performing method of each RDBMS in each of the six experiments concerned with the expression $E2$ and $E3$. First, we consider the varying *ntups* and *selectivity* experiments. These experiments have in common the

[5] In the master thesis we consider a third expression $E4$, but since the data transformation and obtained results are identical to those of $E3$, we omit them.

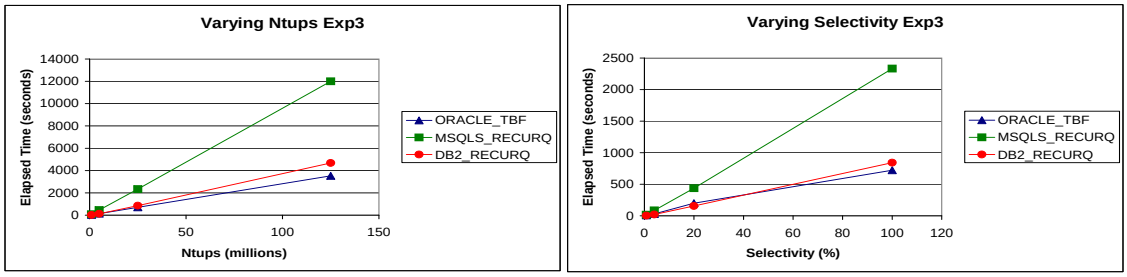
fact that the cost of processing each input record is constant and specific to each implementation method. Their difference resides only in the mechanism used to control the number of records that have to be processed. The varying *ntups* experiment controls the number of records processed by changing the size of the source table; and the varying *selectivity* experiment achieves the same thing by means of a predicate clause. Consequently, the best performing method of a given RDBMS in a specific varying *ntups* experiment should also be the best performing method in the corresponding varying *selectivity* experiment. Second, we analyse the *elapsed time* results of the best implementation method of each RDBMS in the varying *fanout* experiments.

Figures 2 and 3 present the results of the best performing implementation method of each RDBMS in the four varying *ntups* and *selectivity* experiments. As expected, the best performing implementation method of each RDBMS was the same in both experiments.



(a) Best RDBMS $E2$ methods when varying *ntups* (b) Best RDBMS $E2$ methods when varying *selectivity*

Fig. 2. Charts illustrating the best implementation method of each RDBMS for the varying number of records and *selectivity* experiments over $E2$.



(a) Best RDBMS $E3$ methods when varying *ntups* (b) Best RDBMS $E3$ methods when varying *selectivity*

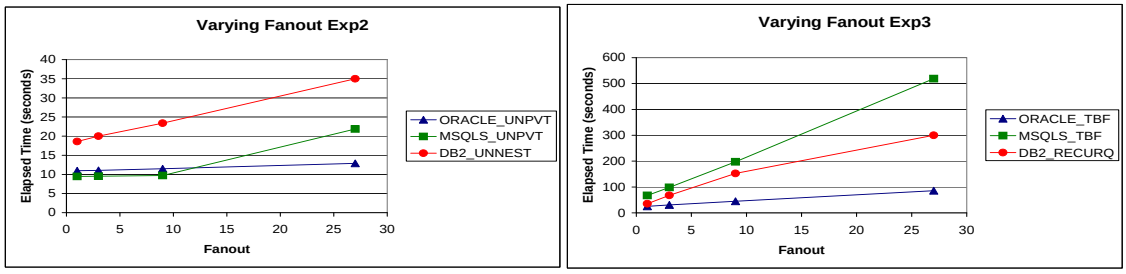
Fig. 3. Charts illustrating the best implementation method of each RDBMS for the varying number of records and *selectivity* experiments over $E3$.

In the varying *ntups* and *selectivity* experiments over the expression $E2$, see the Figure 2 (a) and (b), the Oracle and MSQLS databases presented identical results. The *unpivot* methods of the two RDBMS were equally efficient. The only substantial difference occurred in varying *selectivity* experiment for the *selectivity* value of 20%. In this test, Oracle did not use its clustered index to retrieve the data pages that satisfied the predicate clause, choosing to table scan the source table. Consequently, Oracle incurred in unnecessary IO, which lead to a considerable degradation of the performance of the operator in that test in particular. Overall, the two database systems were equally good in these two experiments, though MSQLS was more intelligent in its data access methods choices. The DB2 *unnest* method was also efficient, with an average elapsed time cost of 1.8 seconds to produce 1M output records. Even so, this value was considerably higher than the approximately 0.5 seconds consumed by the *unpivot* methods of Oracle and MSQLS. DB2 consumed more *cpu time* and incurred in more *wait time* than the other two. In general, DB2 also used more space to materialize both source, target and temporary data, than MSQLS and Oracle.

