INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

RDBMS support for one-to-many data transformations

Nuno Henrique Castro Melo Godinho de Matos

Dissertation for obtaining the Master's degree in
Informatics and Computer Engineering

Jury

President:	Doctor Alberto Manuel Ramos da Cunha
Supervisor:	Doctor Helena Isabel de Jesus Galhardas
Co-Supervisor:	Doctor João Carlos Serrenho Dias Pereira
Examiner:	Doctor Luís Antunes Veiga

Lisbon, 17th June 2010

Acknowledgements

Many thanks to my professors, Helena Galhardas and João Pereira, for guiding me throughout this project, being understanding and tolerant, critic and rigorous, and helping me greatly when I needed. To professor Paulo Carreira, whose contributions in this field were invaluable to me, and from whom I received many great suggestions and help, some of which I did not follow and later came to regret. To professor Andreas Wichert, for his interest, insight and counsel. To Bob Ward, from Microsoft, for helping me understand the IO wait events through latches in SQL Server. To Radhesh Kumar, IBM Certified DB2 Advanced DBA, for helping me figure out how to properly monitor the DB2 UDB statistics that I needed. To the Oracle, SQL Server and DB2 communities, which actively participate in the respective official database forums, where I countless times found answers to various problems I encountered. To Tom Kyte, from Oracle, who has answered countless questions, many of which could have been my own.

Lisbon, 17th June 2010

Nuno Henrique G. de Matos

Dedicated To

To my loving parents, Nuno and Telma, for being patient, supportive and caring, to whom I hold a debt impossible to ever repay. To my grand father Telmo Lopes, for his caring, for the many meals and miles together, and for the comfort of his presence in my life. To my brother Miguel Matos, whose opinions are almost always right, who is the brightest beacon of intelligence, commitment and responsibility in this earth, who is a permanent source of conflict always censoring me, and who occasionally lends a helping hand. To my friend José Chumbo, for enduring my company even when I was most desperate and unbearable. To my dog Anima, and the rest of my family, because you are all very important to me.

I'm writing on a little piece of paper
I'm hoping someday you might find
Well, I'll hide it behind something
They won't look behind

Trent Reznor, Nine Inch Nails

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 (Carreira et al., 2007) constituém uma classe particular de transformação de dados em que cada túplo 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.

Palavras-chave: *Sistemas de Gestão Bases de Dados Relacionais; Transformações de Dados Um-para-muitos (”Bounded” e ”Unbounded”); Parâmetros Críticos (”Fanout”, Selectividade e Número de Tuplos); Benchmarking.*

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 (Carreira et al., 2007) 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.

Keywords: *Relational Database Management Systems; One-to-many Data transformations (Bounded and Unbounded); Critical Parameters (Fanout, Selectivity and Number of Records); Benchmarking.*

Index

List of Figures

List of Tables

Acronyms

1NF	First Normal Form
DTL	Data Transformation Language(s)
DW	Data Warehousing
ETL	Extract-Transform-Load
ISO	International Organization for Standardization
IT	Information Technology
MSQLS	Microsoft SQL Server
N1NF	Non First Normal Form
PL	Programming Language(s)
PSM	Persistent Stored Module(s)
RA	Relational Algebra
RDBMS	Relational Database Management System(s)
SQL	Standard Query Language

1 Introduction

Data transformation is an essential activity in several Information Technology (IT) contexts, namely in data migration, data integration, and in Data Warehousing (DW). *Data migration* is the process of transferring data between systems with different data schemas, most frequently related with the setup of new software applications. *Data integration* is the process of combining data from several data sources into a unified schema. The resulting schema can be either real or virtual. Either the data from the sources resides in a new independent data source, or the data continues to reside in the original data sources being accessed through wrappers and views. *Data Warehousing* is an *Extract-Transform-Load* (ETL) process where relevant data is extracted from one or more sources, transformed and loaded into a data repository with data structured according to a multidimensional model. In all these contexts, data coming from one or more sources, each with its own data schema, has to be transformed to fit into some target output schema.

Data transformations can be accomplished mainly in three different ways: *(i)* Programming Languages (PL); *(ii)* ETL tools; and *(iii)* Relational Database Management Systems (RDBMS).

PL solutions have the most expressive power in the sense that they can perform any computable transformation (Abiteboul et al., 1995). However, PL are procedural by nature, as opposed to declarative languages. Therefore, the code executed is entirely determined by the code written and, consequently, dynamic optimizations normally do not exist as they are hard and time consuming to implement. Moreover, the code written is often extensive and difficult to maintain. Ultimately, the application logic is not easy to understand.

ETL tools can readily access multiple heterogeneous sources of data (Songini, 2004). Their most distinctive and important feature is the support of workflows of data transformations (Vassiliadis et al., 2007). Workflows greatly enhance the comprehension and design of complex ETL processes where several data transformation steps are necessary. However, ETL tools lack conformity/standardisation both in terms of their design modeling techniques and supported transformation languages (Vassiliadis et al., 2007). Hence, different tools may have significantly different functionalities and performance. The inexistence of standards for a transformation language and design modeling of workflows implies that ETL processes are specific to the tools they were implemented in (can not easily be exported). The research community does not have a logical framework for proposing efficient execution algorithms or new operators. As a result, users often code their data transformations with ad-hoc code and the execution of the predefined operators runs, in general, a fixed algorithm.

RDBMS are the main repositories of data in enterprises. This means that they can directly transform the data they are responsible for. RDBMS support an ISO standard declarative language, SQL, used to query and modify the data (Melton & Simon, 2002). Therefore, data transformations coded in SQL are database independent. Moreover, the logical operators supported by the language have been thoroughly investigated over the years and

are supported by several physical execution algorithms (Silberschatz et al., 2001). Language elements classified as *persistent stored modules* (PSM) (ex: Stored Procedures) imbue RDBMS with PL-like capabilities (Melton, 1998). This enables them to overcome some of the limitations present in the *Relational Algebra* (RA). For example, one-to-many data transformations are not fully supported (Carreira et al., 2007). Finally, relational database systems are not meant, at least in concept, to access data in heterogeneous sources, although these systems can normally be extended by drivers such as Oracle Loader to access external sources (ORACLE, 2005).

In this work, we will focus most of our attention and efforts studying the support of a specific class of data transformations in RDBMS.

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, etc) and supporting relationships among users. We assume the existence of a fictional social network called Face5 and work with simplified, yet realistic, relations of the Face5 underlying database. The database is composed by a PROFILE relation storing users' personal information with schema PROFILE=(USER, COUNTRY, CITY, FAVARTISTS) , and a FAVFRIENDS relation storing up to four best friends for each user with schema FAVFRIENDS=(USER, FAVF1, FAVF2, FAVF3, FAVF4) .

Scenario 1. In Face5, every user may promote to the title of best/top 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 readability/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 FAVFRIENDS, see Table 1.1 for an instance of the relation, contains the information required for answering the desired query. However, the data in this instance is not in an adequate format to easily allow the construction of such a query. If the FAVFRIENDS relation could be transformed into an equivalent relation with schema FAVFRIENDS′ = (USER, FAVFRIEND) , then querying for the most liked users would become trivial (e.g. count users grouped by favorite friend). Table 1.2 illustrates an instance of the FAVFRIENDS' relation, which is semantically equivalent to the FAVFRIENDS relation instance of *Table 1.1*. The data transformation that produces the FAVFRIENDS' relation having as input the FAVFRIENDS relation is obtained by making each input tuple yields at most four output tuples, one for each favorite friend.

USER	FAVF1	FAVF2	FAVF3	FAVF4
user1@gmail.com	user2@gmail.com	user4@gmail.com	user3@gmail.com	user6@gmail.com
user2@gmail.com	user1@gmail.com	user4@gmail.com		

Table 1.1: FAVFRIENDS relation instance, used to store each user's favorite friends.

USER	FAVFRIEND
user1@gmail.com	user2@gmail.com
user1@gmail.com	user4@gmail.com
user1@gmail.com	user3@gmail.com
user1@gmail.com	user6@gmail.com
user2@gmail.com	user1@gmail.com
user2@gmail.com	user4@gmail.com

Table 1.2: FAVFRIENDS' relation instance, used to determine the most liked favorite 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.

The relation PROFILE, see Table 1.3 for an instance of the relation, contains the information concerning each user's favorite artists in the string field FAVARTISTS. This schema design was chosen because: *(i)* developers and directors did not foresee the need to query user's favorite artists, *(ii)* users do not want to be limited in the number of artists they are allowed to like and *(iii)* Face5 does not have the resources to create a table of artists containing every artist that users will want to select. The first reason would indicate that nesting the data in one string field would be inadequate. The second reason eliminates the possibility of adopting relation schemas with multiple columns dedicated to distinct favorite artists. The final reason eliminates the possibility of adopting a database schema with a many-to-many association relationship between a PROFILE relation and an ARTIST relation, given that it is impossible to obtain all the necessary data to construct an adequate ARTIST relation instance.

As a result, the current data format makes querying the data for the users interested in a specific artist a challenge. Decoupling the information nested in the FAVARTISTS field would make this query much simpler. This would result in an equivalent relation with schema PROFILE'=(USER, COUNTRY, CITY, FAVARTIST) where the FAVARTIST field contains exactly one artist. Table 1.4 illustrates an instance of the PROFILE' relation. The data transformation that produces the PROFILE' relation having as input the PROFILE relation is obtained by making each input tuple yield as many output tuples as the artists mentioned in the FAVARTISTS field.

USER	COUNTRY	CITY	FAVARTISTS
user1@gmail.com	Portugal	Lisbon	Dead Can Dance; Tori Amos; Amon Amarth; Magnet
user2@gmail.com	Portugal	Lisbon	Cradle of Filth; Amon Amarth; Children of Bodom

Table 1.3: PROFILE relation instance, used to store each user's personal information.

1.2 One-to-many data transformations

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

USER	COUNTRY	CITY	FAVARTIST
user1@gmail.com	Portugal	Lisbon	Dead Can Dance
user1@gmail.com	Portugal	Lisbon	Tori Amos
user1@gmail.com	Portugal	Lisbon	Amon Amarth
user1@gmail.com	Portugal	Lisbon	Magnet
user2@gmail.com	Portugal	Lisbon	Cradle of Filth
user2@gmail.com	Portugal	Lisbon	Amon Amarth
user2@gmail.com	Portugal	Lisbon	Children of Bodom

Table 1.4: PROFILE' relation instance, used to users based on artists preferences.

transformations described in Scenario 1 and 2 are both one-to-many data transformations.

In this Section we characterize the conditions that normally induce the necessity to restructure relations through one-to-many data transformation. Then, we analyse the motivating examples, Scenarios 1 and 2, characterize the essential differences in both scenarios and present the two classes of one-to-many data transformations discussed in (Carreira et al., 2007).

1.2.1 Requiring one-to-many data transformations

Section 1.1 illustrated two examples of one-to-many data transformations. Although the examples are capable of motivating this class of data transformations, they alone do not convey the classes of relations that will, in general, induce the need for these data transformations. Therefore, we now discuss the different classes of relations that need restructuring through one-to-many data transformations.

Essentially, there are three classes of relations over which one-to-many data transformations may prove most useful:

1. *Relations with semantically equivalent attributes:* Relations containing multiple attributes for storing data values with equivalent semantic meaning. These relations may have to be transformed into equivalent relations where only one attribute exists associated to the corresponding semantic domain. Although the information content of the original and transformed relations is the same, the transformed relation uses a schema that is more easily queried. These data transformations are said to be *bounded* as will be discussed in Section 1.2.2. The FAVFRIENDS relation relative to Scenario 1 in Section 1.1, constitutes an example of a relation with *semantically equivalent attributes*. Finally, the *pivot* operator, referred in Chapter 2, can be considered a one-to-many operator meant to transform this class of relations (Cunningham et al., 2004).

2. *Relations with multivalued data:* Relations containing data values where it is possible to identify distinct data elements semantically related, that is, where attribute values can be perceived as sets (normally emulated through the use of strings) containing semantically related data elements. These relations may have to be restructured through one-to-many data transformations in order to decouple the data contained in each tuple. The resulting data transformations are said to be *unbounded* as will be discussed in Section 1.2.2. The PROFILE relation relative to Scenario 2 in Section 1.1, constitutes an example of a relation containing

multivalued data. Finally, the *unnest* operator, referred in Chapter 2, can be considered a one-to-many operator addressing, specifically, the restructuring of this class of relations (Jaeschke & Schek, 1982).

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. One of problems of reverting a *group by* operation is that, in most cases, their inverse can only be approximated, since the original data values used in the *group by* are consumed to originate a new atomic value. Even so, making an estimate of these original values may be a viable option, for example, converting the annual earnings of an enterprise into the respective trimester gains can be approximated by assuming an uniform distribution of gains by trimester. These data transformations are in general *bounded* but *unbounded* examples also exist[1]. For processing this sort of data transformations we can consider the *mapper* and *mapping* operators, referred in Chapter 2, which are capable of expressing every type of one-to-many data transformations (Carreira et al., 2007; Galhardas et al., 2001).

Outside the scope of our motivating scenarios, other examples of the application of one-to-many data transformations could be explored. For example, in (Galhardas et al., 2001) a data cleaning process over bibliographic references (extracted from publicized articles) requires the application of a one-to-many data transformation. In this case, after separating distinct contents into different output columns (title, year, authors, type of publication, etc), the relation has to undergo a one-to-many data transformation to accomplish the separation of the authors in each publication.

1.2.2 Subclasses of one-to-many data transformations

The motivating Scenarios 1 and 2 are similar, since both require the restructuring of relations using one-to-many data transformations. The FAVFRIENDS relation in Scenario 1 falls into the class of relations containing *semantically equivalent attributes*, where each FAVFI attribute is associated to the concept of favorite friend. In Scenario 2, the PROFILE relation falls into the class of relations containing *multivalued data*, since each data value for the attribute FAVARTISTS can be seen as a set containing several similar elements (artists).

What significantly sets these two scenarios apart is the number of records that are obtained when processing each input tuple. In Scenario 1, each tuple can contain at most four favorite friends and so we know that each tuple may yield at most four output records. In Scenario 2, any number of artists may be referred by the user and so a bound for the number of output records is not known in advance.

When a limit k for the maximum number of output records yielded by each input tuple is known, the data transformation is classified as *bounded* one-to-many data transformation. Otherwise, the data transformation is classified as *unbounded* one-to-many data transformation. This distinction is relevant, because *bounded* one-to-many transformations can be accomplished using only standard relational algebra operators, where the *unbounded* can only be accomplished through extensions to this algebra (Carreira et al., 2007).

[1] We know exactly how may trimesters exist in a year, therefore this conversion would be bounded. But for instance, we do not know how many prime numbers constitute the factorization of an arbitrary integer.

The general formula for expressing *bounded* one-to-many data transformations consists in the union of k projections with selections (Carreira et al., 2007). An example of the application of this method can be seen in Code Listing 1.

Code Listing 1. Performing a bounded one-to-many data transformation for Scenario 1 applying the general formula described in (Carreira et al., 2007). This data transformation is composed by the union of four identical projections. The first projection code is delimited by the lines 1 to 3, the second by the lines 5 to 7, the third by the lines 9 to 11 and the fourth by the lines 13 to 15. Each projection performs a one-to-one data transformation, associating each user to one of his favorite friends. By performing the union of four projections, each targeting a different `FAVF` column, we perform a *bounded* one-to-many data transformations that associates each user to all of his favorite friends. We call this technique of performing the *bounded* one-to-many data transformations, the *selection-projection-union* (*SPU*) implementation method, see also the Section 2.1.1.2.

```
01:  SELECT USER, FAVF1 as FAVF
02:  FROM  FAVFRIENDS
03:  WHERE FAVF1 is not null
04: UNION ALL
05:  SELECT USER, FAVF2 as FAVF
06:  FROM  FAVFRIENDS
07:  WHERE FAVF2 is not null
08: UNION ALL
09:  SELECT USER, FAVF3 as FAVF
10:  FROM  FAVFRIENDS
11:  WHERE FAVF3 is not null
12: UNION ALL
13:  SELECT USER, FAVF4 as FAVF
14:  FROM  FAVFRIENDS
15:  WHERE FAVF4 is not null
```

For either case, *bounded* and *unbounded*, there is no relational algebra operator meant explicitly to express them. Therefore, we cannot expect their execution to be adequately optimized by RDBMS technology. Notice that, for Code Listing 1, the relation FAVFRIENDS can potentially be read four times, depending on the cache size and execution plan.

1.3 Purpose

The purpose of this thesis is to study how relational database systems support one-to-many data transformations, in terms of: *(i)* the different implementation methods that these systems offer which can be used to implement these data transformations; *(ii)* to study their efficiency, code complexity, expressive power and the way they are affected by the *critical parameters* that influence the performance of one-to-many data transformations; *(iii)* and based on the previous two points, compare the different database systems with one-another. Three commercial RDBMS will be analysed for this: Oracle 11gR2 (Oracle), Microsoft SQL Server 2008 (MSQLS) and DB2 UDB (DB2).

1.4 Outline of the document

This thesis is comprised of several chapters:

Chapter 2 presents some of the available knowledge related to one-to-many data transformations that is relevant in the context of this thesis. We discuss modern operators present in RDBMS and proposed extensions to the RA that can be used to express one-to-many data transformations.

Chapter 3 reviews the motivational example extending it with a third scenario, so that the three domains of use of one-to-many data transformations identified in the current chapter are covered. This chapter formalizes the three scenarios into a schema-migration problem, where the three one-to-many transformations that solve them are mathematically described by schema mapping expressions using extended relational algebra. The schema mapping expressions that the RDBMS can solve and the different implementation methods at their disposal to solve them are also identified for each specific RDBMS, along with the respective implementation code.

Chapter 4 describes software and hardware conditions in which the benchmarking experiments were run, and specifies the tuning aspects that were controlled in order to ensure the fairness of these experiments among the different RDBMS. We present the experiments results and analyze them. We compare the different RDBMS with one another in an experiment-by-experiment basis, by focusing on the best performing implementation method that each RDBMS supported in each experiment.

Chapter 5 resumes the work undertaken throughout the thesis, presents the conclusions drawn from our work, and proposes two different directions to continue the advancement of the study and support of one-to-many data transformations in RDBMS.

2 State of the Art

In this chapter, we consider the two main data representation models, relational and semi-structured, and describe how the existing technologies for each data model support one-to-many data transformations. For each technology, we analyse its supported language in regard to its capability of expressing one-to-many data transformations. We informally accomplish this by illustrating one-to-many data transformations code examples and by citing references where the expressive power of the language was studied.

2.1 Transforming relational data

Data is most often stored in relational format, that is, in a collection of relations where each relation is perceived as a table and the data stored in each table satisfies a common set of constraints (Codd, 1970). Therefore, data is stored in a structured way and has a certain degree of quality (C. J. Date, 1994).

The first set of tools handling relational data are the RDBMS, responsible for both storing and manipulating the data. First, we analyse the computer language supported by these systems, that is, the *Standard Query Language* (SQL) and its extensions. Or equivalently, if we speak in terms of the mathematical theory behind this computer language, the Relational Algebra (RA) and its extensions. Second, we analyse the procedural constructs offered by the procedural languages supported by different RDBMS products[1]. These procedural languages are used to implement the *Persistent Stored Modules* component of the SQL standard (SQL/PSM) (Melton, 1998).

The second set of tools we analyse are *data cleaning* tools. We consider AJAX and Potter's Wheel (Galhardas et al., 2001). These specific tools are built as a layer of software working on top of RDBMS platforms and providing specialised features, namely: *(i)* new operators to address common data processing problems that arise in the field of quality assurance, and *(ii)* the construction of workflows for modeling the steps of complex data cleaning processes. We focus on the specific operators provided by each of these tools which may be used to perform one-to-many data transformations. As a side note, many of the operators included in these platforms have similarities to those we can expect to find on the varied ETL tools, namely routing operators, as classified and summarized in (Vassiliadis et al., 2007).

2.1.1 The Relational Algebra & and its extensions

To enhance RDBMS with increasingly more powerful query mechanisms the set of operators allowed have been extended beyond what was originally defined in the standard RA (Silberschatz et al., 2001). Some of these exten-

[1] Examples of procedural languages in widespread use are PL/SQL in Oracle, SQL/PL in DB2 and TSQL in SQL Server

sions are standardized and therefore common to all RDBMS which are SQL compliant, for example: the operators for grouping and sorting relations. The remaining extensions are either: *(i)* proposed by researchers in the field of database systems, or *(ii)* defined and implemented by the RDBMS vendors.

We consider both the standard RA and its extensions.

First, we discuss two operators, the *nest* and *unnest*, introduced in an early theoretic extension to the relational model (Jaeschke & Schek, 1982). The *unnest* operator is capable of performing one-to-many data transformations over relations holding *multivalued data*.

Second, we consider a type of RA queries consisting of selection, projection and union operators. Such queries are capable of performing any bounded one-to-many data transformation, namely, processing relations containing *semantically equivalent attributes* and some[2] relations containing *aggregated data*.

Third, we analyse the *unpivot* operator introduced as a Microsoft SQL Server extension *Pivot & Unpivot* (Cunningham et al., 2004). The *unpivot* operator can be used to realize bounded one-to-many data transformations over relations containing *semantically equivalent attributes*.

Four, we explore the *recursive queries* extension introduced in SQL:1999 (Melton & Simon, 2002). These queries can be used to realized both bounded and unbounded data transformations, namely over relations containing *multivalued data* and *aggregated data*. Although this has not yet been formally proven.

Five, we research the *model* extension introduced as an Oracle extension (Oracle, n.d.). This operator can be used to accomplish any bounded one-to-many data transformation, namely over relations containing *multivalued data* and some relations containing *aggregated data*.

Six, we review the *mapper* operator, introduced as modern theoretic extension in (Carreira et al., 2007). This operator can be used to accomplish any type of one-to-many data transformation, namely over relations containing *semantically equivalent attributes*, *multivalued data* and *aggregated data*.

2.1.1.1 Nest & Unnest

G. Jaeschke proposed an extension to the relational model affecting both the data model and algebraic operators (Jaeschke & Schek, 1982). First, the data model is allowed to contain set valued attributes. Second, the algebraic operators are extended by two new operators *nest* and *unnest*.

In this extension, a set is allowed to contain items of two types: *sets* and *atomic values*. If a set contains sets, then every element has to be a set with the same *nesting depth*. Otherwise, if a set contains atomic data, then every element in the set is atomic and belongs to the same domain. Therefore, to formally define the set of domains allowed with Jaeschke's extension we have: if U is some atomic domain supported by the relational data model, then $P^n(U)$, where P denotes the power set and n the nesting depth, is a valid domain in this model (Jaeschke & Schek, 1982).

[2] In these relations, the aggregated data originates from a bounded set of source tuples, for example, a relation containing year salary paycheck which could be divided into twelve monthly payments.

Two operators are proposed to allow the manipulation of set valued data types:

The nest operator can best be understood as a *group by* operation where the aggregation function places all of its input elements grouped in an output set value. In a nest operation a *target attribute* is specified, denoting the attribute that is to be used as input for the aggregation function. The remaining attributes of the relation are used to group the data, constituting a *grouping set*. Therefore, given a relation `R` with schema `R=(A1,...,An,At)` , where At is the target attribute and $GSet = \{Ai|\ i \in \{1,...,n\}\}$ the grouping set, a $nest(R,At)$ is equivalent to a *Group By* over $GSet$ with the aggregation function $F : P(At) \mapsto P(At)$. The aggregation function F is the identity function, in the sense that the set of values fed as input is also returned as output (in the form of a single set). This operator can be seen as a many-to-one operator. The table 2.1 illustrates a nesting data transformation over a source relation `R=(FNAME,SPOKENLANG)` where the target attribute is `SPOKENLANG` . It further illustrates that a *group by* operation with a *strcat* aggregation function produces similar results.

The unnest operator reverts the effects of nesting. If nothing is nested, unnesting will have no effect whatsoever. Therefore, if the first operator is a many-to-one operator and this one is its inverse, it follows that this operator is one-to-many. Unnesting transformations can be said to be unbounded since each set value (to be unnested) can virtually hold an infinite number of elements. In an unnest operation a *target attribute At* is specified. It denotes an attribute whose data values are sets containing data elements we wish to extract. The remaining relation attributes contain data values that are to be copied to each of the generated new tuples. These attributes constitute a *copy set CSet*. Formally, unnesting a tuple over the target attribute At, can be understood as performing the following operation at tuple value: $unnest(\{t\}) = \{t'|\ \forall_{Ai \in CSet}(t'(Ai) = t(Ai))\ \cap\ t'(At) \in t(At)\}$. Finally, it follows that the number of tuples yielded by unnesting a tuple is the same as that of elements at the target attribute At in the source tuple, denoted by $|unnest(\{t\})| = |t(At)|$. The table 2.2 illustrates an unnesting data transformation over a source relation `R=PROFILE` where the target attribute is $At = FAVARTISTS$.

R	
FNAME	SPOKENLANG
Nuno	Portuguese
Niina	Finnish
Niina	Spanish
Christine	French
Christine	Portuguese

→

$nest(R, spokenlang)$	
FNAME	P(SPOKENLANG)
Nuno	{Portuguese }
Niina	{Finnish, Spanish }
Christine	{French, Portuguese }

↘

$groupby(R, \{fname\}, strcat(spokenlang))$	
FNAME	NESTEDLANGS
Nuno	Portuguese
Niina	Finnish, Spanish
Christine	French, Portuguese

Table 2.1: Illustrating the transformation of a source relation R by the *nest* and *groupby* operators, where the target attribute is $At = SPOKENLANG$ and the grouping set is $GSet = \{FNAME\}$. The difference in output produced by the two operators is result of the difference in aggregation function, where the *nest* operator returns a set of elements, and the *group by* returns a string.

The two operators provided in this extended algebra, the *nest* and *unnest*, can be seen as inverse functions

$R = PROFILE$			
USER	COUNTRY	CITY	FAVARTISTS
user1@gmail.com	Portugal	Lisbon	{Dead Can Dance, Tori Amos, Amon Amarth, Magnet }
user2@gmail.com	Portugal	Lisbon	{Cradle of Filth, Amon Amarth, Children of Bodom }

↓

$unnest(R, FAVARTISTS)$			
USER	COUNTRY	CITY	FAVARTIST
user1@gmail.com	Portugal	Lisbon	Dead Can Dance
user1@gmail.com	Portugal	Lisbon	Tori Amos
user1@gmail.com	Portugal	Lisbon	Amon Amarth
user1@gmail.com	Portugal	Lisbon	Magnet
user2@gmail.com	Portugal	Lisbon	Cradle of Filth
user2@gmail.com	Portugal	Lisbon	Amon Amarth
user2@gmail.com	Portugal	Lisbon	Children of Bodom

Table 2.2: Illustrating the transformation of the PROFILE relation, introduced in the motivating example, by the *unnest* operator. The target attribute $At = FAVARTISTS$ contains set values, and the copy set is $CSet = \{USER, COUNTRY, CITY\}$. The result of the data transformation is the same as the obtained by the unbounded one-to-many of Scenario 2.

of each other. However, formally speaking, this statement is incorrect since an unnest operation over a target attribute may not produce any effect. This happens whenever the data values for the target attribute are atomic. More specifically, and using Jaeschke's interpretation of what first normal form relations are, a 1NF relation is not affected by unnesting, that is: $1NF(R) \implies unnest(R) = R$. Inversely, the nest operation will always produce a different output relation by increasing the nesting depth of the elements in some target attribute.

The unnest operator constitutes an evidence of the existence of operators solely designed to execute one-to-many data processing, even if the motivation behind it is not the support of this class of data transformations. In this case, the author was concerned with the support of complex data types in relational database systems (Jaeschke & Schek, 1982). Other researchers conducted studies in the field of non atomic data elements (Abiteboul & Bidoit, 1984).

2.1.1.2 Selection_Projection_Union

Selection_Projection_Unions (SPU) clauses can be used to perform any one-to-many bounded data transformation over any RDBMS (Carreira et al., 2007). The selection clause is used to single out the tuples in the relation that meet some criteria (logic statement), and the projection clause performs a desirable one-to-one data transformation over each tuple.

It is obvious that any one-to-many data transformation is equivalent to the conjunction (union) of an adequate number of one-to-one data transformations. Moreover, any one-to-one data transformation can be expressed through the exclusive use of projections with selections. Despite these results, that should indicate that with SPU clauses we would be able to accomplish any one-to-many data transformation, a significant problem remains: one-to-many data transformations that map input tuples into possibly infinite output tuples, will require the conjunction of an equal number of one-to-one data transformations. This implies that an unbounded one-to-many data transformation is equivalent to an infinite (SQL) expression uniting projections with selections. Hence, *only bounded one-to-many data transformations are supported through this method*. Furthermore, the larger the cardinality of a

bounded one-to-many data transformation, the longer will be the equivalent SQL expression.

This transformation mechanism is illustrated in Section 1.2.2 Code Listing 1, solving the bounded data transformation presented in Scenario 1 Section 1.1.

2.1.1.3 Pivot & Unpivot

Microsoft SQL Server introduced two extensions to the RA through the *pivot* and *unpivot* operators (Cunningham et al., 2004). Neither operator extends the expressive power of the RA (Cunningham et al., 2004; Carreira et al., 2007), but they allow the execution of some of the already supported data transformations in a more efficient and less extensive manner (Cunningham et al., 2004). The two operators have also been adopted by current versions (10g and 11g) of the Oracle database system.

The *pivot* operator performs data aggregations by adding new columns based on specified tuple values. This can be seen as a many-to-one data transformation, just like *Group By* transformations, and is not relevant in the context of this work.

The *unpivot* operator, on the other hand, performs a very specific type of bounded one-to-many data transformations. With the *unpivot* operator an arbitrary number of columns is replaced by a new column whose values will be the names of each of the replaced columns. Furthermore, a second column is added which will contain the values associated to each of previously replaced columns. This means that each input tuple will be mapped to a number of output tuples equal to the number of the replaced columns. For instance, if a relation `R=(A,B,C)` suffers an *unpivot* over the {`B,C`} columns, then the output relation will be of the form `R'=(A, BC, V)` where $Dom(BC) = \{B,C\}$, $Dom(V) = Dom(B) \cup Dom(C)$ and $|R'| = 2 \times |R|$.

The *unpivot* operator is not very flexible in terms of uses in diverse contexts, it can only perform a subset of *bounded* data transformations (those involving the rotation of table columns). However, this implementation method can be particularly useful when dealing with relations containing distinct attributes conveying the same semantic information. The use of this implementation method for this purpose is illustrated in Code Listing 2, just following, solving the bounded data transformation presented in Scenario 1 Section 1.1.

Code Listing 2. Using the *unpivot* implementation method to perform the *bounded* one-to-many data transformation of Scenario 1, Section 1.1. Recall that in this scenario, each input record of the source table associates a user to his favorite friends. The purpose of the data transformation is to produce a new output table, where each record associates one user to one favorite friend. Thus, each input record coming from the source table is transformed into as many output records as the number of favorite friends it was originially associated to. Considering the above short explanation of the unpivot operator, the new columns denoted as `FAVF` and `CNAME` correspond to the attributes V and BC, respectively. In the line 3, we are telling the *unpivot* operator to stripp each input record from its `FAVF1` to `FAVF4` columns, and to replace them by the pair of output columns `FAVF` and `CNAME` . The `FAVF` column of each output record will hold the data value of one of the `FAVFi` input columns, and the `CNAME` column will hold the name of the respective input column from which the `FAVF` data value was copied. If all the input columns that are being deleted hold data, then each input record will be transformed into four output records.

```
01: SELECT USER, FAVF
02: FROM PROFILE
03: UNPIVOT (FAVF FOR CNAME IN (FAVF1, FAVF2, FAVF3, FAVF4)) as TRGT_TABLE
```

2.1.1.4 Recursive Queries

Recursive queries were introduced in the SQL:1999 revision as an extension to the relational algebra (Melton & Simon, 2002). MSQLS and DB2 support them in accordance to the SQL standard, Oracle does not. Their intended purpose was supporting the computation of complex operations, namely the transitive closure of relations containing transitive relationships between attributes (Libkin, 2003). This motivation is interesting because the transitive closure can be seen as a many-to-many data transformation (it takes at least two transitive relationships to yield a new one).

A *recursive query* works in a way similar to that of the general formula for solving bounded one-to-many data transformations, where a number of one-to-one data transformations are united. Similarly, in a *recursive query*, an initial relation containing tuples that will be part of the end result is defined. The query that produces this initial result set is referred to as *anchor*, in the MSQLS terminology. Making an analogy with an *SPU* implementation, the *anchor* can be seen as the first selection-projection query coded in the implementation. After the *anchor* part of the *recursive query* comes a general query referred to as *recursive member*, in the MSQLS terminology. The *recursive member* is query that takes as input the most recently generated tuples by the *recursive query* itself. This *recursive member* is executed over and over, until an iterations finishes without producing any output results. Making an analogy with an *SPU* implementation, each iteration of the *recursive member* can be seen as the addition of another select-projection query to the implementation.

Mathematically, we can understand this implementation mechanism as follows. First, the output of a *recursive query* will be denote as R, standing for the final result set. The output resulting from processing the *anchor* part of a *recursive query* will be denoted as R_0, standing for the initial result set. Now we just need define a recursive function denoted as *rm*, standing for the *recursive member*. This function input domain and output domain are the same, which is that of relations with the same schema of R. What the *recursive member* function does is very simple. It takes as input a result set, say the relation R_i, and computes the following result, denoted as R_{i+1}, such that: $rm(R_i) = R_{i+1} \cup rm(R_{i+1})$. The stop case for the recursive function *rm* is reached when its input is an empty relation, that is: $rm(\{\}) = \{\}$. To conclude, the output of a *recursive query* would be mathematically described as: $R = R_0 \cup rm(R_0)$. For example, if we were processing a one-to-many data transformation where each input record of the source relation yielded two output record (*fanout*=2), by means of a recursive query, than the following would happen.

$$\begin{aligned} R &= R_0 \cup rm(R_0) \\ &= R_0 \cup R_1 \cup rm(R_1) \\ &= R_0 \cup R_1 \cup \{\} \\ &= R_0 \cup R_1 \end{aligned}$$

First, the *anchor* part of the query would processed into the first set of output records. Then, the *recursive member* would be used over this initial result set to process the second set results. Finally, the second result set would be processed by *recursive member* into an empty set of results. At this point, the *recursive query* would end its processing cycle. The reason that would lead the second set of results (R_1) to be processed into an empty result set would be a predicate clause coded into *recursive member* query, which would not be satisfiable by any of the records in this result set.

Recursive queries offer a mechanism of performing up to an unlimited number of query iterations through a finite language expression. The limitation of this powerful query mechanism, if we can call it that, is that the query design for data transformations different then the calculus of transitive closures can be quite complex. Notice that the recursion step is based on a single general query. In contrast, the general formula for bounded data transformations, see Section 1.2, allows each query present in the formula to have it's own specific selection predicate and projection functions. With *recursive queries* we are limited to a very rigid formula. However, it is possible to make bounded and unbounded one-to-many data transformations through the use of *recursive queries*, although this usually requires the specification of a non-evident query.

This transformation mechanism is illustrated in Code Listing 3, just following, solving the unbounded data transformation presented in Scenario 2 Section 1.1.

Code Listing 3. Performing the unbounded one-to-many data transformation for Scenario 2, Section 1.1, using Microsoft SQL Server recursive queries. Recall that this data transformation consisted on producing an output relation where each user was associated to his favourite artists, with each row associating only one user to one artist. Thus, in the output relation each user would be listed in multiple rows. Remember also that in the original source relation, each row associated one user to all of his favourite artisits. As explained above, a *recursive query* implementation (lines 1 to 41) has two components: an *anchor* (lines 4 to 16) and a *recursive member* (lines 18 to 30). The *anchor* produces the first set of results. In this case, it produces an output relation with schema (`UID`, `ANAME`, `RESTARTISTS`), where `ANAME` is the name of the first artist liked by each user, and `RESTARTISTS` are the remaining set of artists liked by the user. We assume that the artists in the source relation attribute `FAVARTISTS` are separated from one-another by means of commas. Finally, the *recursive member* processes an identical computation over its input. However, unlike the *anchor* which used as input the source relation, see line 15, the *recursive member* uses as input the output produced by the recursive query in its previous iteration, see lines 1 and 29. In line 1, we define that the name of the *recursive query* output relation is `RECURSIVE_FAVARTIST`, and in line 29 we use this relation as input of the *recursive member*. The query will eventually stop, when there are no more artists to extract from the `RESTARTISTS` list. That is, when the *recursive member* predicate condition `A.RESTARTISTS != ''` fails for all of its input records.

```
01: WITH RECURSIVE_FAVARTIST (UID, ANAME, RESTARTISTS)
02: AS
03: (
04: SELECT
05:   ID as UID,
06:   -- extract the HEAD ELEMENT
```

```
07:   CAST(RTRIM(LTRIM(SUBSTRING( FAVARTISTS, 1, CHARINDEX(',', FAVARTISTS + ',')-1
08:         ))) as varchar) as ANAME,
09:   -- REMAINING ARTISTS  (If no comma in string there are no more artisits)
10:   CASE WHEN CHARINDEX(',', FAVARTISTS) = 0
11:         THEN ''
12:         ELSE SUBSTRING( FAVARTISTS + ',', CHARINDEX(',', FAVARTISTS)+ 1,
13:                       (LEN(FAVARTISTS)+1) - CHARINDEX(',', FAVARTISTS) )
14:   END as RESTARTISTS
15: FROM PROFILE
16: WHERE FAVARTISTS IS NOT NULL AND FAVARTISTS != ''

17: UNION ALL

18: SELECT
19:   UID,
20:   -- Extract the head element
21:   CAST(RTRIM(LTRIM(SUBSTRING( A.RESTARTISTS, 1, CHARINDEX(',', A.RESTARTISTS)-1
22:         ))) AS varchar) as ANAME,
23:   -- REMAINING ARTISTS (if first comma is at the end of the string there are no more artists)
24:   CASE WHEN CHARINDEX(',', A.RESTARTISTS) = LEN(A.RESTARTISTS)
25:         THEN ''
26:         ELSE SUBSTRING(A.RESTARTISTS, CHARINDEX(',', A.RESTARTISTS)+ 1,
27:                       LEN(A.RESTARTISTS) - CHARINDEX(',', A.RESTARTISTS))
28:   END as RESTARTISTS
29: FROM RECURSIVE_FAVARTIST as A
30: WHERE A.RESTARTISTS != ''
31: )
32: SELECT UID, ANAME
33: FROM RECURSIVE_FAVARTIST
```

2.1.1.5 Model

Oracle's *model* operator was introduced as an analytical operator useful in executing tasks similar to those performed over spread sheets (Oracle, n.d.). With this operator, along with system or user defined functions, many of the operations performed in a spread sheet can easily be performed inside an oracle database. The *model* operator has five fundamental parameters: *(i)* the input relation; *(ii)* the logical partitions; *(iii)* the index dimensions; *(iv)* the measures; and finally *(v)* the rules.

