Universidade de Lisboa
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Mapper: An Efficient Data Transformation Operator

Paulo Jorge Fernandes Carreira

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE ENGENHARIA INFORMÁTICA

2007

Universidade de Lisboa
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Mapper: An Efficient Data Transformation Operator

Paulo Jorge Fernandes Carreira

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE ENGENHARIA INFORMÁTICA

2007

Tese orientada pela Prof.ª Doutora Helena Isabel de Jesus Galhardas
e pelo Prof. Doutor Mário Jorge Costa Gaspar da Silva

Abstract

Data transformations are fundamental operations in *legacy data migration*, *data integration*, *data cleaning*, and *data warehousing*. These operations are often implemented as relational queries that aim at leveraging the optimization capabilities of most DBMSs. However, relational query languages like SQL are not expressive enough to specify *one-to-many data transformations*, an important class of data transformations that produce several output tuples for a single input tuple. These transformations are required for solving several types of data heterogeneities, like those that occur when the source data represents aggregations of the target data.

This thesis proposes a new relational operator, named *data mapper*, as an extension to the relational algebra to address one-to-many data transformations and focus on its optimization. It also provides algebraic rewriting rules and execution algorithms for the logical and physical optimization, respectively. As a result, queries may be expressed as a combination of standard relational operators and mappers. The proposed optimizations have been experimentally validated and the key factors that influence the obtained performance gains identified.

Keywords: Relational Algebra, Data Transformation, Data Integration, Data Cleaning, Data Warehousing.

Sumário

As transformações de dados são operações fundamentais em processos de *migração de dados* de sistemas legados, *integração de dados*, *limpeza de dados* e ao *refrescamento de Data Warehouses*. Usualmente, estas operações são implementadas através de interrogações relacionais por forma a explorar as optimizações proporcionadas pela maioria dos SGBDs. No entanto, as linguagens de interrogação relacionais, como o SQL, não são suficientemente expressivas para especificar as transformações de dados do tipo *um-para-muitos*. Esta importante classe de transformações é necessária para resolver de forma adequada diversos tipos de *heterogeneidades de dados* tais como as que decorrem de situações em que os dados do esquema origem representam uma agregação dos dados do sistema destino.

Esta tese propõe a extensão da álgebra relacional com um novo operador relacional denominado *data mapper*, por forma a permitir a especificação e optimização de transformações de dados um-para-muitos. O trabalho apresenta regras de reescrita algébrica juntamente com diversos algoritmos de execução que proporcionam, respectivamente, a optimização lógica e física de transformações de dados um-para-muitos. Como resultado, é possivel optimizar transformações de dados que combinem operadores relacionais comuns com *data mappers*. As optimizações propostas foram validadas experimentalmente e identificados os factores que influênciam os seus respectivos ganhos.

Palavras Chave: Álgebra Relacional, Transformação de dados, Integração de Dados, Limpeza de Dados, Data Warehousing.

Resumo Alargado

A envolvente económica actual tornou frequente a evolução dos sistemas de informação. Esta evolução é desencadeada pela aquisição de novos pacotes de software ou pela necessidade de integrar múltiplos sistemas heterogéneos num único sistema.

Quando a evolução dos sistemas de informação é efectuada através da sua substituição, torna-se necessário migrar os dados do sistema legado para o novo sistema. Este processo é conhecido como *migração de dados*. A *integração de sistemas heterogéneos* requer a integração de múltiplas fontes de dados numa base de dados unificada (Halevy *et al.*, 2005).

Uma outra actividade importante nos sistemas de informação é a *prospecção de informação*, que consiste na exploração dos dados para deduzir conhecimento para apoio à tomada de decisão. Esta actividade assenta em duas operações fundamentais: a já mencionada integração de dados, que visa juntar os dados provenientes de fontes distintas, e a *limpeza de dados*, cujo objectivo é assegurar a qualidade dos dados.

Os processos de migração, de integração e de limpeza de dados, bem como de refrescamento de Data Warehouses são constituídas por diversas etapas que empregam *transformações de dados* como operações fundamentais (Rundensteiner, 1999). De uma forma geral, uma transformação de dados converte dados de uma determinada representação, ou *esquema origem*, numa outra representação, ou *esquema destino*.

Verifica-se na prática que os mesmos dados são representados de maneiras diferentes em sistemas diferentes, especialmente se estes sistemas foram desenvolvidos usando técnicas de análise distinctas ou por profissionais com formações diversas. Estas discrepâncias de representação são conhecidas na literatura como *heterogeneidades dos dados* e determinam a complexidade de transformações dos dados: diferenças mais substanciais de representação requerem transformações mais elaboradas (Kim *et al.*, 2003; Rahm & Do, 2000).

Alguns tipos comuns de heterogeneidades são, por exemplo:

i) a utilização de unidades de medida diferentes — por exemplo, a conversão de dólares em euros;

ii) diferenças nas representações de dados compostos — por exemplo, a representação de uma data utilizando atributos distintos para dia, mês e ano por oposição a um único atributo do tipo date;

iii) representações distintas do mesmo domínio — por exemplo, diferentes representações para Booleanos: {true, false} por oposição a {yes, no};

iv) representação dos dados segundo diferentes níveis de agregação — por exemplo, dados que representam eventos com frequência diária e que têm de ser representados como eventos com frequência horária noutro esquema.