In the varying *ntups* and *selectivity* experiments over the expression $E3$, see the Figure 3, the *elapsed time* for all concerned implementation methods increased linearly with the varying factor. Oracle was the most efficient database system presenting the smallest *elapsed time* cost with its *table function* implementation. However, the DB2 *recursive query* was sill competitive. Oracle consumed in average 5.6 seconds to produce 1M output records, the DB2 *recursive query*

consumed 6.9 seconds. While the DB2 *recursive query* consumed less *cpu time* than the Oracle *table function*, it performed significant temporary IO that the Oracle *table function* did not[6]. Finally, the MSQLS *recursive query* was considerably more expensive than the methods of the other two database systems. Its *elapsed time* in the larger tests (e.g. *ntups* 125M and *selectivity* 100%) was approximately 3 times the *elapsed time* of the Oracle *table function*. The MSQLS *recursive query* had inferior IO costs than the DB2 *recursive query*, with MSQLS performing approximately 22 GB of temporary read and write IO, while DB2 performed 84 GB. Despite this, both system had identical *wait time* costs. In terms of *cpu time*, the MSQLS *recursive query* was extremely costly, its *cpu time* was approximately 4 times that of the DB2. It consumed in average 18.9 seconds to produce 1M output records. Consequently, Oracle outperformed the other two database systems because its *table function* was associated to a simple query execution plan with optimal IO cost: neither the *cpu time* nor *wait time* were too high; the source table was read only once, it did not use temporary data, and minimal logging was incurred.

Figure 4 presents the results of the best performing implementation method of each RDBMS in each of the two varying *fanout* experiments. Concerning the expression $E2$, see Figure 4 (a), the best performing implementation method in Oracle and MSQLS was the same, the *unpivot*. However, they exhibited distinct behaviors. Oracle linearly increased its *elapsed time* from test-to-test, resulting from the fact that its *wait time* remained approximately constant while the *cpu time* increased slowly. MSQLS maintained its *elapsed time* approximately constant, resulting from the fact that the *cpu time* increase was being compensated by an equal valued *wait time* decrease. As in the varying *selectivity* and *ntups* experiments over expression $E2$, the two systems presented practically identical *elapsed time* performance from test 1 to test 3. However, in the final test the *unpivot* suffered an inexplicable *cpu time* increase which was not linear with the *fanout*, and which provoked the spike in the *elapsed time*. The *cpu time* of the MSQLS *unpivot* in the final test should have been close to 18 seconds. Instead, it was close 27 seconds (almost 10 seconds over our expectations), which practically corresponded to the *elapsed time* increase that we see in the final test. The DB2 *unnest* method was not as efficient as the *unpivot* of the other two database systems: its *cpu time* increased very slowly with the *fanout*, however, unlike in Oracle and MSQLS, its *wait time* increased steadily with the *fanout*, since the write operations incurred in delays, and their number linearly increased with the *fanout*. Given the almost identical *elapsed time* performance of the Oracle and MSQLS *unpivot* method in the first three tests, and the better Oracle performance in the final test, we consider that Oracle was the database system that implemented this data transformation most efficiently.



(a) Best RDBMS $E2$ Methods (b) Best RDBMS $E3$ Methods

Fig. 4. Charts illustrating the best implementation method of each RDBMS for the varying *fanout* experiments.

In the varying *fanout* experiments over the expressions $E3$, see the Figure 4 (b), the best performing implementation method in Oracle was the *table function*. In MSQLS, it was also the *table function*. DB2 was alike in *elapsed time* performance to MSQLS in the experiment over the expression $E3$, despite having performed significantly more read and write IO in the temporary raw partition. However, its temporary IO increased less than linearly with *fanout*, and thus its *elapsed time* also increased less than linearly.

Overall, the Oracle *table function* was better in handling the $E3$ expression. This is due to the fact that the *recursive query* is an expensive implementation method, as explained before,

[6] Recall that *recursive queries* materialize the results of each processing cycle.

while the MSQLS implementation of the *table function* also materializes the output results into the temporary raw partition, all the write IO it performs on it is fully logged, and it also has a very high cpu cost. Consequently, the Oracle *table function* was the best performing implementation method in the experiment.

7 Conclusions

We concluded that both *bounded* and *unbounded* one-to-many data transformations are supported by modern commercial RDBMS, by means of different implementation methods. The *unbounded* are not supported by any implementation method common to all three RDBMS, while the *bounded* are commonly supported exclusively by the *SPU*, which has an inefficient query execution plan. In the conditions our benchmarks were performed, the *critical parameters ntups*, *fanout* and *selectivity*, each affected the performance of these data transformations linearly, regardless of the implementation method. The DB2 *recursive query* in the varying *fanout* experiment was the only exception to this generalisation, since for the $E3$ data transformation the temporary IO performed by this method is the sum of an arithmetic progression, where in each processing cycle the number bytes written per tuple is reduced in size of one favorite artist. Oracle and MSQLS were the RDBMS with the best performance implementing the *bounded*, both using the *unpivot*. Finally, Oracle and DB2 had the best performance implementing the *unbounded*, using the *table function* and *recursive query*, respectively.