First, the input relation contains the data to be processed. The result of processing the input relation will not yield output relations with distinct relational schema. However, the contents of some of the cells in the input relation may be changed in the output relation, or even further, new rows may have been introduced. In summary, the *model clause* will process a relational instance into a possibly new one, while maintaining the input and output relational schemas unchanged.

Second, the *partition clause* defines a set of attributes responsible for identifying logical partitions. The physical creation of logical partitions can be seen as the process of placing tuples in clusters defined by a set of attributes with equal values. This is important because the data placed in each cluster can be processed independently of the

data contained in distinct partitions. This means that oracle can parallelize the processing of the data in each logical partition, which may have impact on the performance of the overall computation (Oracle, n.d.). However, it is important to mention that the developer is not required to specify attributes for logical partitions. As a side note, for processing one-to-many data transformations, any logical partition will, in theory, be acceptable, since the data we are interested for processing tuples (into many) is totally contained in each individual tuple.

Third, the *dimensions clause* specifies the set of attributes which will be used as vector coordinates for the internal physical structures created by oracle (Oracle, n.d.). The set of attributes in the *dimensions clause* is disjoint in relation to the attributes contained in the *partition clause*. This is because, for each logical partition created the values of the attributes contained in it are fixed and can not be changed. Moreover, the *dimensions clause* is forcefully a non empty set. It is necessary to have one or more attributes serving as coordinates for the values to be addressed in the processing stage. More importantly, the conjunction of attributes in both the *partition* and *dimensions clauses* have to constitute a *candidate key* for the input relation. The reason for this, is that in the internal vector representation generated by oracle, the cells whose values we desire to affect/change have to be uniquely identified. Otherwise, for a given vector coordinate we could, possibly, be addressing the values contained in more than one tuple in the defined *cluster/partition* of tuples.

Fourth, the *measures clause* identifies a set of attributes for which the corresponding cell values in the tuples we want to affect, normally they correspond the attributes known as *facts* in *OLAP*. Once again, this set of attributes is disjoint in relation to the attributes specified in the *partition* and *dimensions* clauses. Oracle generates an equal number of vectors for each partition to that of *measures* attributes. This is because at each vector coordinate we are only interested in reading or writing values concerning one specific attribute.

Fifth, the *rules* are the mechanism of affecting measure cell values. Each rule consists of describing what changes should be applied to each measure vector. The rules specified are not partition specific, they are applied to every logical partition. In conclusion, the model operator offers an alternative mechanism of updating row values. At the end of the execution every logical partition is converted back in to relational format and the data in distinct partitions is joined together into the output relational instance.

Since the operator is rather complex, we will try to describe the way it works through a more mathematical perspective. Let us consider a relation R with relational schema `R=(A,B,C,V)` , and that the following functional dependency is true $FD : A \times B \times C \mapsto V$. If so, we can, for example, partition the data in the relation according to the attributes A and B, so that $P = \{A, B\}$. The first consequence of doing this is that there will be N distinct partitions where every tuple has the same data values in cells associated to the attributes A and B. The second consequence is that we can no longer affect the values of those attributes. And the third and most important consequence is that in each partition the functional dependency $FD2 : C \mapsto V$ is true. After all, each functional dependency $FD2$ is a domain restriction of FD, where the attributes A and B are equal to the constants $a_i \in A$ and $b_i \in B$, respectively. Finally, we define our dimension set as $D = \{C\}$. Oracle then will create for each logical partition a vector traducing the functional dependency $FD2$ of that partition. Then, we can manipulate the functional dependency $FD2$ of each logical partition to affect the cell values relative to the attribute V of the records assigned to the partition. Rules will be of constructions of this nature: $FD2(c_1) = v_5$, where $c_1 \in DOM(FD2)$ and $v_5 \in V$; $\forall_{c \in DOM(FD2)} FD2(c) = FD2(c_1)$; etc. It is important to mention that the oracle vectors are dynamic, permitting

us to increase the number of rows in the output relation by writing a rule such as $FD2(c_{200}) = v_5$, where c_{200} is a valid C value while not being a valid index position for the partition, that is $c_{200} \in C$ but $c_{200} \notin DOM(FD2)$.

This operator can be used to accomplish *bounded* and *unbounded* one-to-many data transformations. Code Listings 4 and 5, illustrate how the *bounded* and *unbounded* data transformations relative to Scenarios 1 and 2 of Section 1.1, respectively, can be solved using this implementation method.

Code Listing 4. Performing the *bounded* one-to-many data transformation for Scenario 1 using Oracle's *model* implementation method. The first component of the *model* clause, see line 06, tells Oracle that we want to divide the input records by an unique attribute `ID`. Thus, initially, each cluster of data is only comprised by one input record. This will not last, because when we start processing the input record in each partition (lines 9 to 14), we make that record yield four output records, one for each favorite friend. The second component of the *model* clause, see line 07, defines the set of attributes that we have to use when we wish to access the information contained in the records of the partition. Recall, that the `PARTITION BY` attributes together with the `DIMENSION BY` attributes, must uniquely identify each record in a partition. Therefore, we must define a dummy attribute that we call `FINDEX`, which will permit us to add new records into each partition. The third and final initialization component of the *model* clause, see line 08, tells Oracle exactly which attributes of the input records we want to read or write to. With the three components put together, Oracle can create one array for each *measure* (e.g. `FAVF1`, etc). Each of these arrays is indexed by the `FINDEX` attribute. To complete our query (see line 9), we tell oracle in the `RULES` clause to fetch the information of each `FAVFi` array, and to copy it into the `FAVF` array at four different index positions. For instance, the first rule copies into the `FAVF` array at the `FINDEX` 1 position the contents of the `FAVF1` attribute of the input record being processed; the second rule copies into the `FAVF` array at the `FINDEX` 2 position the contents of the `FAVF2` attribute of the input record; and so on. Since each index position in an array corresponds to a different record, we are adding 3 new tuples to each partition (e.g. after the initialisation of the *model* structures, the `FINDEX` values of 2 to 4 are new). When the query finishes, the output records are constructed by doing the inverse process that created the several arrays during the initialisation. In this case, the output records have the schema (`ID`, `FINDEX`, `FAVF1`,.., `FAVF4`, `FAVF`) attributes. In the final output result, se line 1, we project only the data contained in the `ID` and `FAVF` attributes. Although the code is simple to implement, this implementation mechanism even for the *bounded* data transformations is very complex to understand, as the above explnation portrays.

```
01: SELECT USRID, FAVF
02: FROM (
03: SELECT ID AS USRID, FAVF
04: FROM PROFILE P
05: MODEL
06:     PARTITION BY (ID)
07:     DIMENSION BY (1 as FINDEX)
08:     MEASURES (FAVF1, FAVF2, FAVF3, FAVF4, 0 AS FAVF)
09:     RULES ( FAVF[1] = FAVF1[1], FAVF[2] = FAVF2[1], FAVF[3] = FAVF3[1], FAVF[4] = FAVF4[1] )
10: )FAVFRIEND
11: WHERE FAVF IS NOT NUL
```

Code Listing 5. Performing the unbounded one-to-many data transformation for Scenario 2, Section 1.1. The initilisation of the *model* clause (lines 5 ro 7) is identical to the one described in Code Listing 4. The difference being that for this data transformation we only need to refer to the values in the FAVARTISTS attribute of the input relation, see line 7. However, the data transformation processing clause follows a quite distinct strategy. To implement the data transformation we only require one *model* rule, see lines 9 to 12. It recursively copies the list of favorite artists of each user from one record to another, deleting the head element of the list being copied. This process starts by copying into the FINDEX position 2 of the RESTARTISTS array, the list of artists contained in the same array at the FINDEX position 1. It continues to do so until a last *FINDEX* position refers to a list consisting of a single artist. For example, if a user likes 7 artists, the user's list is going to be copied 6 times. In this case, the RESTARTISTS array is going to contain a list of seven artists at the FINDEX position of 1, and only one artist at the FINDEX position of 7. We clarify that this iteration over FINDEX positions is defined in the left side of the *model* rule assignment, lines 9 to 10. The code LENGTH(REGEXP _REPLACE(RESTARTISTS[1], '[^,]',NULL)) , is used to count the number of comma characters in the list of favorite artists at the first RESTARTISTS position. Knowning the number of commas is equivalent to knowing the number of artists in the list, which lets us know how many output records we need to produce. The right side of the assignment, lines 11 to 12, use the code CV()-1 to identify the last FINDEX position, from which we extract its list of artist with the head element removed. Finally, when we print our final output results, lines 1 and 2, we only present the head element of list of artists of each output record (line 2), thus completing our desired data transformation. This is the most complex implementation method, and the code we discussed is obviously non evident.

```
01: SELECT ID as USRID,
02:     RTRIM(LTRIM(SUBSTR( RESTARTISTS, 1, INSTR(RESTARTISTS, ',') - 1))) as ANAME
03: FROM PROFILE P
04: MODEL
05:     PARTITION BY (ID)
06:     DIMENSION BY (1 as FINDEX)
07:     MEASURES (CONCAT(FAVARTISTS, ',') as RESTARTISTS )
08:     RULES(
09:         RESTARTISTS[ FOR FINDEX
10:             from 2 TO LENGTH(REGEXP_REPLACE(RESTARTISTS[1], '[^,]',NULL)) increment 1]
11:         = SUBSTR(   RESTARTISTS[CV()-1], INSTR(RESTARTISTS[CV()-1], ',') + 1,
12:             LENGTH(RESTARTISTS[CV()-1]) - INSTR(RESTARTISTS[CV()-1], ','))
13:     );
```

2.1.1.6 Unnest

The SQL standard 2003 introduced the support of a new collection type, the *multiset*. Fundamentally, two collection types are supported in this standard: *arrays* and *multisets*. While arrays are constricted by a maximum size and are dense, multisets are conceptually unbounded and they can be sparse[3] (ORACLE & Moore, 2008). The unnest operator is the mechanism by which a collection is converted into a table, making it possible to use collections in the *from* clause of a *select* statement(Eisenberg et al., 2004).

[3]In the Oracle RDBMS for example, the SQL collections types *array* and *multiset* directly map to Oracle *vararray* and *nested table* types.

Oracle supports two collection data types, nested tables and arrays. Nested tables are the return type of *table functions*. Arrays are collections that can be converted into tables by means of the *unnest*. DB2 also supports an *unnest* operator with the purpose of allowing the conversion of arrays into tables. However, this system only allows the use of this operator in the body of a stored procedure. In DB2, the *unnest* can be used as a query mechanism similarly to Oracle. Since the declaration of an array must specify its maximum size, this data type cannot be used to support the *unbounded* one-to-many data transformations, because we do not know before-hand the number of output records that each input record will yield.

This transformation mechanism is illustrated in Code Listing 6, just following, solving the *bounded* data transformation presented in Scenario 1 Section 1.1.

Code Listing 6. Performing the *bounded* one-to-many data transformation for Scenario 1, Section 1.1, applying the unnest operator of Oracle. The `TABLE` operator used in the `FROM` clause at line 5, has the same semantics as the SQL 2003 *unnest* operator. It permits the use of any collection type in this clause.

```
--1st create a row type
01: CREATE OR REPLACE TYPE t_favfriend_row AS OBJECT (FAVF int);
--2nd create an array with elements of type "row of favfriends"
02: CREATE OR REPLACE TYPE t_favfriend_varray AS VARRAY(4) OF t_favfriend_row;
--3rd the query:
03: SELECT ID, FAVF
04: FROM   PROFILE P,
05:         TABLE(t_favfriend_varray( t_favfriend_row(P.FAVF1), t_favfriend_row(P.FAVF2),
06:                 t_favfriend_row(P.FAVF3), t_favfriend_row(P.FAVF4) ))
07: WHERE FAVF IS NOT NULL
```

2.1.1.7 The mapper extension

Paulo Carreira (Carreira et al., 2007) has conducted very significant work on the subject of one-to-many data transformations. His work was two fold: practical and theoretic. In his practical work, he concluded that RDBMS did not provide an adequate declarative support for this category of data transformations, and that an SQL extension would be most relevant. It was also concluded that the execution of complex one-to-many queries, where other relational algebra operators such as selections are present, do not result in optimal query plans.

The theoretic/formal research consisted on defining an SQL extension operator, named *mapper*, to support the execution of one-to-many data transformations in a declarative manner, providing an adequate SQL Language syntax for the operator, and a set of algebraic optimizations compatible with this category of data transformations.

The *mapper* operator is defined as a triple $\langle A, B, F \rangle$, where A specifies a set of output attributes, B a set of relevant attributes existing in the input relation and F is a set of *proper mapper functions*. Each *mapper function*, composing the *mapper operator*, takes as input data conforming to a subset of attributes contained in B. The data returned, consisting of a set of output tuples, conforms to a subset of attributes contained in A. Therefore, the simplest *mapper operator* only contains a *mapper function* which would be a triple of the form $\langle A, B, f \rangle$.

Here, f is a function of the form $f : Dom(A) \mapsto P(Dom(B))$ (Carreira et al., 2007). The simplest form of the *mapper* is obviously expressive enough to allow any one-to-many data transformation. However, it was the author's understanding that having the *mapper* operator allow the use of a set of distinct mapping functions would be better. This is due to the fact that complex one-to-many data transformations, where several distinct attributes may originate several output tuples, can be made easier through the definition of several individual mapping functions. The only constraint is that the output attributes of each *mapper function* are non intercepting. It is also proven that a *mapper operator* using several distinct mapping functions is equivalent to a *mapper operator* using a single mapping function, with mathematical expression equal to the *Cartesian* product of each *mapper function* belonging to the more complex mapper operator.

Finally, several algebraic optimizations are given for one-to-many data transformations, such as anticipating selections. These optimizations would take into account if the filters were defined as *a-priori filters* or *a-posteriori filters*. An *a-priori filter* can be applied before the data transformation without violating the expected results. On the other hand, *a-posteriori filter* can only be applied after the data transformation takes place. However, if the conditions for applying the proposed algebraic optimizations are satisfied then, for example, anticipating selections before the Cartesian product of mapper functions will yield the expected results.

The work in (Carreira et al., 2007) constitutes the most complete study produced on the subject of one-to-many data transformations.

2.1.2 RDBMS Persistent Stored Modules

2.1.2.1 Stored Procedures

A *stored procedure* is a construct type supported by the SQL Persistent Stored Modules (SQL/PSM) extension introduced in SQL 1999 revision (Melton & Simon, 2002). SQL/PSM extension is supported by varied procedural languages depending on the specific RDBMS, namely T-SQL in Microsoft SQL Server, SQL/PL in Oracle an PL SQL in DB2. The extension enhances RDBMS with programming language capabilities such as the use of control statements, e.g. If-Then-Else and While statements. Stored procedures process data (usually stored on one or more relations) and can store the resulting data into one or more relations. In this procedural design, each tuple can be processed into many output tuples. Therefore, *stored procedures* are expressive enough to support one-to-many data transformations of any kind. However, the more dependent the created routines are on procedural constructs, such as While loops, the less prone they are to being optimizable. Moreover, *stored procedures* cannot be used in the `FROM` clause of queries. They are not meant to be used nor can they directly be used as a query mechanism.

The used of *stored procedures* is illustrated in Code Listing 7, just following, solving the *unbounded* data transformation presented in Scenario 2 Section 1.1.

Code Listing 7. Performing the *unbounded* one-to-many data transformation for Scenario 2, Section 1.1, using an Oracle's stored procedures. This particular solution uses a cursor (line 3) to iterate (lines 6 to 15) through the input table. It writes the output of each processe row into a temporary global table named *Temp_P_Results*, see line 12. This solution omits the creation of the temporary table.

```
01: CREATE OR REPLACE PROCEDURE hr.SP_OneToMany_On_Artists
02: IS
03:         CURSOR c_Profile IS (select EMAIL, COUNTRY, CITY, FAV_Artists from  profile);
04:         resultRow TEMP_P_Results%rowtype;
05: BEGIN
06:         for row_profile in c_Profile loop
07:                   while row_profile.fav_artists IS NOT NULL loop
08:                      resultRow.user := row_profile.email;
09:                      resultRow.country := row_profile.country;
10:                      resultRow.city := row_profile.city;
11:                      resultRow.fav_artist  := udf_firstitem(row_profile.fav_artists);
12:                      insert into TEMP_P_Results values resultRow;
13:                      row_profile.fav_artists := udf_restitems(row_profile.fav_artists);
14:                   end loop;
15:        end loop;
END;
```

2.1.2.2 Table Functions

Functions are similar to *stored procedures* in the sense that they are also a SQL/PSM type of construct (Melton & Simon, 2002), and they can take advantage of most types of statements permitted in stored procedures (e.g. while loops). However, functions and *stored procedures* are fundamentally different in their uses. *Stored procedures* are meant to be used as mechanism to perform auxiliary routines in a database system (e.g. verifying that data satisfies complex integrity constraints before being inserted into a table, by means of a trigger). They are not meant to be used as a query mechanism. Functions on the other hand, are meant to be used wherever they are needed (e.g. in queries, other functions, etc). When they are used, despite being procedural constructs, they may not even compromise seriously the declarative nature of a query. A query can invoke functions at multiple levels, for example, in the `SELECT` , `FROM` and `WHERE` clauses.

Table functions, which are a special type of functions, are particularly useful in the context of designing one-to-many data transformations. They are meant to be invoked in the `FROM` clause of a query, since their output is a relation instance. This feature is especially interesting because it gives the user a query mechanism with Turing-complete expressive power. For instance, it allows us to code data transformations where one input record is processed into several output records.

Depending on the RDBMS, *table functions* may not even compromise the declarative design of a query. For instance, in Oracle, a *table function* can be coded as a simple multi-value function, where the user only codes the manner by which single-valued input (e.g. a record) can be transformed into the desired set of output records (e.g. a relation), see Code Listing 8. In this case, only a very small portion of the query would be coded by means of procedural constructs, without compromising the declarative design of the query anymore than a *scalar function* appearing in the `SELECT` clause of the query would. However, in other RDBMS (e.g. MSQLS), a *table function* may have to deal with multiple aspects that compromise the declarative design of the queries that use it. For example, a MSQLS *table function* has to iterate over its input records by means of cursors, since it does not support the invocation of the *table function* in a tuple-by-tuple basis. Consequently, the *table function* would have

to be recompiled in a number of situations. For instance, whenever the source table name changed, its predicate clauses on data access where modified, etc.

We illustrate in Code Listing 8, the use of an Oracle *table function* for solving the *unbounded* data transformation presented in Scenario 2 Section 1.1.

Code Listing 8. Performing the *unbounded* one-to-many data transformation for Scenario 2, Section 1.1, using the Oracle *table function* implementation method. Code lines 1 and 2 define the Oracle data types that are necessary to declare the *table function*. The first can be interpreted as the delcartion of a row type. The second as the declartion of relation type where rows must have the previously defined row type. This *table function* accepts as input a string representing a list of favorite artists, and processes this input into a an output relation containing rows with the individual artist names from that list. The function body is, in essence, a cycle where in each iteration an artist is removed from the list of artists and returned as output, see lines 19 to 25. The invocation of this *table function* is illustrated in the lines 28 and 29. Notice that the *table function* does not know where its input comes from, its the declarative query in the last two lines that specifies which input should be passed to the *table funtion*.

```
01: CREATE OR REPLACE TYPE t_favartist_row AS OBJECT (ANAME varchar(300));
02: CREATE OR REPLACE TYPE t_favartist_table AS TABLE OF t_favartist_row;

03: CREATE OR REPLACE FUNCTION exp3_totable(FAVARTISTS varchar)
04: RETURN t_favartist_table PIPELINED
05: AS
06:   list_artists varchar(301);
07:   head_element varchar(300);
08:   separator_pos int;
09    len int;
10: BEGIN
11:   IF FAVARTISTS IS NULL OR FAVARTISTS = '' THEN
12:         RETURN ;
13:   END IF;
14:   list_artists := CONCAT(FAVARTISTS, ',');
      -- the 1st iteration happens outside the cycle: 'artist,' would break the while
15:   len := LENGTH(list_artists);
16:   separator_pos := INSTR(list_artists, ',');
17:   head_element := RTRIM(LTRIM(SUBSTR(list_artists, 1, separator_pos - 1)));
18:   PIPE ROW (t_favartist_row(head_element));
      -- condition = true => there are still some artists left to process
19:   WHILE len > separator_pos LOOP
20:         list_artists := SUBSTR(list_artists, separator_pos + 1, len - separator_pos);
21:         separator_pos := INSTR(list_artists, ',');
22:         len := LENGTH(list_artists);
23:         head_element := RTRIM(LTRIM(SUBSTR(list_artists, 1, separator_pos - 1)));
24:         PIPE ROW (t_favartist_row(head_element));
25:   END LOOP;
26: RETURN ;
27: END;
```

```
-- Query Ivocation of the table function
28: SELECT  P.ID as USRID, T.ANAME
29: FROM PROFILE P, TABLE(exp3_totable(P.FAVARTISTS)) T;
```

2.1.3 AJAX

The AJAX framework builds upon an RDBMS extending the SQL language transformation capabilities by providing five new logical operators[4] (Galhardas et al., 2001). AJAX was designed to support a declarative data cleaning processes where, a data cleaning process is defined as a workflow of data transformations using the supported operators.

In the context of our work, AJAX provides a *mapping* operator capable of performing both bounded and unbounded data transformations. Its mapping clause is composed by the following three sections: *(i)* create clause, *(ii)* let clause and *(iii)* output clause. *The create clause* assigns both a name to the workflow step and designs the input (relation) of the transformation step. *The let clause*, consisting of an arbitrary number of assigned relational variables, applies external functions to tuples in the input relation and any previously defined relation variables. We can informally interpret that the operator is computing one big temporary relation, denoted as Tn, where each tuple in the previously computed relations $Tn-i$ is joined to the output of the current external function. Formally, we start with $T0 = I$, where I denotes the input relation. Afterwards, we compute the first relation variable $T1$ by performing the following computation $T1 = \cup_{t0 \in T0}\{t0\} \times f1(t0)$. And so on. Therefore, we can recursively define the obtained temporary table computed in a mapping as: $Tn = \cup_{tn-1 \in Tn-1}(\{tn-1\} \times fn(tn-1))$ with $T0 = I$. A physical algorithm for computing this temporary relation could be a nested loop with depth $n+1$, where at depth one the input table is being iterated, at depth two the output of the external function $f1$ is being iterated and at depth n+1 the output of the external function fn is being iterated. Finally, the mapping algorithm ends by creating an arbitrary number of output relations where only the desired attributes from the relation Tn are projected into.

As an example, see Code Listing 9, we perform the unbounded one-to-many data transformation from scenario 2 by specifying a function $f1 = udf\_unbounded\_tf$ and joining each input tuple with the output of the given function.

Code Listing 9. Processing the unbounded one-to-many data transformation introduced in Scenario 2 using the AJAX mapping operator. Lines 1 and 2 constitute the create clause, line 3 constitutes a let cause defining only one relation variable (separated_artists) and lines 4 and 5 constitute one output clause routing the temporary results to only one output relation (PROFILE_TRANSFORMED). For more complex illustrations of mapping routines refer to (Galhardas et al., 2001).

```
01: CREATE MAPPING unbounded_favartists_to_favartist
02: FROM profile
03: LET separated_artists = udf_unbounded_tf(PROFILE.FAVARTISTS)
04: {SELECT profile.email, profile.country, profile.city, separated_artists.artist
05: INTO PROFILE_TRANSFORMED}
```

[4] Mapping, view, matching, clustering and merging.

2.2 Transforming XML data

XML is presently both *(i)* a logical data model and *(ii)* a physical data format. It is a logical data model, as are the relational and object-oriented, since data in XML has an underlying logical structure, that of a tree of labeled nodes (for simple XML files), which provides a platform for the emergence of query languages compatible with it, namely XPath and XQUERY (Abiteboul et al., 1999). XML is also a physical data format since, the logical data elements consisting of a set of strings representing tags, attributes and values, are stored directly in physical files as streams of bytes following an arbitrary character encoding (normally UTF-8).

Although logical data models are in theory independent of the physical data structures representing them, in practice, with XML, both the physical and logical representations are virtually the same. In fact, the distinction between a logical model and physical model for Web data, namely HTML and XML, is presently not recognized (Abiteboul et al., 1999). However, Abiteboul argues that with the increasing use of XML, eventually, data will be stored in more efficient ways, such as compressed text files so as to improve the performance of handling XML data (Abiteboul et al., 1999). If and when this happens, XML will be exclusively considered a semi-structured logical data model.

Presently, XML is widely used as a data exchange format for three reasons: *(i)* flexibility, *(ii)* portability and *(iii)* powerful query mechanisms. First, the XML language can flexibly represent data and data relationships. Second, the physical representation of the said data is platform independent[5], as with all Web formats, making XML data files compatible with heterogeneous systems. Finally, the existence of powerful standards for processing XML data, namely XSLT and XQuery, make the data highly usable.

We will be reviewing two standard technologies for processing XML data, the XSLT and XQuery.

2.2.1 XSLT

XSLT can be regarded as a functional programing language with syntax composed of predefined XML tags. The developer, through the definition of a set of XSLT templates, will be able to process input XML files into some output text file, normally HTML or XML.

While XML files are content oriented, as opposed to HTML files which are presentation oriented, the XSLT standard is specifically meant to allow the easy transformation of XML content into a human readable presentation format in HTML (Abiteboul et al., 1999). As such, there are some limitations that arise when using XSLT as a query language: Abiteboul states that XSLT is not relational complete since it does not the support the relational join operation (Abiteboul et al., 1999).

The XSLT standard has considerable expressive power, in fact, it is Turing complete [6]. Not only can the templates be regarded as recursive functions as, the use of special loop statements, such as the *xsl:for-each*, is

[5]Most character encodings are heterogeneous as they normally use 8-bit/1byte character codes not arising problems of endianess/byte-order. Text files encoded UTF-16, for example, start with a hidden byte-order mark (BOM) determining the endianess of the file. Therefore, text files are in general heterogeneous data sources.

[6]IBM URL: `http://www.ibm.com/developerworks/xml/library/x-tiploop.html`

supported. Furthermore, even if there is no primitive for performing the cartesian product of XML documents, the developer may always access external XML documents through the *document* function and program by hand (ad-hoc) the semantics of a join operation in a template. Still, there are more convenient means of processing such data transformations, for instance, by using other XML languages meant for query processing. As far as selections and projections are concerned, XSLT can very conveniently perform these over a given input XML file, the first by using XPath expressions and the later by defining the desired output content in the body of a template.

We will illustrate in code listing 10 the execution of the unbounded one-to-many data transformation relative to Scenario 2.

Code Listing 10. Performing the unbounded one-to-many data transformation for Scenario 2, Section 1.1, using the XSLT standard version 2.0. We tested this stylesheet to process an input data file that contained a *profile* root element, which contained several *row* elements, each composed by one *id* element and one *favartists* element. This stylesheet contains a template that matches the *profile* root element (see lines 3 to 12), and creates an output *favartist* root element representing the target output table (see lines 4 and 11). It processes each *row* element inside the *profile* (see lines 5 and 10), and returns one output *row* for each artist token in the *favartists* element of the input *row* (see lines 7 to 9).

```
01: <?xml version="2.0" encoding="UTF-8"?>
02: <xsl:transform version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
03:     <xsl:template match="/profile">
04:         <favartist>
05:             <xsl:for-each select="row">
06:                 <xsl:variable name="uid"><xsl:copy-of select="id"/></xsl:variable>
07:                 <xsl:for-each select="tokenize(favartists, ',' )">
08:                     <row> <xsl:copy-of select="$uid"/> <favf><xsl:value-of select="."/></favf> </row>
09:                 </xsl:for-each>
10:             </xsl:for-each>
11:         </favartist>
12:     </xsl:template>
13: </xsl:transform>
```

2.2.2 XQuery

XQuery which was introduced in 2007 is a W3C standard, as is XSLT, although introduced much earlier in 1999, and like XSLT, XQuery is also Turing complete (Kepser, 2004). However, contrary to XSLT, XQuery is from origin specifically designed to be used as a query language over XML data. More specifically, this query language is meant to easily support common query processing activities such as: selecting, filtering, joining, grouping and sorting XML data from a collection of documents or any other sources (Walmsley, 2007).

Presently, XQuery is supported in a variety of platforms. There are stand alone XQuery processors capable of handling XML data files stored in a filesystem, and there are embedded XQuery processors in relational database systems such as Oracle, DB2 and SQL Server and, of course, in native XML database systems (Walmsley, 2007).

The syntax of XQuery is much simpler than that of XSLT and more familiar to developers that have used the SQL language. Typically, a simple query consists of a *flwor* expression, that is, there is a *for* clause for iterating over the elements in a XML file, there is a *let* clause for assigning values to variables, there is a *where* clause for filtering undesired elements, there is an *order by* clause for sorting the output data and finally there is a *return* statement for expressing the output data and its respective format (Walmsley, 2007). Furthermore, nesting XQueries is also possible, as will be seen in the succeeding code example. Still, a XQuery may be as simple as a single XPath expression.

We will illustrate in code listing 11 the execution of the unbounded one-to-many data transformation relative to Scenario 2.

Code Listing 11. Performing the unbounded one-to-many data transformation for Scenario 2, Section 1.1, using the XQuery standard. The query starts by defining that the output result set is enclosed by a root *favartist* element (see lines 1 and 9). It opens the input data file and processes each input *row* element individually (see line 3). For each input *row*, we tokenize the *favartists* string value (see line 5). Finally, for each artist token, we return one output *row* comprised by one *id* and *aname* element (see line 7).

```
01: <favartist>
02: {
03:     for $r in doc("profile_rel.xml")//row
04:     return
05:         for $tok in tokenize($r/favartists, ", *")
06:         return
07:             <row> {$r/id} <aname>{$tok}</aname> </row>
08: }
09: </favartist>
```

3 Implementation of one-to-many data transformations

The main goal of this thesis is to demonstrate how relational technology supports the implementation of one-to-many data transformations. In this chapter, we illustrate the relevant types of one-to-many data transformations and the corresponding implementations using three commercial RDBMS. With the purpose of illustrating one-the-many data transformations in the three main contexts that they are applied, we enhance the motivating example of the Face5 company presented in Section 1.1.

In Section 3.1, we start by reviewing the database schema of Face5 legacy database and its problems. Then, we propose a different database schema that can adequately support the Face5 workload. Finally, we define the extended *Relational Algebra* (RA) expressions that implement the schema mapping from the source to the target database. In Section 3.2, we present the implementations of the schema mapping expressions, which involve one-to-many data transformations, using the different RDBMS implementation routes that were detailed in Section 2.1.1 and Section 2.1.2.

3.1 Database schema transformation

In Chapter 1, we introduced a motivating example where a company named Face5 is having difficulty to perform a number of queries due to the underlying database schema. The solution we proposed in Section 1.1 was to create a new database schema and migrate the data from the old to the new database. In this section, we start by reviewing the Face5 database schema and the queries that cannot easily be performed over it. Then, we propose the data model for the target database, which should easily support the query requirements of Face5, and present the corresponding database schema mapping. Finally, we detail the extended RA expressions that implement the schema mapping from the source to the target database.

3.1.1 Source data schema

In Section 1.1, we presented the database schema of Face5. In this section, we use an enhancement of that schema that consists of a single relation, the `PROFILE` relation. This relation contains all the information that existed in the previous database schema and additional information that is used to present a third data transformation problem that was omitted in Chapter 1.

The purpose of this thesis is to present and study the performance of one-to-many data transformations in the context of the three essential problems that they solve: *(i)* eliminating identical columns in relations containing semantically equivalent attributes; *(ii)* unnesting data in relations containing multivalued data; and *(iii)* reversing

group by operations in relations containing aggregated data. Out of these three classes of problems, only the first two were introduced in Section 1.1. Hence, we start this section by reviewing the two scenarios that were previously discussed, and we introduce the third and final scenario, which is representative of the third class of problems that one-to-many data transformations can solve: reversing *group by* operations.

The three scenarios that follow illustrate the workload that Face5 wishes to execute and for which the current database schema has proven to be an obstacle.

1. Determining the users that are the most popular in terms of their favorite friendship links, that is, Face5 wants to count, for each user, the number of times that he/she has been picked as a favorite friend by other users. This problem has been discussed in Scenario 1 of Section 1.1. It results from the fact that there exists more than one attribute for mapping this favorite friend association, namely the attributes ranging from FAVF1 to FAVF4 . This scenario illustrates the problem of having semantically equivalent attributes in a relation.

2. Determining the artists that are the most liked in the Face5 community, that is, Face5 wants to determine the number of times that each artist has been picked as a favorite artist by distinct users. This problem has been discussed in Scenario 2 of Section 1.1. It results from the fact that the artists that each user likes are nested into a single string value in the attribute FAVARTISTS . This scenario illustrates the problem of having aggregated data in a relation.

3. In Face5, every user has a record of the number of times that his/her profile page has been visited by other users. This number is stored in the attribute NVIEWS of the PROFILE relation and displayed on the top of each user's page, labeled as "number of views". Face5 is preparing a system of selective placement of ads in a restricted number of users' profile pages. The board of directors has decided that, each month, one hundred pages should hold internet ads. At the beginning of each month, the one hundred pages with the highest average of profile visits per month must be selected to hold the ads. It is also desirable that, in the computation of this result, the most current months of activity of a given user have an higher influence in the user's obtained result. That is, a weighted average should be applied benefiting the evaluation of users that have been popular in the most recent months of activity, over the users that were popular in the past. The pages determined by this process are supposedly more likely to lead to ad clicks.

The problem imposed by Scenario 3 is that, in order to implement the desired ad placement system, Face5 has to record the number of profile visits per month for each user. Currently, the Face5 database only registers the total number of visits of each profile, in the NVIEWS attribute. This number concerns a time period ranging from the user's registration date, recorded in the REGDATE attribute, until the current system date. Therefore, this information has to be transformed so that the number of profile visits are stored in monthly records. That is, the information should then be stored in a relation VIEWHISTORY=(UID, YEAR, MONTH, VCOUNT) , where UID identifies the user, YEAR and MONTH identify the time period that a record describes, and VCOUNT specifies the number of views that were counted during the time period specified by the pair (YEAR , MONTH) for the user.

Table 3.1 and 3.2 illustrate the data transformation imposed by Scenario 3 , where we assume that the current system date is some day in MAY/2009. The source relation instance PROFILE that is illustrated in Table 3.1 only

shows a subset of its attributes, those that are relevant for the example. The data transformation is an unbounded one-to-many data transformation since the number of output tuples depends on two variables: the current system date and the register date of a user. Furthermore, we assume that the total number of views contained in the input tuples are uniformly distributed by the resulting output tuples. That is, if an input tuple t originates a set S of output tuples t', then for each t', its VCOUNT attribute value should be

$$t'[\text{VCOUNT}] = \frac{t[\text{NVIEWS}]}{|S|} \tag{3.1}$$

However, this last formula may produce values that are not integers. Therefore, for every output tuple t' with respect to a (YEAR ,MONTH) time period less than that of the current system date, we round-down the result of the previous formula, thus making

$$t'[\text{VCOUNT}] = \lfloor \frac{t[\text{NVIEWS}]}{|S|} \rfloor \tag{3.2}$$

Finally, to the output tuple t' with attributes (YEAR ,MONTH) correspondent to the current system date, it is added the remainder of the division, thus making

$$t'[\text{VCOUNT}] = \lfloor \frac{t[\text{NVIEWS}]}{|S|} \rfloor + t[\text{NVIEWS}] \; mod \; |S| \tag{3.3}$$

Let us consider the application of this formula when processing the PROFILE record $t =$ (1, Popular Girl, 15/FEBRUARY/2009, 4002), illustrated in Table 3.1. First, we have to determine the number of output tuples, denoted as $|S|$, produced by the one-to-many data transformation applied to this record. Since between the user's registration date, given by $t[\text{REGDATE}]$, and the current date, given by some built-in function of the RDBMS, there are four months: February, Mars, April and May; the transformation will produce four output tuples, that is $|S| = 4$. Each of the output tuples will contain a valid month in the described time period, its corresponding year and finally an estimated number of profile visits. Second, having determined that processing the tuple t will yield four output tuples, we can calculate the VCOUNT value of each tuple by applying the formulas described above. Since this user's profile has been visited $t[\text{NVIEWS}] = 4002$ times, the user's average number of visits per month is $\frac{4002}{4} = 1000.5$. This value results from applying Formula 4.1 to the tuple t. However, the value produced by this formula is not an integer, hence we have to apply the floor operation to the obtained result. This implies that for the output tuples with months ranging from February to April the VCOUNT value is $\lfloor 1000.5 \rfloor = 1000$, obtained by applying Formula 4.2 to t; and it also implies that the last output tuple produced, that corresponds to May, has a VCOUNT value of $\lfloor 1000.5 \rfloor + 4002 \; mod \; 4 = 1002$, obtained by applying Formula 4.3. These output tuples are illustrated in Table 3.2.

ID	NAME	REGDATE	NVIEWS
1	Popular Girl	15/FEBRUARY/2009	4002

Table 3.1: Scenario 3 input: PROFILE instance.

From now on, we consider that Face5 legacy database consists of a single relation PROFILE with schema:

PROFILE=(ID__, NAME, COUNTRY, CITY, FAVARTISTS, NVIEWS, REGDATE, FAVF1, ..., FAVF4)

FAVF1 : FK(PROFILE.ID) , ..., FAVF4 : FK(PROFILE.ID)

UID	YEAR	MONTH	VCOUNT
1	2009	2	1000
1	2009	3	1000
1	2009	4	1000
1	2009	5	1002

Table 3.2: Scenario 3 output: VIEWHISTORY instance.

ID is the primary key of the relation and it uniquely identifies each user of Face5. NAME is the attribute that contains the user's name. COUNTRY and CITY are two attributes that geographically locate the user. These attributes can be used for facilitating queries that search for users in specific regions. For instance, they can be used whenever a user searches for other users that live close to him/her, namely in the same country and city. NVIEWS holds the number of times that a user's profile was visited by other users of Face5. People with a high view count are users that are popular or attractive inside the Face5 community. REGDATE is the date of registration of a user in the Face5 community. Members with an older registration date have been members of the community for a longer period than users with a more recent registration date. FAVF1 to FAVF4 are attributes that identify, for each user, his/her favorite friends. Therefore, these four attributes allow each user, individually, to select up to four other users as his/her favorite friends.

3.1.2 Target data schema

The set of problems described in Section 3.1.1 led Face5 to migrate its database to a new one with a more appropriate relational schema. In this section, we describe the Entity-Relationship (ER) model and the corresponding relational schema that Face5 has adopted to replace the older relational schema, which was presented in Section 3.1.1. Finally, we present the *schema matching* between the source and target relational schemas, that illustrates the correspondence between the attributes of the source and target relational schemas.