Os diferentes tipos de heterogeneidades de dados são resolvidos empregando classes distintas de transformações de dados. De acordo com Galhardas (2001) e Cui & Widom (2001), uma transformação de dados pode ser classificada de acordo com o tipo de mapeamento que ela representa em termos da *multiplicidade* dos tuplos de entrada e de saída.

As transformações *um-para-um* produzem exactamente um tuplo de saída para cada tuplo da entrada. Esta classe de mapeamentos pode ser usada, por exemplo, para resolver as heterogeneidades dos dados decorrentes da utilização de diferentes unidades de medida. As transformações *um-para-muitos* produzem diversos tuplos na saída para cada tuplo na entrada. Esta classe de transformação de dados é empregue sempre que os dados de fonte representam uma agregação dos dados do destino (por exemplo, dados agregados por ano na fonte e dados mensais no destino). As transformações *muitos-para-um* são as que geram no máximo um tuplo de saída para cada conjunto de tuplos da entrada. Esta classe de transformações ocorre quando grupos de tuplos da fonte têm que ser consolidados, por exemplo, através do comando `GROUP BY` do SQL, que pode ser aplicado, por exemplo, para transformar os salários dos empregados nos montantes brutos correspondentes. As transformações *muitos-para-muitos* caracterizam as transformações de dados que geram conjuntos de tuplos a partir de conjuntos de tuplos, tais como sejam as operações de *ordenação* e de *normalização*. Estas últimas,

são operações matemáticas que convertem um conjunto de tuplos num novo conjunto com determinadas características, sendo utilizadas fundamentalmente em contextos de limpeza de dados ou na preparação de dados para prospecção de informação (Han & Kamber, 2001, Section 3.3.2).

Esta tese debruça-se sobre a problemática das transformações de dados um-para-muitos, que, apesar da sua predominância no contexto da migração, integração e limpeza de dados, não foram até à data, estudadas de forma sistemática.

Descrição do problema

Tendo em vista a minimização do esforço de desenvolvimento e a maximização do desempenho das transformações de dados, é altamente desejável que estas sejam descritas recorrendo a um formalismo simultaneamente *declarativo*, *expressivo*, e *optimizável*.

Os benefícios da utilização do paradigma declarativo para a especificação de transformações dos dados são destacados por Rahm & Do (2000). Um aspecto importante das linguagens declarativas é poderem ser equipadas com um conjunto de construções específicas para um domínio (van Deursen *et al.*, 2000). A utilização de construções específicas de domínio nas transformações de dados, torna-as mais fáceis de descrever e de compreender, uma vez que estas não são poluídas com detalhes desnecessários.

De facto, o desacoplamento entre as especificações das transformações de dados e a sua execução abre diversas oportunidades interessantes do ponto de vista da optimização, uma vez que muitos aspectos complexos da execução podem ser deduzidos automaticamente. Por exemplo, uma vez que nem todos os planos de acesso têm o mesmo tempo de execução, os mais eficientes podem ser determinados automaticamente.

Finalmente, as construções de linguagens declarativas atrás mencionadas podem ser combinadas para expressar uma multiplicidade de transformações de dados distintas. Entretanto, esta expressividade não surge gratuitamente: mais expressividade significa também maior complexidade em termos de optimização.

O desenho de linguagens que maximizem a declaratividade, a expressividade e a optimizabilidade constitui um problema de investigação complexo. No que diz

respeito à especificação de dados um-para-muitos, nenhum formalismo foi proposto até agora que seja simultaneamente, declarativo, expressivo e optimizável.

Limitações das soluções actuais

Actualmente, as transformações de dados um-para-muitos são desenvolvidas recorrendo a uma das seguintes alternativas:

i) elaboração de um programa de transformação de dados utilizando uma linguagem de programação de âmbito geral, tal como o C (Kernighan & Ritchie, 1988), o Java (Gosling *et al.*, 2005) ou o Perl (Wall *et al.*, 2000);

ii) modelação da transformação utilizando uma ferramenta de ETL;

iii) utilização de uma linguagem proprietária de base de dados, tal como, por exemplo,PL/SQL do Oracle (Feuerstein & Pribyl, 2005);

iv) desenvolvendo uma interrogação, por exemplo, em SQL.

Cada uma destas alternativas apresenta um conjunto de inconvenientes. Considerando as linguagens de âmbito geral, estas não fornecem, apesar da sua expressividade, uma separação clara entre a lógica da transformação e sua execução, resultando daqui que as transformações de dados se tornam difíceis de compreender e de manter. Adicionalmente, à parte das optimizações estáticas de código, muitas optimizações significativas inerentes ao domínio das transformações dos dados não são passíveis de identificação pelo compilador ou pelo interpretador de uma linguagem de âmbito geral. Quando às ferramentas de ETL, embora forneçam bibliotecas extensivas de operadores de transformação dos dados, a sua composição não é optimizável (Simitsis *et al.*, 2005). Além disso, em algumas ferramentas de ETL, tais como o FileAid Express[1], os operadores têm um poder expressivo bastante limitado. Para superar as limitações de expressividade, é necessário recorrer a scripts complexos utilizando linguagens proprietárias ou à codificação de funções externas. Em alternativa, as transformações dos dados executadas como extensões de um SGBD, tais como os *Persistent Stored Modules*

[1] http://www.compuware.com/products/fileaid/express.html

(Garcia-Molina *et al.*, 2002, Section 8.2), tanto na forma de *stored procedures* como de *function tables* do SQL 2003 (Eisenberg *et al.*, 2004), utilizam uma combinação das construções procedimentais e declarativas que são extremamente difíceis de optimizar.