7.1 Future Work

RDBMS support adequately one-to-one, one-to-none, many-to-one, Cartesian products and transitive closures, by means projections, selections, group by, all manners of joins and recursive queries, respectively. However, they do not support in any adequate manner one-to-many data transformations. For this reason, we consider that an important practical work in this field would be to implement a specialised one-to-many operator in a modern RDBMS using the contributions given in [5]. The study we applied over the support of one-to-many data transformations in commercial RDBMS should be continued for open-source database systems. The benchmark used in this work should be refined. It could be extended to encompass the data transformation $E3$ in a *bounded* scenario. This could be fruitful in determining whether the *bounded* implementation methods continue to outperform the *unbounded*, as concluded with the benchmarks over $E2$; or in determining if MSQLS continues to be among the best RDBMS solving the *bounded* data transformations when it cannot solve them by means of the *unpivot*. Also, a new set of experiments should be undertaken, varying both the *ntups* and *fanout critical parameters* together, so as to confirm if their joint variation yields a quadratic influence in the implementation methods performance behaviour.

References

1. Abiteboul, S., Bidoit, N.: Non first normal form relations to represent hierarchically organized data. In: PODS '84: Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems. pp. 191–200. ACM, New York, NY, USA (1984)
2. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann, San Francisco, CA, USA (1999)
3. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
4. Bohannon, P., Elnahrawy, E., Fan, W., Flaster, M.: Putting context into schema matching. In: VLDB '06: Proceedings of the 32nd international conference on Very large data bases. pp. 307–318. VLDB Endowment (2006)
5. Carreira, P.: Mapper: An Efficient One To Many Operator. Ph.D. thesis, Universidade de Lisboa, Faculdade de Ciências (2007)
6. Carreira, P., Galhardas, H., Lopes, A., Jo a.P.: One-to-many data transformations through data mappers. Data Knowl. Eng. 62(3), 483–503 (2007)
7. Codd, E.: A relational model of data for large shared data bases. Comm. ACM 13(6), 377–387 (1970)

8. Cunningham, C., Galindo-Legaria, C.A., Graefe, G.: Pivot and unpivot: optimization and execution strategies in an rdbms. In: VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases. pp. 998–1009. VLDB Endowment (2004)
9. Darwen, H.: In reply to domains, relations and religious wars. SIGMOD Rec. 25(4), 6–7 (1996)
10. Date, C.: Date on database: writings 2000-2006. Apress (2006)
11. Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.E., Zemke, F.: Sql:2003 has been published. SIGMOD Rec. 33(1), 119–126 (2004)
12. Elmasri, R.A., Navathe, S.B.: Fundamentals of Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
13. Galhardas, H., Florescu, D., Shasha, D., Simon, E., Saita, C.A.: Declarative data cleaning: Language, model, and algorithms. In: Apers, P.M.G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., Snodgrass, R.T. (eds.) VLDB. pp. 371–380. Morgan Kaufmann (2001)
14. Iyengar, S., Sudarshan, S., 0002, S.K., Agrawal, R.: Exploiting asynchronous io using the asynchronous iterator model. In: Das, G., Sarda, N.L., Reddy, P.K. (eds.) COMAD. pp. 127–138. Computer Society of India / Allied Publishers (2008)
15. Jaeschke, G., Schek, H.J.: Remarks on the algebra of non first normal form relations. In: PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems. pp. 124–138. ACM, New York, NY, USA (1982)
16. Kepser, S.: A simple proof for the turing-completeness of xslt and xquery. In: Extreme Markup Languages (2004)
17. Libkin, L.: Expressive power of sql. Theor. Comput. Sci. 296(3), 379–404 (2003)
18. Mehta, D.P., Sahni, S.: Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.). Chapman & Hall/CRC (2004)
19. Melton, J., Simon, A.: SQL: 1999-Understanding Relational Language Components. Morgan Kaufmann (2002)
20. Melton, J.: Understanding SQL's stored procedures: a complete guide to SQL/PSM. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1998)
21. ORACLE: Oracle Database SQL Language Reference 11g Release 1 (11.1) (Semptember 2008), uRL: http://www.oracle.com/pls/db102/homepage
22. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill Science/Engineering/Math (2002)
23. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database Systems Concepts. McGraw-Hill Science/Engineering/Math, fourth edn. (October 2001), http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0072554819
24. Songini, M.L.: ETL. Computerworld 38(5), 23 (2004)
25. Vassiliadis, P., Karagiannis, A., Tziovara, V., Simitsis, A.: Towards a benchmark for etl workflows. In: Ganti, V., Naumann, F. (eds.) QDB. pp. 49–60 (2007)
26. Walmsley, P.: XQuery. O'Reilly Media, Inc. (2007)