Face5 started its data migration solution by modeling the entities that should exist in the target database and their relationships. The ER model is illustrated in Figure 3.1.

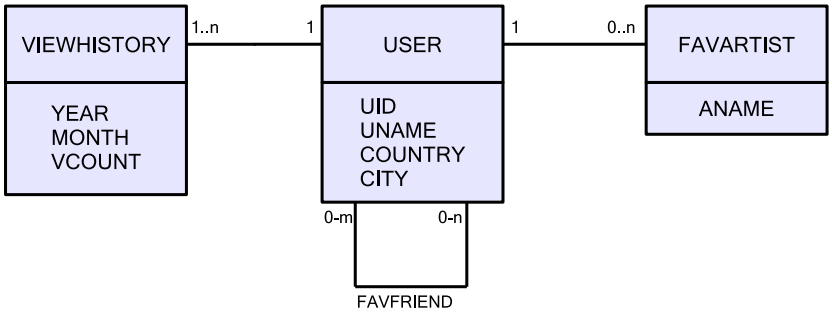


Fig. 3.1: ER model of Face5 target database.

The model entities are the following:

1. The USER entity holds the user's unique identification in UID, the user's name in UNAME and the user's location in the COUNTRY and CITY attributes.

2. The FAVARTIST entity holds each user's favorite artist name in ANAME .

3. The VIEWHISTORY entity holds information concerning the time period specified by (YEAR , MONTH) attributes, and describes the number of times a user's profile was visited during that time period, in the VCOUNT attribute.

The model entities are related to one another by the following relationships:

1. The association FAVFRIEND relates a user to his/her favorite friends. Each USER instance is related to possibly many USER instances, constituting the user's set of favorite friends.

2. The association between the USER and FAVARTIST entities relates a user to his/her artist preferences. Each USER instance is related to possibly many FAVARTIST instances, and in turn, each FAVARTIST instance is related to a single USER instance.

3. The association between the USER and VIEWHISTORY entities relates a user to his/her monthly history of profile visits. Each USER instance is related to as many VIEWHISTORY instances as months passed since the user's registration, and in turn, each VIEWHISTORY instance is related to a single USER instance.

A possible relational schema that corresponds to the ER model illustrated in Figure 3.1 consists of the following four relations:

USER=(UID, UNAME, COUNTRY, CITY)

FAVFRIEND=(UID, FAVF)
UID:FK(USER.UID) , FAVF:FK(USER.UID)
IC: one UID is associated to a maximum of four FAVF values.

FAVARTIST=(UID, ANAME)
UID:FK(USER.UID)

VIEWHISTORY=(UID, YEAR, MONTH, VCOUNT)
UID:FK(USER.UID)

FK stands for Foreign Key, IC for Integrity Constraint, and the attributes that constitute the Primary Key of every relation are underlined.

Finally, Table 3.3 contains the set of matching pairs that fully describe which attributes of the source schema map to attributes in the target schema.

3.1.3 Schema mapping

We use the Relational Algebra extended with the mapper operator (Carreira et al., 2007), presented in Section 2.1.1.7, to write schema mapping expressions that implement the schema matchings in Table 3.3. These schema mapping expressions can be seen in Table 3.4.

PROFILE		USER
ID	↦	UID
NAME	↦	UNAME
COUNTRY	↦	COUNTRY
CITY	↦	CITY

PROFILE		VIEWHISTORY
ID	↦	UID
	↦	YEAR
REGDATE	↦	MONTH
	↘	
NVIEWS	↦	VCOUNT

PROFILE		FAVARTIST
ID	↦	UID
FAVARTISTS	↦	ANAME

PROFILE		FAVFRIEND
ID	↦	UID
FAVF1		
FAVF2	↦	FAVF
FAVF3		
FAVF4		

Table 3.3: Schema matchings between source and target attributes.

	Extended RA expression	Target
$E1$	$\rho_{\texttt{ID}\mapsto\texttt{UID},\texttt{NAME}\mapsto\texttt{UNAME}}\,(\Pi_{\texttt{ID},\texttt{NAME},\texttt{COUNTRY},\texttt{CITY}}\,(\texttt{PROFILE}))$	USER
$E2$	$\mu_{renameid,favf}(\texttt{PROFILE})$	FAVFRIEND
$E3$	$\mu_{renameid,unpacknames}(\texttt{PROFILE})$	FAVARTIST
$E4$	$\mu_{renameid,monthlyviews}(\texttt{PROFILE})$	VIEWHISTORY

Table 3.4: Extended RA expressions for transforming the PROFILE source relation into the distinct target relations.

The data transformation expressed by $E1$ consists of a projection over the attributes ID, NAME, COUNTRY, CITY of the PROFILE relation, followed by the renaming of the attributes ID and NAME to UID and UNAME respectively.

The data transformation expressed by $E2$ consists of a mapper operator that uses two proper mapper functions to process its input: $renameid_{\texttt{UID}}$ and $favf_{\texttt{FAVF}}$. The mapper function $renameid_{\texttt{UID}}$ is a single-valued renaming mapper function (Carreira et al., 2007). This function produces a single output UID value that is equal to the value of the input attribute ID. The mapper function $favf_{\texttt{FAVF}}$ is a multi-valued mapper function. This function produces one output value for each favorite friend of a given user. It consumes the information contained in the attributes FAVF1 to FAVF4 and copies it to the attribute FAVF of each output value produced. By applying a Cartesian Product over the results produced by the two mapper functions, the mapper outputs a set of tuples of the form (UID, FAVF), thus relating each user to his favorite friends.

The data transformation expressed by $E3$ consists of a mapper operator that uses two proper mapper functions to process its input: $renameid_{\texttt{UID}}$ and $unpacknames_{\texttt{ANAME}}$. The mapper function $renameid_{\texttt{UID}}$ is the same function used in expression $E2$. The mapper function $unpacknames_{\texttt{ANAME}}$ is a multi-valued mapper function. This function is essentially a string tokenizer function: it produces one ANAME value for each artist contained in the FAVARTISTS attribute of the input tuple. By applying a Cartesian Product over the results produced by the two mapper functions, the mapper outputs a set of tuples of the form (UID, ANAME), thus relating each user to his favorite artists.

The data transformation expressed by $E4$ consists of a mapper operator that uses two proper mapper functions to process its input: $renameid_{\texttt{UID}}$ and $monthlyviews_{\texttt{YEAR, MONTH, VCOUNT}}$. The mapper function $renameid_{\texttt{UID}}$ is the same function used in expression $E2$. The mapper function $monthlyviews_{\texttt{YEAR, MONTH, VCOUNT}}$ is a multi-valued mapper function. This function produces one output value for each month of activity of a given user. The function consumes the information contained in the attributes REGDATE and NVIEWS of the input tuple, and uniformly distributes the value of NVIEWS by the multiple (YEAR, MONTH) pairs. By applying a Cartesian Product over the results

produced by the two mapper functions, the mapper outputs a set of tuples of the form (`UID` , `YEAR` , `MONTH` , `VCOUNT`), thus relating each user to his profile visits history.

The three expressions $E2$, $E3$ and $E4$ contain one-to-many data transformations, and will be used to evaluate the behaviour of three commercial RDBMS. In Section 3.2, we describe the implementations for each of these expressions supported by each RDBMS. We explore the implementation approaches of one-to-many data transformations in RDBMS discussed in Chapter 2.

3.2 Implementations of the schema mapping RA expressions

Multiple RDBMS implementation methods that can be used to support one-to-many data transformations, generically or specialised subtypes, were discussed in detail in Section 2. In this section, we summarize which of these methods that can be used to implement solutions for the schema mapping expressions $E2$, $E3$ and $E4$ (see Section 3.1.3). Table 3.5 matches each of these expressions with the alternative implementation methods capable of supporting them, and identifies the locations in this document where the coded implementations for each specific RDBMS can be found. The commercial RDBMS for which we provide these solutions are Microsoft SQL Server 2008 Enterprise Edition (MSQLS), Oracle 11G R2 Enterprise Edition (Oracle) and IBM DB2 Universal Database (DB2).

	Implementation Mechanisms					
	SPU	*unpivot*	*recursivequery*	*tablefunction*	*model*	*unnest*
$E2$	Section 1.2.2 Code Listing 1	Section 2.1.1.3 Appendix A Code Listing 2, 13		Section A Code Listing 16, 17	Section 2.1.1.5 Code Listing 4	Section 2.1.1.6 Appendix A Code Listing 6
$E3$			Section 2.1.1.4 Appendix A Code Listing 3, db2	Section 2.1.2.2 Appendix A Code Listing 8, 18	Section 2.1.1.5 Code Listing 5	
$E4$			Section A Code Listing 15, db2	Section A Code Listing 19, 20	Section A Code Listing 21	

Table 3.5: RDBMS implementation mechanisms that support the schema mapping expressions presented in Table 3.4

Two important exceptions must be mentioned. First, we do not provide any implementation solution for any of the three schema mapping expressions using the DB2 *table function*. This mechanism has limited expressive power in this RDBMS. Unlike Oracle and MSQLS, this RDBMS does not support any type of internal table variable onto which output results can be appended record-by-record. Instead, its syntax enforces the *table function* return statement to be a standard query. Second, we do not provide any implementation solution for the schema mapping expression $E2$ using the *recursive query*, although this method has sufficient expressive power to support it. However, the solution would be unreasonably complex for such simple data transformation. There are other implementation methods that can readily provide simpler solutions (e.g. the *SPU*).

4 Experiments: system configuration and tests

The main goal of this chapter is to report on the performance of different RDBMS when implementing one-to-many data transformations. We tested the following three commercial database systems: Oracle 11g Enterprise Edition (ORACLE), Microsoft SQL Server 2008 Enterprise Edition (MSQLS), and DB2 Universal Database (DB2). In Section 2.1.1 and Section 2.1.2, we discussed the implementation mechanisms that can be used to execute these data transformations in RDBMS. Afterwards, in Section 3.1, we discussed the operational requirements of a fictional company Face5, and presented a conceptual solution for them in which three one-to-many data transformations expressions - $E2$, $E3$ and $E4$ - had to be processed. Finally, in Section 3.2, we discussed how each specific RDBMS implement these expressions.

In this chapter, we discuss the set of experiments undertaken to benchmark the support and performance of the three expressions $E2$, $E3$ and $E4$ in the three RDBMS. We start, in Section 4.1, by describing the generic RDBMS software installation, which ensures that the RDBMS were run in similar conditions. For this purpose, we consider both the hardware, Operating System (OS) and database configurations.

In Section 4.2, we describe the set of statistics that were gathered during query execution, and which are used, or can be used, to present and understand the benchmark results. A benchmark is an experiment in an RDBMS, where different implementation methods are used to perform the same data transformation and their associated costs are compared. In each experiment, we elect a best performing implementation method based on its *elapsed time* cost.

In Section 4.3, we enumerate the *critical parameters* that affect the performance of one-to-many data transformations and whose effect we wish to measure in our benchmarks (Carreira et al., 2007). We further explain that we organized our experiments in three sets, one for each *critical parameter*, and that each set was comprised by three experiments, one for each data transformation expression.

In Section 4.4, we detail the software application we developed to support the different tests associated to the experiments of our one-to-many data transformations benchmark. We characterize this software tool in terms the essential the processes it supports and the fundamental ideas behind its design.

Finally, in the Sections 4.5, 4.6 and 4.7, we report the obtained results of our experiments and analyse them. Section 4.5 is dedicated to the experiments where we varied the number of input records of a data transformation, Section 4.6 to the experiments where we varied the number of output records produced by a data transformation, and Section 4.7 to experiments where varied the number of input records that were filtered by a selection predicate clause.

4.1 Installation setup

This section describes the hardware specification and software configurations under which the experiments were performed. The hardware and OS conditions were the same for the three RDBMS. As for the RDBMS specific configurations, we tried to equalize them so as to ensure the fairness of the experiments. We describe the configuration parameters that we controlled - and which are common to the three RDBMS; any other configuration parameter was left with the RDBMS vendor default value.

In terms of hardware, the machine running the tests is a Dell OptiPlex 960 computer. The machine is comprised by: an Intel 64 bit (EM64T) Core 2 E8500 processor, with each core running at 3.16GHz; 2 GB of Ram supported by 2x 1GB DDR2 800 MHz dims; and a 250 GB Sata Hard Disk Drive (HDD). The machine is extended by a secondary HDD with 80 GB of storage capacity. This small modification enabled us to distribute the read and write I/O by two independent disks, each with its own communication bus. Thus, reading and writing operations could be performed simultaneously.

In terms of OS, we used the 32 bit Windows 2003 Server Release 2 edition with the Microsoft Windows Service Pack2 installed. We disabled the use of the Windows SWAP/Page file. This has the effect of minimizing the swapping out of used physical memory to disk.

Finally, we present the essential RDBMS tuning aspects to ensure the fairness of our experiments:

Block Size The RDBMS were configured to use a database block size of 8KB (8*1024 Bytes). A database datafile is logically composed by a set of blocks with well defined sizes. Fetching information from a table or any database object that is stored in a datafile can be measured in terms of blocks retrieved, since a block is the smallest unit of information that the RDBMS can retrieve from persistent storage. By controlling its size, we are ensuring that the minimum I/O cost (1 block) is the same for all the databases.

Raw partitions. We used *raw partitions* as database datafiles, instead of the conventional approach that consists of using standard files of a file system as the database datafiles. A *raw partition* is a disk region that comprises a primary or logical partition that has no file system installed on it. Consequently, the OS has no claim over this space. The RDBMS can use each partition as it would use any regular database datafile stored on a file system. By using raw partitions we achieved the following goals: *(i)* the fragmentation of the database datafiles was managed by the RDBMS themselves and not by the OS and its file system; *(ii)* the OS does not use its *system cache* for the disk accesses to regions in raw partitions. Therefore, we were assured that whenever the RDBMS performed an I/O operation, it corresponded to a physical disk access since the memory was not written to, or read from, the OS *system cache*.

Physical location of database objects By using raw partitions instead of OS files ensures that the database objects are stored on well defined disk regions. We defined a set of raw partitions for clustering different types of database objects: *(i)* a *system* raw partition for system objects such as the database catalog, built-in functions and stored procedures, dynamic views for statistics, etc; *(ii)* a *log* a raw partition for storing the database log; *(iii)* a *temporary* raw partition where temporary data, such as materialized tables in non pipelined

operations, was written to and read from; *(iv)* a *source* raw partition where the source data (e.g., the `PROFILE` table of the motivating example introduced in Section 1.1) was stored; *(v)* and a *target* raw partition where the query output results were saved. The raw partition architecture, describing the size and location of each Raw partition, is represented in Table 4.1. Some small adaptations to this base architecture had to be undertaken in each RDBMS, because each system has specificities that make them not fully compatible with this architecture. Further details on this can be consulted in Appendix B.1.

Buffer Cache. We set the database buffer cache to a size of 700 MB. This buffer was used to rapidly access data required by read operations. Accessing data stored in the buffer cache is faster than retrieving it from hard disk.

Logging. Logging cannot be fully disabled. In all RDBMS, a certain amount of logging is always generated to ensure that system critical information, such as the catalog, remains in a consistent state. However, certain operations, for instance updating or creating a table with the SQL statements `INSERT INTO AS SELECT` and `CREATE TABLE AS SELECT` , which are called bulk operations, are normally designed to support long running transactions, that generate - if properly configured - minimal amounts of log. In our experiments, we tried to minimize the log I/O to its minimum in each RDBMS. Therefore we used the above mentioned SQL statements to materialize the output results of our data transformations.

Hard Disk Drive	Partition Name	File System Type	Partition Size
HDD0	Windows 2003	NTFS	88 GB
HDD0	Target	Raw	32 GB
HDD0	Log	Raw	32 GB
HDD1	System	Raw	4 GB
HDD1	Source	Raw	32 GB
HDD1	Temporary	Raw	32 GB

Table 4.1: Architecture of the raw partitions.

4.2 Gathered Statistics

Modern RDBMS automatically gather a wide range of statistics during their execution time. Most of the statistical variables in a RDBMS represent the accumulated value of the occurrences of a measurable event, counted since the moment the database started running. For example, a RDBMS may keep track of the number of CPU cycles it consumed since it was started. Thus, to measure the activity of a database in a specific time interval, it is necessary to record the set of statistic variables we want to control at two different moments in time. Each of these images of variables is called a *snapshot*. Finally, by calculating the difference between two snapshots of a statistical variable, and we obtain a detailed description of how the measured computational resource was used during the time period that separated the two snapshots.

In our experiments, we measured the cost of processing a specific data transformation by executing the following set of steps: *(i)* we took a first snapshot of the set of statistical variables we wanted to control; *(ii)* we run the data transformation we wished to benchmark; *(iii)* we took a second snapshot that recorded the same statistical variables comprised in the first snapshot; *(iv)* we calculated the difference between the second and the first snapshots, and used the obtained values to understand the cost of the implementation mechanism that processed the

data transformation. In Appendix B.2, we present the code used to snapshot the database statistics of the different RDBMS. In Appendix B.3, we present the code used to calculate the corresponding snapshot difference.

RDBMS statistics are made accessible through a set of built-in system views. The level of detail at which these statistics can be gathered may vary from RDBMS to RDBMS. Some database systems gather statistics only at the lowest level of granularity (*database wide statistics*); others are capable of gathering statistics at higher levels of granularity (*session level statistics*). The first type of statistics describe the global usage of computational resources consumed in a RDBMS, while the later describe the computational resource consumption that is specific to an individual database connection.

Database wide statistics constitute the common level at which statistics can be retrieved in all the RDBMS. For this reason, we gathered database wide statistics to benchmark the performance of one-to-many data transformations. Since in our experiments the RDBMS were only exposed to our individual sessions (no one else had access to the databases), we could use the system wide statistics to interpret the resource consumption imposed by each test. However, in a multi-user environment, we would have to use session level statistics to perform this type of study.

In what follows, we present the statistics we collected for the purposes of benchmarking the different implementation methods capable of performing one-to-many data transformations, and understanding the obtained results.

Elapsed Time In Seconds (ET) represents the time the RDBMS takes to execute a query.

CPU Time In Seconds (CPUT) represents the time the RDBMS spends processing while a query was being executed.

Wait Time In Seconds (WT) corresponds to the total time that *worker threads* (the threads that handle the workload of a database) have to be blocked waiting, without processing, for an event to terminate. For example, the time a thread processing a query spends waiting for database pages to be retrieved from disk before it can process their contents. We only considered the wait events of each system that were related to IO wait time.

Total Physical Reads And Writes are the number of physical read and write operations, respectively, performed on every raw partition handled by the RDBMS. The raw partition architecture of each RDBMS can be seen in Appendix B.1. A *physical read* or *write*, corresponds to a disk I/O operation where an unspecified number of database pages are read from, or written to disk. Database systems normally support physical operations capable of reading and writing single and multiple database pages, used for different purposes. For example, performing a *table scan* (read an entire database table) is normally accomplished by means of multi-block read operations; while *index range scans* over non clustered indexes may be performed using single block reads. This statistic may be relevant when we try to understand the efficiency of the I/O access methods used when executing a query. For example, for a given a query, it is normally desirable to read only the necessary database pages using a minimal amount of read operations.

Total Physical Reads And Writes In Bytes correspond to the number of bytes read from and written to disk, respectively, on every raw partition handled by the RDBMS. These statistics were computed by multiplying the number of database pages read/written from/to disk by the database block size (8KB). Furthermore, these statistics should not be confused with the physical reads and writes statistic explained above. There is no direct correlation between a physical read or write operation, and the corresponding number of database blocks read or written by that operation. Therefore, database systems normally keep record both of the number of physical operations performed and their corresponding number of database blocks. They convey different information.

The following buffer manager statistics detail the physical memory space allocated to the buffer manager cache, and the efficiency with which it was used.

Buffer Manager Cache Size In Bytes represents the number of data pages that can be allocated to physical memory at a given moment in time.

Buffer Manager Cache Lookups measures the number of block requests received by the buffer manager. The page requested by a database operation may have to be retrieved from disk if it does not exist in the buffer.

Buffer Manager Cache Hits measures the number of block requests that did not incur into a disk I/O read operation.

Buffer Manager Cache Hit Ratio (CHR) measures the rate of cache hits by cache lookups, and is calculated as $CHR = CHits/CLookups$. Although this statistic can be inferred from the two statistics described before, for some RDBMS (e.g. MSQLS) it is only possible to retrieve the CHR, which is automatically calculated by the system.

Finally, we gathered more than just the aggregated values of the different I/O statistics explained above. We kept record of the physical reads, physical reads in bytes, physical writes, and physical writes in bytes for each individual raw partition handled by the RDBMS. This way, we could analyse in detail the I/O performed in the individual raw partitions dedicated to *system*, *temporary*, *log*, *source* and *target* data.

4.3 Introduction of the experiments

In Chapter 2, we conducted a study about the query mechanisms supported by Oracle, SQL Server and DB2 that may be used to implement one-to-many transformations. Afterwards, in Chapter 3, we detailed three one-to-many transformation scenarios to be tested in each RDBMS, and enumerated the implementation methods at the disposal of each RDBMS to meet this challenge. Table 4.2 summarizes both the extended Relational Algebra representation and the different implementation methods capable of supporting the three one-to-many data transformations.

This section introduces the generalities about the experiments undertaken. First, in Section 4.3.1, we present the *critical parameters* that have a significant impact on the performance of one-to-many data transformations (Carreira et al., 2007). Second, in Section 4.3.2, we discuss the purpose of the experiments undertaken and their

		Implementation Mechanisms					
		union	*unpivot*	*recursivequery*	*tablefunction*	*model*	*unnest*
$E2$	$\mu_{renameid,favf}$(PROFILE)	X	X		X	X	X
$E3$	$\mu_{renameid,unpacknames}$(PROFILE)			X	X	X	
$E4$	$\mu_{renameid,monthlyviews}$(PROFILE)			X	X	X	

Table 4.2: Summary of the algebra expressions for one-to-many data transformations and the implementation mechanisms supporting them.

organization within this text. Third, in Section 4.3.3, we describe the essential metrics that we used to present and compare the results of the different experiments. Finally, in Section 4.3.4, we report on the special case of the Oracle *model* implementation method, which manifested a very unstable behaviour during our experiments.

4.3.1 Critical parameters

The time a data transformation takes to execute, its CPU cost or the amount of IO incurred are statistics directly influenced by a set of critical parameters: *(i)* the number of input records, denoted by *ntups*, and their corresponding size in bytes; *(ii)* the *fanout* of the transformation; *(iii)* and the *selectivity* over the input relation (Carreira et al., 2007). Recall that the *fanout* of a one-to-many data transformation is the average number of output tuples produced by each input tuple. The *selectivity* of a data transformation is the probability that each input record has of satisfying the predicate clause in a given query (Silberschatz et al., 2001).

These parameters are important for the following reasons. By increasing the number of records (*ntups*) of the input relation, the data transformation has to process more data. So, the overall cost of the data transformation should also increase. Likewise, increasing the value of the *selectivity* means that more input records satisfy the predicate clauses of a query. Consequently, more input records have to be processed. Together, the *ntups* and *selectivity* parameters determine the work batch of the implementation method processing the data transformation. The *fanout* of a data transformation does not influence the amount of records that are handled by the algorithm. It bares influence on 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.

4.3.2 Experiments: their purpose and organization

The purpose of our experiments is twofold: *(i)* to evaluate the effects of *critical parameters* on the performance of the implementation methods capable of supporting one-to-many data transformations; *(ii)* and compare three commercial RDBMS on their performance of these data transformations. To accomplish this, each experiment undertaken targeted a single data transformation ($E2$, $E3$ or $E4$) and measured the effect of only one critical parameter on the performance of each implementation method supported by the RDBMS under evaluation.

We organized our experiments according to the following sets: *(i)* experiments concerning the effect of varying the *ntups* parameter, described in Section 4.5; *(ii)* experiments concerning the effect of varying the *fanout* parameter, described in Section 4.6; and *(iii)* experiments concerning the effect of varying the *selectivity* parameter, described in Section 4.7. For each of these groups, we performed three experiments, one for each one-to-many

data transformation $E2$, $E3$ and $E4$.

Recall that $E2$ is responsible for producing the FAVFRIEND table of the FACE5 target database schema. In this table, each user of the social network is associated to his favourite friends. $E3$ is responsible for producing the FAVARTIST table where each user is associated to his favourite artists. $E4$ is responsible for producing the VIEWHISTORY table where each user is associated to the several months passed since the user's registration, and to the number of viewers that visited the user profile in each month.

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. For example, for a given experiment in a given RDBMS, the *elapsed time* results presented for *table function* implementation method in each of the experiment tests, corresponds to the average *elapsed time* of the five runs undertaken by the *table function* to perform the said test. Finally, all the implementation methods capable of supporting the one-to-many data transformation targeted by the experiment were exposed to the four tests and five runs per test that comprised it (the experiment).

The three critical parameters *ntups*, *fanout* and *selectivity* are determined by characteristics of the data contained in the source relation. Therefore, before the execution of each test, the source relation PROFILE was bulk loaded with data automatically generated to rigorously satisfy the *critical parameters* configuration of the test.

4.3.3 Main metrics used

Several statistics were gathered throughout the tests (see Section 4.2). Out of these, we focus our attention on the time-related statistics, namely the *elapsed time*, the *cpu time* and the *wait time*, to interpret and compare the experiments.

The *elapsed time* is an essential statistic because it describes the total execution time of a query. A straightforward way of comparing two systems is measuring which of them can complete the same task using less time. It is also important to understand that there are two essential time events that influence the *elapsed time* (ET) for a query. One is the *wait time* (WT) which measures the time that threads had to spend waiting for computational resources to become available. Most frequently, *wait time* events are I/O related. The second is the *cpu time* (CPUT), which is the amount of time the CPU was allocated servicing the query.

In multi-threaded software systems, like RDBMS, there is no formula to represent the *elapsed time* as a function of the other two time-related statistics (*cpu time* and *wait time*): two threads can potentially be processing at the same time in a computer with more than one CPU; and likewise, two threads can potentially be stopped waiting for an event to finish. Nevertheless, for a single threaded program, it is true that $ET = WT + CPUT$. This of course is illustrative of the fact that both the *cpu time* and *wait time* statistics play a determinant role influencing the *elapsed time*. So, in order to best understand the *elapsed time* for a query, we should also consider the other two time-related events.

RDBMS may run a query execution plan using one of two parallelization policies: serial plans and parallel

plans/queries. Serial plans are one threaded. That is, the entire *query execution plan* is run in a single thread. For these plans the equation $ET = WT + CPUT$ holds. Either the thread running the query is processing, or it is waiting for a resource. In parallel queries, multiple threads are launched to support the *query execution plan*. For instance, one thread per operation in the query plan, which is called inter-parallelism. Sometimes even a single operator may be supported by several threads, which is called intra-parallelism. In these plans IO is performed asynchronously. That is, unless all the threads that support the query execution plan are waiting for resources, the threads that are in a runnable state may continue processing. Thus, a parallel *query execution plan* does not hold true to the equation $ET = WT + CPUT$.

In Oracle, the experiments were executed serially, which is the default query execution policy of the database system. This policy was chosen because the parallel query policy performed 25% slower, in average, than the serial. In MSQLS, the experiments were run using the default scheduling policy of the RDBMS. The database system normally used three threads to support a parallel execution of the query plan. Using serial scheduling or parallel scheduling produced identical results, the only implementation method that appeared to benefit from this policy was the *unpivot*, having in average a 15% smaller *elapsed time* when run in parallel. In the end, all the RDBMS run their experiments under their default scheduling policies, which seemed to work best.

The presentation of the experiments results is always preceded by an introduction where we describe: *(i)* the test configurations; *(ii)* our expectations for the three main time related statistics as well as the IO consumption. Whenever our expectations are not met, we additionally describe the results obtained for other auxiliary statistics gathered which may help understanding the obtained results.

4.3.4 The exceptional case of the model operator

In Table 4.2, we can see that the *model* method - supported by the Oracle RDBMS - can be used to perform the three one-to-many data transformations. However, the operator seems unreliable to process relations with a size beyond a certain threshold[1]. Reaching a threshold point, the operator sometimes completes the query, and other times lingers indefinitely. When the lingering behaviour occurs, the system appears to be locked in an infinite cycle of read I/O on the temporary raw partition. We have let the system linger for as long as one terabyte of read I/O on this disk partition.

To perform the experiments with this operator, we were forced to divide a single query expression that would process the entire data transformations, into multiple queries that process one million records from the source table. Thus, to process 125 million records, we had to generate a script composed of 125 model queries, each one processing distinct sets of 1 million records. In Code Listing 12, we illustrate how the *model* implementation method had to be adapted to process the bounded one-to-many data transformations with multiple queries, processing restricted quantities of data.

Code Listing 12. The following partial SQLPlus script illustrates how the *model* implementation query had to be adapted to process the bounded one-to-many data transformation $E2$, when the `PROFILE` table was configured

[1]For the data transformation $E2$ with a *fanout* of five: no more than 5 million records could be processed using the operator. For the data transformations $E3$ and $E4$, the threshold was reached at 25 million records.

for a *fanout* of five. The script is not complete because it only shows two similar queries. The first SQL code corresponding to the first query is delimited by lines 1 to 16. The second between the lines 17 to 30. The full script, however, was composed by five such queries. Notice from the lines 8 and 22, that each of these queries is limited to process one million records. Had it not been necessary to limit the number of records handed to the *model* method, and the implementation of the expression would be composed solely by the first of these two queries without the inclusion of the *where clause* at line eleven. The model code used for implementing this data transformation has already been explained in Section 2.1.1.5 Code Listing 4.

```
1: CREATE TABLE TB_TARGET
2: NOLOGGING TABLESPACE TARGET
3: AS
4: SELECT USRID, FAVF
5: FROM (
6:       SELECT ID AS USRID, FAVF
7:       FROM PROFILE P
8:       WHERE ID <= 1000000
9:       MODEL
10:              PARTITION BY (ID)
11:              DIMENSION BY (1 as FINDEX)
12:              MEASURES (FAVF1, FAVF2, FAVF3, FAVF4, FAVF5, 0 AS FAVF)
13:              RULES ( FAVF[1] = FAVF1[1], FAVF[2] = FAVF2[1], FAVF[3] = FAVF3[1],
14:                       FAVF[4] = FAVF4[1], FAVF[5] = FAVF5[1] )
15:     )FAVFRIEND
16: WHERE FAVF IS NOT NULL;

17: INSERT /*+ append */ INTO TB_TARGET NOLOGGING
18: SELECT USRID, FAVF
19: FROM (
20:     SELECT ID AS USRID, FAVF
21:     FROM PROFILE P
22:     WHERE ID > 1000000 AND ID <= 2000000
23:     MODEL
24:              PARTITION BY (ID)
25:              DIMENSION BY (1 as FINDEX)
26:              MEASURES (FAVF1, FAVF2, FAVF3, FAVF4, FAVF5, 0 AS FAVF)
27:              RULES ( FAVF[1] = FAVF1[1], FAVF[2] = FAVF2[1], FAVF[3] = FAVF3[1],
28:                       FAVF[4] = FAVF4[1],FAVF[5] = FAVF5[1] )
29:     )FAVFRIEND
30: WHERE FAVF IS NOT NULL;
```

4.4 Java application for query performance benchmarking

In order to implement our benchmarks, we developed the DatabaseTests application. This is a flexible lightweight Java application for benchmarking the performance of queries in RDBMS. In essence, it accepts a test configuration, prepares the RDBMS environment accordingly, executes the desired query a fixed number of times and, after each run, saves the statistics collected from the RDBMS into an output spreadsheet file. The overall view of the query benchmarking process supported by this application is illustrated in Figure 4.1, using the Business

Process Modeling Notation (BPMN). We can see that the application performs two essential steps: *(i)* the RDBMS environment initialization; and *(ii)* the query performance benchmarking.

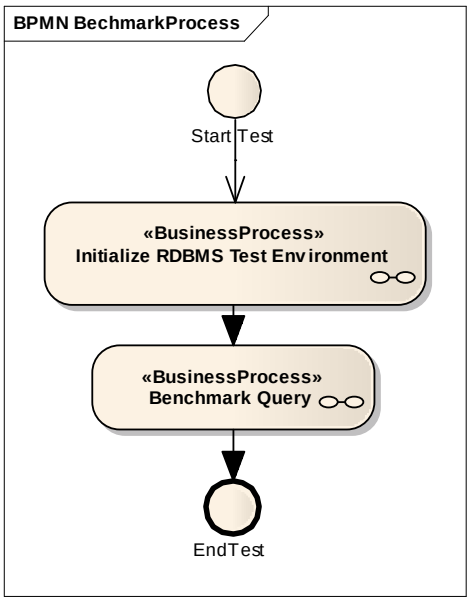


Fig. 4.1: Overall query benchmarking process supported by the DatabaseTests Java application.

The RDBMS environment initialization process is illustrated in Figure 4.2. The filled arrows show the sequence of activities undertaken, while the dotted lines show the essential data objects produced or consumed by each activity. Essentially, this process loads the database with test data, orders the RDBMS to compute statistics on the source table and calculates the query execution plan of the query that is to be benchmarked in the test.

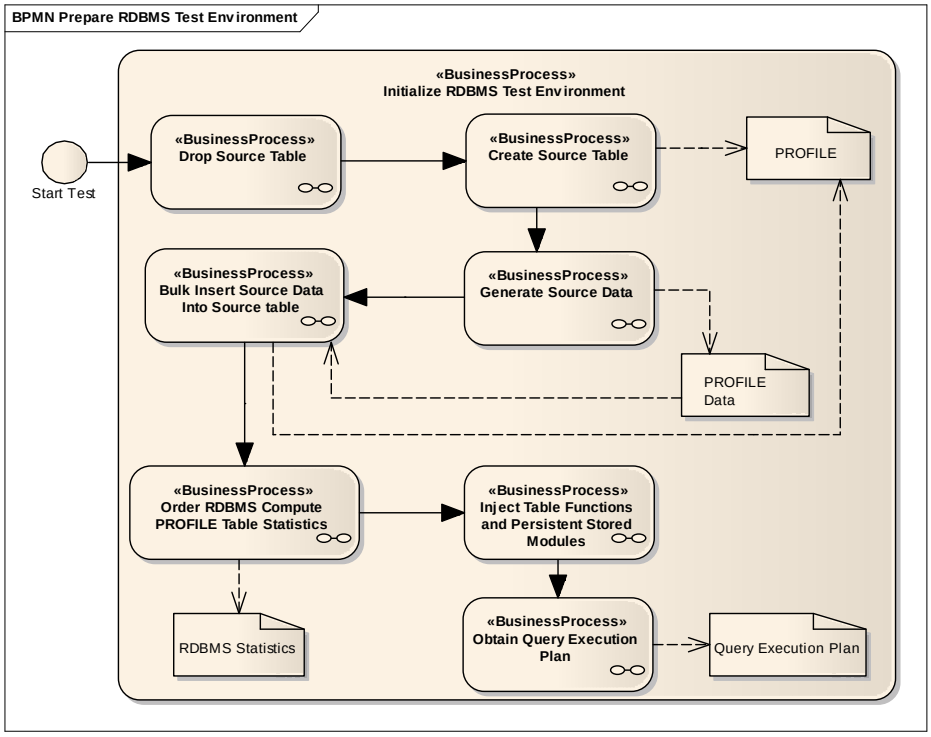


Fig. 4.2: Initialize RDBMS test environment process supported by the DatabaseTests Java application.

The query performance benchmarking process is illustrated in Figure 4.3. Essentially, this process ensures that the database cache is clean and no output data exists before executing the query to be benchmarked (e.g: by dropping output tables and restarting the database instance). Afterwards, the query is executed with RDBMS statistics snapshots being taken before and after query execution; finally, the snapshot difference is computed and the query benchmark statistics are gathered and saved into an output spread sheet. This process is repeated a specific number of types, depending on the number-of-runs configuration of the test.

Out of all the activities seen in the main processes described here, only the gathering of the benchmark statistics, and their subsequent saving into an output spreadsheet file, were implemented in Java. It was necessary to use JDBC connections to retrieve the snapshot difference from the RDBMS into Java, and subsequently, using the Apache-POI API, save them into an output excel file. All the other activities were implemented using bash script files. For instance, we created bash scripts that generated Dynamic SQL scripts, which in turn were processed by the RDBMS command line clients (e.g: it is impractical use static SQL scripts, when the number of columns in the SQL statements varies with the *fanout*).

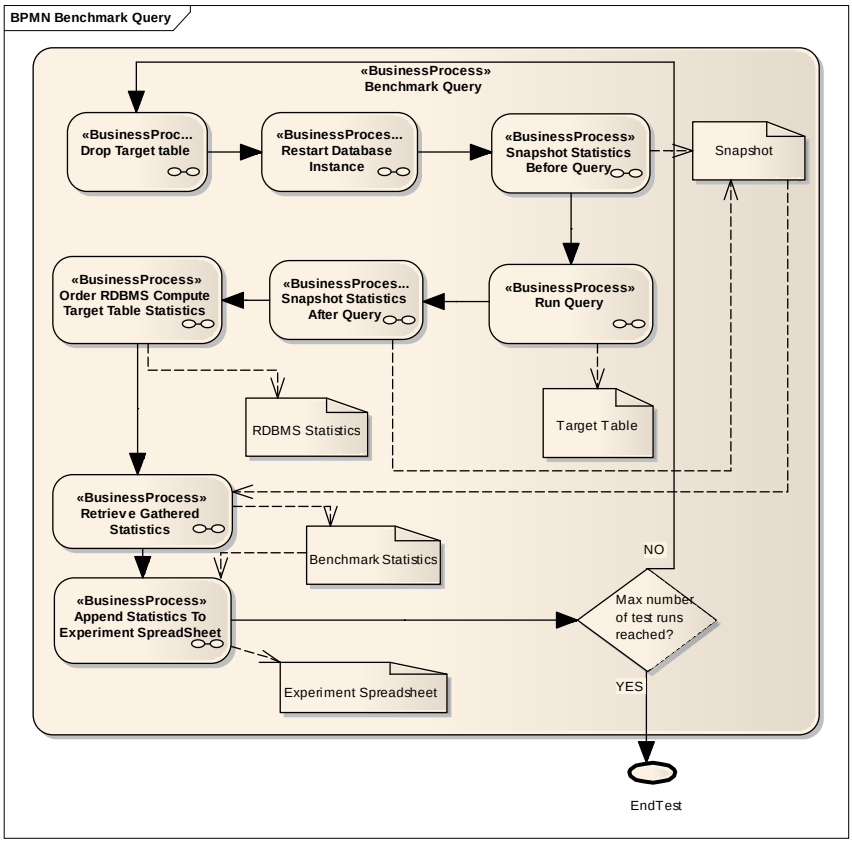


Fig. 4.3: Benchmark query process supported by the DatabaseTests Java application.