As transformações dos dados podem também ser especificadas declarativamente como interrogações (ou vistas) sobre os dados de origem. A linguagem de escolha para expressar transformações dos dados é geralmente o SQL, que é baseado na *álgebra relacional* (Codd, 1970). Uma vantagem de usar o SQL e a álgebra relacional é a disponibilidade de um vasto corpo de conhecimento sobre a sua optimização (Chaudhuri, 1998; Graefe, 1993). No entanto, muitas transformações de dados pertinentes não podem ser descritas através de expressões relacionais (Lakshmanan *et al.*, 1996), devido ao limitado poder expressivo da álgebra relacional (Aho & Ullman, 1979). Em particular, a álgebra relacional não permite expressar a classe das transformações de dados um-para-muitos (facto que é demonstrado formalmente nesta tese).

Solução proposta

A tese propõe a extensão da álgebra relacional com um novo operador unário, denominado *data mapper*. Esta extensão supera as limitações de expressividade da álgebra relacional tirando partido, simultaneamente, da sua estrutura declarativa e do seu potencial de optimização. Como resultado, obtém-se um formalismo que permite especificar transformações de dados um-para-muitos de uma forma declarativa, expressiva e optimizável.

Informalmente, o operador data mapper, uma vez aplicado a uma relação de entrada produz uma relação da saída. De uma forma semelhante a outras extensões à álgebra relacional, tais como o operador generalizado de projecção ou o operador de agregação (Klug, 1982), o operador mapper utiliza funções externas. O mapper permite criar múltiplos tuplos de saída dinamicamente a partir da avaliação dos conteúdos de cada tuplo de entrada. Este tipo da operação tem aparecido implicitamente em sistemas de transformação de esquemas e de dados, tais como os propostos por Amer-Yahia & Cluet (2004), por Cui & Widom (2001),

Cunningham *et al.* (2004), Galhardas *et al.* (2000), e Raman & Hellerstein (2001). No entanto, não foi ainda estudado como um operador relacional.

As linguagens de interrogação de SGBD, bem como as linguagens subjacentes a ferramentas de ETL e de limpeza de dados são baseadas na álgebra relacional (Amer-Yahia & Cluet, 2004; Galhardas *et al.*, 2000; Labio *et al.*, 2000; Raman & Hellerstein, 2001; Simitsis *et al.*, 2005; Zhou *et al.*, 1996). Neste contexto, o objectivo de equipar a álgebra relacional com o operador mapper reveste-se de um elevado interesse prático. Em primeiro lugar, porque dota as ferramentas de transformação de dados baseadas na álgebra relacional com um operador com maior poder expressivo. Em segundo lugar, porque aumenta a eficiência da execução das transformações de dados um-para-muitos.

A tese propõe extensões para deduzir estratégias melhoradas da execução de interrogações que combinam operadores relacionais com mappers, que estendem as estratégias de optimização para interrogações relacionais já estudadas na literatura (Chaudhuri, 1998).

Contribuições

A tese propõe uma extensão à álgebra relacional para tratamento das transformações um-para-muitos, propondo um novo operador relacional e respectivos mecanismos para a sua optimização. Os mecanismos propostos consistem em regras de optimização algébrica complementados por algoritmos de execução física visando a optimização lógica e física, respectivamente. As propostas são validadas experimentalmente, sendo identificados os factores determinantes dos ganhos obtidos. De uma forma mais detalhada, as principais contribuições deste trabalho são as seguintes:

Um operador especializado para transformações um-para-muitos. Para melhor compreender o operador mapper, foi desenvolvida a sua definição formal e, a partir desta formalização, demonstradas diversas propriedades importantes dos mappers. Entre as mais importantes destaca-se a demonstração de que a semântica do mapper pode ser simulada através do produto cartesiano dos resultados das funções avaliadas, conduzindo a um algoritmo físico de execução extremamente simples. Na sequência deste estudo,

o poder expressivo da álgebra relacional estendida com o operador mapper é também estudado, demonstrando-se formalmente que a álgebra relacional estendida é estritamente mais expressiva do que a álgebra relacional padrão.

É proposta também uma extensão directa à sintaxe da linguagem SQL que possibilita a especificação, na forma de interrogações, de transformações que combinam mappers com outros operadores relacionais.

Regras de optimização algébrica demonstradas formalmente. Propõe-se um conjunto de regras de reescrita algébrica que são complementadas com as correspondentes demonstrações formais de correcção. As regras apresentadas visam a optimização lógica de expressões de transformações de dados que combinam operadores relacionais com mappers, evitando avaliações redundantes de funções. São propostos dois conjuntos de regras. O primeiro conjunto consiste em regras para comutar selecções que visam filtrar à entrada tuplos desnecessários. O segundo conjunto consiste em regras para comutar projecções que evitam a propagação de atributos irrelevantes para avaliação de operadores subsequentes. Com base nestas regras é possível gerar planos lógicos alternativos para a execução duma expressão relacional envolvendo mappers.

Algoritmos físicos da execução. A optimização lógica do novo operador é complementada com algoritmos físicos para execução do operador mapper. Embora a semântica formal do operador mapper sugira a execução *tuplo-a-tuplo*, designada como *Algoritmo Naïve*, esta, apesar de atractiva devido à sua simplicidade, revela-se muito ineficiente em situações reais. Os problemas de ineficiência são especialmente notórios sempre que um mapper é composto por funções com custos de avaliação elevados, como as utilizadas em contextos de limpeza de dados. Por esta razão, a pesquisa de algoritmos eficientes para execução de mappers reveste-se da máxima importância. Para superar esta dificuldade, a tese fornece dois algoritmos de execução que tiram partido da presença de valores duplicados nos atributos das relações. O princípio de operação de ambos assenta na redução do custo total de avaliação do mapper, evitando avaliações supérfluas das

funções. O primeiro algoritmo, designado por *Algoritmo de Shortcircuiting*, tira proveito da semântica do mapper: sempre que o resultado de uma função é o conjunto vazio, a avaliação das restantes funções é dispensável. O segundo algoritmo, designado por *Algoritmo Baseado em Cache*, explora a presença de valores duplicados na relação de entrada, recorrendo a uma cache em memória actualizada com os resultados das funções do mapper.

Para superar as limitações de um mecanismo de cache em memória, são consideradas políticas de substituição de cache. Inicialmente é considerada uma variante do algoritmo baseado em cache, utilizando a política de substituição *least recently used* (LRU), frequentemente empregue na gestão de caches de bases de dados e em sistemas operativos e duas novas políticas de substituição específicas para a avaliação de mappers: *least usefull replacement* (LUR) e *relaxed least usefull replacement* (XLUR). A política LUR, baseia as suas decisões de substituição na maximização de uma função de utilidade que tem como parâmetros o número das referências, a distância inter-referências e o custo médio de avaliação da função. Uma vez que a política de substituição LUR tem um custo de execução elevado, é proposta uma nova política, designada XLUR, que minimiza uma aproximação da função de utilidade.

Conclusões finais

A tese propõe um novo operador relacional para fazer face ao problema da especificação de transformações de dados um-para-muitos, explorando com sucesso as oportunidades de optimização lógica e física. Em relação ao operador mapper, demonstra-se formalmente a sua pertinência, dado que nenhuma expressão relacional é suficientemente potente para exprimir a classe de transformações de um-para-muitos. Logo, a extensão com um novo operador é não só desejável mas necessária. Adicionalmente, valida-se o interesse prático do mapper através da sua incorporação na ferramenta comercial de transformação de dados "Data Fusion", seleccionada para diversos projectos de grande relevância no sector bancário ibérico e na administração pública portuguesa.

Contrastando o desempenho de transformações de dados um-para-muitos que aplicam selecções aos mappers com as suas equivalentes algébricas optimizadas, conclui-se que a introdução de optimizações algébricas se traduz em elevados ganhos de desempenho. Relativamente aos novos algoritmos propostos para o operador mapper concluiu-se que quer o Algoritmo de Shortcircuiting, quer o Algoritmo Baseado em Cache são vantajosos na redução do custo da avaliação do mapper, produzindo importantes melhorias nos tempos de execução das transformações um-para-muitos. A pesquisa desenvolvida nesta tese tem impacto na tecnologia utilizada para executar transformações de dados, demonstrando que mais uma classe de transformações dos dados pode ser exprimida e optimizada utilizando as boas práticas da independência lógica e física dos SGBD. Tendo em conta que hoje em dia os SGBD desempenham papéis cada vez mais complexos, quer como motores de transformação, quer como gestores de áreas armazenamento intermédio em diversas actividades da gestão de dados, este tese contribui para o alargamento das suas aplicações. Na prática, a introdução do operador mapper amplia a classe das transformações de dados que podem ser asseguradas de forma eficaz. O operador mapper é também uma adição valiosa a uma ferramenta de transformação de dados, explicitando transformações um-para-muitos embutidas em scripts de transformação, tornando, dessa forma, mais fácil de compreender e manter as transformações de dados.

Acknowledgements

Following the dissertations' best practices, this is the place where I voluteerly acknowledge those who, either by chance or misfortune, were involved in my PhD.

I start acknowledging Prof. Mário J. Silva for calling me to pursue my PhD.

It would have been impossible to carry out my research without the support from the company where I always working in, Oblog. I must acknowledge the management of Oblog for considering my post-graduate programme relevant for the company and unconditionally supporting me. I am also grateful to Julião Duartenn and Ana Ferrão who have also supported me on several occasions. A special word goes out to the *dream team* that embarked with me in the Data Fusion project: Alejandro Tamalet, Leonardo Bartocci, João Fitas, Fernando Martins, Pedro Lopes and André Gonçalves.

After getting the support of the company, Prof. Helena Galhardas accepted to advise me. Her patience, wisdom and support made it possible.

Special thanks also go to Antónia Lopes and João Pereira for their time in many discussions and proofreading. For their support in many different occasions, my thanks go also to my colleagues at IST: Pavel Calado, Carla Ferreira and Andreas Wichert, and to my colleagues at FCUL: Marcírio Chaves, Daniel Gomes and Bruno Martins.

Finally, I have been blessed with a family that made my life much richer during these last few years. I'll be forever in debt to my wife Susana and to my daughter Carolina to whom much of the time spent in writing this thesis is owed.

Aos pais fantásticos Celsino e Rosa por investirem tudo nos filhos.

À memória da avó Rosa pelo legado de tenacidade e trabalho árduo.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Today's business landscape is fast-changing. Since company mergers and joint-ventures became common headlines, the information systems that support their operation have been required to evolve at a similar pace. The evolution is achieved either by simply replacing old systems by newer ones, or by integrating multiple heterogeneous systems into a new single one.