The DatabaseTests application was designed with three main concerns:

RDBMS independence. This application was used to test the three RDBMS, and only minor modifications had to be made to the software code to include the support of each new RDBMS. These modifications were essentially of one type: realizing three abstract classes, which were nothing more than document libraries specifying the file system locations of shell script files (in our case, we used BASH scripts supported by the

MSYS implementation of the BASH shell for Windows).

Minimal Java code. By minimizing the amount of Java code, we reduce bugs and simplify the future refinement of this application by other users. The focus of the application was the support of a Workflow of activities, which should, whenever possible, be implemented externally to Java (e.g: by means of shell scripts and other external applications). Evidently, the intended purpose of this Workflow of activities was the execution of a query benchmarking process. Therefore, we can summarize our Java application as an orchestrator with very small concerns as to how the Workflow activities were performed.

Exploiting robust applications. By delegating the implementation of the Workflow activities to external processes, we were, in most cases, taking full advantage of highly specialised and robust tools to perform the necessary tasks (e.g: communicating with the RDBMS by means of their own command line clients for the execution of various SQL statements; as opposed to implementing these tasks with potentially incorrect code using the RDBMS JDBC driver).

Finally, the experiments that were undertaken to evaluate the performance of the different implementation methods in the three RDBMS were implemented as JUnit test suites. Each test class called upon by a test suite specified an experiment for a specific implementation method and RDBMS, see Section 4.5 4.6 and 4.7. These test classes exercised the DatabaseTests API, passing it the desired tests configuration (e.g: RDBMS, type of experiment, *ntups*, *fanout*, *selectivity* and implementation method).

4.5 Experiments varying the number of records

This section evaluates the effect of increasing the source relation size, denoted as *ntups*, when the *fanout* and *selectivity* parameters remain constant (set to 5 and 1 respectively), on the performance of the data transformations $E2$, $E3$ and $E4$. In the four tests that comprised each experiment, the parameter *ntups* increased exponentially by five, starting with a value of 1 million records (in test 1) up to 125 million records (in test 4).

We expected that the absolute values of the *cpu time* (CPUT) and *wait time* (WT) would increase linearly with the size of the input relation. This is because, by increasing only the number of records of the input relation, we would only be influencing the dimension of the workload handled by the implementation methods. Thus, we expected that the statistics followed a linear behaviour ($k_1 + m_1 ntups$), where k_1 and m_1 are constants that depend on the RDBMS and implementation mechanism. The constant k_1 represents the fixed costs of processing a data transformation that are not affected by the increase in size of the input relation (e.g. determining a query plan, allocating fixed resources like buffers and variables for the different operations in the query execution pan, etc), and m_1 is the function slope in seconds per input record. Concerning the *elapsed time*, in a serial plan we expect that it is also affected linearly by *ntups*. In a parallel query this can happen or not.

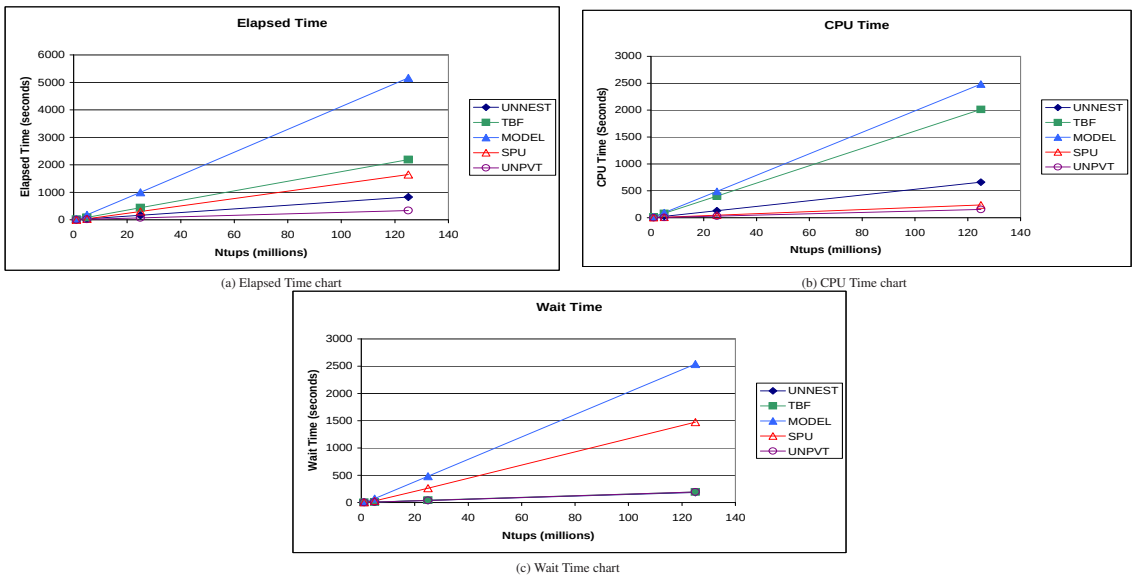
The following sections present the results obtained when varying the *ntups* value for the three one-to-many data transformations.

4.5.1 Experiment 1: execution of $E2$ varying the number of records

In this section, we report and analyse the results obtained when varying the number of records during the execution of the data transformation $E2$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation. We recall that the data transformation $E2$ produces the `FAVFRIEND` output table, which is written to the target raw partition, from processing the records of the input table `PROFILE` , which is read from the source raw partition. In this data transformation, each output record corresponds to a pairing of a user with one of his favourite friends.

4.5.1.1 Oracle results

In Oracle, the $E2$ expression can be implemented using the *SPU*, *unpivot*, *table function*, *unnest* and *model* implementation methods. The results of this experiment are shown in Figure 4.4. In this experiment, the *unpivot* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.4: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E2$ in the varying number of records experiment.

The results were according to our expectations. The *elapsed time*, *cpu time* and *wait time* charts (Figure 4.4 a, b and c respectively) show that all the implementation methods were affected linearly by the increase of the number of input records. Recall that in Oracle we used serial queries.

In terms of *cpu time* (Figure 4.4 b), we can see that the *unpivot*, *SPU* and *unnest* were the best performing implementation methods, with the *unpivot* and *SPU* having very similar costs. The difference in their *elapsed time* performance is explained by their considerable difference in their *wait time* performance. The *SPU* did not have an efficient IO behaviour. Finally, the *table function* and *model* methods had very costly algorithms, they had the worst *cpu time* performance.

The *unpivot*, *table function* and *unnest* implementation methods exhibited the same IO behavior. The three implementation methods read the source table only once, and also performed the same IO to write the output of

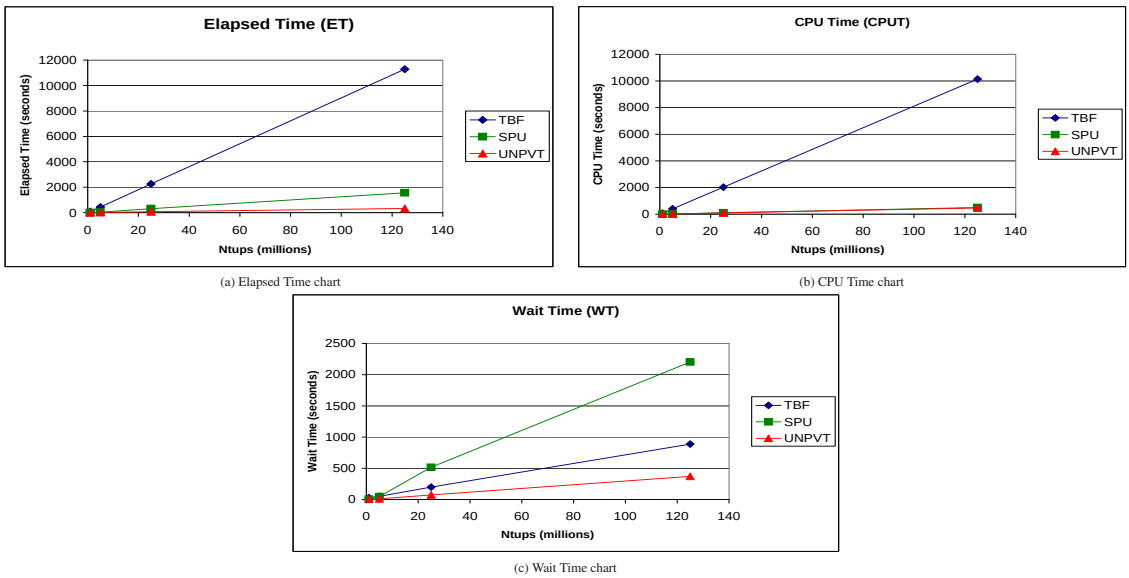
the data transformation on the target table.

The *selection projection and union* (*SPU*) implementation method is a special case. Although this operator is very efficient in terms of *cpu time*, the operator still comes out as one of the worst methods that could possibly be chosen to solve this data transformation. The *SPU* method has a very large *wait time* - larger than that of the *table function*, *unnest* and *unpivot* methods - which significantly handicaps its overall performance. This unique behaviour can be explained by the fact that the query plan of the *SPU* implementation method performs 5 table scans (the same as the *fanout*). The other methods only require to scan the table once. As a result, when the source table can fit in the database buffer cache, which happens only for the first test with 1M records, it is only read once. As the table gets larger than the database cache, less and less read operations find the requested block on cache. Finally, the buffer cache is so small when compared to the relation size that in practice the table is scanned entirely from disk for each *table scan*.

The *model* implementation method, had the highest *wait time*. This is explained by the fact that this method used the temporary raw partition to perform its computations. This characteristic is responsible for making the *model* implementation method considerably expensive when dealing with large volumes of data. In the last test, this method performed close to 66 GB of IO on the temporary raw partition.

4.5.1.2 SQL Server results

In MSQLS, the $E2$ expression can be implemented using the *SPU*, *unpivot*, and *table function* implementation methods. The results of this experiment are shown in Figure 4.5. In this experiment, the *unpivot* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.5: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E2$ in the varying number of records experiment.

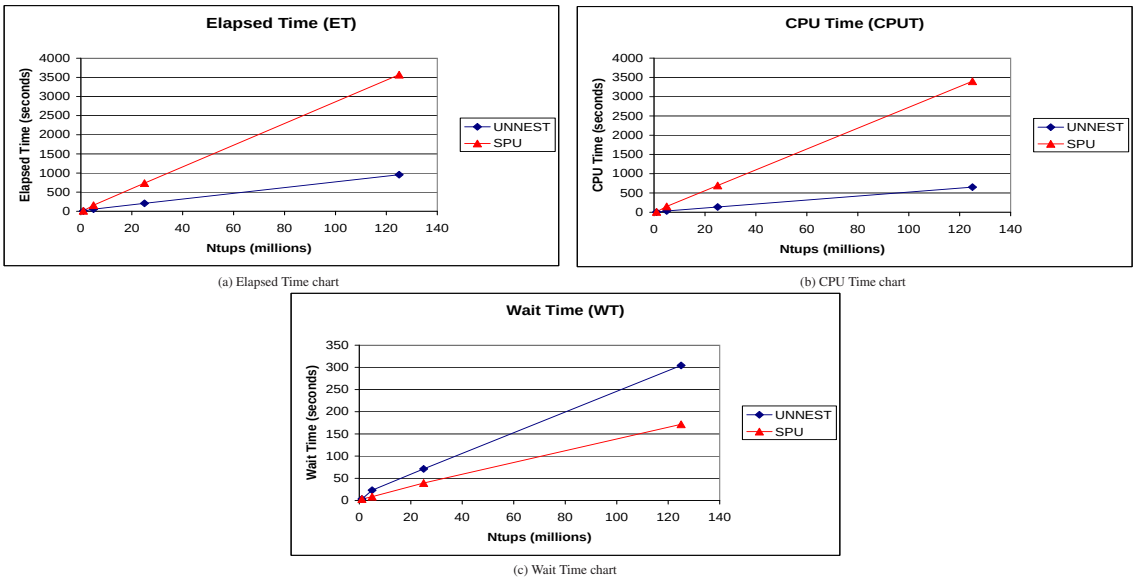
The results were according to our expectations, and, for the most part, the explanations given for this same experiment in Oracle, see Section 4.5.1.1, also apply to interpretation of the results manifested by the three MSQLS

implementation methods. Their differences are explained bellow. As expected, since the query plan was parallel, the *elapsed time* is lower than the addition of the *cpu* and *wait time*.

The *table function* in MSQLS, presented high *cpu time* cost, just like what we have seen in Oracle. However, in MSQLS, *table functions* implemented in TSQL are not pipelined. They materialize their output results in the temporary raw partition. Moreover, these write operations are fully logged. Thus, the *table function* has much higher IO in MSQLS than in Oracle.

4.5.1.3 DB2 results

In DB2, the $E2$ expression can be implemented using the *SPU*, and *unnest* implementation methods. The results of this experiment are shown in Figure 4.6. In this experiment, the *unnest* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.6: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E2$ in the varying number of records experiment.

The results were according to our expectations, and identical to those of Oracle and MSQLS in this same experiment, Section 4.5.1.1 and Section 4.5.1.2. The only significant differences were that in DB2 the *SPU* had considerably higher *cpu time* cost than Oracle and MSQLS, while its *unnest* manifested results identical the *unpivot* solutions of Oracle and MSQLS.

Regarding the *wait time* statistics collected, we noticed an interesting fact. DB2 generally presents negligible read *wait time*, unlike Oracle and MSQLS. Throughout the various experiments, the read *wait time* was approximately 3 seconds. This value was virtually unaffected by increases in the number of bytes read by the implementation methods in the various experiments, and is most likely associated with overhead costs of taking the statistics snapshots and commencing to read the source relation. This means that DB2 was capable of using its read-ahead mechanism (called pre-fetch), with maximum efficiency. The varying number of records experiment with 125M input records was one of the few experiments where methods presented read IO *wait time* higher than 3 seconds. Generally, the *wait time* values measured were related with delays in write operations. It is likely that the

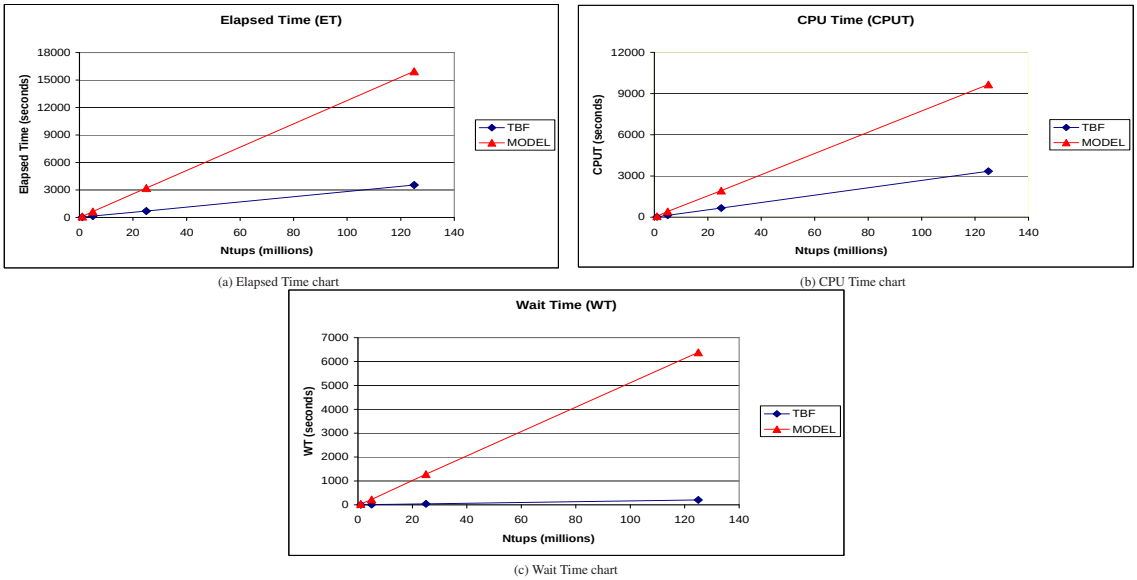
demand for buffer pages imposed by the read-ahead mechanism, in conjunction with that of the write operations, lead the buffer to eventually become full with non free pages. In such a case, asynchronous write IO is no longer possible. The fact that DB2 wait events tend to be related with free pages starvation in write operations and not in read operations is not surprising. After all, the queries produce one output page at a time, while the asynchronous read operations pre-fetch in anticipation 128 pages at a time.

4.5.2 Experiment 2: Execution of $E3$ when varying the number of records

In this section, we report and analyse the results obtained when varying the number of records during the execution of the data transformation $E3$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation. We recall that the data transformation $E3$ produces the `FAVARTIST` output table, which is written to the target raw partition, when processing the records of the input table `PROFILE` , which is read from the source raw partition. In this data transformation, each output record corresponds to a pair constituted by a user and one of his favourite artists.

4.5.2.1 Oracle results

In Oracle, the $E3$ expression can be implemented using the *table function* and the *model* implementation methods. The results of this experiment are shown in Figure 4.7. In this experiment, the *table function* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

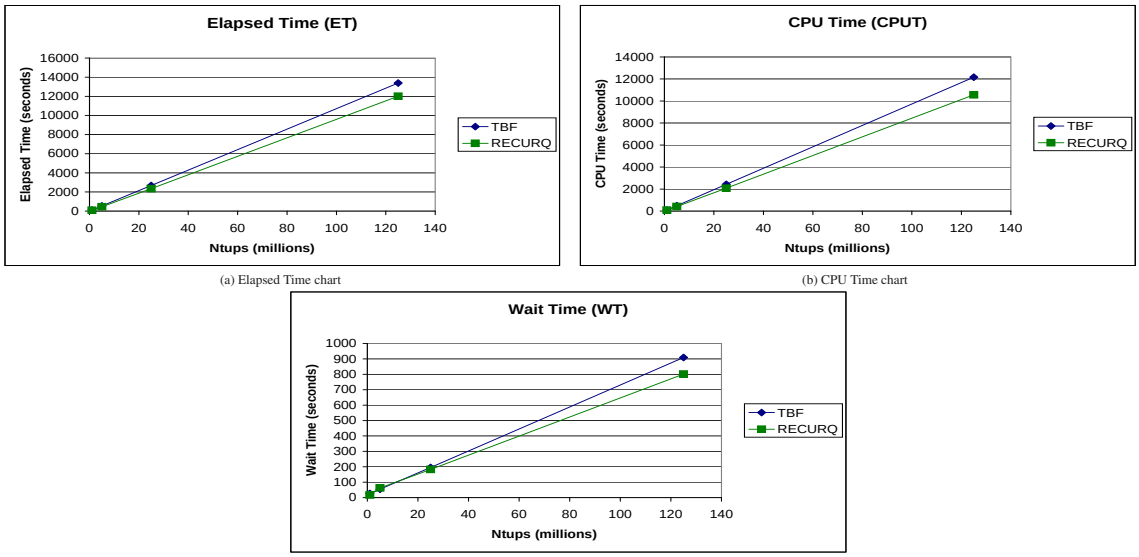
(c) Wait Time chart

Fig. 4.7: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E3$ in the varying number of records experiment.

The two implementation methods evidenced results that matched our expectations. Both the *table function* and *model* methods linearly increased their *elapsed time*, *cpu time* and *wait time* when the number of records increased.

4.5.2.2 SQL Server results

In MSQLS, the $E3$ expression can be implemented using the *table function* and the *recursive query* implementation methods. The results of this experiment are shown in Figure 4.8. In this experiment, the *recursive query* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.8: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E3$ in the varying number of records experiment.

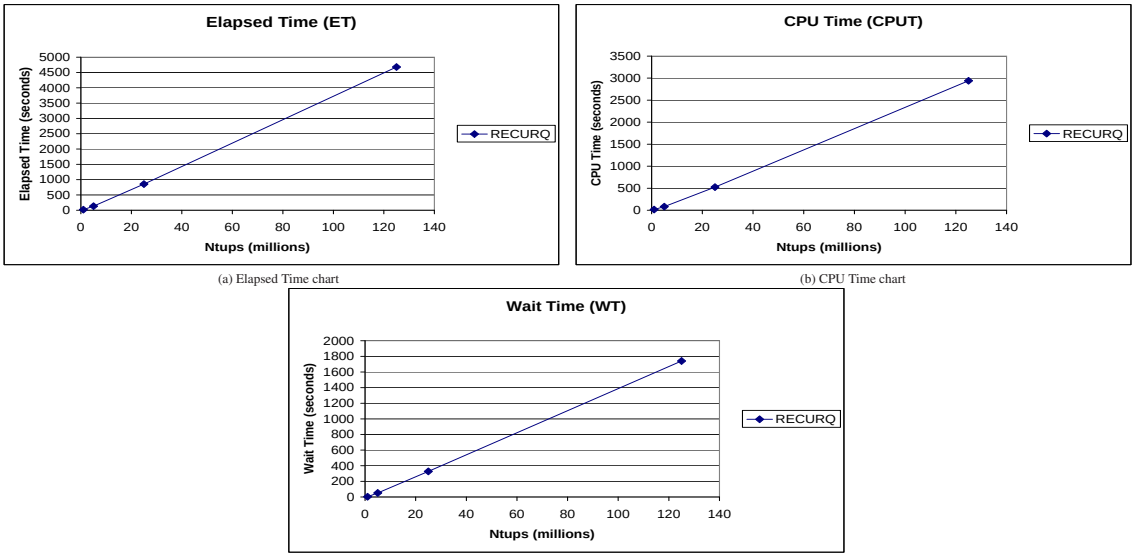
The two implementation methods evidenced results that matched our expectations: both the *table function* and *recursive query* methods linearly increased their *cpu time* and *wait time* when the number of records increased. The *table function* behaviour in this experiment was identical to its behaviour in the varying number of input records experiment over the expression $E2$, which was explained in Section 4.5.1.2.

In MSQLS, the *recursive query* processes a data transformation in several iterations. The output of each iteration has two destinations. One, the records are projected into the target output table (only the relevant columns are chosen). Two, the records are inserted into a temporary spool file. A *spool file* is a data file that is created dynamically during the execution of a query, and is normally used as the output stream of an operator to be later used as the input stream of another. When the spool file becomes too large to fit into physical memory, it is written to disk in the temporary raw partition. None of this write IO is logged. Finally, the contents of the spool file that were added in the last iteration, are then read and processed in the following. This process only ends when an iteration terminates without adding any contents to the spool file.

The *recursive query* not only presents high *cpu time* cost, but it may also perform high IO, equally divided in reads and writes, in the temporary raw partition. However, unlike the *table function*, this implementation method performed the data transformation with minimal logging.

4.5.2.3 DB2 results

In DB2, the $E3$ expression can only be implemented using the *recursive query* implementation method. The results of this experiment are shown in Figure 4.9.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.9: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E3$ in the varying number of records experiment.

The *recursive query* evidenced results that matched our expectations: the *elapsed time*, *cpu time* and *wait time* increased linearly with the number of records. It should be said that the DB2 *recursive queries* are associated to query execution plans identical to those of MSQLS, see Section 4.5.2.2.

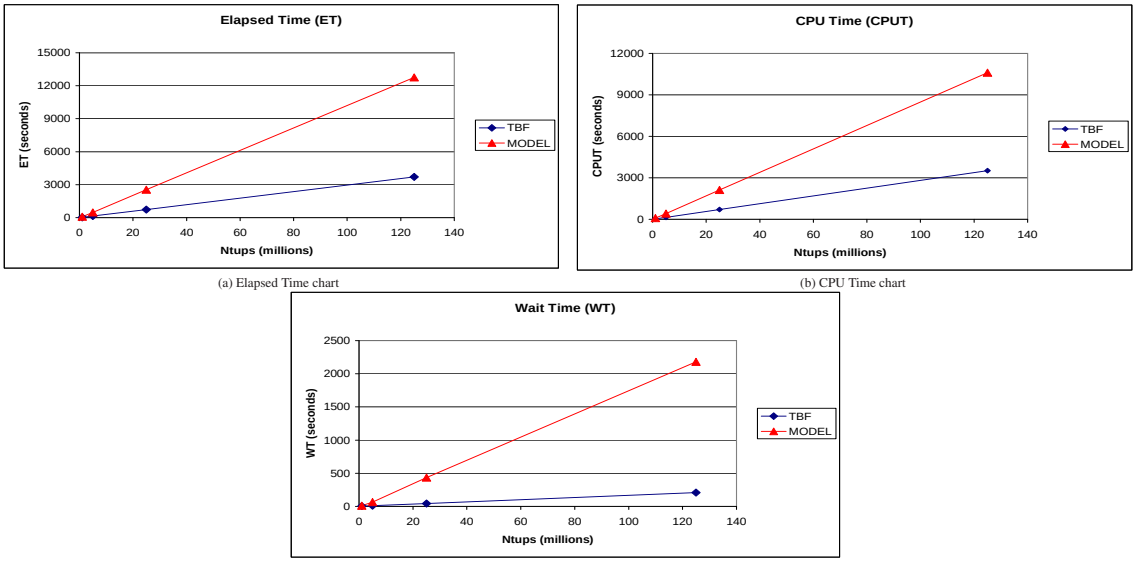
4.5.3 Experiment 3: Execution of $E4$ when varying the number of records

In this section, we report and analyse the results obtained when varying the number of records during the execution of the data transformation $E4$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation. We recall that the data transformation $E4$ produces the `VIEWHISTORY` output table, which is written to the target raw partition, when processing the records of the input table `PROFILE` , which is read from the source raw partition. In this data transformation, each output record corresponds to a pairing of a user with the number of visits to user's profile in a valid year and month time period.

4.5.3.1 Oracle results

In Oracle, the $E4$ expression can be implemented using the *table function* and the *model* implementation methods. The results of this experiment are shown in Figure 4.10. In this experiment, the *table function* was the best implementation method, having the smallest *elapsed time*.

The two implementation methods evidenced results that matched our expectations. Both the *table function* and *model* methods linearly increased their *elapsed time*, *cpu time* and *wait time* when number of records increa-



(a) Elapsed Time chart

(b) CPU Time chart

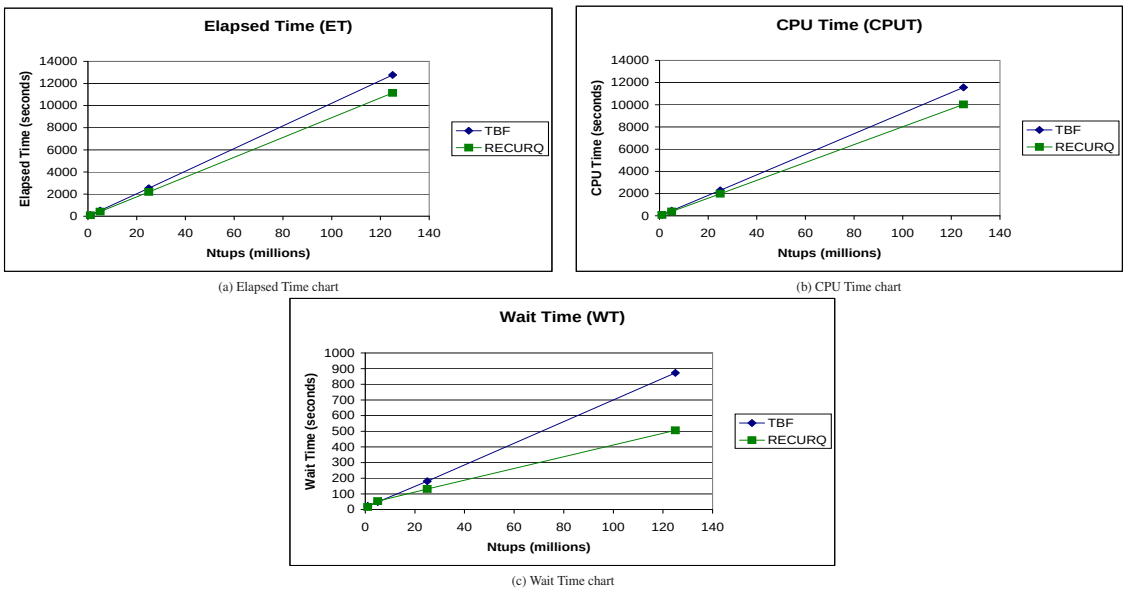
(c) Wait Time chart

Fig. 4.10: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E4$ in the varying number of records experiment.

sed. Their behaviour in this experiment was identical to their behaviour in the varying number of input records experiment over the expression $E2$, which was explained in Section 4.5.1.1.

4.5.3.2 SQL Server results

In MSQLS, the $E4$ expression can be implemented using the *table function* and the *recursive query* implementation methods. The results of this experiment are shown in Figure 4.11. The *recursive query* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

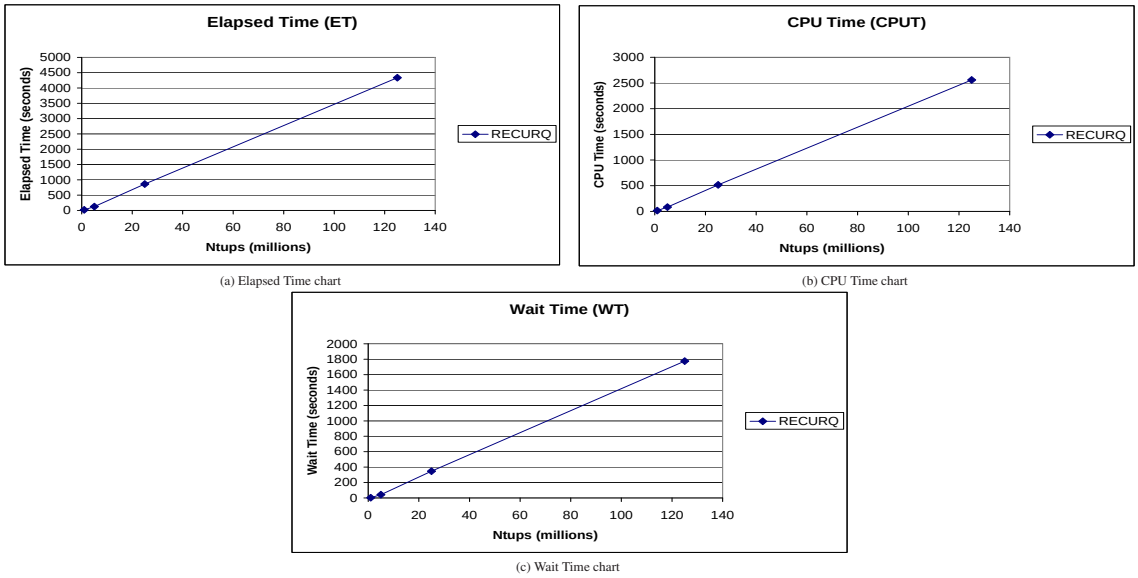
Fig. 4.11: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E4$ in the varying number of records experiment.

The two implementation methods evidenced results that matched our expectations: both the *table function* and *recursive query* methods linearly increased their *elapsed time*, *cpu time* and *wait time* when number of re-

cords increased. The *table function* and *recursive query* behaviour shown in this experiment was identical to the behaviour observed in the previous experiments.

4.5.3.3 DB2 results

In DB2, the $E4$ expression can only be implemented using the *recursive query* implementation method. The results of this experiment are shown in Figure 4.12.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.12: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E4$ in the varying number of records experiment.

The *recursive query* evidenced results that matched our expectations: the method linearly increased its *elapsed time*, *cpu time* and *wait time* when number of records increased.

4.6 Experiments varying the fanout

This section evaluates the effect of increasing the *fanout*, when the *ntups* and *selectivity* parameters remain constant (set to 1 Million and 1 respectively), on the performance of the various implementation methods supported by each RDBMS capable of implementing the one-to-many data transformations $E2$, $E3$ and $E4$. In the four tests that comprised each experiment, the parameter *fanout* increased exponentially by three, starting with a value of 1 (in test 1) up to 27 (in test 4).

For all the experiments presented in this section, the input relation remained the same. The source relation was populated only once, with data configured for a *fanout* of 27. By doing this, we ensured that the amount of data retrieved to process each input record had no influence in the variation of the test results, since the number of bytes read from the source relation remained constant during the whole set of experiments (not being affected by the change in *fanout* imposed on the tests). However, since the source relation was configured for a *fanout* of 27, we had to modify the code of each data transformation implementation to take into account the desired *fanout* value of the test. For instance, the processing cycle of the *table functions* was extended with an additional condition that

ensured the interruption of the processing once the number of output records generated by it matched the desired *fanout*.

Our predictions for these experiments were the following:

1. The value of *physical read in bytes* in the source raw partition should remain constant, since the source relation size also remained constant throughout the tests.

2. The value of *physical write in bytes* in the target raw partition should increase by three from test-to-test, proportionally to the *fanout* increase. This should occur because, as a result of the change in *fanout*, three times more output records are written from test to test. Nevertheless, the increase of this statistic should have minimum influence in the measured *wait time* because RDBMS perform asynchronous output, flushing automatically the dirty pages to disk while the input is being processed. Furthermore, since the disk onto which these output data pages are flushed is separate from the one containing the source records, it is unlikely that the worker threads processing our query become constricted by the output IO.

3. The value of *cpu time*, on the other hand, should increase with the *fanout*. All the operations comprised in a query plan contribute to the overall *cpu time*. However, some of them are not affected by the variation of the *fanout* (e.g: table scanning the `PROFILE` relation); and others should vary linearly with the fanout (e.g applying a table function to each input record or writing the output data). Thus, for these experiments, we can say that the cost of processing each tuple should be modelled by the formula $CPURecord = k_1 + m_1 * fanout$, where k_1 and m_1 are constants that depend on the implementation method and data transformation. The constant k_1 denotes the *cpu time* cost associated to any activity not influenced by the *fanout* when processing each individual record (e.g the initialization of variables before a processing cycle in a *table function* implementation), and m_1 denotes the *cpu time* spent for producing each output tuple.

4. The value of *wait time* associated to the IO wait events should decrease as a consequence of the higher *cpu time* spent per input page with the increase in *fanout*. RDBMS perform read operations asynchronously while they process the input data pages available in the cache. The longer a data page takes to be processed, the less time we expect to wait for subsequent data pages to be retrieved. This process is called *read-ahead*, and is generally used for data access methods like table scans and index range scans. Different RDBMS use different policies for reading-ahead. In Oracle, once a cache miss occurs, the RDBMS performs multi-block read operations to retrieve the data requested by a table scan (e.g. read 60 consecutive data pages). Once the first data page is made available on the buffer, the query can start processing it without waiting for the remaining 59. Consequently, if the *cpu time* spent per input page is sufficiently high, the query should only wait for the first input page of each multi-block read. The minimum *wait time* should be $inputPages/mblockReadCount * wtPage$, where $inputPages$ denotes the number of data pages of the input table, $mblockReadCount$ denotes the number of pages fetched per multi-block read operation, and $wtPage$ is the average time spend reading one random data page. In MSQLS, data pages are read in anticipation even before a cache miss occurs. In this case, the minimum *wait time* should be 0, if we ignore possible IO overhead costs. Since the number of input data pages is constant, the *wait time* should also remain constant with the increase in *fanout* once the minimal *wait time* is reached. However, the increase in *fanout* also leads

to increased write IO. Thus, it is possible that once an unknown threshold of read and write IO is reached, the issued IO operations start to bottleneck and the *wait time* performance degrade.

5. Finally, the value of *elapsed time* is expected to behave in the same manner as explained in our expectations for the varying number of records experiments (see Section 4.5). If the queries are processed serially, and both the *cpu time* and *wait time* change linearly with the *fanout*, then the *elapsed time* should also behave linearly. If the queries are processed in parallel, it is not possible to guarantee the linearity of the *elapsed time* even in the presence of linear *cpu time* and *wait time* behaviour.

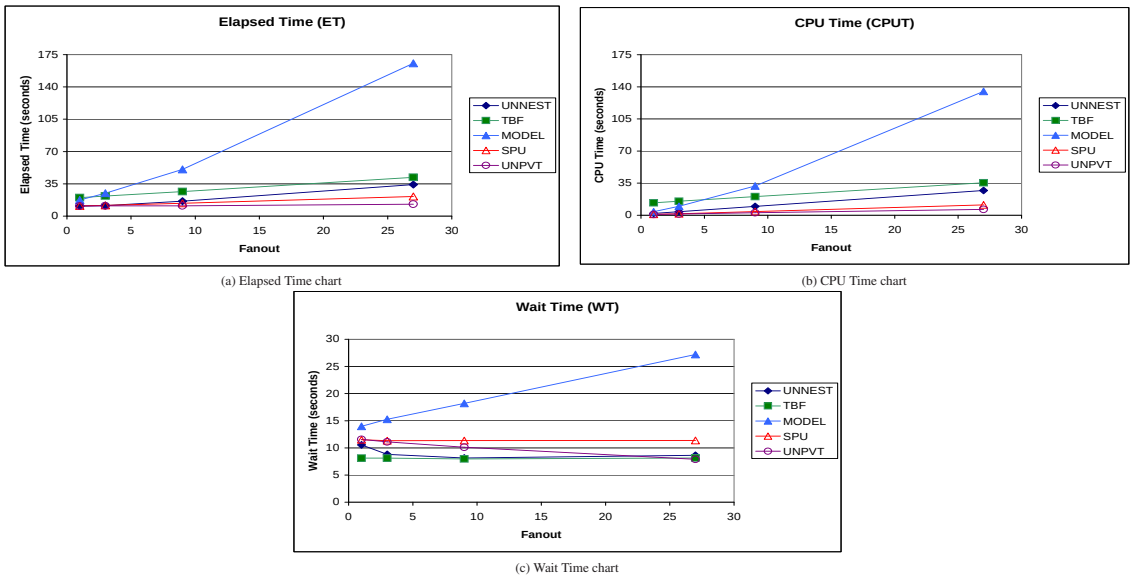
The following sections present the results obtained when varying the *fanout* value for the three one-to-many data transformations.

4.6.1 Experiment 4: Execution of $E2$ with varying fanout

In this section, we report and analyse the results obtained when varying the *fanout* during the execution of the data transformation $E2$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation.

4.6.1.1 Oracle results

In Oracle, the $E2$ expression can be implemented using the *SPU*, *unpivot*, *table function*, *unnest* and *model* implementation methods. The results of this experiment are shown in Figure 4.13. The *unpivot* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.13: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E2$ in the varying *fanout* experiment.

The methods *unpivot*, *spu*, *table function* and *unnest* evidenced results that match our expectations. The four methods presented a linear increase of *cpu time* and a relatively constant *wait time*, (Figure 4.13 b and c,

respectively). Consequently, *elapsed time* of these methods increased linearly with the *fanout*. The *model* method was exceptional, in the sense that neither the *wait time* remained constant nor the *cpu time* increased linearly.

All implementation methods had different *cpu time* costs. The *unpivot* and *SPU* were the most efficient, both with very low and similar *cpu time* costs. The *unnest* method started with a low *cpu time* cost, close to that of the previous two. But the increase in processing time was high with the change in the *fanout*. We can see in the *cpu time* chart that the slope of this method is steeper than that of the *table function*. For a *fanout* value of 27, the *unnest* presented a *cpu time* cost similar to the *table function*, and beyond this point its cost would most likely surpass it. The *model* method had a *cpu time* cost much higher than the other methods.