When the evolution of an information system consists of its replacement by a newer one, the underlying data must be migrated into the new system. This process is known as *legacy-data migration*. The alternative of integrating multiple heterogeneous systems, nowadays referred as *enterprise information integration*, often relies on *data integration*, which consists of integrating multiple sources of data into one unified database (Halevy *et al.*, 2005).

Data brought together from different systems must be explored to derive new knowledge for decision making, which constitutes another important activity on information systems known as *business intelligence*. Two well-known cornerstone activities of business intelligence are: data integration, mentioned above, which aims at bringing together data from distinct sources; and *data-cleaning* applications, whose goal is to ensure data quality.

Data management activities such as legacy-data migration, data-integration, data-cleaning and the refreshment of data stored in a data warehouse are implemented as sequences of steps that employ data transformations as fundamental operations (Rundensteiner, 1999). Broadly speaking, a data transformation takes

source data that obeys to a given representation and converts it into a distinct *target* representation.

As a matter of fact, the same data is often represented in fundamentally different ways in systems, in particular if these systems were developed using distinct analysis techniques by different people. These differences in representation are known in the literature as *data heterogeneities* and they determine the complexity of data transformations: more substantial differences in representing data that requires more elaborate data transformations (Kim *et al.*, 2003; Rahm & Do, 2000). Several kinds of data heterogeneities have been identified, for instance:

i) the use of different units of measurement —e.g., the conversion from dollars to euros;

ii) the use of different representations of compound data —e.g., multiple attributes representing day, month and year information *vs* a single date attribute;

iii) distinct representations of the same data domain —e.g., {true, false} *vs* {yes, no} for Boolean values;

iv) the representation of data according to different aggregation levels —e.g., in one schema, some data represents an *hourly* measure, while the same data represents *daily* measure in the other schema.

The various types of data heterogeneities can be resolved by employing distinct classes of data transformations. According to Galhardas (2001) and Cui & Widom (2001), one way to classify a data transformation is to consider the type of mapping it represents in terms of the multiplicity of its input and output tuples. A *one-to-one* mapping produces exactly one output tuple for each input tuple. This class of mappings can be used, for example, to solve the data heterogeneities caused by the existence of different units of measurement. A *one-to-many* mapping produces several output tuples for each input tuple. This class of data transformations is employed, for example, when the source data represents an aggregation of the target data (e.g., yearly aggregated data in the source and monthly data in the target). A *many-to-one* mapping corresponds

to data transformations that generate at most one output tuple from a set of input tuples. This class of mapping takes place when groups of source tuples have to be consolidated, for example, through an SQL group and aggregation, e.g., transforming employees salaries into their corresponding total incomes. Finally, *many-to-many* mappings characterize data transformations that generate sets of tuples from sets of tuples, like *sorting* and *normalization* operations. Normalizations are mathematic transformations that take sets of input tuples and produce new sets of tuples that meet specific requirements. An example of a normalization operation, which is frequently required when preparing data for data mining, consists of converting all input values proportionally, so that they fall within specific upper and lower limits (Han & Kamber, 2001, Section 3.3.2).

This dissertation is particularly concerned with data transformations classified as one-to-many mappings, henceforth designated as *one-to-many data transformations*.

1.1 One-to-Many Data Transformations

One-to-many data transformations will be introduced through two examples, which are based on real-world data migration problems previously identified (Carreira & Galhardas, 2004a). These examples are presented here in a simplified form for illustration purposes.

EXAMPLE 1.1.1: *Consider a relational table* `LOANEVT` *that, for each given loan, keeps the events that occur since the establishment of a loan contract until it is closed. A loan event consists of a loan number, a type and several columns with amounts. For each loan and event, one or more event amounts may apply. The field* `EVTYPE` *maintains the event type, which can be* `OPEN` *when the contract is established,* `PAY` *meaning that a loan installment has been payed,* `EARLY` *when an early payment has been made,* `FULL` *meaning that a full payment was made, or* `CLOSED` *meaning that the loan contract has been closed. In the target table named* `EVENTS`*, the same information is represented by adding one row per event with the corresponding amount. An event row is added only if the amount is greater than zero.*

Relation LOANEVT

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

Relation EVENTS

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

Figure 1.1: A bounded one-to-many data transformation. The records of the source relation LOANEVT concerning the loan number 1234 (on the left) and the corresponding target relation EVENTS (on the right).

In the data transformation described in Example 1.1.1, each *input row* of the LOANEVT table corresponds to several *output rows* in the EVENTS table, as illustrated in Figure 1.1. For a given input row, the number of output rows depends on whether the contents of the CAPTL, TAX, EXPNS, BONUS columns are positive. Thus, each input row can result in at most four output rows. This means that there is a known *bound* on the number of output rows produced for each input row. We designate this type of data transformations as *bounded* one-to-many data transformations. However, in other one-to-many data transformations, such bound cannot always established *a-priori*, as shown in the following example:

EXAMPLE 1.1.2: *Consider the source relation* ***LOANS***[***ACCT***, ***AM***] *(represented in Figure 1.2) that stores the details of loans per account. Suppose* ***LOANS*** *data must be transformed into* ***PAYMENTS***[***ACCTNO***, ***AMOUNT***, ***SEQNO***]*, the target relation, according to the following requirements:*

i) In the target relation, all the account numbers are left padded with zeroes. Thus, the attribute ***ACCTNO*** *is obtained by (left) concatenating zeroes to the value of* ***ACCT****.*

Relation LOANS

ACCT	AM
12	20.00
3456	140.00
901	250.00

Relation PAYMENTS

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

Figure 1.2: An unbounded data transformation. The source relation LOANS for loan number 1234 (on the left), and the corresponding target relation PAYMENTS (on the right).

ii) The target system does not support payment amounts greater than 100. *The attribute* ***AMOUNT*** *is obtained by breaking down the value of* ***AM*** *into multiple parcels with a maximum value of 100, in such a way that the sum of amounts for the same* ***ACCTNO*** *is equal to the source amount for the same account. Furthermore, the target field* ***SEQNO*** *is a sequence number for the parcel. This sequence number starts at one for each sequence of parcels of a given account.*

The implementation of data transformations similar to those requested for producing the target relation PAYMENTS of Example 1.1.2 is challenging, since the number of output rows, for each input row, is determined by the value of the attribute AM. Thus, unlike Example 1.1.1, the upper bound on the number of output rows cannot be determined by the data transformation specification. We designate these data transformations as *unbounded* one-to-many data transformations. Other sources of unbounded data transformations exist like, for example, when converting collection-valued attributes of SQL:1999 covered by Melton & Simon (2002), where each element of the collection is mapped to a new row in the target table. A common data transformation in data-cleaning consists of converting a variable length string attribute, that encodes a set of values with a varying number of elements, into rows. This data transformation is unbounded because the exact number of output rows can only be determined by analyzing the string.

Despite their prominence in the context of data migration, integration and cleaning, one-to-many data transformations have never been addressed in the literature as a first-class relational operation.

1.2 Problem Statement

To minimize the development effort and maximize their performance, data transformations must be written in a language that is *declarative*, *expressive*, and *optimizable*.

The benefits of using the declarative paradigm for specifying data transformations have been highlighted by Rahm & Do (2000). One important aspect of declarative languages is that, since they are equipped with a set of high-level domain-specific constructs, they encourage users to focus on the problem domain (van Deursen *et al.*, 2000). As a result, data transformations become easier to write and to understand because the specifications are not cluttered by unnecessary details.

Decoupling data transformation specifications from their implementations also opens many interesting optimization opportunities, because many complex implementation aspects can be derived automatically. For example, not all execution plans have the same execution time and the most efficient ones can be better determined automatically.

Finally, the above-mentioned constructs of declarative languages can be combined to express a manifold of data transformations. However, this expressivity is governed by a compromise with optimizability: greater expressivity also means greater complexity on optimization.

The design of languages that maximize declarativeness, expressivity and optimizability is an ongoing research problem. Nevertheless, up to now, no formalism has been proposed, which is simultaneously, declarative, expressive and optimizable, for addressing one-to-many data transformations.

1.3 Overview of Existing Solutions

Currently, to develop one-to-many data transformations, one has to resort to one of the four alternatives:

i) implementing data transformation programs using a general purpose programming language, such as C (Kernighan & Ritchie, 1988), Java (Gosling *et al.*, 2005) or Perl (Wall *et al.*, 2000);

	General Purpose Language	ETL Tool	RDBMS Extensions	Relational Algebra/SQL
Declarativeness	−	−	+/−	+
Optimizability	−	−	+/−	+
Expressivity	+	+	+	+/−

Table 1.1: Comparison of the different approaches for performing one-to-many data transformations.

ii) developing data transformation workflows using a commercial ETL (Extract-Transform-Load) tool;

iii) using some database server procedural language like Oracle PL/SQL (Feuerstein & Pribyl, 2005); or

iv) using an SQL query.

Each alternative poses a number of drawbacks (see Table 1.1). The use of general purpose languages is hindered, despite their expressivity, by the lack of a clear separation between the transformation logic and its implementation. This makes data transformations difficult to understand and maintain. Moreover, apart from static code optimizations, many significant optimizations inherent to the domain of data transformations are not identified by the compiler or interpreter of a general purpose language.

Although ETL tools provide an extensive library of data transformation operators, their composition is not optimizable (Simitsis *et al.*, 2005). Moreover, in some tools, like FileAid Express[1], the provided operators have limited expressive power. To overcome this situation, one has to resort either to writing complex server scripts using proprietary languages or to coding external functions. Data transformations implemented through RDBMS extensions, such as *Persistent Stored Modules* (Garcia-Molina *et al.*, 2002, Section 8.2), like stored procedures or SQL 2003 table functions (Eisenberg *et al.*, 2004), rely on a mix of procedural and declarative constructs that are not amenable to optimization.

Data transformations can also be declaratively specified as queries (or views) over the source data. The language of choice to express data transformations is

[1]http://www.compuware.com/products/fileaid/express.html

usually SQL, which is based on Relational Algebra (RA) (Codd, 1970). A compelling aspect of using SQL and RA is the availability of a vast body of knowledge about its optimization (Chaudhuri, 1998; Graefe, 1993). However, many important data transformations cannot be expressed in this way (Lakshmanan *et al.*, 1996). This is due to the limited expressive power of RA (Aho & Ullman, 1979). In particular, RA cannot express the full class of one-to-many data transformations (this will be formally demonstrated in this thesis).