The *spu* and *table function* were the only methods that did not present any visible changes in *wait time*, appearing practically constant. While the *unpivot* and *unnest* started with a *wait time* identical to the *spu* and rapidly converged it to that of the *table function*. These results are according to our expectations. The *table function wait time* was constant because its *cpu time* was already high for a *fanout* of 1 and the time to process an input page was high enough to asynchronously load the next one. The *unpivot* and *unnest* having very low initial *cpu time* costs processed each input page faster than they could be retrieved. Thus, by increasing the *fanout*, the two methods converged their *wait time* to that of the *table function*. It can also be seen that the *unnest* method converged its *wait time* to that of the *table function* much faster than the *unpivot*. This is because the *unnest* also increased its *cpu time* much faster. Finally, the *SPU* maintained its *wait time* constant despite having a very low initial *cpu time* cost. This happened because the *cpu time* increase did not affect in any way the rate at which the *SPU* method requested data pages from the cache. Recall that the *SPU* performs a data transformations by means of multiple table scans. The algorithm is computed fully at the table scan level. It is the first table scan that loads all the data pages into the cache, and this table scan is performed at exactly the same rate whether for a *fanout* of 1 or 27.

The *model* method, on the other hand, contrary to our expectations, increased its *elapsed time* time in a non linear manner. The chart seems to hint towards an exponential increase of this statistical variable, justified by a likewise tendency of the *cpu time* to increase exponentially with the change in *fanout*. Furthermore, the very significant gap in performance of this operator, in comparison to other implementation methods, is not only amplified by the non linear increase in *cpu time*, but also by the very significant IO cost of reading and writing operations undertaken over the temporary raw partition. In this experiment, the amount of IO performed on the temporary raw partition increased linearly with the *fanout*, thus explaining the unexpected linear increase in *wait time* show in Figure 4.13 (c) for this operator. The remaining Oracle operators did not access this raw partition, and manifest identical IO statistics over the source and target raw partitions.

We have to clarify that in the following Oracle varying *fanout* experiments $E3$ and $E4$, discussed in Sections 4.6.2.1 4.6.3.1 respectively, we are shown clear evidence that the change in *fanout* is not directly responsible both for the linear increase in *wait time* and for the non linear increase in *cpu time* that we have shown for this experiment. In these experiments, the *model method* showed results that matched our expectations. In truth, the *fanout* was only indirectly responsible for causing the unexpected results of this experiment. Each time the *fanout* increased, the query code for the data transformation $E2$ of this experiment changed. There was a linear increase in the number of *measures* and *rules* used in the *model clause* of the said query. This is illustrated in Figure 4.14.

The *measures* of a *model clause* determine which data fields from the input records have data values needed for processing the query. Thus, the linear increase of IO performed on the temporary raw partition is explained by the increment on data fields (`FAVFI`) required for processing the query. On the other hand, the *rules* of a *model clause* determine the processing that the *model* operator is commanded to do. It appears that the linear increase of *rules* led to a non linear increase on the *cpu cost* of the query. Section 2.1.1.5 explains in detail the various components of a *model clause*.

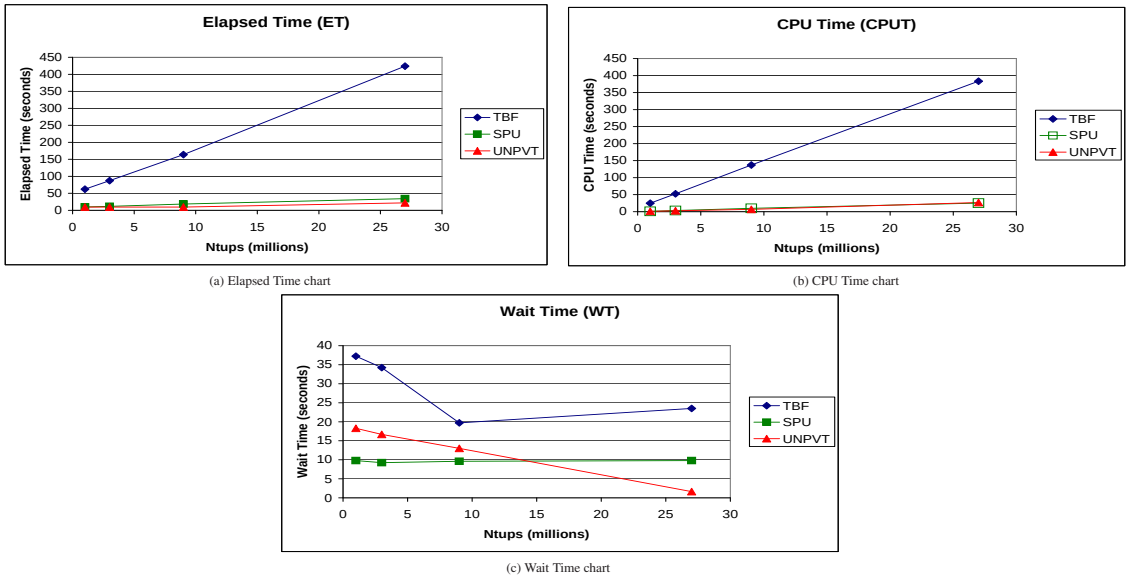
$fanout = 1$	$fanout = 3$

`MODEL`
`  PARTITION BY (ID)`
`  DIMENSION BY (1 as FINDEX)`
`  MEASURES (FAVF1, 0 AS FAVF)`
`  RULES (`
`  FAVF[1] = FAVF1[1]`
`  )`
	`MODEL`
	`  PARTITION BY (ID)`
	`  DIMENSION BY (1 as FINDEX)`
	`  MEASURES (FAVF1, FAVF2, FAVF3, 0 AS FAVF)`
	`  RULES (`
	`  FAVF[1] = FAVF1[1], FAVF[2] = FAVF2[1], FAVF[3] = FAVF3[1]`
	`  )`

Fig. 4.14: One-to-many data transformation $E2$ changes its *model clause* code with the *fanout* increase.

4.6.1.2 SQL Server results

In MSQLS, the $E2$ expression can be implemented using the *SPU*, *unpivot*, and *table function* implementation methods. The results of this experiment are shown in Figure 4.15. The *unpivot* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.15: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E2$ in the varying *fanout* experiment.

The *elapsed time* and *cpu time* charts (Figure 4.15 (a) and (b)) show that the *SPU* and *table function* implementation methods were affected linearly by the increase of *fanout* showing the same *cpu time*. However, the *unpivot* maintained an approximately constant *elapsed time*, and only increased it in the last test. From the *wait time* chart (Figure 4.15 c) we see different behaviours. Overall, the *unpivot* and *SPU* were affected linearly by the change in *fanout*, while the *table function* suffered a *wait time* increase in the last test. We try to explain these occurrences bellow.

The *unpivot* decreased its *wait time* linearly with the change in *fanout*, as expected. We can see that the *unpivot* practically had 0 IO wait time in the last test. The increase in processing time dispensed per input data page, permitted the read-ahead mechanism to have more time to fetch data pages in anticipation.

The *elapsed time* behaviour of the *unpivot* was not linear with the *fanout*. In the first three tests it remained approximately constant, while in the final test it increased. The fact that the increase in *cpu time* in the first three tests did no lead to an increase in *elapsed time* is probably explained by the decrease in *wait time*. That is, the time that was being gained processing the query was being lost waiting for IO. However, in the final test the *elapsed time* showed a significant increase in value since the *cpu time* increased much more than the *wait time* decreased.

The *SPU* maintained its *wait time* constant with the change in *fanout* for the same reasons as in the Oracle experiment (see Section 4.6.1.1).

The *table function* did not present a regular *wait time* behavior. Initially, it behaved like the *unpivot* method, decreasing its *wait time* with the change in *fanout*. However, in the fourth test (*fanout* 27) its *wait time* increased. This should be explained by the fact that the IO increased linearly with the *fanout* in the target and log raw partitions, and that from the third test and onward the *table function* output had to be fully materialized into the temporary raw partition. Since the *fanout* increased exponentially from test-to-test, so did the total IO performed by this implementation. That is, the relation between the number of data pages written and read per input data page increased sharply from test-to-test. Consequently, it is very likely that in the last test the RDBMS was freeing buffer pages at a slower rate than they were being needed by the query execution algorithm. In this case, reads in advance could not always be performed before other IO events terminated. From this point onward increasing the *fanout* would only lead to further degradation of the IO performance, despite the increased *cpu time* per input page.

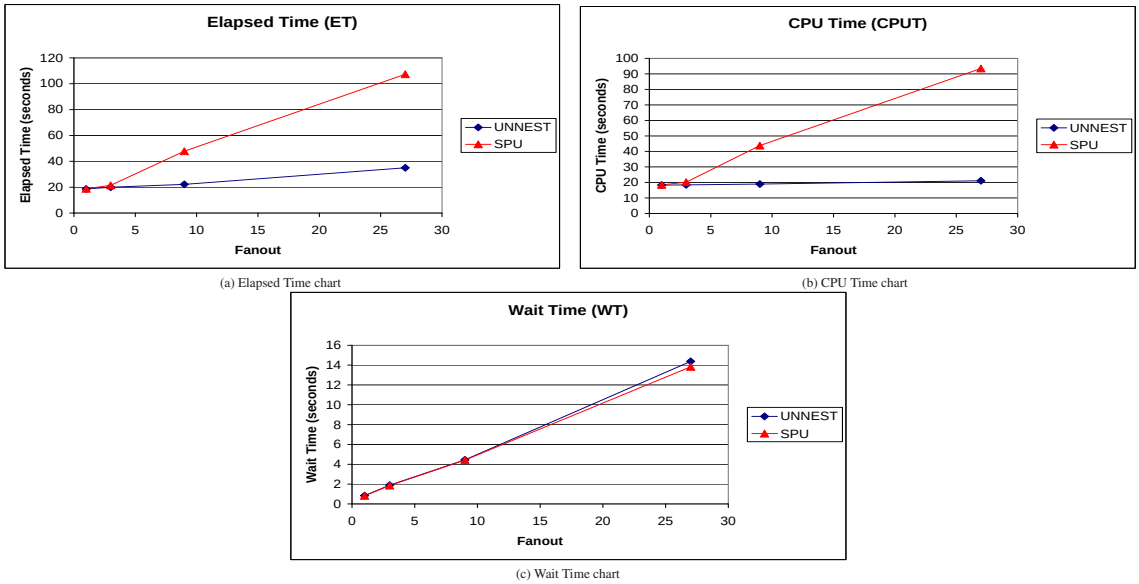
4.6.1.3 DB2 results

In DB2, the $E2$ expression can be implemented using the *SPU* and *unnest* implementation methods. The results of this experiment are shown in Figure 4.16. The *unnest* was the best implementation method.

The two methods presented a different behaviour concerning the *elapsed time* and *cpu time*.

The *unnest* varied the three statistics linearly with the *fanout*. This was not as expected, because for the varying *fanout* experiments we did not predict the possibility of an increasing *wait time* behaviour, as it happened with both methods. In our expectations, we always consider the wait events associated with the reading of input pages as the main cause of delays, since the write operations are normally performed asynchronously and should have negligible contribution to the *wait time*. However, in DB2 the read time was already optimal, it could not improve from any increase in *cpu time*. DB2 maintained its optimal read time performance throughout the experiment for both methods. But it seems that, for both methods, the database system could not sustain its asynchronous write time policy, which lead to a linear increase in the delays in write time.

The *SPU* presented an irregular *elapsed time* slope, which was similar to the *cpu time* slope. In turn, the *cpu time* behaviour was identical to the total IO behaviour (the sum of all physical bytes read and written), both



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.16: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E2$ in the varying *fanout* experiment.

elements presented similar rates of change from test to test. This seems to indicate that the largest portion of the *cpu time* consuming operations for the *SPU* method are IO related. Finally, the irregular IO behaviour resulted from the fact that the DB2 *SPU* changed the number of times it read the source relation with the *fanout*, unlike the Oracle and MSQLS *SPU* method which processed this experiment reading the source table only once. DB2 read the source table once up to a *fanout* of 3. For the *fanout* values of 9 and 27, the data pages of the input relation were read approximately two and three times, respectively. Therefore, the *SPU* presented two different *cpu time* behaviours: the first is comprehended between the *fanout* values 1 to 3; and the second between 9 to 27.

4.6.2 Experiment 5: Execution of $E3$ with varying fanout

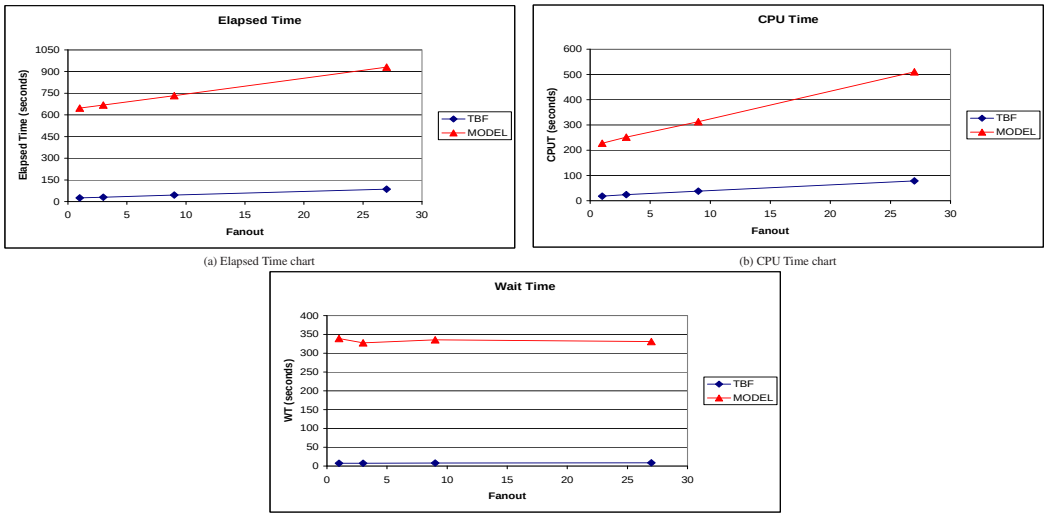
In this section, we report and analyse the results obtained when varying the *fanout* during the execution of the data transformation $E3$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation.

4.6.2.1 Oracle results

In Oracle, the $E3$ expression can be implemented using the *table function* and *model* implementation methods. The results of this experiment are shown in Figure 4.17. The *table function* was the best implementation method, having the smallest *elapsed time*.

The two implementation methods evidenced results that matched our expectations. Both the *table function* and *model* methods linearly increased their *elapsed time* and *cpu time* while maintaining their *wait time* approximately constant when the *fanout* increased. The *table function* behaviour in this experiment was identical to its behaviour in the varying number of input records experiment over the expression $E2$, which was explained in Section 4.6.1.1.

In this experiment, unlike in the previous one (see Section 4.6.1.1), the *model* implementation method perfor-



(a) Elapsed Time chart

(b) CPU Time chart

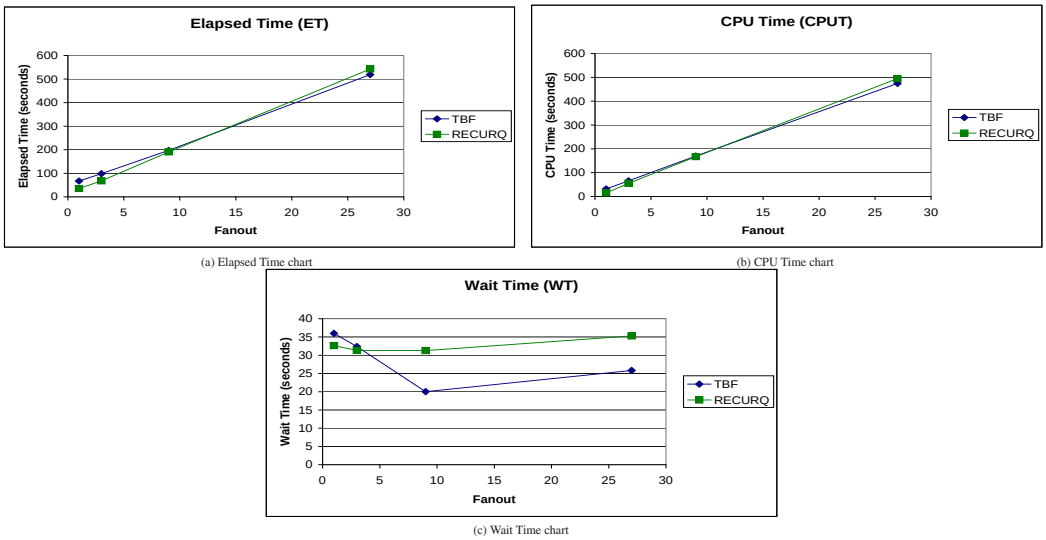
(c) Wait Time chart

Fig. 4.17: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E3$ in the varying *fanout* experiment.

med with a regular behaviour. This results from the fact that the implementation code for this data transformation did not have to be adapted according to the *fanout* of each test. Recall that in the previous experiment the *rules* and *dimension model clauses* were adapted to the *fanout*. Finally, the *wait time* of this method was significantly larger than that of the *table function* due to the high cost of IO performed over the temporary raw partition.

4.6.2.2 SQL Server results

In MSQLS, the $E3$ expression can be implemented using the *table function* and *recursive query* implementation methods. The results of this experiment are shown in Figure 4.18. The *table function* was the best implementation method for high values of *fanout* while the *recursive query* was better for small values.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.18: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E3$ in the varying *fanout* experiment.

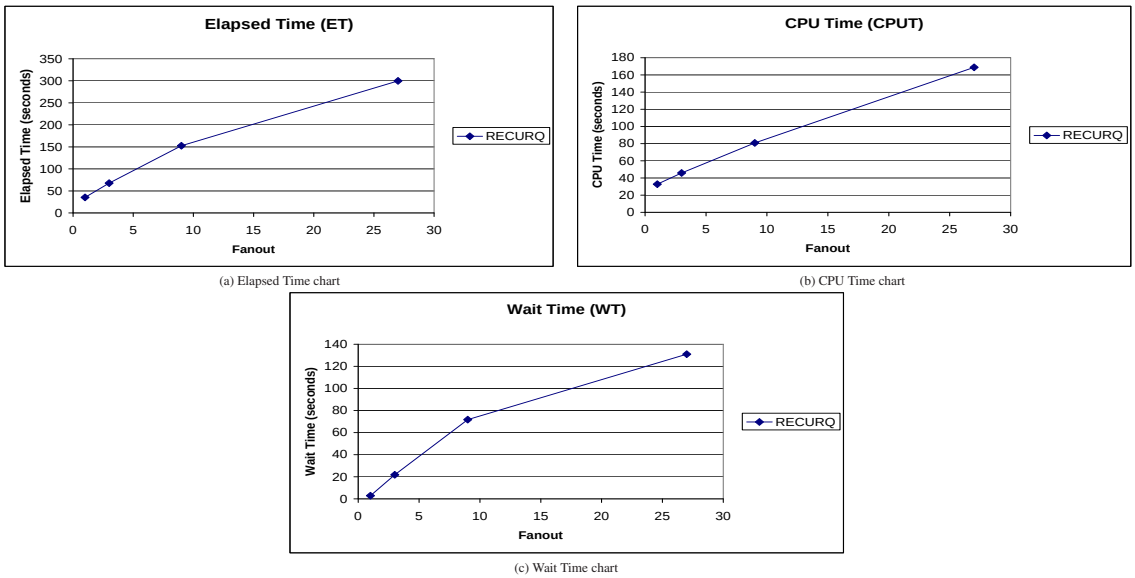
The *elapsed time* and *cpu time* charts (Figure 4.18 (a) and (b)) show that all the implementation methods were affected linearly by the increase of *fanout*. From the *wait time* chart (Figure 4.18 c) we see different behaviours.

The *table function* presented a *wait time* variation identical to the one discussed in the previous MSQLS experiment, see Section 4.6.1.2. The *recursive query* was irregular in its *wait time* variation.

The *recursive query* manifested regular IO behaviour throughout the tests. First, the source table was read only once, making the IO on the source raw partition remain constant. Second, the IO on the target raw partition increased linearly, and was identical to the IO performed by the *table function* in the same raw partition. Third, the temporary raw partition suffered read and write IO in equal proportions, which increased linearly with the *fanout*. However, the presented *wait time* was affected in a non regular manner, increasing slightly its value in the last test. This implementation method processes each input page always at the same rate just like the *SPU*, since in each iteration each input record is only transformed into one output record. Like the *SPU*, this method also seems to maintain an approximately constant *wait time*. It is likely that the increase in IO in the temporary raw partition also leads this method to increase its *wait time*. This would be logical since the read-ahead mechanism does not benefit from increased time to process each input page with the change in *fanout*, but the number of pages to process increase with the *fanout*. This method only materializes a small portion of the data pages it produces while processing. In the fourth test approximately 700 MB were written in the target raw partition, while the temporary raw partition suffered close to 400 MB IO evenly distributed by reads and writes.

4.6.2.3 DB2 results

In DB2, the $E3$ expression can only be implemented using the *recursive query* implementation method. The results of this experiment are shown in Figure 4.19.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.19: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E3$ in the varying *fanout* experiment.

The *elapsed time* chart shows that the *recursive query* increased its cost almost linearly with the *fanout*. This behaviour was the result of a linear increase in *cpu time*, while the *wait time* increased slower than linearly. The *wait time* behaviour is justified by a sublinear increase of the IO in the temporary raw partition with the *fanout*. This was unexpected. We explain these results bellow.

After careful consideration of the IO results, we concluded that *recursive queries* do not vary their temporary IO costs linearly with the *fanout* whenever one or more attributes of the temporary results changes its size according to the recursive iteration steps. More specifically, we realized that in this specific experiment the number of favorite artists written to the materialized temporary table changes in each *recursive query* iteration. For instance, in the first iteration twenty-seven artists are written to the temporary table, 1 artist in the `ANAME` column and 26 artists in the `RESTARTISTS` column. In the second iteration, an artist is removed from the `RESTARTISTS` column, thus only twenty-six artists are written. Following this logic, we know that in the last iteration of each test, $27-(fanout+1)=28-fanout$ artists are written. Consequently, the temporary IO cost of this method in this experiment, is modelled as the sum of an arithmetic progression (e.g: 27 favArists + 26 favArtists + ... + 1 favArtist, if $fanout=27$). Now, by applying the formula of the sum of an arithmetic progression, we conclude that per input tuple the Total Number of Artists Written (TNAW) is given by $TNAW=\frac{fanout}{2}*(55-fanout)$. Knowing this, we can model the Total Temporary Bytes Written (TTBW) as $TTBW=TNAW*m_1*ntups$, where m_1 is the ratio (Bytes/Artist).

Of course, none of these conclusions hold for the $E4$ data transformation, which behaves precisely as expected, since its temporary materialized table is comprised exclusively by columns of fixed size (e.g: Integers).

4.6.3 Experiment 6: Execution of $E4$ with varying fanout

In this section, we report and analyse the results obtained when varying the *fanout* during the execution of the data transformation $E4$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation.

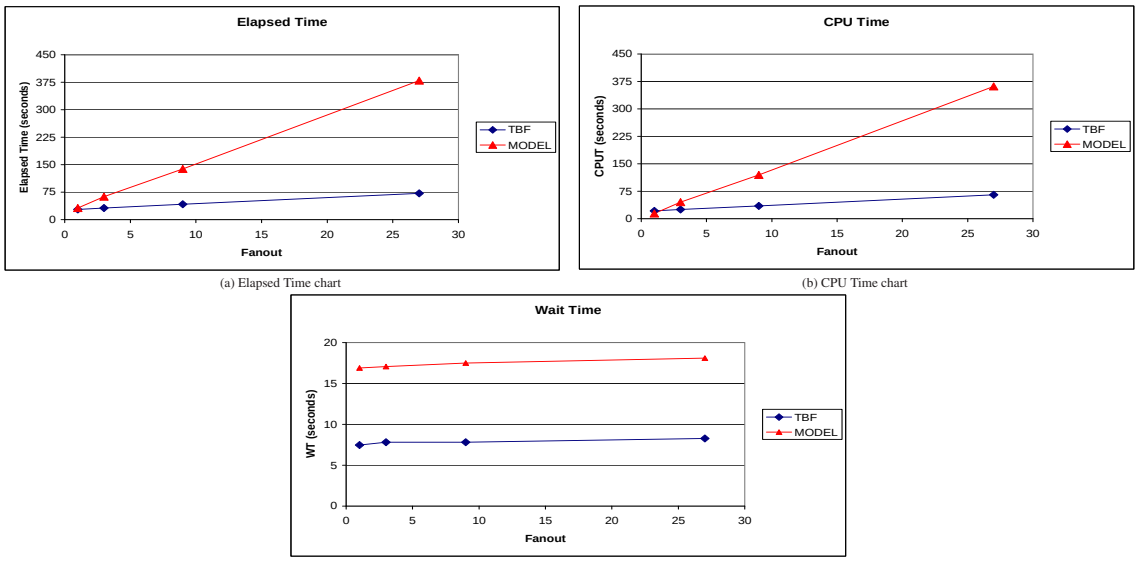
4.6.3.1 Oracle results

In Oracle, the $E4$ expression can be implemented using the *table function* and *model* implementation methods. The results of this experiment are shown in Figure 4.20. The *table function* was the best implementation method, having the smallest *elapsed time*.

The two implementation methods evidenced results that matched our expectations. Both the *table function* and *model* methods linearly increased their *elapsed time* and *cpu time* while maintaining their *wait time* approximately constant when the *fanout* increased. The *table function* and *model* behaviour was identical to the behavior shown in Section 4.6.1.1 and Section 4.6.2.1, respectively.

4.6.3.2 SQL Server results

In MSQLS, the $E4$ expression can be implemented using the *table function* and *recursive query* implementation methods. The results of this experiment are shown in Figure 4.21. The *recursive query* was the best implementation method, having the smaller *elapsed time* in the four tests, however its *elapsed time* slope was slightly steeper than the *table function*, which might mean that for very high values of *fanout* the *table function* might have a better performance.

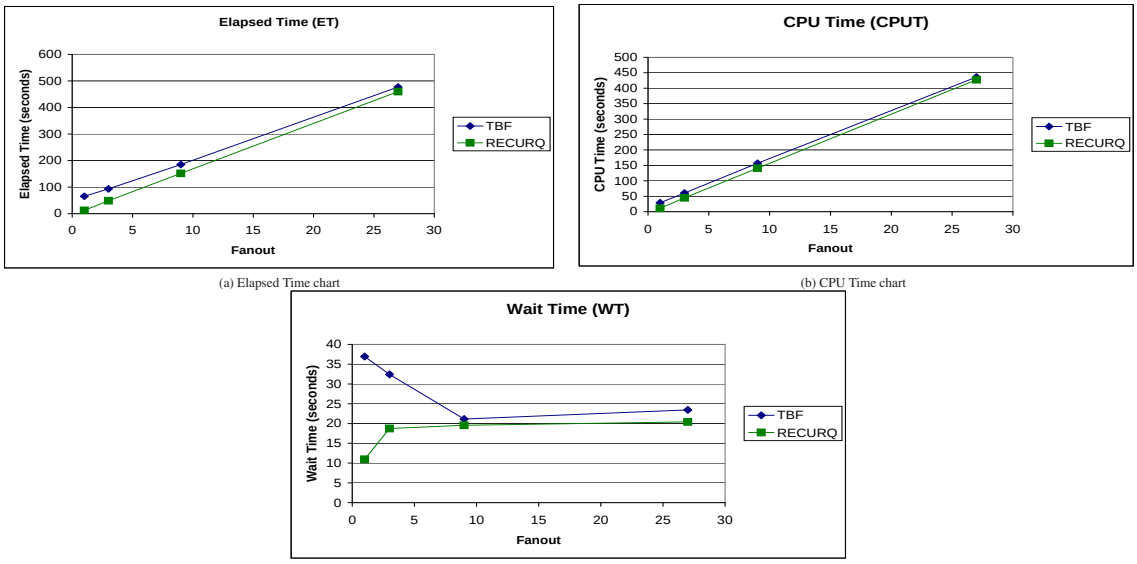


(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.20: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E4$ in the varying *fanout* experiment.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

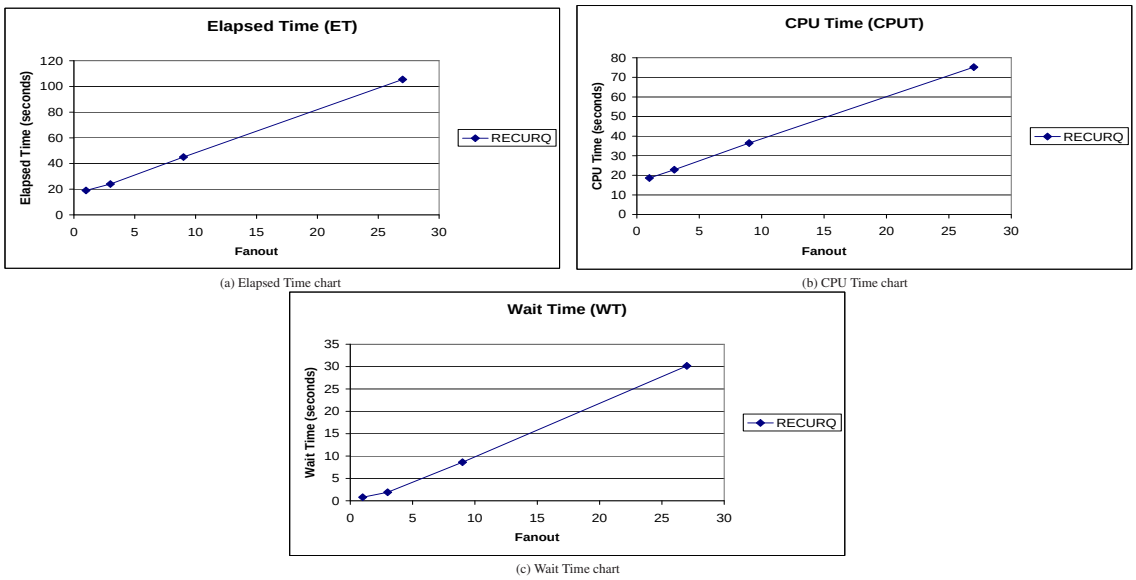
Fig. 4.21: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E4$ in the varying *fanout* experiment.

The *elapsed time* and *cpu time* charts (Figure 4.21 (a) and (b)) show that both the implementation methods were affected linearly by the increase of *fanout*. From the *wait time* chart (Figure 4.21 c) we see different behaviours. The *table function* and *recursive query* presented a *wait time* variation identical to the one discussed in Section 4.6.1.2 and Section 4.6.2.2, respectively.

4.6.3.3 DB2 results

In DB2, the $E4$ expression can only be implemented using the *table function* and *recursive query* implementation methods. The results of this experiment are shown in Figure 4.22.

The *elapsed time* and *cpu time* charts increased linearly with *fanout*, as expected. The *wait time* also increased



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.22: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E4$ in the varying *fanout* experiment.

linearly with the *fanout*. This happened since the write operations were the cause of most of the measured *wait time*, and they increased their number linearly with the *fanout*. The read *wait time* was negligible, it remained constant with the value of approximately 3 seconds throughout the whole experiment.

4.7 Experiments varying the selectivity

This section evaluates the effect of increasing the *selectivity* of a predicate clause over the source relation field `COUNTRY` , when the *ntups* and *fanout* parameters remain constant (set to 25 Million and 5 respectively), on the performance of the various implementation methods supported by each RDBMS capable of implementing the one-to-many data transformations $E2$, $E3$ and $E4$. In the four tests that comprised each experiment, the parameter *selectivity* increased exponentially by five, starting with a value of 1/125 (in test 1) up to 1 (in test 4).

To enforce the desired selectivity, we generated the source table (`PROFILE`) records uniformly distributed by country, and we controlled the number of distinct countries in our data set. Afterwards, each of the queries we run against the database held the equality predicate clause: `WHERE PROFILE.COUNTRY = 'Portugal'` . Thus, the selectivity associated to each query was $1/ncountries$. For instance, for a *selectivity* of $1/5$, the data generated had five distinct countries with each country being referred by $ntups/5$ records.

Finally, to perform this experiment and make the results as predictable as possible in terms of IO, we created the source table `PROFILE` clustered by country. This way using an index to fetch all the records that match the predicate clause will incur in minimum IO. Otherwise, each record mapped by a non clustering index could potentially be held in distinct and non consecutive table pages, rendering the IO results completely unpredictable. Therefore, the approach we followed ensures that no database block is fetched more than once; and if the index is used by the RDBMS, only the necessary blocks are retrieved [2].

[2]Even if the RDBMS holds a usable index, the optimizer may still choose to ignore the index and perform a full table scan instead. This

In terms of expectations, our predictions for these experiments were the same as for the varying number records experiments described in Section 4.5. The two experiences are alike, one controls the number of input records that must processed by affecting the size of the source relation, the other does the same by controlling the number of input records that satisfy a predicate clause. In both cases, what is being changed is the number of input records that must be processed by the data transformation. Therefore, the *cpu time* should increase linearly with the *selectivity*. The *wait time* should increase linearly with the *selectivity* if the RDBMS decides that it is worthwhile to access the input relation through the index. However, if it decides to table scan the input relation, then the *wait time* should remain approximately constant, while showing a tendency to decrease its value as the *cpu time* increases, as discussed in *wait time* expectations for the varying *fanout* experiments in Section 4.6.

The purpose of this experiment is do determine if for one-to-many data transformations the RDBMS are capable of optimizing the data access to a source relation given the presence of a predicate clause in the query. The following sections present the results obtained when varying the *selectivity* value for the three one-to-many data transformations.

4.7.1 Experiment 7: Execution of $E2$ with varying selectivity

In this section, we report and analyse the results obtained in the varying *selectivity* experiment targeting the data transformation $E2$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation.

4.7.1.1 Oracle results

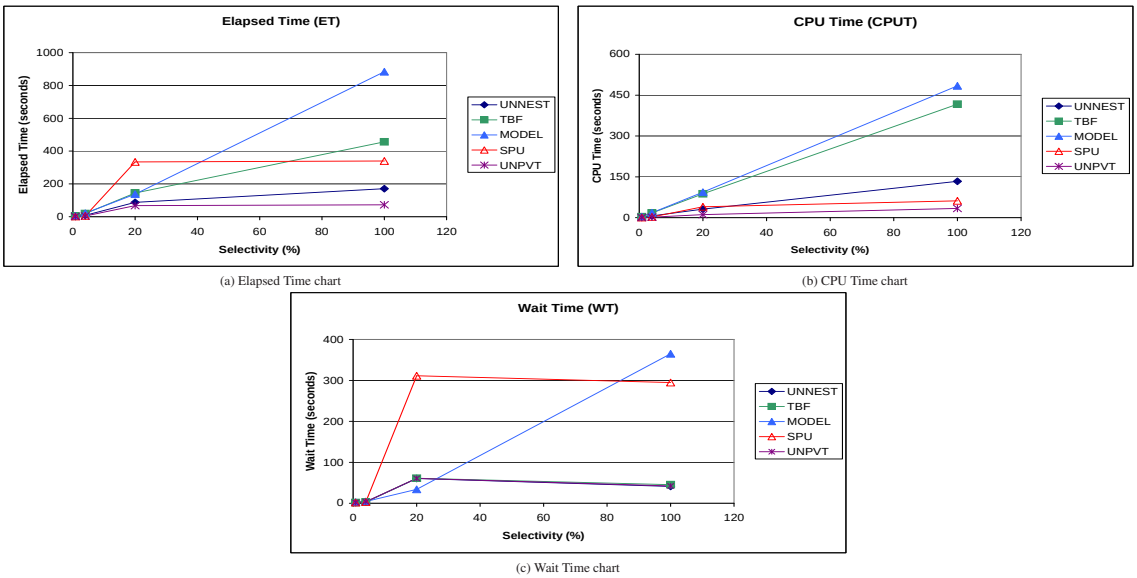
In Oracle, the $E2$ expression can be implemented using the *SPU*, *unpivot*, *table function*, *unnest* and *model* implementation methods. The results of this experiment are shown in Figure 4.23. The *unpivot* was the best implementation method, having the smallest *elapsed time*.

The experiment results show that with the exception of the *model* implementation method, none of the others evidenced a linear increase of the *elapsed time* with the *selectivity*. To understand this we will explain bellow the *cpu time* and *wait time* behaviour of these methods.

With the exception of the *model*, we can see that all the implementation methods showed identical *wait time* behavior. That is, their *wait time* increased sharply into a maximum at the selectivity value of 20%, and then it decreased slightly at the *selectivity* value of 100%. This happened because despite the clustered index organization of the input table, Oracle only used clustered index seeks for *selectivity* values lower than 20%. Instead, for *selectivity* values of 20% or above, Oracle opted to use table scans. Consequently, at the *selectivity* value of 20%, Oracle was reading the full 25M input records contained in the source table. This is what explains the sudden peak in *wait time* at 20%. The small reduction in *wait time* that can be seen for the *selectivity* value of 100% is according to our expectations, and is due to the *cpu time* increase at this value[3]. Finally, while the *unpivot*, *unnest* and *table*

has happened for the selectivity of $1/5$.

[3]Recall that by increasing the *cpu time* of a data transformation while maintaining the number of data pages read, the rate at which the query execution operators request input pages from the buffer is reduced.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.23: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E2$ in the varying *selectivity* experiment.

function presented identical *wait time* values, the *SPU* presented higher *wait time*. The reason for this is that the first three methods only read the source table once, while the *SPU* read it multiple times (5 table scans).