1.4 Proposed Solution

This thesis proposes to address one-to-many data transformations by extending RA with a new unary operator, the *data mapper*. This extension addresses the expressivity issue of RA concerning one-to-many data transformations, while taking advantage both of its well-founded declarative framework and its optimization potential.

Informally, a data mapper, henceforth designated as *mapper*, is applied to an input relation and produces an output relation. It iterates over each input tuple and generates one or more output tuples by applying a set of domain-specific functions. The mapper supports the dynamic creation of tuples based on the evaluation of each source tuple contents. This kind of operation appears implicitly in systems that implement schema and data transformations, like those proposed by Amer-Yahia & Cluet (2004), Cui & Widom (2001), Cunningham *et al.* (2004), Galhardas *et al.* (2000), and Raman & Hellerstein (2001). However, it has never been handled as a relational operator. The introduction of such relational operator opens interesting optimization opportunities, since expressions that combine the mapper operator with standard relational algebra operators can be optimized.

The query languages supported of RDBMSs as well as those offered by some data cleaning and ETL tools are based on RA (Amer-Yahia & Cluet, 2004; Galhardas *et al.*, 2000; Labio *et al.*, 2000; Raman & Hellerstein, 2001; Simitsis *et al.*, 2005; Zhou *et al.*, 1996). Equipping RA with the mapper operator is of great practical interest for two reasons. First, it endows data transformation applications based on RA with a more powerful transformation specification language.

Second, it speeds up the execution of data transformations expressed as combinations of standard relational operators with mappers, by proposing appropriate extensions to the logical and physical optimization strategies for relational queries already studied in literature (Chaudhuri, 1998).

1.5 Contributions

This thesis champions the extension of relational algebra for addressing one-to-many data transformations. Such extension is achieved through the proposal of a new relational operator named *data mapper* jointly with a framework for its optimization. The optimization framework consists of a set of algebraic rewriting rules and alternative execution algorithms that enable the logical and physical optimization of data transformation operations, respectively. A more detailed break-down of the contributions follows:

Validation of the relevance of one-to-many transformations. The usefulness of one-to-many data transformations was validated through an implementation of the mapper operator in a commercial tool named Data Fusion, which was developed under the supervision of the author. Data Fusion was used in the implementation of several large-scale database migration projects (Carreira & Galhardas, 2004a). The most relevant aspects of this data migration tool were published as a workshop paper in the Semantic Integration Workshop (Carreira & Galhardas, 2003) and then as a demonstration paper at ACM SIGMOD'04 (Carreira & Galhardas, 2004a).

Evaluation of one-to-many transformations. The adoption of RDBMSs to perform data transformations motivates an evaluation of how they handle one-to-many data transformations. In this study, the main factors that influence the performance of one-to-many data transformations are identified. As a further contribution, the potential benefits from the performance side come from using a dedicated operator for handling one-to-many data transformations are validated.

The new data mapper operator. A detailed formalization of the mapper operator as an extension to the relational algebra is developed building on an initial proposal presented at the IQIS'04 ACM's International Data Quality Workshop (Carreira & Galhardas, 2004b). The formal definition of the operator is given and several important properties are described. The thesis compares the expressive power of the operator with the traditional relational algebra operators. Then, it is formally demonstrated that mappers subsume the renaming, projection and selection relational operators. In the sequel, the expressive power of relational algebra extended with the mapper operator is considered. In this realm, it is formally demonstrated that relational algebra extended with the mapper operator is more expressive than standard relational algebra. This contribution was presented at SBBD'05, the Brazilian Symposium on Databases (Carreira *et al.*, 2005b).

A set of provably correct algebraic optimization rules. A set of algebraic rewriting rules that enable the logical optimization of data transformation expressions, which combine relational operators with mappers, are supplied with their corresponding formal proofs of correctness. The rules presented aim at avoiding superfluous function evaluations. There are two sets of rules. The first set consists of rules for pushing selections through mappers that aim at filtering unnecessary input tuples. The second set of rules aims at pushing projections through mappers, avoiding the propagation of attributes that are not used by subsequent operators. The development of the proposed rules was presented at SBBD'05, the Brazilian Symposium on Databases (Carreira *et al.*, 2005b) and their validation at DAWAK'05, the International Conference on Data Warehousing and Knowledge Discovery (Carreira *et al.*, 2005a).

Distinct physical execution algorithms. The logical optimization is complemented with different physical execution algorithms for the mapper operator. Although the formal semantics of the mapper operator suggests a straightforward tuple-at-a-time execution semantics, this naïve execution algorithm may be very inefficient in many real-world settings. Finding efficient algorithms to execute mappers becomes of utmost importance. The

thesis provides different execution algorithms that take advantage of the existence of duplicates in the input data.

The proposed logical optimizations and physical execution algorithms were validated on an number of experiments. Concerning logical optimizations, this study contrasts the computation effort required to evaluate expressions involving the mapper operator and its optimized equivalents. It identifies the factors that have greater influence on the performance gains obtained through the logical optimizations. Concerning the physical algorithms, each of the conditions that favor each variant of the proposed algorithm have been identified. Then, based on the observations, a cost model was proposed that may enable a cost-based optimizer to select the most appropriate optimization rule and execution algorithm. The accuracy of the cost model is also validated through experimentation. The proposal and the validation of the cost model were published in the Data and Knowledge Engineering Journal (DKE) (Carreira *et al.*, 2007).