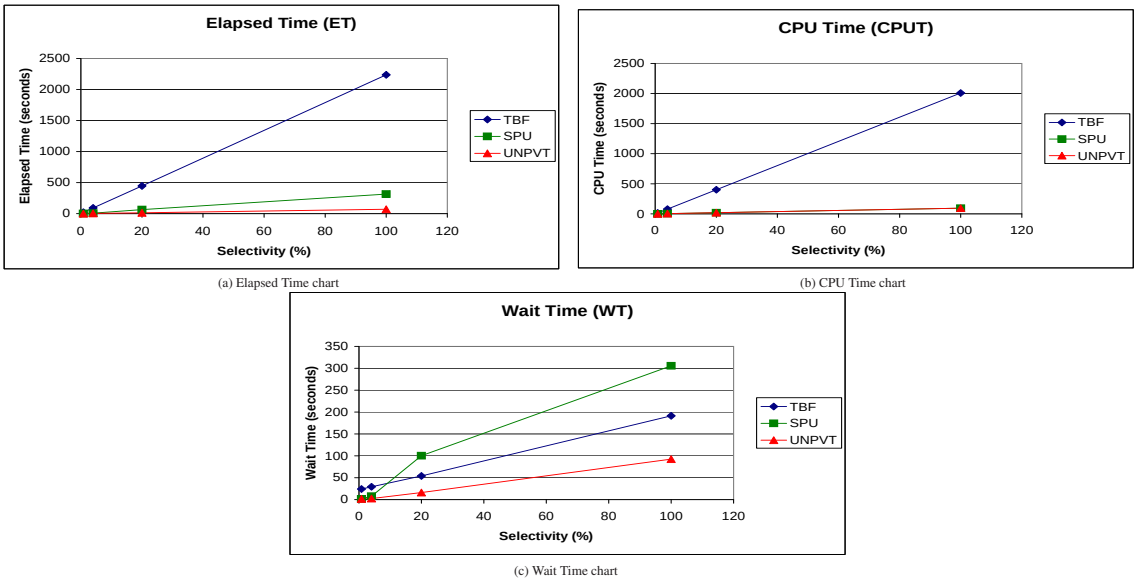
Concerning the *cpu time* chart, all the implementation methods but the *SPU*, displayed a linear increase of this statistic, as expected. However, the *SPU* displayed identical *cpu time* at the *selectivity* values of 20% and 100% (with a slightly higher *cpu time* at 100%). The reason for this is simple. As we have explained before, see Section 4.6.1.1, the *SPU* query execution plan consists in the concatenation of the output of a set of table scans, followed by a *select into table* operation. For a *selectivity* value of 20% and 100%, this method performs 5 table scans that process the whole input table. All the other implementation methods do most of their processing time in the operators that follow their data access method. Since the output of the data access method (e.g. table scan) is five times higher for a *selectivity* value of 100% than 20%, the *cpu time* of these operators increased significantly in the final test, unlike the *SPU*.

The *model* implementation method displayed results that matched our expectations. Its regularity in behaviour is due to its use of fixed query execution plan in all the tests of this experiment. Unlike the other implementation methods that changed their access methods from index seeks to table scans when reaching the *selectivity* value of 20%, the *model* always used index rage scans as an access method, ensuring that only the necessary data pages were retrieved. The reason for this difference is that the *model* method used two predicate the clauses. One, the predicate on the `COUNTRY` attribute of the source relation (like all the other implementation methods). Two, a predicate on the `ID` attribute. Recall that in Section 4.3.4, we explained that the data transformations, when implemented with the *model* method, had to be subdivided into multiple sub-queries. Each sub-query processed at most one million input records. Thus, the predicate clause over the `ID` always had an associated *selectivity* of $1/25$ per sub-query. Jointly, the two filters managed to always induce the query optimizer to prefer index range scans over table scans, taking full advantage of the `COUNTRY, ID` clustered organization of the source table. Nevertheless, we can see in *wait time* chart that this method manifested a significant slope increase after the *selectivity* value of 20%. This

happened because the temporary raw partition started being used at this point. Before that, no IO was performed in this raw partition.

4.7.1.2 SQL Server results

In MSQLS, the $E2$ expression can be implemented using the *SPU*, *unpivot*, and *table function* implementation methods. The results of this experiment are shown in Figure 4.24. The *unpivot* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

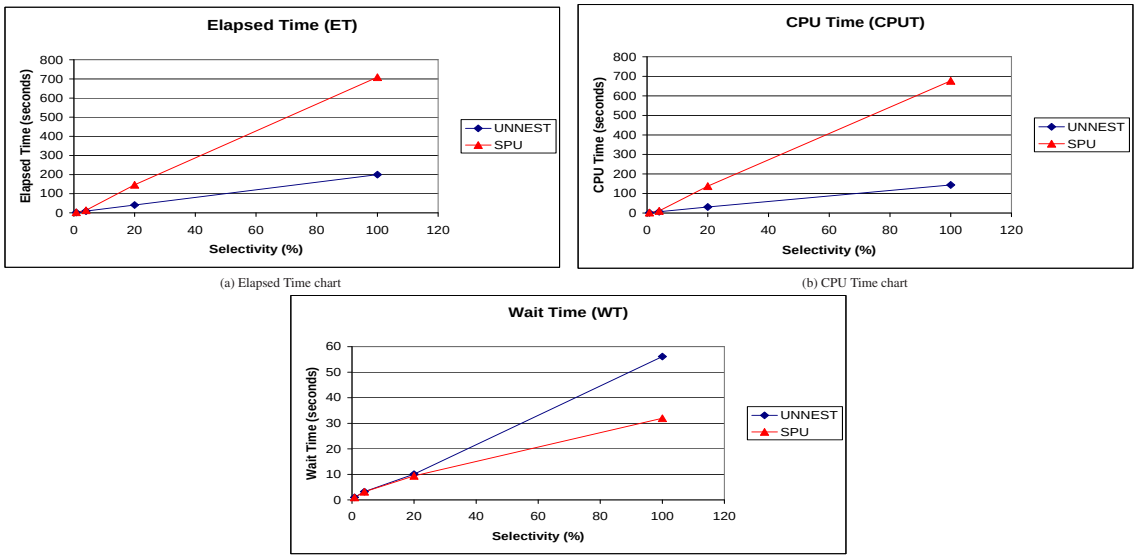
Fig. 4.24: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E2$ in the varying *selectivity* experiment.

The results were according to our expectations. The *elapsed time* and *cpu time* charts (Figure 4.24 (a) and (b), respectively) show that all the implementation methods were affected linearly by the increase in *selectivity*. The *wait time* results (Figure 4.24 c) increased linearly for the *unpivot* and *table function*. However, the *SPU* had a significant increase in *wait time* when the test of *selectivity* of 20% was reached. This is explained by the fact that a relation with 20% the size of source table is too large to entirely fit in the 700MB cache. Consequently, input data pages have to be read more than once, since the query execution plan of this mechanism consists in five clustered index scans.

All the methods increased linearly their IO in all the raw partitions they used. But unlike Oracle, see Section 4.7.1.1, which opted to use table scans to access the source table data pages for *selectivity* greater or equal than 20%, MSQLS always used clustered index seeks to retrieve the records that satisfied the predicate clause. Therefore, in MSQLS no changes in the query execution plans ever occurred, and minimal IO was performed.

4.7.1.3 DB2 results

In DB2, the $E2$ expression can be implemented using the *SPU* and *unnest* implementation methods. The results of this experiment are shown in Figure 4.25. The *unnest* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.25: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E2$ in the varying *selectivity* experiment.

The results were according to our expectations. The *elapsed time* and *cpu time* charts (Figure 4.25 (a) and (b), respectively) show that both implementation methods were affected linearly by the increase in *selectivity*. The *wait time* results (Figure 4.25 c) increased linearly for the *spu*, while the *unnest* presented a significant *wait time* increase in the last test.

There is no obvious explanation for the irregular *wait time* increase of the *unnest* in the last test. The IO in the raw partitions used (source and target) increased linearly. However, we noticed that in this last test, the *unnest* evidenced an extremely high standard deviation value for this statistic over the five runs. Its value was approximately 24 seconds. Moreover, the lowest *wait time* observed was 30 seconds, which would be within our expectations, while the highest was 81 seconds.

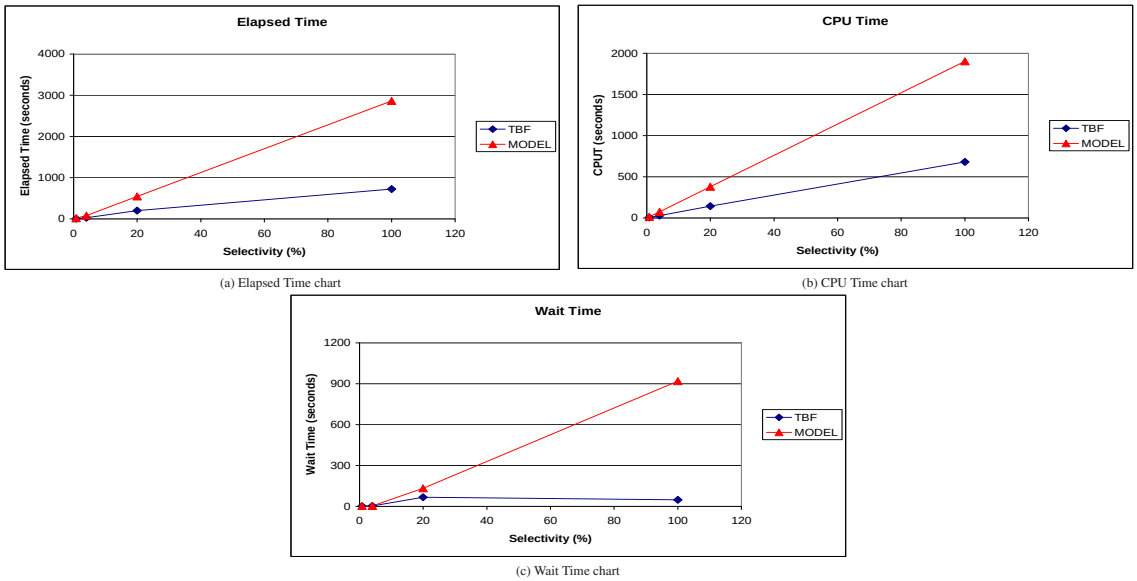
Also interesting is the fact that the *SPU wait time* behaviour was very regular, even though when the *selectivity* of 20% was reached the necessary source data was physically read 5 times. This happened because DB2 was, both for the *SPU* and *unnest*, capable of using its read-ahead mechanism (called pre-fetch), with maximum efficiency, as explained in Section 4.5.1.3. For both methods, the read wait time was approximately 3 seconds throughout the whole experiment.

4.7.2 Experiment 8: Execution of $E3$ with varying selectivity

In this section, we report and analyse the results obtained in the varying *selectivity* experiment targeting the data transformation $E3$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation.

4.7.2.1 Oracle results

In Oracle, the $E3$ expression can be implemented using the *table function* and *model* implementation methods. The results of this experiment are shown in Figure 4.26. The *table function* was the best implementation method, having the smallest *elapsed time*



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

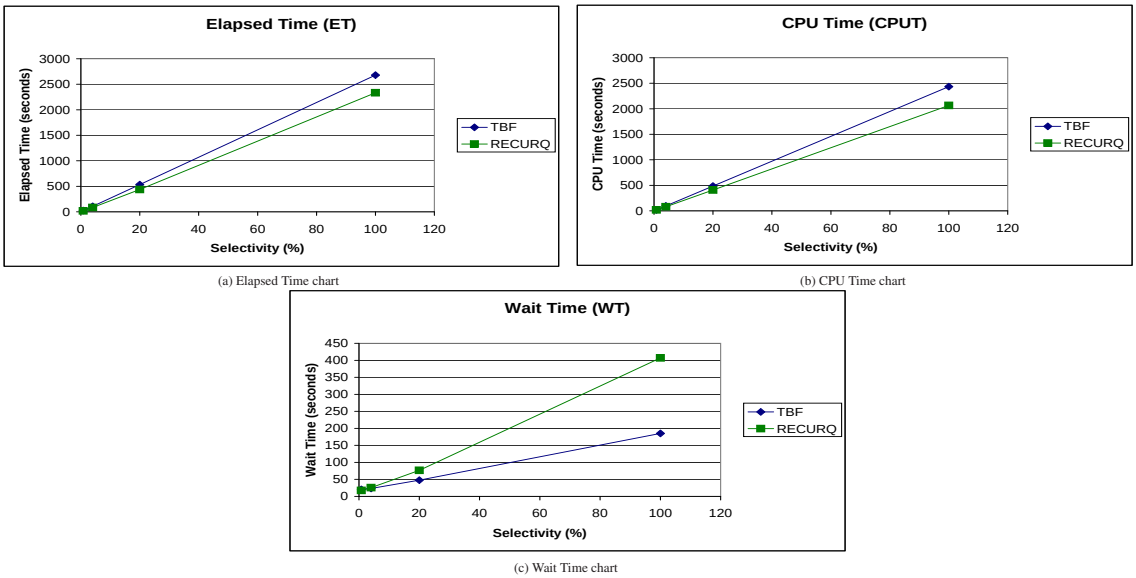
Fig. 4.26: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E3$ in the varying *selectivity* experiment.

The *elapsed time*, *cpu time* and *wait time* results of the two implementation methods were identical to the ones obtained in the Oracle varying *selectivity* experiment over the expression $E2$, see Section 4.7.1.1. The *cpu time* increased linearly for the two methods. Their *wait time* varied non linearly for the reasons that were explained in the previous experiment: the *table function* started performing table scans instead of index seeks when it reached the *selectivity* of 20%; and the *model* only started performing IO in the temporary raw partition when it reached the *selectivity* of 20%.

4.7.2.2 SQL Server results

In MSQLS, the $E3$ expression can be implemented using the *table function* and *recursive query* implementation methods. The results of this experiment are shown in Figure 4.27. The *recursive query* was the best implementation method, having the smallest *elapsed time*.

The *elapsed time*, *cpu time* and *wait time* charts (Figure 4.27 (a), (b) and (c), respectively) show that all the implementation methods were affected linearly by the increase of *selectivity* of the data transformation, as expected. The slight increase in *wait time* slope of the *recursive query* occurring between the *selectivity* values of 20% to 100%, is explained by that the fact that method only required the use of the temporary raw partition reaching the selectivity of 100%.



(a) Elapsed Time chart

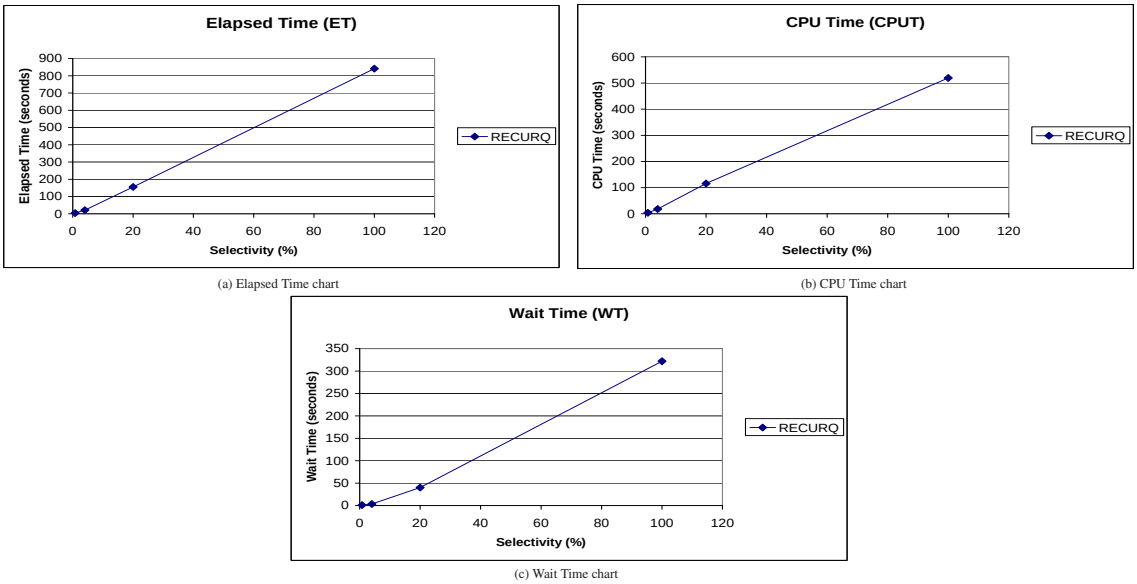
(b) CPU Time chart

(c) Wait Time chart

Fig. 4.27: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E3$ in the varying *selectivity* experiment.

4.7.2.3 DB2 results

In DB2, the $E3$ expression can only be implemented using the *recursive query* implementation method. The results of this experiment are shown in Figure 4.28.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

Fig. 4.28: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E3$ in the varying *selectivity* experiment.

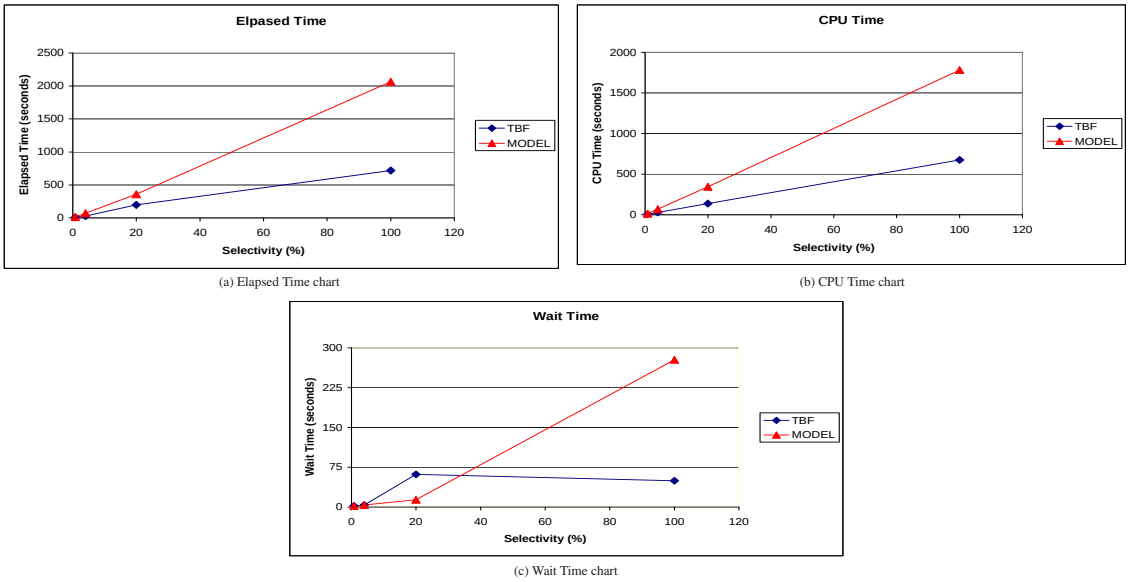
The *elapsed time* and *cpu time* charts (Figure 4.28 (a) and (b), respectively) show that the implementation methods were affected linearly by the increase of *selectivity* of the data transformation, as expected. The irregular slope changes in the *wait time* chart are explained by the fact that this method only required the use of the temporary raw partition reaching the selectivity of 20%. Thus, the IO behaviours from 0.8% to 4% of *selectivity* and 20% to 100% differed from one-another.

4.7.3 Experiment 9: Execution of $E4$ with varying selectivity

In this section, we report and analyse the results obtained in the varying *selectivity* experiment targeting the data transformation $E4$. The results portray the performance of the different implementation methods that each RDBMS could use to perform this data transformation.

4.7.3.1 Oracle results

In Oracle, the $E4$ expression can be implemented using the *table function* and *model* implementation methods. The results of this experiment are shown in Figure 4.29. The *table function* was the best implementation method, having the smallest *elapsed time*.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

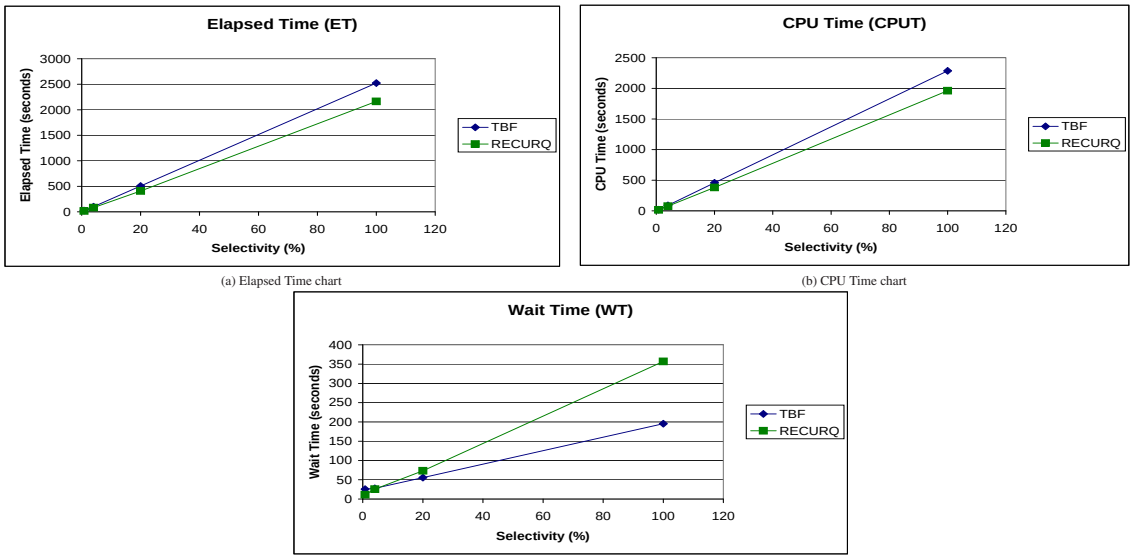
Fig. 4.29: Elapsed Time, CPU Time and Wait Time charts of Oracle, when implementing $E4$ in the varying *selectivity* experiment.

The *elapsed time*, *cpu time* and *wait time* results of the two implementation methods were identical to the ones obtained in the Oracle varying *selectivity* experiment over the expression $E2$, see Section 4.7.1.1. The only substantial difference was that *model* method only started performing IO in the temporary raw partition when it reached the *selectivity* of 100%, whereas in the previous experiments it started it when reaching the *selectivity* of 20%. For this reason, the spike in the *wait time* slope is only evidenced later.

4.7.3.2 SQL Server results

In MSQLS, the $E4$ expression can be implemented using the *table function* and *recursive query* implementation methods. The results of this experiment are shown in Figure 4.30. The *recursive query* was clearly the best implementation method for high values of *selectivity*. For low values, both methods have similar behavior.

The *elapsed time*, *cpu time* and *wait time* results of the two implementation methods matched our expectations, and were identical to the ones obtained in the MSQLS varying *selectivity* experiment over the expression $E3$, see Section 4.7.2.2.



(a) Elapsed Time chart

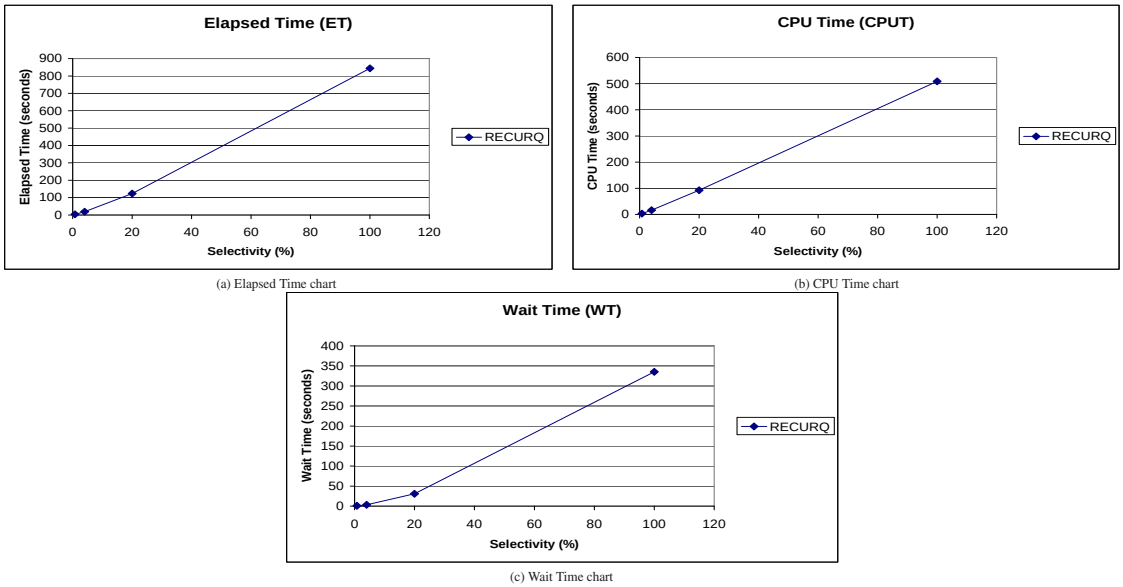
(b) CPU Time chart

(c) Wait Time chart

Fig. 4.30: Elapsed Time, CPU Time and Wait Time charts of SQL Server, when implementing $E4$ in the varying *selectivity* experiment.

4.7.3.3 DB2 results

In DB2, the $E4$ expression can only be implemented using the *recursive query* implementation method. The results of this experiment are shown in Figure 4.31.



(a) Elapsed Time chart

(b) CPU Time chart

(c) Wait Time chart

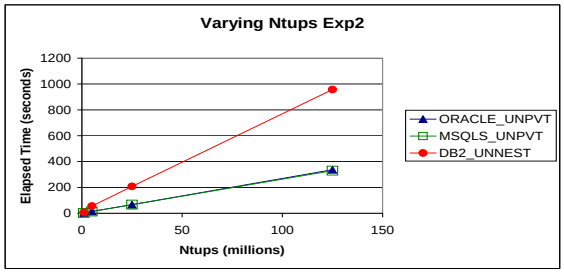
Fig. 4.31: Elapsed Time, CPU Time and Wait Time charts of DB2, when implementing $E4$ in the varying *selectivity* experiment.

The *elapsed time* and *cpu time* results of the two implementation methods matched our expectations, while the and *wait time* presented irregular slope changes. These results were identical to the ones obtained in the DB2 varying *selectivity* experiment over the expression $E3$, see Section 4.7.2.3.

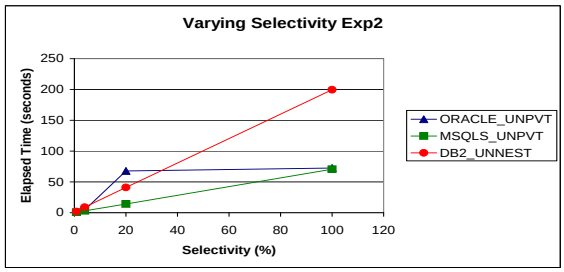
4.8 Comparing the performance of the three RDBMS

In this section we compare the performance demonstrated by the best performing method of each RDBMS in each of the nine experiments described in the previous sections. First, we consider the varying *ntups* and *selectivity* experiments. These experiments have in common the 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. As explained in the Sections 4.5 and 4.7, 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 4.32, 4.33 and 4.34 present the results of the best performing implementation method of each RDBMS in each of the six varying *ntups* and *selectivity* experiments. As expected, the best performing implementation method of each RDBMS was the same in both experiments.

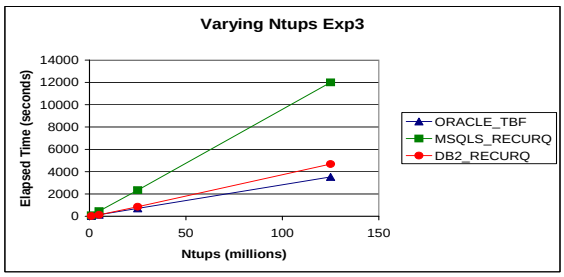


(a) Best RDBMS $E2$ methods when varying *ntups*

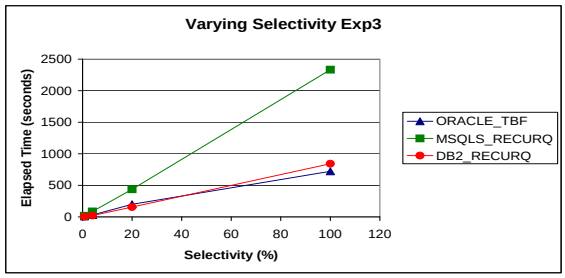


(b) Best RDBMS $E2$ methods when varying *selectivity*

Fig. 4.32: Charts illustrating the best implementation method of each RDBMS for the varying number of records and *selectivity* experiments over $E2$.



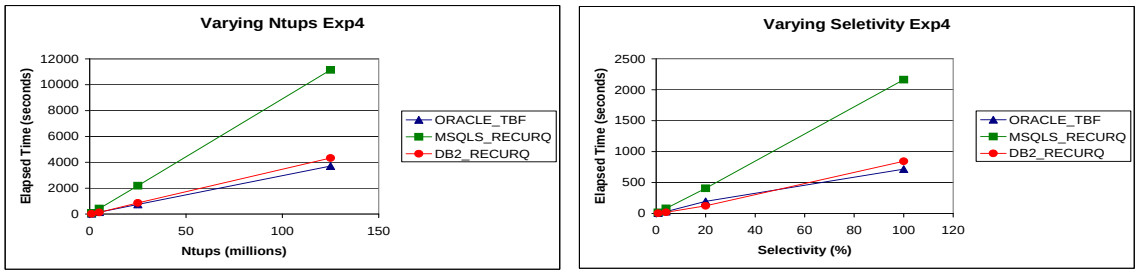
(a) Best RDBMS $E3$ methods when varying *ntups*



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

Fig. 4.33: 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 4.32 (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



(a) Best RDBMS $E4$ methods when varying *ntups*

(b) Best RDBMS $E4$ methods when varying *selectivity*

Fig. 4.34: Charts illustrating the best implementation method of each RDBMS for the varying number of records and *selectivity* experiments over $E4$.

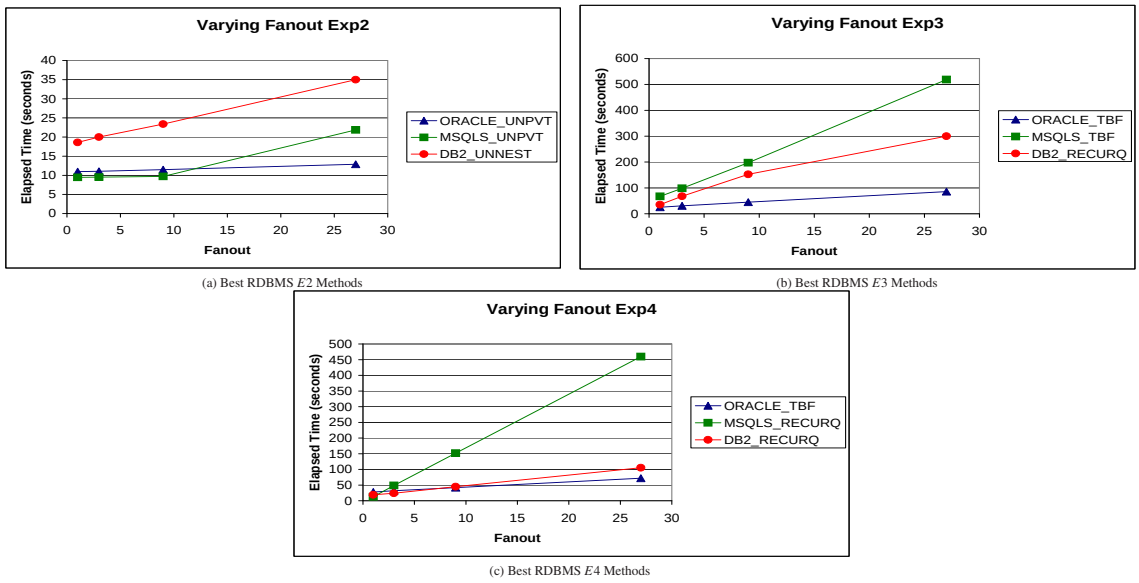
systems were equally good in these two experiments, though MSQLS was more intelligent in its data access methods choices. The DB2 *unnest* method was also efficient, with an average elapsed time cost of 1.8 seconds to produce 1M output records. Even so, this value was considerably higher than the approximately 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, it 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 expressions $E3$ and $E4$, see the Figures 4.33 and 4.34, 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*. 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[4]. 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. 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 4.35 presents the results of the best performing implementation method of each RDBMS in each of the three varying *fanout* experiments. Concerning the expression $E2$, see Figure 4.35 (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 (see Section 4.6.1.1). 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 (see Section 4.6.1.2). As in the varying *selectivity* and *ntups* experiments over expression

[4]Recall that *recursive queries* materialize the results of each processing cycle into a temporary table.

$E2$, 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* cost 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

(c) Best RDBMS $E4$ Methods

Fig. 4.35: Charts illustrating the best implementation method of each RDBMS for the three varying *fanout* experiments.

In the varying *fanout* experiments over the expressions $E3$ and $E4$, see the Figure 4.35 (b) and (c), the best performing implementation method in Oracle was the *table function* in both experiments. In MSQLS, the *table function* and *recursive query* methods had very similar results. In the expression $E3$ the best one was the *table function*, while in $E4$ the best one was the *recursive query*. 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. Over the expression $E4$, DB2 was alike in *elapsed time* performance to Oracle. This happened because DB2 performed negligible IO in the temporary raw partition in this experiment. Overall, the Oracle *table function* was better in handling both the $E3$ and $E4$ expressions. This is due to the fact that the *recursive query* is an expensive implementation method, as explained before, while the MSQLS implementation of the *table function* also materializes the output results into the temporary raw partition, all the write IO on it is fully logged, and it also has very high cpu cost. Consequently, the Oracle *table function* was the best performing implementation method in the two experiments.

5 Conclusion

Throughout this thesis we have tried to understand the capabilities of different RDBMS to support one-to-many data transformations:

- Chapter 1 introduced the concept of one-to-many data transformations by means of a simplified motivating example, where we presented and discussed the two fundamental classes of these data transformations: *bounded* and *unbounded*. We identified three domains where the application of these data transformations are of use: *(i)* restructuring relations containing semantically related attributes (e.g. a relation with more than one favorite friend column); *(ii)* restructuring relations containing multivalued data (e.g. a relation with an attribute whose column values are lists of artists); and *(iii)* restructuring relations containing aggregated data (e.g. a relation with an attribute whose column values are the output of a `GROUP BY` operation, for instance, the sales by trimester of company). The purpose of the first chapter was to acknowledge the existence of this class of data transformations, and to identify potential domains for their use.

- Chapter 2 presented different technologies capable of processing large volumes of data, focusing in RDBMS, and their means of supporting one-to-many data transformations. Specifically, we identified that the *SPU*, *unpivot*, *unnest*, *table function*, *model* and *recursive query* are the mechanisms supported by some of the RDBMS[1] that enable us to process *bounded* or *unbounded* - or both - one-to-many data transformations.

- Chapter 3 reviewed the motivating example, formalizing it into a schema migration problem. We enlarged the data transformation scenarios with a third challenge, which is representative of the domain of one-to-many data transformations over relations containing aggregated data. Thus, our data migration challenge incorporated three distinct one-to-many data transformations, that were representative of the three domains of use of one-to-many data transformations identified in Chapter 1. Each of the data transformations was formally described by an extended algebra expression, using the notation introduced in (Carreira et al., 2007). Finally, we presented all the possible implementation solutions for the three extended algebra expressions using the RDBMS implementation methods that had been identified and discussed in Chapter 2.

- Chapter 4 presented the benchmark results of the different implementation solutions discussed in Chapter 3. The benchmark purpose was the evaluation of the performance of different implementation methods capable of supporting one-to-many data transformations, in each individual RDBMS.

In the following sections, we draw the conclusions to which our practical study of one-to-many data transformations has lead us; and we discuss future work of interest in the area of one-to-many data transformations.

[1] Not all of them are supported by the three RDBMS (Oracle, MSQLS and DB2).

5.1 Conclusions

This thesis provides four essential contributions: *(i)* we assessed 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 terms of supporting the *bounded* and *unbounded* subclasses of these data transformations; *(ii)* we discussed how each method can be used to implement solutions for these problems, and their code complexity; *(iii)* we analysed the computational resources consumed by each implementation method; *(iv)* we benchmarked the performance of each implementation method supported by each RDBMS, studied how they were affected by changes of each individual *critical parameter*, and compared the performance of the best implementation method in each experiment of the three RDBMS.

Concerning our first goal, Table 5.1 shows the different RDBMS implementation methods that can support one-to-many data transformations, their expressive power, and the RDBMS where they are available. We concluded the following:

(i) Both *bounded* and *unbounded* data transformations are supported in the three RDBMS;

(ii) The implementation methods that support *unbounded* also support *bounded* data transformations;

(iii) There are no universally common solutions with sufficient expressive power to support *unbounded* data transformations. Among the RDBMS, only three different implementation methods can support *unbounded* data transformations. None of them is common to all three RDBMS. They are: the *model*, exclusive to Oracle; the *table function*, in Oracle and MSQLS; and the *recursive query*, in MSQLS and DB2.

IMP. METHOD	ORACLE	MSQLS	DB2
SPU	B	B	B
unpivot	$\bar{B}$	$\bar{B}$	
unnest	B		B
model	BU		
table function	BU	BU	
recursive query		BU	BU

Legend

B	*bounded*
$\bar{B}$	subset of *bounded*
U	*unbounded*

Table 5.1: Summary of the RDBMS implementation methods capable of supporting one-to-many data transformations, the database systems that support them and their respective expressive power.

Our second goal was to evaluate these implementation methods concerning the complexity of specifying one-to-many data transformations. Table 5.2 resumes our conclusions:

(i) The *bounded* one-to-many data transformations are simple to implement using either the *SPU*, *unpivot* and *unnest* implementation methods.

(ii) The *unbounded* one-to-many data transformations can go from simple to implement (e.g. using the *table function*) to very complex to implement (e.g. using the *model*), depending on the implementation method chosen and the RDBMS used.

(iii) With the exception of the MSQLS *table function*, all the implementation methods provide declarative solutions for the implementation of one-to-many data transformations.

IMP. METHOD	CODE COMPLEXITY
SPU	simple
unpivot	simple
unnest	simple
model	very complex
table function	simple
recursive query	complex

Table 5.2: Summary of the RDBMS implementation methods code complexity when implementing one-to-many data transformations, in a three degrees scale of complexity (simple, complex and very complex).

Our third goal was to have an understanding of the computational resources consumption associated to the physical execution algorithm of each implementation method. Table 5.3 resumes our evaluation of the resource consumption profiles of the different implementation methods. We concluded that:

(i) Only the implementation methods that support *bounded* one-to-many data transformations exclusively have low CPU cost, namely the *SPU* (in Oracle and MSQLS), *unnest* and *unpivot*.

(ii) All the implementation methods with sufficient expressive power to support *unbounded* one-to-many data transformations suffer from query execution plans with high or very high CPU cost, namely the *table function* and *model*. The *recursive query* results over the $E3$ and $E4$ data transformations, when compared to the *table function* in the same experiments, indicate that this implementation method has high and very high *cpu time* cost in DB2 and MSQLS, respectively.

(iii) With the exception of the Oracle *table function*, all of the implementation methods with sufficient expressive power to support *unbounded* data transformations depend on the use of auxiliary memory[2] (in the temporary tablespace) to perform their computations. This is undesirable, as it leads to a most inefficient computation process of the *unbounded* data transformations.

(iv) The ideal query execution plan for one-to-many data transformations should have a computational resource consumption profile with low CPU cost when processing simple *mapper functions* (e.g. our expression $E2$), that logically processes the input in a single-pass (e.g. unlike the *SPU*[3]), and that does not depend on the use of auxiliary memory (e.g. unlike the *model*, *recursive query* and the MSQLS *table function*).

Concerning our third goal, we also summarize in Table 5.4 the nature the time components (CPU, IO wait, or both), which were most determinant to the *elapsed time* performance of different implementation methods in the varying number of records experiments. We concluded that:

[2]The use of auxiliary memory by the *model* and *recursive query* methods is comprehensible since they were not designed to support one-to-many data transformations specifically, and their intended purposes require their dependency on auxiliary memory.

[3]For that matter, the *recursive query* does not process its input in a single-pass as well. If we consider that the output of an iteration is used as the input of the following iteration.

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.

Table 5.3: 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).

IMP. METHOD	$E2$	$E3$	$E4$
SPU (Oracle)	IO-Bounded	–	–
SPU (MSQLS)	IO&CPU-Bounded	–	–
SPU (DB2)	CPU-Bounded	–	–
unpivot	IO&CPU-Bounded	–	–
unnest (Oracle)	CPU-Bounded	–	–
unnest (DB2)	IO&CPU-Bounded	–	–
table function	CPU-Bounded	CPU-Bounded	CPU-Bounded
model	CPU-Bounded	IO&CPU-Bounded	CPU-Bounded
recursive query (MSQLS)	–	CPU-Bounded	CPU-Bounded
recursive query (DB2)	–	IO&CPU-Bounded	IO&CPU-Bounded

LEGEND:

IO-Bounded More than 75% of the *elapsed time* in the varying number of records experiment was spent waiting for IO completion.

CPU-Bounded More than 75% of the *elapsed time* in the varying number of records experiment was spent processing.

IO&CPU-Bounded Neither of the previous two conditions was met.

Table 5.4: Summary of the nature of the time components most determinant to the *elapsed time* performance of each implementation method in the varying number of records experiments.

(i) We cannot establish any clear relationship between the resource consumption profile of one implementation method, as illustrated in Table 5.3, and the nature of the time components that are most determinant to its *elapsed time* performance. However, the implementation methods with high or very high CPU cost are either CPU-Bounded or IO&CPU-Bounded , see Tables 5.3 and 5.4.

(ii) The *SPU* was most efficient in Oracle, where it was also `IO-Bounded` . Contrarily, this method was least efficient in DB2, where it was `CPU-Bounded` . Unlike in Oracle and MSQLS, the DB2 *SPU* presented high *cpu time* cost, see Table 5.3. Moreover, the DB2 *SPU* manifested the lowest *wait time*.

(iii) The *unnest* in DB2 was `IO&CPU-Bounded` while in Oracle it was `CPU-Bounded` , this was a consequence of DB2 having higher *wait time* associated to its write operations.

(iv) The *recursive query* in DB2 was `IO&CPU-Bounded` while in MSQLS it was `CPU-Bounded` , this happened because MSQLS had considerably higher *cpu time* consumption.

Our next goal was to study how the *critical parameters*, individually, affect the performance of one-to-many data transformations. We concluded that:

(i) Overall, all the implementation methods that were tested in the several experiments evidenced that they were linearly affected by the changes in the *ntups*, *fanout* and *selectivity* parameters, as expected. The only exceptions were: *(i)* the Oracle implementation methods in the varying *selectivity* experiments, which suffered from changes in the data access methods used in their query execution plans with the change of the *selectivity*; *(ii)* the MSQLS *unpivot* in the varying *fanout* experiment over the expression $E2$, which suffered from a non linear *cpu time* increase in the last test; and *(iii)* the DB2 *recursive query* in the varying *fanout* experiment over the expression $E3$, because in this data transformation, specifically, its temporary IO is the sum of an arithmetic progression, where in each recursive iteration the number of bytes written per tuple is reduced by the size of one favorite artist.

(ii) The *ntups* and *selectivity* parameters were virtually identical, having exactly the same influence in the performance of the various methods by determining the number of records needed to be processed. These two parameters affect the computational resource consumption in all dimensions linearly. The *fanout* parameter, while having a linear impact in the performance of these data transformations, namely increasing their processing cost and *elapsed time*, has a smaller influence in the IO cost of the queries since, in our benchmark conditions, it does not affect the number of input pages read, only the number of output pages written.

(iii) The *ntups* and *selectivity* parameters have greater impact than the *fanout* in the performance of these data transformations. This is a natural conclusion. For example, if we double the value of any of these parameters, in all three scenarios we are doubling the number of output records produced by the data transformation and the processing time of the data transformation. However, in the varying *ntups* and *selectivity* scenarios, we are also doubling the number of input pages that have to be processed. Therefore, a greater number of pages have to be accessed, while in the varying *fanout* experiments the number of input pages remain always the same. We are assuming that the cost of processing one input record is not affected significantly by the increase in *fanout* (e.g. exponentially), nor the size of the input records is affected by this parameter.

Our final goal was to compare the different RDBMS with one another. For this purpose, we used the information gathered from the previous steps: the experiments undertaken to study the influence of *critical parameters* also allowed us to compare the performance of the RDBMS with one another; and the computational resource

consumption profiles of the different implementation methods in each RDBMS allowed us, for the most part, to understand and explain the results disparity of the different databases when performing the same experiments (see Section 4.8). Again, the results were very clear. We concluded that:

(i) The *bounded* one-to-many data transformations of a type identical to the expression $E2$ (rotating columns) are best supported in Oracle and MSQLS by means of the *unpivot*. Both RDBMS presented identical performance.

(ii) The *unbounded* one-to-many data transformations (generically), are best supported in Oracle and DB2 by means of the *table function* and *recursive query*, respectively. However, the Oracle *table function* clearly outperformed the DB2 *recursive query* in the varying *fanout* experiments.

(iii) The Oracle *table function* is the only implementation method with sufficient expressive power to support the *unbounded* data transformations and with an adequate resource consumption profile: it reads the source table only once, does not use auxiliary memory, and is not forcefully logged.

(iv) The implementation methods that support exclusively *bounded* data transformations have better performance than any implementation method with sufficient expressive power to support the *unbounded*. Namely, the *SPU* was the best implementation method after the *unpivot* (in Oracle and MSQLS), and the *unnest* (in Oracle and DB2).

(v) From (iv) we also concluded that with the exception of the *unpivot* and the *unnest*, all the implementation methods are inefficient means of supporting *bounded* data transformations, since the *SPU* outperforms them, and it has an inefficient query execution plan which logically reads the source table *fanout* times.

5.2 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 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 (Carreira, 2007). 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.

Appendix I

A Alternative code implementations

This appendix contains all the code implementations, which are RDBMS specific, that are omitted from Chapter 3 Section 3.2. In the discussion of the different implementation mechanisms for the schema mapping expressions $E2$, $E3$ and $E4$ (see Section 3.1.3), we only presented one code implementation for each expression and implementation mechanism, leaving out equivalent code implementations for the remaining RDBMS systems that support the same implementation mechanism. Therefore, in this appendix the omitted code is presented. Thus, all the RDBMS and their supported implementation mechanisms are in the end fully covered.

A.1 Unpivot

Out of the three one-to-many schema mapping expressions, the *unpivot* can only be used to implement a solution for $E2$. The solution for $E2$ using the MSQLS *unpivot* was presented in Section 2.1.1.3 Code Listing 2. In Code Listing 13, we present the implementation solution for the same schema mapping expression $E2$ in Oracle.

Code Listing 13. Oracle implementation of the schema mapping expression $E2$ through the use of an unvpivot query. See Section 2.1.1.3 for detailed description of the solution.

```
01: SELECT USRID, FAVF
02: FROM (
03:       SELECT ID AS USRID, FAVF1, FAVF2, FAVF3, FAVF4
04:       FROM PROFILE) P
05: UNPIVOT EXCLUDE NULLS (FAVF FOR CNAME IN (FAVF1, FAVF2, FAVF3, FAVF4)) FAVFRIEND
```

A.2 Unnest

Out of the three one-to-many schema mapping expressions, the *unnest* can only be used to implement a solution for $E2$. The solution for $E2$ using the Oracle *unnest* was presented in Section 2.1.1.6 Code Listing 6. In Code Listing 14, we present the implementation solution for the same schema mapping expression $E2$ in DB2.

Code Listing 14. DB2 implementation of the schema mapping expression $E2$ through the use of an *unnest* query. The solution is identical to the one presented for the Oracle RDBMS. The only significant difference is that the DB2 syntax permits the application of the `TABLE` keyword over a list of values without encapsulating them into objects and arrays (see line 3), as in an *INSERT INTO TABLE* statement.

```
01: SELECT P.ID as USRID, Q.FAVF
02: FROM PROFILE P,
03:       TABLE(VALUES (P.FAVF1), (P.FAVF2), (P.FAVF3), (P.FAVF4), (P.FAVF5))
```

```
04:       AS Q(FAVF)
05: WHERE FAVF IS NOT NULL
```

A.3 Recursive query mechanism

Out of the three one-to-many schema mapping expressions, we use *recursive query* to implement solutions for $E3$ and $E4$. The solution for $E3$ using the MSQLS *recursive query* was presented in Section 2.1.1.4 Code Listing 3. In Code Listing 15, we present the implementation solution for the same schema mapping expression $E4$ in MSQLS.

Code Listing 15. MSQLS implementation of the schema mapping expression $E4$ through the use of a *recursive query*. The solution is identical to the used for the implementation of the expression $E3$, see Section 2.1.1.4 Code Listing 3. In this solution, the stop case for the recursive query is met when the current *recursive query* iteration surpasses the number of monthes a user has been registered, see line 46. We summarize the implementation solution as follows. The *achor* produces the output records referring to the first month of registration of each user. The *recursive member* does the same for all the other monthes that user has been registered. In each iteration, a specific month, year and view count are computed, see the *anchor* lines 19, 20, and 21 to 30; and the *recursive member* lines 39, 40, and 41 to 44. The solution has some sensible points that demanded careful attention relative to the handling of dates. Every time we used dates to compute date-differences in monthes (lines 11 to 14), or advance a date to the following month (lines 16 to 18), we took special care that each date was reseted to first day of its month. This was necessary because the following is true for most RDBMS. If we calculate a date difference between the last day of january and the first day of feburary, the result will not be equal to 1. However, for our intended purposes, the user has been registered for two monthes: January and February. Thus, we need to make sure that when we compute a date difference, we obtain the expected results.

```
01: WITH recursive_query(
02:   UID, NVIEWS, NMONTHS,
03:   NEXT_ITERATION, NEXT_DATE,
04:   YEAR, MONTH, VCOUNT
05: )
06: AS
07: (
---------------
--RECURSIVE STEP 1 (R1=PROJECT_CLAUSE)
---------------
08: SELECT
      --------------------
      -- COLUMN 1: ID-> UID
      --------------------
09:   ID AS UID,
      --------------------
      -- COLUMN 2: NVIEWS
      --------------------
10:   NVIEWS,
      --------------------
```

```
       -- COLUMN 3: NMONTHS (obtained as the difference in monthes between two dates with the day reset to 01)
       --------------------
11:    FLOOR(DATEDIFF(MONTH,
12:           DATEADD(DAY, -(DAY(REGDATE) - 1), REGDATE),
13:           DATEADD(DAY, -(DAY(GETDATE()) - 1), GETDATE())
14:    )) + 1 AS NMONTHS,
       --------------------
       -- COLUMN 4: NEXT_ITERATION
       --------------------
15:    2  AS NEXT_ITERATION,
       --------------------
       -- COLUMN 5: NEXT_DATE (addition of one month to the registration date, day in month is reseted to 01)
       --------------------
16:    DATEADD(MONTH, 1,
17:           DATEADD(DAY, -(DAY(REGDATE) - 1), REGDATE) --this expression = REGDATE, but with day reset to 01
18:    ) AS NEXT_DATE,
       --------------------
       -- COLUMN 6: YEAR
       --------------------
19:    YEAR(REGDATE) as YEAR,
       --------------------
       -- COLUMN 7: MONTH
       --------------------
20:    MONTH(REGDATE) as MONTH,
       --------------------
       -- COLUMN 8: VCOUNT [CASE WHEN 1 = NMONTHS; THEN NVIEWS; ELSE NVIEWS/NMONTHS]
       --------------------
21:    CASE WHEN 1 = (FLOOR(DATEDIFF(MONTH,
22:                                DATEADD(DAY, -(DAY(REGDATE) - 1), REGDATE),
23:                                DATEADD(DAY, -(DAY(GETDATE()) - 1), GETDATE())
24:                                )) + 1)
25:           THEN NVIEWS
26:    ELSE NVIEWS / (FLOOR(DATEDIFF(MONTH,
27:                                DATEADD(DAY, -(DAY(REGDATE) - 1), REGDATE),
28:                                DATEADD(DAY, -(DAY(GETDATE()) - 1), GETDATE())
29:                                )) + 1)
30:    END as VCOUNT
31: FROM PROFILE as anchor
32: UNION ALL
---------------
--RECURSIVE STEPS 2 to NMONTHS: definition of frecursive.
---------------
33: SELECT
34:    UID,
35:    NVIEWS,
36:    NMONTHS,
37:    previousTable.NEXT_ITERATION + 1,
38:    DATEADD(MONTH, 1, previousTable.NEXT_DATE),
39:    YEAR(previousTable.NEXT_DATE),
```

```
40:   MONTH(previousTable.NEXT_DATE),
41:   CASE WHEN previousTable.NEXT_ITERATION = NMONTHS
42:          THEN VCOUNT + (NVIEWS % NMONTHS)
43:          ELSE VCOUNT
44:   END
45: FROM recursive_query as previousTable
46: WHERE previousTable.NEXT_ITERATION <= NMONTHS
47: )
48: SELECT UID, YEAR, MONTH, VCOUNT
49: FROM recursive_query
```

A.4 Table function mechanism

We use *table functions* to implement solutions for the three one-to-many schema mapping expressions $E2$, $E3$ and $E4$. The solution for $E3$ using the Oracle *table function* was presented in Section 2.1.2.2 Code Listing 8. In the following Code Listings, we present the remaining Oracle and MSQLS solutions using the *table function* for the expression $E2$, $E3$ and $E4$.

Code Listing 16. Oracle implementation of the schema mapping expression $E2$ using the *table function* implementation method, see lines 2 to 20. The solution is trivial. Each favorite friend passed to the function by the several input attributes (see line 3) is directly written out to the output, by means of the Oracle `PIPE ROW` statement.

```
--1st: Define an object type that will be used as the row type for a nested table type
01: CREATE OR REPLACE TYPE t_favfriend_row AS OBJECT (FAVF int);


--2nd create a nested table type that will be use as return value by the table function
02: CREATE OR REPLACE TYPE t_favfriend_table AS TABLE OF  t_favfriend_row;

03: CREATE OR REPLACE FUNCTION exp2_totable(FAVF1 int, FAVF2 int, FAVF3 int, FAVF4 int)
04: RETURN t_favfriend_table PIPELINED
05: AS

06: BEGIN
07: IF FAVF1 IS NOT NULL
08:    THEN PIPE ROW (t_favfriend_row(FAVF1));
09: END IF;
10: IF FAVF2 IS NOT NULL
11:    THEN PIPE ROW (t_favfriend_row(FAVF2));
12: END IF;
13: IF FAVF3 IS NOT NULL
14:    THEN PIPE ROW (t_favfriend_row(FAVF3));
15: END IF;
16: IF FAVF4 IS NOT NULL
```

```
17:    THEN PIPE ROW (t_favfriend_row(FAVF4));
18: END IF;
19: RETURN ;
20: END;

--INVOCATION
21: SELECT ID as USRID, FAVF
22: FROM PROFILE, TABLE(exp2_totable(FAVF1, FAVF2, FAVF3, FAVF4))
```

Code Listing 17. MSQLS implementation of the schema mapping expression $E2$ using the *table function* implementation method, see lines 1 to 31. Again, the solution is trivial. Each favorite friend, once fetched, is directly written out to the output, by means of the MSQLS `INSERT INTO` statement. However, the fact that the input records have to be fetched inside the table function by means of a cursor (see the lines 11 to 15 and 26), instead of being passed at a higher level by the query invoking the *table function* (see lines 32 and 33), is a considerable disadvantage. It compromises the declartiveness of the solution. In essence, the full query is coded inside the *table function*.

```
01: CREATE FUNCTION exp2_totable ()
02: RETURNS @FAVFRIEND TABLE
03: (UID int, FAVF int)
04: AS
05: BEGIN
06:   DECLARE @uid int,
07:          @favf1 int,
08:          @favf2 int,
09:          @favf3 int,
10:          @favf4 int;

11:   DECLARE p_cursor CURSOR
12:   LOCAL FORWARD_ONLY READ_ONLY FAST_FORWARD
13:   FOR (SELECT ID, FAVF1, FAVF2, FAVF3, FAVF4 FROM PROFILE);

14:   OPEN p_cursor;
15:   FETCH NEXT FROM p_cursor INTO @uid, @favf1, @favf2, @favf3, @favf4;
16:   WHILE @@FETCH_STATUS = 0
17:   BEGIN
18:          IF @favf1 IS NOT NULL
19:                INSERT INTO @favfriend VALUES (@uid, @favf1);
20:          IF @favf2 IS NOT NULL
21:                INSERT INTO @favfriend VALUES (@uid, @favf2);
22:          IF @favf3 IS NOT NULL
23:                INSERT INTO @favfriend VALUES (@uid, @favf3);
24:          IF @favf4 IS NOT NULL
25:                INSERT INTO @favfriend VALUES (@uid, @favf4);
26:          FETCH NEXT FROM p_cursor INTO @uid, @favf1, @favf2, @favf3, @favf4;
27:   END
28:   CLOSE p_cursor;
29:   DEALLOCATE p_cursor;
```

```
30:   RETURN
31: END


-- INVOCATION
32: SELECT UID, FAVF
33: FROM exp2_totable()
```

Code Listing 18. MSQLS implementation of the schema mapping expression $E3$ using the *table function* implementation method, see lines 1 to 37. This solution is identical to the Oracle solution presented in Section 2.1.2.2 Code Listing 8, refer to it for a more detailed explanation of the code. Once more, the only significant difference between the Oracle and MSQLS implementations, lies in the fact that MSQLS has to fetch directly the input records that it needs to process. Otherwise, the function body from the lines 18 to 32 is identical to that of the Oracle implementation.

```
01: CREATE FUNCTION exp3_totable ()
02: RETURNS @FAVARTIST TABLE
03: (UID int, FAVARTIST varchar(400))
04: AS
05: BEGIN
06:   DECLARE @uid int,
07:         @list_artists varchar(401),
08:         @head_element varchar(400),
09:         @separator_pos int,
10:         @len int;

11:   DECLARE p_cursor CURSOR
12:         LOCAL FORWARD_ONLY READ_ONLY FAST_FORWARD
13:         FOR (SELECT ID, FAVARTISTS FROM PROFILE);
14:   OPEN p_cursor;
15:   FETCH NEXT FROM p_cursor INTO @uid, @list_artists;

16:   WHILE @@FETCH_STATUS = 0
17:   BEGIN
            -- 1st CHECK IF THE ARTIST LIST IS NULL
18:         IF ISNULL(@list_artists, '') = ''
19:               CONTINUE;

            -- 2nd APPEND , to the end of the string
20:         SET @list_artists = @list_artists + ',';

            -- 3rd 1st iteration happens outside the cycle: 'artist,' would break the while
21:         SET @len = LEN(@list_artists);
22:         SET @separator_pos = CHARINDEX(',', @list_artists);
23:         SET @head_element = RTRIM(LTRIM(SUBSTRING(@list_artists, 1, @separator_pos - 1)));
24:         INSERT INTO @favartist VALUES (@uid, @head_element);
```

```
            -- 4th while condition = true => there are still artists left to process
25:         WHILE @len > @separator_pos BEGIN
26:               SET @list_artists = SUBSTRING(@list_artists, @separator_pos + 1, @len - @separator_pos);
27:               SET @separator_pos = CHARINDEX(',', @list_artists);
28:               SET @len = LEN(@list_artists);
29:               SET @head_element = RTRIM(LTRIM(SUBSTRING(@list_artists, 1, @separator_pos - 1)));
30:               INSERT INTO @favartist VALUES (@uid, @head_element);
31:         END;
32:         FETCH NEXT FROM p_cursor INTO @uid, @list_artists;
33:   END

34:   CLOSE p_cursor;
35:   DEALLOCATE p_cursor;
36:   RETURN;
37: END


-- Invocation:
38: SELECT UID, FAVARTIST
39: FROM exp3_totable();
```

Code Listing 19. Oracle implementation of the schema mapping expression $E4$ using the *table function* implementation method, see lines 3 to 34. The function receives as input the date of registration and the total number of visits of a user's profile (line 3). The function body executes a processing cycle (lines 19 to 29), with as many iterations as the number of months, minus 1, passed since the date of the registration (line 24). In each iteration, the function returns one output record (line 27). Finally, the last iteration occurs outside the cycle (lines 30 to 32). This happens because the view count of the last output record is computed with a different formula. Each output record specifies one year and month of activity of the user, and the average number of views of the user's profile (lines 27 and 32).

```
--1st: Define an object type that will be used as the row type for a nested table type
01: CREATE OR REPLACE TYPE t_viewhistory_row AS OBJECT (YEAR int, MONTH int, VCOUNT int);

-- 2nd create a nested table type that will be use as return value by the table function
02: CREATE OR REPLACE TYPE t_viewhistory_table AS TABLE OF t_viewhistory_row;


03: CREATE OR REPLACE FUNCTION exp4_totable(REGDATE date, NVIEWS int)
04: RETURN t_viewhistory_table PIPELINED
05: AS
06: first_day_of_month date;
07: first_day_of_currdt date;
08: year_ int;
09: month_ int;
10: enddate date;
11: nmonths int;
12: lastvcount int;
13: stdvcount int;
14: BEGIN
```

```
15: enddate := CURRENT_DATE;
16: IF REGDATE - 3/86400 > enddate
17:   THEN RETURN ;
18: END IF;

19: first_day_of_month := REGDATE - (EXTRACT( DAY FROM REGDATE) - 1);
20: first_day_of_currdt := CURRENT_DATE - (EXTRACT( DAY FROM CURRENT_DATE) - 1);
21: nmonths := FLOOR(MONTHS_BETWEEN(first_day_of_currdt, first_day_of_month)) + 1;
22: stdvcount := FLOOR(nviews/nmonths);
23: lastvcount := stdvcount + mod(nviews,nmonths);
-- iterate from [1 to NMONTHS-1], all the tuples have VCOUNT= stvcount
24: FOR i IN 1 .. nmonths - 1 LOOP
25:   year_ := EXTRACT( YEAR FROM first_day_of_month);
26:   month_ := EXTRACT( MONTH FROM first_day_of_month);
27:   PIPE ROW (t_viewhistory_row(year_, month_, stdvcount));
28:   first_day_of_month := ADD_MONTHS(first_day_of_month, 1);
29: END LOOP;

30: year_ := EXTRACT( YEAR FROM first_day_of_month);
31: month_ := EXTRACT( MONTH FROM first_day_of_month);
32: PIPE ROW (t_viewhistory_row(year_, month_, lastvcount));
33: RETURN ;
34: END;



-- INVOCATION:
35: SELECT ID as USRID, YEAR, MONTH, VCOUNT
36: FROM PROFILE, TABLE(exp4_totable(REGDATE, NVIEWS)) T;
```

Code Listing 20. MSQLS implementation of the schema mapping expression $E4$ using the *table function* implementation method, see lines 1 to 45. The solution is identical to that of the Oracle database system for the same expression, see Code Listing 19.

```
01: CREATE FUNCTION exp4_totable ()
02: RETURNS @VIEWHISTORY TABLE
03: (UID int, YEAR int, MONTH int, VCOUNT int)
04: AS
05: BEGIN
06:   DECLARE @uid int,
07:         @regDate smalldatetime,
08          @nviews int,
09:         @first_day_of_month smalldatetime,
10:         @first_day_of_currdt smalldatetime,
11:         @year_ int,
12:         @month_ int,
13:         @nmonths int,
14:         @lastvcount int,
15:         @stdvcount int;

16:   DECLARE p_cursor CURSOR
```

```
17:          LOCAL FORWARD_ONLY READ_ONLY FAST_FORWARD
18:          FOR (SELECT ID, REGDATE, NVIEWS FROM PROFILE);
19:   OPEN p_cursor;
20:   FETCH NEXT FROM p_cursor INTO @uid, @regDate, @nviews;

21:   WHILE @@FETCH_STATUS = 0
22:   BEGIN
             -- 1st INITIALIzE VARIABLES: reset the DAY of the regdate and system date,etc.
23:          SET @first_day_of_month = DATEADD(DAY, -(DAY(@regdate) - 1), @regDate);
24:          SET @first_day_of_currdt = DATEADD(DAY, -(DAY(GETDATE()) - 1), GETDATE());
25:          SET @nmonths = FLOOR(DATEDIFF( MONTH , @first_day_of_month, @first_day_of_currdt)) + 1;
26:          SET @stdvcount = FLOOR(@nviews/@nmonths);
27:          SET @lastvcount = @stdvcount + (@nviews%@nmonths);

             -- 2nd iterate from [1 to NMONTHS-1], all the tuples have VCOUNT= stvcount
28:          DECLARE @counter int;
29:          SET @counter = 1;
30:          WHILE @counter < @nmonths BEGIN
31:                SET @year_ = YEAR(@first_day_of_month);
32:                SET @month_ = MONTH(@first_day_of_month);
33:                INSERT INTO @viewhistory VALUES (@uid, @year_, @month_, @stdvcount);
34:                SET @first_day_of_month = DATEADD(MONTH, 1, @first_day_of_month);
35:                SET @counter = @counter + 1;
36:          END;
             --3rd output the last record with last vcount number of views.
37:          SET @year_ = YEAR(@first_day_of_month);
38:          SET @month_ = MONTH(@first_day_of_month);
39:          INSERT INTO @viewhistory VALUES (@uid, @year_, @month_, @lastvcount);

40:          FETCH NEXT FROM p_cursor INTO @uid, @regDate, @nviews;
41:   END;
42:   CLOSE p_cursor;
43:   DEALLOCATE p_cursor;
44:   RETURN;
45: END


-- INVOCATION
46: SELECT UID, YEAR, MONTH, VCOUNT
47: FROM exp4_totable()
```

A.5 Model transformation mechanism

In Section 2.1.1.5, we discussed the *model* operator of Oracle in detail, which can be used to perform the three one-to-many data transformation expressions $E2$, $E3$ and $E4$. The implementation solutions for the first the schema mapping expressions ($E2$ and $E3$) were given in Section 2.1.1.5 Code Listing 4 and Code Listing 5. In this section, implementation of the schema mapping expression $E4$ in Code Listing 21 .

Code Listing 21. Oracle solution for the schema mapping expression $E4$ using the *model* implementation method. The solution carries various similarities with the two previous solutions for the $E2$ and $E3$ expressions, see Section 2.1.1.5 Code Listing 4 and Code Listing 5: the `PARTION BY` and `DIMENSION BY` clauses are the same, see lines 3 and 4; and the multiple *model* rules processes the attribute-values of the output records in a recursive manner, alike the solution of the $E3$ expression. The first rule (lines 15 to 18), initializes the `VCOUNT` measure of the first output record of each partition. The second rule (lines 19 to 23), recursively computes the `VCOUNT` of the remaining output records produced in each partition. The third rule (lines 24 to 25), recursively computes incrementing date values assigned to the `TMPDATE` measure of the output records of each partition. The fourth and fifth rules (lines 26 to 27 and 28 to 29), compute the `YEAR` and `MONTH` measures by extracting the year and month values of the corresponding dates in the `TMPDATE` measure (which were computed in the third rule). When all the rules are processed, the several arrays are converted back into an output relation with schema (`ID, FINDEX, TMPDATE, NMONTHS, VCOUNT, NVIEWS`). We only project the `ID`, `YEAR`, `MONTH` and `VCOUNT` attributes of this relation, see line 1.

```
01: SELECT  ID as USRID, YEAR, MONTH, VCOUNT
02: FROM PROFILE P
03: MODEL
03:    PARTITION BY    (ID)
04:    DIMENSION BY    (1 as FINDEX)
05:    MEASURES         (REGDATE - (EXTRACT(DAY FROM REGDATE) - 1) as TMPDATE,
06:                       EXTRACT(YEAR FROM REGDATE) as YEAR,
07:                       EXTRACT(MONTH FROM REGDATE) as MONTH,
08:                       FLOOR(MONTHS_BETWEEN(
09:                    CURRENT_DATE - (EXTRACT(DAY FROM CURRENT_DATE) - 1),
10:                    REGDATE - (EXTRACT(DAY FROM REGDATE) - 1)
11:                       )) + 1 as NMONTHS,
12:                       0 as VCOUNT,
13:                       NVIEWS as NVIEWS)
14:    RULES SEQUENTIAL ORDER(
            --Remark: When entering the rules clause, the 1st tuple of each partition
            --already has the attributes TMPDATE, YEAR, MONTH and NMONTHS with the appropriate
            --values. The VCOUNT attribute was set to 0 and has yet to be correctly set up

            --Rule1: Set The VCOUNT VALUE OF THE FIRST TUPLE IN EACH PARTITION
15:            VCOUNT[1] = CASE WHEN 1 = NMONTHS[1]
16:                             THEN NVIEWS[1]
17:                             ELSE FLOOR(NVIEWS[1]/NMONTHS[1])
18:                        END,

            --Rule2: SET THE VCOUNT VALUE OF EVERY OTHER OUTPUT TUPLE
19:            VCOUNT[FOR FINDEX FROM 2 TO NMONTHS[1] INCREMENT 1]
20:            = CASE  WHEN CV(FINDEX) = NMONTHS[1]
21:                          THEN VCOUNT[1] +  MOD(NVIEWS[1],NMONTHS[1])
22:                          ELSE VCOUNT[1]
23:                    END,
```

```
            --Rule3: SET THE TMPDATE OF EVERY OUTPUT TUPLE
24:             TMPDATE[FOR FINDEX FROM 2 TO NMONTHS[1] INCREMENT 1]
25:             = ADD_MONTHS(TMPDATE[CV(FINDEX)-1], 1),

            --RULE4: SET THE YEAR VALUE OF EVERY OUTPUT TUPLE
26:             YEAR[FOR FINDEX FROM 2 TO NMONTHS[1] INCREMENT 1]
27:             = EXTRACT(YEAR FROM TMPDATE[CV(FINDEX)]),

            --RULE5: SET THE MONTH VALUE OF EVERY OUTPUT TUPLE
28:             MONTH[FOR FINDEX FROM 2 TO NMONTHS[1] INCREMENT 1]
29:             = EXTRACT(MONTH FROM TMPDATE[CV(FINDEX)])
30:     );
```

B Installation and tests

This appendix explains more in depth some of the tuning decisions that were made during the configuration of the computer system and RDBMS that were described in Chapter 4 Section 4.1.

B.1 RDBMS-Specific Raw Partitions Architectures

In Section 4.1, we presented a general raw partitions architecture, to ensure that the same disk regions are being used by every RDBMS to access or save information when similar activities are performed by each of them. However, there are some significant differences from RDBMS to RDBMS that sometimes enforce us to deviate a little from this strict logical architecture. Essentially, two aspects are at the root of this problem: *(i)* a particular RDBMS has some unique feature that creates or handles data objects that do not fall into any of the data types we defined before (System, Log, Temp, Source and Target); *(ii)* the RDBMS may prefer to divided some of these dimensions into subsets. In the following sections we explain how the raw partitions architecture illustrated in Table 4.1 was adapted to the three commercial RDBMS under evaluation.

Nevertheless, the changes to the base raw partitions architecture that are presented in the following subsections are in practice not that significant. Similar database objects - through the use of more or less raw partitions - are all placed into similar physical locations.

B.1.1 Oracle Raw Partitions Architecture

In Oracle, we had to adapt the generic raw partitions architecture presented in Section 4.1 Table 4.1.

First, Oracle has a unique feature: it has a RollBack mechanism called *undo* that is used instead of the log data to revert changes in the database. In Oracle, the before imagines of a record are not only stored in the log datafile, but also in an Undo datafile. This datafile is used to enhance the speed of rollbacking a running transaction when the database is online. The undo information is also used as an active read consistency mechanism, enabling transactions to read the before-images of records when another transaction is in the process of updating the said records. We cannot qualify the data handled by the undo mechanism as making part of the database logging mechanism (they serve different purposes). Consequently, in Oracle, not only do we need a raw partition for log data - as we would expect in any RDBMS - but we also need a raw partition to store the undo data objects. So, our raw partitions architecture has to be adapted, by extending it with an additional Raw Partition (the Undo Raw Partition).

Second, Oracle also forces us to partition two of the raw partitions defined in our generic raw partitions

architecture: *(i)* the log raw partition; and *(ii)* the system raw partition . Oracle uses a minimum of two log datafiles, switching from one to the other when the active log file becomes full. Consequently, our Log Dimension has to be divided into at least two raw partitions. Furthermore, Oracle recommends its system objects to be separated into two distinct *tablespaces*: one called *System*, which is where the catalog along with other core objects are stored; and another called *SysAux*, where other secondary system information related to non essential database components are stored. To support this, we have to divide our System Raw partition into two smaller Raw Partitions, which can accommodate this particular architecture.

This raw partitions architecture is illustrated in Table B.1. We can see that the System raw partition, originally intended to be 4 GB large, was divided into two adjacent raw partitions -System and SysAux - with 2 GB size each. The Log raw partition, originally intended to be 32 GB large, was divided into two adjacent raw partitions - redo log 1 and redo log 2 - each 16 GB large. The unique Undo feature of the Oracle database is supported by a new raw partition called Undo that was 32 GB large, and placed after the two Log raw partitions.

Hard Disk Drive	Partition Name	File System Type	Partition Size
HDD0	Windows 2003	NTFS	88 GB
HDD0	Target	Raw	32 GB
HDD0	Redo Log 1	Raw	16 GB
HDD0	Redo Log 2	Raw	16 GB
HDD0	Undo	Raw	32 GB
HDD1	System	Raw	2 GB
HDD1	SysAux	Raw	2 GB
HDD1	Source	Raw	32 GB
HDD1	Temporary	Raw	32 GB

Table B.1: Locations and sizes for the Raw Partitions serving as database datafiles in Oracle.

B.1.2 MSQLS Raw Partitions Architecture

In MSQLS, we had to adapt the generic raw partitions architecture presented in Section 4.1 Table 4.1. This RDBMS has two types of databases: built-in system databases, and user defined databases. The built-in system databases are created automatically upon an installation of an SQL Server instance. The built-in system databases are: *(i)* the master; *(ii)* the model; *(iii)* and the tempdb. The master database holds the catalog information of all other databases supported by the MSQLS instance. The model database contains templates that are used when new databases are created (and other such operations). The tempdb is always created anew when the server instance starts, and destroyed when it shutdowns. This database is used to store temporary information, such as the materialization of data during a query execution. Since we have no control over the creation of these databases, they are always installed in file system where the MSQLS instance software is installed. In our case, that was the disk volume C:\. User databases are created by the user, and are meant to hold the user's information. Consequently, our SQLServer instance supported 4 databases. The first three were the built-in system databases, and the fourth was a user defined database with one datafile for the source data, another for the target data and third for the logging information.

The user defined database was created according to our generic raw partitions architecture. That is, the data files for source data, target and log data where created in the source, target and log raw partitions. However, it was impossible to place the data files of the built-in databases in the desirable raw partitions. The system raw partition was not used, we could not to move the file system data files of the master and model databases into this

raw partition. This is a fact of little consequence, since the IO over data in the system databases is practically negligible, or inexistent, when compared to all the IO performed in the data files of the user defined database. Since the tempdb is always recreated on restart, by changing the catalog information concerning the location of datafiles of this database, we managed to move the datafiles into the proper raw partitions. The temporary data was placed in the temporary raw partition, and the temporary log was placed in an additional raw partition that did no exist in the generic raw partitions architecture. This was unavoidable, since in MSQLS each database has at least two datafiles. One for holding the work data, the other for log data.

This raw partitions architecture is illustrated in Table B.2. In regard to the system data, belonging to the master and model databases, we can see that it was placed in the HDD0 in the Windows 2003 partition. The 4GB raw partition that should have held this information in HDD1 continued to exist but was empty. The source, target and log data were placed in their adequate 32 GB raw partitions, which were named Source, Target and User Database Log, respectively. Finally, the tempdb database log file was written to the raw partition named Temp Log . In Oracle, this exact same raw partition was used to contain the undo data. Overall, we believe to have kept an adequate equilibrium between the possible raw partitions architecture in this RDBMS, and the desired generic raw partition architecture described Section 4.1 Table 4.1. We were able to satisfy the generic architecture for the source, target and temp data.

Hard Disk Drive	Partition Name	File System Type	Partition Size
HDD0	Windows 2003, System	NTFS	88 GB
HDD0	Target	Raw	32 GB
HDD0	User Database Log	Raw	32 GB
HDD0	Tempdb Log	Raw	32 GB
HDD1	Empty	Raw	4 GB
HDD1	Source	Raw	32 GB
HDD1	Temporary	Raw	32 GB

Table B.2: Locations and sizes for the Raw Partitions serving as database datafiles in MSQLS.

B.1.3 DB2 Raw Partitions Architecture

The DB2 raw partitions respected the generic raw partitions architecture presented in Section 4.1 Table 4.1. The catalogue and system information, the temporary data, the source data and the target data, were stored in four distinct table spaces, which administered the system, temporary, source and target raw partitions, respectively. The log data could not be placed in any raw partition, because when a DB2 database is created a set file-system log files are also automatically created with it. This architecture is illustrated in Table B.3. Of notice is the fact that the temporary raw partition in this architecture is 38 GB in size, larger than for the other two database system. This was done in order to accommodate all the write IO performed by DB2 *recursive queries* when the source table held 128M records.

Hard Disk Drive	Partition Name	File System Type	Partition Size
HDD0	Windows 2003, Log	NTFS	88 GB
HDD0	Target	Raw	32 GB
HDD1	System	Raw	4 GB
HDD1	Source	Raw	32 GB
HDD1	Temporary	Raw	38 GB

Table B.3: Locations and sizes for the Raw Partitions serving as database datafiles in DB2.

B.2 Gathering RDBMS specific statistics

This section presents the code used to take snapshots of the statistics described in Section 4.2 in the different RDBMS. When a query was run snapshots were taking before and after the query execution, so that the snapshot difference could later be used to interpret the effect of the query in the various computational resources used by the RDBMS. We only present the code used to snapshot statistics before a query was run, since the code used to snapshot the same statistics after query was in essence the same.

B.2.1 Gathering Wait Event statistics

The Oracle wait events are made accessible by the `V$SYSTEM_EVENT` system view. In Code Listing 22, we present a small routine for taking a snapshot of these statistics.

Code Listing 22. First, in the code lines 1 to 13, we start by checking the existence of the table where we want to save our wait events snapshot. If the table already exists, we drop it. Second, in the code lines 14 to 18, we run a query that saves the wait event statistics, accessible from the Oracle system view `V$SYSTEM_EVENT` , into our snapshot output table.

```
1: DECLARE
2: tb_count NUMBER;
3: BEGIN
4:  SELECT COUNT(TABLE_NAME)
5:  INTO tb_count
6:  FROM USER_TABLES
7:  WHERE TABLE_NAME = 'MY_WAIT_EVENTS_STATS_START';

8:  IF tb_count > 0
9:  THEN
10:    EXECUTE IMMEDIATE 'DROP TABLE MY_WAIT_EVENTS_STATS_START PURGE';
11:  END IF;
12: END;
13: /

14: CREATE TABLE MY_WAIT_EVENTS_STATS_START
15: TABLESPACE TARGET NOLOGGING
16: AS
17: SELECT vse.EVENT, vse.TIME_WAITED, EVENT_ID, WAIT_CLASS
18: FROM V$SYSTEM_EVENT vse;

19: ALTER TABLE MY_WAIT_EVENTS_STATS_START
20: ADD CONSTRAINT pk_waitevents_start PRIMARY KEY (EVENT_ID);
```

The MSQLS wait events are made accessible by the `sys.dm_os_wait_stats` system view. In Code Listing 23, we present a small routine for taking a snapshot of these statistics.

Code Listing 23. First, in the code lines 1 to 4, we check the existence of the table where we want to save our wait events snapshot. If the table already exists, we drop it. Second, in the code lines 6 to 11, we copy the contents of the dynamic system view sys.dm_os_wait_stats to a new snapshot table.

```
01: BEGIN
02:     if exists(select 1 WHERE object_id ('MY_WAIT_EVENTS_STATS_START') IS NOT NULL )
03:             DROP TABLE MY_WAIT_EVENTS_STATS_START;
04: END;
05: GO
06: SELECT IDENTITY(int,1,1) AS ROW_ID, WAIT_TYPE, WAIT_TIME_MS
07: INTO MY_WAIT_EVENTS_STATS_START
08: FROM sys.dm_os_wait_stats
09: ORDER BY WAIT_TYPE;
10: GO
11: ALTER TABLE MY_WAIT_EVENTS_STATS_START ADD CONSTRAINT pk_waitevents_start PRIMARY KEY (ROW_ID)
```

The DB2 wait events are made accessible by the mon_get_connection table function. In Code Listing 24, we present a small routine for taking a snapshot of these statistics. Note: The DB2 catalogue views offer very poor statistical information. Instead, DB2 provides a set of table functions which detail the statistical information monitored by DB2 agents. This statistical information is lost once a session disconnects from the database. Thus, statistics have to be collected in the same session that runs the query.

Code Listing 24. This DB2 command line script uses the statement termination character @, to mark the ending of each SQL statement. In line 1, we drop the target snapshot table, to ensure that the table onto which we want to save our wait events snapshot does not exist. In line 2 to 3, we create the target snapshot table. In the code lines 4 to 31, we query the DB2 monitoring table function mon_get_connection , and save all the wait event statistics that it monitors into our snapshot table. Line 30, we selects the statistical information of the current session, only. It uses the scalar function myapphandle , which retrieves the unique identifier of the present connection. A better understanding of this query requires the user to read the DB2 hierarchical wait time model.

```
01: DROP TABLE MY_WAIT_EVENTS_STATS_START@
02: CREATE TABLE MY_WAIT_EVENTS_STATS_START ( ID int, WAIT_TYPE varchar(30), WAIT_TIME_SEC double)
03: IN USERSPACE1 NOT LOGGED INITIALLY@

04: INSERT INTO MY_WAIT_EVENTS_STATS_START (ID, WAIT_TYPE, WAIT_TIME_SEC)
05: SELECT Q.ID, Q.WAIT_TYPE, CAST(Q.WAIT_TIME_SEC as double) / 1000 as WAIT_TIME_SEC
06: FROM table(mon_get_connection(null,-1)) x,
07:     TABLE(VALUES
08:             (1, 'total_compile_proc_time', x.total_compile_proc_time),
09:             (2, 'total_section_proc_time', x.total_section_proc_time),
10:             (3, 'total_commit_proc_time', x.total_commit_proc_time),
11:             (4, 'total_rollback_proc_time', x.total_rollback_proc_time),
12:             (5, 'total_runstats_proc_time', x.total_runstats_proc_time),
13:             (6, 'total_reorg_proc_time', x.total_reorg_proc_time),
14:             (7, 'agent_wait_time', x.agent_wait_time),
15:             (8, 'wlm_queue_time_total', x.wlm_queue_time_total),
```

```
16:              (9, 'lock_wait_time', x.lock_wait_time),
17:              (10, 'log_buffer_wait_time', x.log_buffer_wait_time),
18:              (11, 'log_disk_wait_time', x.log_disk_wait_time),
19:              (12, 'pool_read_time', x.pool_read_time),
20:              (13, 'pool_write_time', x.pool_write_time),
21:              (14, 'direct_read_time', x.direct_read_time),
22:              (15, 'direct_write_time', x.direct_write_time),
23:              (16, 'tcpip_recv_wait_time', x.tcpip_recv_wait_time),
24:              (17, 'tcpip_send_wait_time', x.tcpip_send_wait_time),
25:              (18, 'fcm_recv_wait_time', x.tcpip_recv_wait_time),
26:              (19, 'fcm_send_wait_time', x.tcpip_send_wait_time),
27:              (20, 'ipc_recv_wait_time', x.tcpip_recv_wait_time),
28:              (21, 'ipc_send_wait_time', x.tcpip_send_wait_time))
29:     AS Q(ID, WAIT_TYPE, WAIT_TIME_SEC)
30: WHERE x.application_handle =  myapphandle()@
31: commit@
```

B.2.2 Gathering Datafile Statistics

The Oracle datafile statistics are made accessible by multiple system views. First, the regular datafile statistics can be obtained from the V$FILESTAT and V$DATAFILE views. Second, the temporary datafile statistics can be obtained from the V$TEMPSTAT and V$TEMPFILE views. Lastly, the log datafile statistics can be obtained from the V$SYSSTAT view. In Code Listing 25, we present a small routine for snapshot taking these statistics.

Code Listing 25. In this code listing we illustrate how we can snapshot Oracle data file statistics. To do so, we start by checking the existence of the table where we want to save our wait events snapshot (lines 1 to 13). If the table already exists, we drop it. Afterwards, we run a query that saves the data file statistics, accessible from the Oracle system views V$FILESTAT (lines 14 to 21), V$TEMPSTAT (lines 23 to 28) and V$SYSSTAT (lines 30 to 38), into our snapshot output table.

```
01: DECLARE
02: tb_count NUMBER;
03: BEGIN
04:  SELECT COUNT(TABLE_NAME)
05:  INTO tb_count
06:  FROM USER_TABLES
07:  WHERE TABLE_NAME = 'MY_DATAFILE_STATS_START';

08:  IF tb_count > 0
09:  THEN
10:    EXECUTE IMMEDIATE 'DROP TABLE MY_DATAFILE_STATS_START PURGE';
11:  END IF;
12: END;
13: /

14: CREATE TABLE MY_DATAFILE_STATS_START
```

```
15: TABLESPACE TARGET NOLOGGING
16: AS
--DATAFILE STATS:
17: SELECT
18:      vfs.FILE#, vfs.PHYRDS, vfs.PHYWRTS,
19:      vfs.PHYBLKRD, vfs.PHYBLKWRT, vfs.READTIM,
20:      vfs.WRITETIM, df.BLOCK_SIZE, df.NAME
21: FROM V$FILESTAT vfs INNER JOIN v$datafile df ON vfs.FILE# = df.FILE#

22: UNION ALL

--TEMP FILE STATS:
23: SELECT
24:      maxID.MAX_ID + 1 AS FILE#, vts.PHYRDS, vts.PHYWRTS,
25:      vts.PHYBLKRD, vts.PHYBLKWRT, vts.READTIM,
26:      vts.WRITETIM, tf.BLOCK_SIZE, tf.NAME
27: FROM (SELECT MAX(FILE#) AS MAX_ID FROM V$FILESTAT) maxID,
28:  v$TEMPSTAT vts INNER JOIN v$TEMPFILE tf ON vts.FILE# = tf.FILE#

29: UNION ALL

--REDO LOG FILE(S) STATS:
30: SELECT
31:      maxID.MAX_ID + 2 AS FILE#, 0 AS PHYRDS, tb1.VALUE AS PHYWRTS,
32:      0 AS PHYBLKRD, tb2.VALUE AS PHYBLKWRT, 0 AS READTIM,
33:      0 AS WRITETIM, tb3.OS_BLOCK_SIZE AS BLOCK_SIZE, 'REDO_LOG' AS NAME
34: FROM
35:   (SELECT MAX(FILE#) AS MAX_ID FROM V$FILESTAT) maxID,
36:   (SELECT VALUE  FROM V$SYSSTAT  WHERE  CLASS=2 AND NAME = 'redo writes') tb1,
37:   (SELECT VALUE  FROM V$SYSSTAT  WHERE  CLASS=2 AND NAME = 'redo blocks written') tb2,
38:   (SELECT 512 AS OS_BLOCK_SIZE FROM dual) tb3;


39: ALTER TABLE MY_DATAFILE_STATS_START
40: ADD CONSTRAINT pk_datafiles_start PRIMARY KEY (FILE#);
```

The MSQLS datafile statistics are made accessible by the sys.databases , sys.master _files and ::fn _virtualfilestats dynamic system views. In Code Listing 26, we present a small routine for snapshot taking these statistics.

Code Listing 26. In this code listing we illustrate how we can snapshot MSQLS data file statistics. To do so, we start by checking the existence of the table where we want to save our wait events snapshot (lines 1 to 4). If the table already exists, we drop it. Afterwards, we run a query that saves the data file statistics, accessible from the MSQLS system views sys.databases , sys.master _files and ::fn _virtualfilestats (lines 6 to 17), into our snapshot output table.

```
01: BEGIN
```

```
02:   if exists(select 1 WHERE object_id ('MY_DATAFILE_STATS_START') IS NOT NULL )
03:         DROP TABLE MY_DATAFILE_STATS_START;
04: END;
05: GO
06: SELECT
07:   IDENTITY(int,1,1) AS ROW_ID,
08:   TB_DBS.name DB_NAME, SUSER_SNAME(TB_DBS.owner_sid) DB_OWNER,                    -- databases info
09:   TB_MF.name FLOGICAL_NAME,  TB_MF.physical_name FPHYSICAL_NAME, TB_MF.Type FTYPE,    -- datafiles info
10:   TB_VFN.NumberReads, TB_VFN.BytesRead, TB_VFN.NumberWrites, TB_VFN.BytesWritten,     -- datafiles stats
12: INTO MY_DATAFILE_STATS_START
13: FROM
14:   sys.databases TB_DBS
15:   INNER JOIN ::fn_virtualfilestats(-1, -1) TB_VFN ON TB_DBS.database_id=TB_VFN.dbid
16:   INNER JOIN sys.master_files TB_MF ON TB_VFN.fileid = TB_MF.file_id AND TB_VFN.dbid = TB_MF.database_id
17: ORDER BY DB_NAME, FLOGICAL_NAME;

18: GO
19: ALTER TABLE MY_DATAFILE_STATS_START  ADD CONSTRAINT pk_MY_DATAFILE_STATS_START  PRIMARY KEY (ROW_ID);
```

The DB2 datafile statistics are made accessible by the SYSIBMADM.SNAPTBSP and SYSIBMADM.SNAPDB dynamic system views. The first containing the tablespace and the latter the log files statistics. In Code Listing 27, we present a small routine for snapshot taking these statistics.

Code Listing 27. In this code listing we illustrate how we can snapshot DB2 data file statistics. We start by the table where we want to save our wait events snapshot (line 1). Afterwards, create the target snapshot table lines 2 to 4. The ID column of this table is automatically generated in each insert (line 3). Finally, we run a query that saves the data file statistics, accessible from the DB2 system views SYSIBMADM.SNAPTBSP and SYSIBMADM.SNAPDB (lines 5 to 15), into our snapshot output table. Note: While in Oracle and MSQLS the physical reads and physical writes counters measure the number of IO operations performed, in DB2 these two statistics measure the number of pages read from and written to disk, respectively.

```
01: DROP TABLE MY_DATAFILE_STATS_START@
02: CREATE TABLE MY_DATAFILE_STATS_START (
03:   ID int GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),
04:   TBSP_NAME  VARCHAR(20), TBSP_PAGE_SIZE int, PREADS int, PWRITES int) IN USERSPACE1 NOT LOGGED INITIALLY@

05: INSERT INTO MY_DATAFILE_STATS_START (TBSP_NAME, TBSP_PAGE_SIZE, PREADS, PWRITES)
06: SELECT
07:   TBSP_NAME, TBSP_PAGE_SIZE,
08:   POOL_DATA_P_READS + POOL_TEMP_DATA_P_READS + POOL_TEMP_INDEX_P_READS
      + POOL_INDEX_P_READS + DIRECT_READS AS PREADS,
09:   POOL_DATA_WRITES + POOL_INDEX_WRITES + DIRECT_WRITES AS PWRITES
10: FROM SYSIBMADM.SNAPTBSP
11: UNION ALL
12: SELECT 'LOG' AS TBSP_NAME, 4096 AS TBSP_PAGE_SIZE, LOG_READS AS PREADS, LOG_WRITES AS PWRITES
13: FROM SYSIBMADM.SNAPDB
14: ORDER BY TBSP_NAME@
```

```
15: COMMIT@
```

B.2.3 Gathering other Relevant Statistics

Having retrieved both the data files statistics and the system wait events, there are still some other relevant statistics that should be gathered, namely the CPU time used, date of the snapshot and size of the database buffer[1]. In Code Listing 28, we present a small routine for taking a snapshot of these statistics in Oracle, and in Code Listing 29 and Code Listing 30 we to the same for MSQLS and DB2, respectively.

Code Listing 28. In this code listing we illustrate how we can snapshot the CPU time used, date of the snapshot, size of the database buffer, cache lookups and cache hits in Oracle. First, we check the existence of the table where we want to save our wait events snapshot (lines 1 to 13). If the table already exists, we drop it. Afterwards, we query the Oracle system view V$SYS_TIME_MODEL (lines 21 to 22) and V$SYSSTAT (lines 23 to 24), to gather the CPU time statistics and number of Log Switches into our snapshot output table. In lines 25 to 35, we query multiple buffer statistics, namely cache hits, cache lookups and the buffer size.

In order to understand the query for the CACHE_HITS and CACHE_LOOKUPS attributes (lines 26 and 27), it is important to first explain that, in Oracle: *(i)* the number of cache misses is considered to be the number of physical blocks read; *(ii) logicalIO* is the term used to describe the cache lookups. Hence, the official Oracle formula for calculating the cache hit ratio is: $CHR = (logicalIO - physicalReads)/logicalIO$. Of course, the difference $logicalIO - physicalReads$ yields the number of cache hits. Finally, the *logical io* is the sum of two statistic variables: *db consistent gets* and *db block gets*. They are both database block requests made to the buffer manager (with slight differences in terms of view consistency properties of the data).

```
01: DECLARE
02: tb_count NUMBER;
03: BEGIN
04:  SELECT COUNT(TABLE_NAME)
05:  INTO tb_count
06:  FROM USER_TABLES
07:  WHERE TABLE_NAME = 'MY_OTHER_STATS_START';

08:  IF tb_count > 0
09:  THEN
10:    EXECUTE IMMEDIATE 'DROP TABLE MY_OTHER_STATS_START PURGE';
11:  END IF;
12: END;
13: /

14: CREATE TABLE MY_OTHER_STATS_START
15: TABLESPACE TARGET NOLOGGING
16: AS
```

[1]Fetching the database buffer size only makes sense if the system was configured to a fixed buffer size. Normally, RDBMS use by default varying buffer sizes according to the workload.

```
17: SELECT
18:   tb1.CPU_TIME_SEC, sysdate as SNAPSHOT_TIME, tb2.LOG_SWITCHES,
19:   tb3.CACHE_HITS, tb3.CACHE_LOOKUPS, tb3.CACHE_BUFF_SIZEB
20: FROM
  -- CPU TIME
21:  (SELECT value/1000000 AS CPU_TIME_SEC FROM v$sys_time_model
22:  WHERE stat_name like '%CPU%') tb1,
  -- LOG SWITCHES (ORACLE DOES NOT HAVE LOG TRUNCATIONS.IT SWITCHES LOG FILE)
23:  (SELECT sstat.VALUE AS LOG_SWITCHES FROM V$SYSSTAT sstat
24:  WHERE CLASS = 2 AND NAME = 'redo log space requests') tb2,
  -- BUFFER STATS:
25:  (SELECT
26:   (P1.value + P2.value - P3.value) AS CACHE_HITS,
27:   (P1.value + P2.value) AS CACHE_LOOKUPS,
28:   P4.value AS CACHE_BUFF_SIZEB
29:   FROM   v$sysstat P1, v$sysstat P2, v$sysstat P3, V$PARAMETER P4
30:   WHERE
31:    P1.name = 'db block gets'   AND
32:    P2.name = 'consistent gets' AND
33:    P3.name = 'physical reads' AND
34:    P4.name = 'db_cache_size'
35:    ) tb3;
```

Code Listing 29. In this code listing we illustrate how we can snapshot the CPU time used, date of the snapshot, size of the database buffer, cache lookups and cache hits in MSQLS. First, we check the existence of the table where we want to save our wait events snapshot (lines 1 to 13). If the table already exists, we drop it. Afterwards, we obtain the MSQLS CPU time and current date directly using built-in system functions and specialised tokens (e.g. @@CPU_BUSY), se lines 7 and 8. Lines 13 to 25, join multiple single record nested queries. The first of these (lines 14 to 15) retrieves the number of times the log file was truncated. The second (lines 16 to 17), retrieves the buffer size (we configured the minimum buffer size to be equal to the maximum buffer size). The third (lines 18 to 25), calculates the cache hit ratio. MSQLS, does not provide any views to retrieve neither the cache hits nor the cache looks. The only option we have is to use the dynamic system view `sys.dm_os_performance_counters` to calculate directly the cache hit ratio. All the information is saved into the snapshot output table (lines 6 to 12).

```
01: BEGIN
02:   if exists(select 1 WHERE object_id ('MY_OTHER_STATS_START') IS NOT NULL )
03:         DROP TABLE MY_OTHER_STATS_START;
04: END;
05: GO
06: SELECT
07:   @@Cpu_Busy * CAST(@@TIMETICKS AS FLOAT) /1000000 AS CPU_TIME_SEC,
08:   GETDATE() AS SNAPSHOT_TIME,
09:   tb1.LOG_TRUNCATIONS,
10:   tb2.MIN_MEM  AS CACHE_SIZE_MB,            -- MAX MEMORY SET EQUAL TO MIN MEMORY
11:   tb3.CACHE_HIT_RATIO
12: INTO MY_OTHER_STATS_START
```

```
13: FROM
14:    (SELECT cntr_value AS LOG_TRUNCATIONS FROM sys.dm_os_performance_counters
15:    WHERE instance_name = '_Total' AND counter_name = 'Log Truncations') tb1,
       -----
       -- BUFF MANAGER STATS
       -----
16:    (SELECT value AS MIN_MEM FROM sys.configurations
17:    WHERE configuration_id = 1543 AND name = 'min server memory (MB)') tb2,
18:    (SELECT
19:           CAST(A.NUMERATOR AS NUMERIC)/CAST(B.DENOMINATOR AS NUMERIC) AS CACHE_HIT_RATIO
20:    FROM
21:           (SELECT cntr_value AS NUMERATOR FROM sys.dm_os_performance_counters
22:           WHERE object_name = 'SQLServer:Buffer Manager' AND counter_name = 'Buffer cache hit ratio') A,
23:           (SELECT cntr_value AS DENOMINATOR FROM sys.dm_os_performance_counters
24:           WHERE object_name = 'SQLServer:Buffer Manager' AND counter_name = 'Buffer cache hit ratio base') B
25:    ) tb3
```

Code Listing 30. In this code listing we illustrate how we can snapshot the CPU time used, date of the snapshot, size of the database buffer and cache hit ratio in DB2. In line 1, we drop the table onto which we will save these statistics. In the lines 2 and 3, we create the empty snapshot table. In the lines 4 to 16, we take a snapshot of these statistics and save them into the snapshot table. The CPU_TIME_SEC, lines 9 to 11, is taken from the mon_get_connection table function, and corresponds to the sum of all the different types of processing times monitored by the DB2 agents. The total *wait time* is also gathered from this table function (line 8). The buffer size is taken in line 14, and it is inferred from the number of pages of the DB2 default buffer pool IBMDEFAULTBP. When a tablespace is not assigned any specific buffer pool, its data pages are cached in the default buffer pool. DB2 automatically calculates the cache hit ratio statistic for each buffer pool (lines 15 and 16).

```
01: DROP TABLE MY_OTHER_STATS_START@
02: CREATE TABLE MY_OTHER_STATS_START ( CTIME timestamp, CPU_TIME_SEC  double, WAIT_TIME_SEC double,
03:    CACHE_SIZE_BYTES int, CACHE_HIT_RATIO double) IN USERSPACE1 NOT LOGGED INITIALLY@

04: INSERT INTO MY_OTHER_STATS_START
05: SELECT CURRENT TIMESTAMP as CTIME, CPU_TIME_SEC, WAIT_TIME_SEC, CACHE_SIZE_BYTES, CACHE_HIT_RATIO
06: FROM
       ----------------
       -- COLLECT CPU TIME and WAIT TIME
       ----------------
07:    (SELECT
08:           CAST(total_wait_time as double) / 1000 as WAIT_TIME_SEC,
09:           CAST( total_compile_proc_time  + total_section_proc_time  +  total_commit_proc_time  +
10:                  total_rollback_proc_time  +  total_runstats_proc_time  + total_reorg_proc_time  +
11:                  total_load_proc_time  as double) / 1000 AS CPU_TIME_SEC
12:    FROM table(mon_get_connection(null,-1))
13:    WHERE application_handle =  myapphandle()),
       ----------------
       -- COLLECT CACHE SIZE and HIT RATIO
```

```
      ----------------
14:   ( SELECT NPAGES * PAGESIZE AS CACHE_SIZE_BYTES FROM SYSCAT.BUFFERPOOLS WHERE BPNAME = 'IBMDEFAULTBP' ),
15:   (SELECT CAST(TOTAL_HIT_RATIO_PERCENT as double) AS CACHE_HIT_RATIO FROM SYSIBMADM.BP_HITRATIO
16:   WHERE BP_NAME = 'IBMDEFAULTBP')@
17: commit@
```

B.3 Extracting the statistics from the snapshots

In this section we present, for the different RDBMS, the set of queries that allowed us to retrieve, from the snapshots we took, the statistics we discussed in Chapter 4 Section 4.2. With the exception of the wait event statistics, these queries convert every statistic we are interested in into a set of RDBMS independent pairs: $\langle$STAT_NAME , STAT_VALUE $\rangle$, where STAT_NAME is an RDBMS independent statistic name, and STAT_VALUE is the *double* value of the statistic.

The wait event statistics cannot be normalized, each RDBMS has as a set of very distinct wait event types. The way they are triggered also depends on the RDBMS. Some RDBMS mainly consider the wait events associated to bottlenecks in the execution of foreground threads (e.g. Oracle); others mix the wait events of foreground threads with background threads and make it nearly impossible to assess the wait time that is specific to the threads executing a query. Nevertheless, wait events are an important tool to comprehend the resources constricting the smooth execution of the tasks supported by an RDBMS. In general, we are only interested in considering the wait events related to the IO of data pages.

B.3.1 Extracting the wait events statistics from the snapshots

The wait event statistics were saved in two snapshot tables: MY_WAIT_EVENTS_STATS_END and MY_WAIT_EVENTS_STATS_START. The wait time value of each wait event was normalized into seconds, but the wait event names were maintained as they existed in the RDBMS. The wait event statistics we gathered corresponds to the snapshot difference of these two correlated snapshots. In Code Listing 31, we present a small routine for gathering the wait events statistics in Oracle. These statistics are obtained in an identical manner in MSQLS and DB2, therefore we omit the respective code used.

Code Listing 31. This code listing illustrates how we computed the snapshot difference of the tables MY_WAIT_EVENTS_STATS_END and MY_WAIT_EVENTS_STATS_START in Oracle. The Oracle Wait Event statistics are time-valued in hundreds of the second, thus we had to divide these values by 100 in order to convert them to seconds (line 3). *Idle* wait events, traducing the time spend by threads while they were not busy, were not considered (line 9).

```
--------
-- COMPUTES THE WAIT EVENTS TIME IN SECONDS FROM THE TWO SNAPSHOTS TAKEN
--------
01: SELECT
02:  waitStatsE.EVENT AS WAIT_TYPE,
```

```
03:  (waitStatsE.TIME_WAITED - waitStatsS.TIME_WAITED) / 100 AS WAIT_TIME_SEC
04: FROM
05:  MY_WAIT_EVENTS_STATS_END waitStatsE
06:  INNER JOIN MY_WAIT_EVENTS_STATS_START waitStatsS
07:  ON waitStatsE.EVENT_ID = waitStatsS.EVENT_ID
08: WHERE
09:  waitStatsE.WAIT_CLASS != 'Idle' AND
10:  waitStatsE.TIME_WAITED - waitStatsS.TIME_WAITED > 0
11: ORDER BY WAIT_TIME_SEC DESC
```

B.3.2 Extracting the system wide statistics from the snapshots

The System Wide (SW) statistics are inferred from all the snapshots tables. The SWIDE_WAIT_TIME_SECONDS is inferred from the snapshot tables MY_WAIT_EVENTS_STATS_END and MY_WAIT_EVENTS_STATS_START. The SWIDE_CPU_TIME_SECONDS is inferred from the snapshot tables MY_OTHER_STATS_END and MY_OTHER_STATS_START. The statistics SWIDE_TOTAL_PHYR, SWIDE_TOTAL_PHYR_IN_BYTES, SWIDE_TOTAL_PHYW and SWIDE_TOTAL_PHYW_IN_BYTES are inferred from the snapshot tables MY_DATAFILE_STATS_END and MY_DATAFILE_STATS_START. In Code Listing 32, we present a routine for gathering these statistics in Oracle. These statistics are obtained in an identical manner in MSQLS and DB2, therefore we omit the respective code used.

Code Listing 32. This code listing illustrates how we can query the various snapshot tables to obtain the system wide statistics described in Chapter 4 Section 4.2, in Oracle. The code lines 1 to 20 define two distinct tables: the table DATAFSTATS, which yields the snapshot difference of the gathered data-files statistics; and the table OTHERSTATS, which yields the difference of other important statistics gathered (e.g. CPU time). These two tables are referred in the lines 21 to 29 to produce the statistics: *SWIDE_CPU_TIME_SECONDS*, *SWIDE_TOTAL_PHYR_IN_BYTES*, *SWIDE_TOTAL_PHYW_IN_BYTES*, *SWIDE_TOTAL_PHYR* and *SWIDE_TOTAL_PHYW*. The *ELAPSED_TIME_SECONDS* is calculated directly in our Java benchmarking platform.

```
01: WITH DATAFSTATS AS
02: (
03: SELECT
04:  dfilesStatsE.PHYRDS - dfilesStatsS.PHYRDS AS PHYSICAL_READS,
05:  dfilesStatsE.PHYWRTS - dfilesStatsS.PHYWRTS AS PHYSICAL_WRITES,
06:  (dfilesStatsE.PHYBLKRD - dfilesStatsS.PHYBLKRD)*dfilesStatsE.BLOCK_SIZE AS BYTES_READ,
07:  (dfilesStatsE.PHYBLKWRT - dfilesStatsS.PHYBLKWRT)*dfilesStatsE.BLOCK_SIZE AS BYTES_WRITTEN,
08:  dfilesStatsE.NAME
09: FROM
10:  MY_DATAFILE_STATS_END dfilesStatsE INNER JOIN
11:  MY_DATAFILE_STATS_START dfilesStatsS
12:  ON dfilesStatsE.FILE# = dfilesStatsS.FILE#
13: ),
14: OTHERSTATS AS
15: (
16: SELECT
```

```
17:  tbE.CPU_TIME_SEC - tbS.CPU_TIME_SEC AS CPU_TIME_SEC
18: -- (tbE.SNAPSHOT_TIME - tbS.SNAPSHOT_TIME) * 24*60*60  AS ELAPSED_TIME_SECONDS,
19: FROM MY_OTHER_STATS_END tbE, MY_OTHER_STATS_START tbS
20: )
----------------------------
-- SYSTEMWIDE STATISTICS:
----------------------------
21: SELECT 'SWIDE_CPU_TIME_SECONDS' AS STAT_NAME , CPU_TIME_SEC AS STAT_VALUE FROM OTHERSTATS
22: UNION ALL
23: SELECT 'SWIDE_TOTAL_PHYR' AS STAT_NAME, SUM(PHYSICAL_READS) AS STAT_VALUE FROM DATAFSTATS
24: UNION ALL
25: SELECT 'SWIDE_TOTAL_PHYW'  AS STAT_NAME, SUM(PHYSICAL_WRITES) AS STAT_VALUE FROM DATAFSTATS
26: UNION ALL
27: SELECT 'SWIDE_TOTAL_PHYR_IN_BYTES' AS STAT_NAME, SUM(BYTES_READ) AS STAT_VALUE FROM DATAFSTATS
28: UNION ALL
29: SELECT 'SWIDE_TOTAL_PHYW_IN_BYTES'  AS STAT_NAME, SUM(BYTES_WRITTEN) AS STAT_VALUE FROM DATAFSTATS
```

B.3.3 Extracting the buffer manager statistics from the snapshots

The buffer manager statistics were saved in the snapshot tables MY_OTHER_STATS_END and MY_OTHER_STATS_START. From them we can retrieve the statistics BUFFMAN_CACHE_SIZE_BYTES, BUFFMAN_CACHE_LOOKUPS and BUFFMAN_CACHE_HITS, see Chapter 4 Section 4.2. In Code Listing 33, we present a routine for gathering these statistics in Oracle. These statistics are obtained in an identical manner in MSQLS and DB2, therefore we omit the respective code used.

Code Listing 33. This code listing illustrates how we can query the snapshot tables MY_OTHER_STATS_END and MY_OTHER_STATS_START to obtain the buffer manager statistics in Oracle. In the code lines 1 to 8, define the snapshot difference table OTHERSTATS. In the code lines 9 to 10, we rotate the collumns BUFFMAN_CACHE_HITS, BUFFMAN_CACHE_LOOKUPS and BUFFMAN_CACHE_SIZE_BYTES into rows. Each of the rows having one of the column names in the STAT_NAME attribute, and the corresponding column value in the STAT_VALUE attribute.

```
-----------------
-- BUFFER MANAGER STATS
------------------
01: WITH OTHERSTATS AS
02: (
03: SELECT
04:  tbE.CACHE_HITS - tbS.CACHE_HITS AS BUFFMAN_CACHE_HITS,
05:  tbE.CACHE_LOOKUPS - tbS.CACHE_LOOKUPS AS  BUFFMAN_CACHE_LOOKUPS,
06:  CAST(tbE.CACHE_BUFF_SIZEB AS NUMBER) AS BUFFMAN_CACHE_SIZE_BYTES
07: FROM MY_OTHER_STATS_END tbE, MY_OTHER_STATS_START tbS
08: )
09: SELECT STAT_NAME, STAT_VALUE
10: FROM OTHERSTATS
11: UNPIVOT ( STAT_VALUE FOR STAT_NAME IN
```

```
12:          (BUFFMAN_CACHE_HITS, BUFFMAN_CACHE_LOOKUPS, BUFFMAN_CACHE_SIZE_BYTES)) unpvt
```

B.3.4 Extracting the datafiles statistics from the snapshots

The data files statistics were saved in two snapshot tables MY_DATAFILE_STATS_END and MY_DATAFILE_STATS_START . In Code Listing 34 , we present the routine used to gather the statistics of the source data file in Oracle. These statistics are obtained in an identical manner in MSQLS and DB2, therefore we omit the respective code used. The routines for gathering the statistics of the remaining data files are identical and are not illustrated.

Code Listing 34. This code listing illustrates how we can query the snapshot tables MY_DATAFILE_STATS_END and MY_DATAFILE_STATS_START to obtain the source datafile statistics described in Chapter 4 Section 4.2, in Oracle. In the lines 1 to 14 we define the DATAFSTATS table, representing the snapshot difference of all the data-files statistics. In the lines 15 to 26, we focus on the source data-file (line 23), and rotate the statistic columns SRC_DTF_PHYR , SRC_DTF_PHYR_IN_BYTES , SRC_DTF_PHYW and SRC_DTF_PHYW_IN_BYTES into multiple rows. In the lines 28 to 30, we gather source table size. In this case, the USER_TABLES in line 29 is an Oracle built-in view.

```
-----------------------------
-- BASE DATAFILE STATS:
-----------------------------
01: WITH DATAFSTATS AS
02: (
03: SELECT
04:  dfilesStatsE.PHYRDS - dfilesStatsS.PHYRDS AS PHYSICAL_READS,
05:  dfilesStatsE.PHYWRTS - dfilesStatsS.PHYWRTS AS PHYSICAL_WRITES,
06:  (dfilesStatsE.PHYBLKRD - dfilesStatsS.PHYBLKRD)*dfilesStatsE.BLOCK_SIZE AS BYTES_READ,
07:  (dfilesStatsE.PHYBLKWRT - dfilesStatsS.PHYBLKWRT)*dfilesStatsE.BLOCK_SIZE AS BYTES_WRITTEN,
09:  dfilesStatsE.NAME
10: FROM
11:  MY_DATAFILE_STATS_END dfilesStatsE INNER JOIN
12:  MY_DATAFILE_STATS_START dfilesStatsS
13:  ON dfilesStatsE.FILE# = dfilesStatsS.FILE#
14: )
-----------------------------
-- SOURCE DATAFILE STATISTICS:
-----------------------------
15: SELECT unpvt.STAT_NAME, unpvt.STAT_VALUE
16: FROM(
17: SELECT
18:  PHYSICAL_READS  AS  SRC_DTF_PHYR,
19:  PHYSICAL_WRITES AS  SRC_DTF_PHYW,
20:  BYTES_READ      AS  SRC_DTF_PHYR_IN_BYTES,
21:  BYTES_WRITTEN   AS  SRC_DTF_PHYW_IN_BYTES
22: FROM DATAFSTATS
23: WHERE NAME LIKE '%USER'
24: ) temp
```

```
25: UNPIVOT (STAT_VALUE FOR STAT_NAME
26:  IN (SRC_DTF_PHYR, SRC_DTF_PHYR_IN_BYTES, SRC_DTF_PHYW, SRC_DTF_PHYW_IN_BYTES)) unpvt
27: UNION ALL
--------
-- PROFILE TABLE SIZE (measured in terms of used table blocks)
-------
28: SELECT 'SRC_TAB_SIZE_IN_BYTES' AS STAT_NAME,   blocks*8192  as STAT_VALUE
29: FROM user_tables
30: WHERE table_name='PROFILE'
```

Bibliography

Abiteboul, S., & Bidoit, N. (1984). 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). New York, NY, USA: ACM.

Abiteboul, S., Buneman, P., & Suciu, D. (1999). *Data on the web: From relations to semistructured data and xml*. San Francisco, CA, USA: Morgan Kaufmann.

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of databases*. Addison-Wesley.

Bohannon, P., Elnahrawy, E., Fan, W., & Flaster, M. (2006). Putting context into schema matching. In *Vldb '06: Proceedings of the 32nd international conference on very large data bases* (pp. 307–318). VLDB Endowment.

Carreira, P. (2007). *Mapper: An efficient one to many operator*. Unpublished doctoral dissertation, Universidade de Lisboa, Faculdade de Ciências.

Carreira, P., Galhardas, H., Lopes, A., & Jo a. P. (2007). One-to-many data transformations through data mappers. *Data Knowl. Eng.*, *62*(3), 483–503.

Codd, E. (1970). A relational model of data for large shared data bases. *Comm. ACM*, *13*(6), 377–387.

Cunningham, C., Galindo-Legaria, C. A., & Graefe, G. (2004). 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.

Darwen, H. (1996). In reply to domains, relations and religious wars. *SIGMOD Rec.*, *25*(4), 6–7.

Date, C. (2006). *Date on database: writings 2000-2006*. Apress.

Date, C. J. (1994). *Introduction to database systems* (I. E. Board, Ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.-E., & Zemke, F. (2004). Sql:2003 has been published. *SIGMOD Rec.*, *33*(1), 119–126.

Elmasri, R. A., & Navathe, S. B. (1999). *Fundamentals of database systems* (C. Shanklin, Ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Galhardas, H., Florescu, D., Shasha, D., Simon, E., & Saita, C.-A. (2001). Declarative data cleaning: Language, model, and algorithms. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, & R. T. Snodgrass (Eds.), *Vldb* (p. 371-380). Morgan Kaufmann.

Iyengar, S., Sudarshan, S., 0002, S. K., & Agrawal, R. (2008). Exploiting asynchronous io using the asynchronous iterator model. In G. Das, N. L. Sarda, & P. K. Reddy (Eds.), *Comad* (p. 127-138). Computer Society of India / Allied Publishers.

Jaeschke, G., & Schek, H. J. (1982). 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). New York, NY, USA: ACM.

Kepser, S. (2004). A simple proof for the turing-completeness of xslt and xquery. In *Extreme markup languages.*

Libkin, L. (2003). Expressive power of sql. *Theor. Comput. Sci.*, *296*(3), 379–404.

Mehta, D. P., & Sahni, S. (2004). *Handbook of data structures and applications (chapman & hall/crc computer and information science series.).* Chapman & Hall/CRC.

Melton, J. (1998). *Understanding sql's stored procedures: a complete guide to sql/psm.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Melton, J. (2002). *Advanced sql 1999: Understanding object-relational, and other advanced features*. New York, NY, USA: Elsevier Science Inc.

Melton, J., & Simon, A. (2002). *SQL: 1999-Understanding Relational Language Components*. Morgan Kaufmann.

Oracle. (n.d.). *Chapter 22 sql for modeling.* Oracle. (Web location: http://download.oracle.com/docs/ in Database Data Warehousing Guide 10g Release 2 (10.2))

ORACLE. (2005). *Oracle database utilities 10g release 2 (10.2).* (URL: http://www.oracle.com/pls/db102/homepage)

ORACLE. (2008). *Oracle database sql language reference 11g release 1 (11.1).* (URL: http://www.oracle.com/pls/db102/homepage)

ORACLE, & Moore, S. (2008). *Oracle database pl/sql language reference 11g release 1 (11.1).* (URL: http://www.oracle.com/pls/db102/homepage)

Ramakrishnan, R., & Gehrke, J. (2002). *Database management systems*. McGraw-Hill Science/Engineering/Math.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2001). *Database systems concepts* (Fourth ed.). McGraw-Hill Science/Engineering/Math.

Songini, M. L. (2004). ETL. *Computerworld*, *38*(5), 23.

Vassiliadis, P., Karagiannis, A., Tziovara, V., & Simitsis, A. (2007). Towards a benchmark for etl workflows. In V. Ganti & F. Naumann (Eds.), *Qdb* (p. 49-60).

Walmsley, P. (2007). *Xquery*. O'Reilly Media, Inc.