1.6 Organization of the Thesis

The remaining of the thesis is organized into six chapters. In Chapter 2, the different possible approaches to address the problem of one-to-many data transformations are analyzed. In Chapter 3, the formal details of the mapper operator are developed. Chapter 4 presents the algebraic rewriting rules for logically optimizing queries involving mappers, together with their corresponding proofs of correctness. Then, in Chapter 5, alternative physical algorithms to execute mappers are explored. Chapter 6 presents the experimental validations. Finally, Chapter 7 summarizes and outlines directions for further research.

Chapter 2

Implementing One-to-many Transformations

This chapter presents alternatives for implementing one-to-many data transformations. Since data often resides on relational database systems, these often double as data transformation systems. Several implementations of one-to-many data transformations that use relational databases are reviewed in detail. Then, alternatives for implementing data transformations, like languages for restructuring semi-structured and XML data are considered. Schema mapping, data integration, data cleaning and ETL tools are also analyzed. The different alternatives are contrasted, giving special attention to their declarativeness and expressivity for specifying one-to-many data transformations.

2.1 Introduction

Several data management activities, like legacy-data migration, data-integration and data-cleaning, require data transformations to support modifications in the structure, representation or content of data (Rundensteiner, 1999). Since data supporting different applications is encoded in fundamentally different ways, the aforementioned data transformations are frequently quite complex. The above mentioned data management activities are now detailed:

Legacy-data migration. In this activity, projects are triggered by a common pattern in which, proprietary applications are discontinued in favor of new applicational packages. Organizations often buy applicational packages (like SAP, for instance) that replace existing ones (e.g., supplier management). This situation leads to data migration projects that must transform the data model underlying old applications into a new data model that supports new applications.

The data transformations presented in Example 1.1.1 and in Example 1.1.2 are examples of data transformations found in legacy data migration contexts where each tuple of the source relation has to be converted into potentially many tuples in the target relation.

Data integration. Heterogeneous information systems operate as a single unified system, conveying the user the illusion of interacting with a larger whole (Ziegler & Dittrich, 2004). Nowadays, such undertaking is essential for organizations to take full advantage of their IT infrastructure when it is deployed on multiple disparate systems. Data integration is implemented through a virtual homogeneous system with a single integrated schema (Batini *et al.*, 1986), also known as *mediated schema*, against which, queries are evaluated.

Although the technique may vary, the processing of queries posed over the virtual schema requires that data in the sources be combined and transformed in order to be presented according to the virtual schema. One-to-many data transformations are often required for mapping the tuples of the fused source relations into the virtual schema. Consider, for example, fusing two relations from disparate sources in which, records are organized by year and then feeding the obtained fused relation to a virtual schema, where the information is organized by month. Each tuple of the fused relation, corresponding to a year, is represented according to several tuples in the virtual schema relation, each corresponding to a month.

Data cleaning. Data quality is a critical aspect of applications that support business operations (Rahm & Do, 2000). Several tasks of the data cleaning

process comprise data transformations to produce clean data that apply a set cleaning functions to tuples containing dirty data.

When performing data cleaning, one-to-many transformations arise for example when data pertaining to multiple tuples have to be extracted from the contents of one single attribute. Consider a cleaning transformation that takes input tuples of a bug tracking system, where multiple detail lines about the bug are kept on a text attribute named `DETAILS`, and produces a relation with one tuple for each issue.

These activities are implemented with RDBMSs, or specialized tools and languages, which are all required to perform data one-to-many data transformations. In the following sections the adequacy of the different solutions for expressing one-to-many data transformations is analyzed in further detail. It is important to distinguish between bounded and unbounded data transformations, as introduced in Section 1.1. A *bounded* one-to-many data transformation admits an upper bound k in the number of output tuples generated for each input tuple. This upper bound is known before the execution of the data transformation. Conversely, *unbounded* one-to-many data transformations do not have such an upper bound known a-priori.

The classification of data transformations into bounded and unbounded is interesting, because it serves the purpose of classifying the alternative approaches for implementing one-to-many data transformations. As it turns out, bounded data transformations can be expressed as Relational Algebra (RA) expressions as introduced by Codd (1970), while unbounded data transformations cannot (this will be demonstrated later, in Section 3.5).

2.2 Relational Algebra

The normalization theory underlying the relational model imposes the division of data among several relations in order to eliminate redundancy and inconsistency of information. In the original model proposed by Codd (1970), new relations are derived from the database by selecting, joining and unioning relations. Despite the fact that RA expressions denote data transformations among relations, the

notion that presided the design of RA, as noted by Aho & Ullman (1979), was that of data retrieval. However, this notion, is insufficient for reconciling the substantial differences in the representation of data that occur between fixed source and target schemas (Miller, 1998).

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

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

In order to clarify the concept, Figure 2.1 presents the SQL implementation of the bounded data transformation presented in Example 1.1.1 using multiple **union all** statements (lines `5`, `9` and `13`). Each **select** statement (lines `2`–`4`, `6`–`8`, `10`–`12` and `14`–`16`) encodes a separate condition and potentially contributes with an output tuple. The drawback of this solution is that the size of the query grows proportionally to the maximum number of output tuples k that have to be generated for each input tuple. If this bound value k is high, the query becomes too big. Expressing one-to-many data transformations in this way has a lot of repetition, especially if many columns are involved.

Despite this drawback, many useful data transformations can be appropriately defined in terms of relational expressions, especially when considering relational algebra equipped with a generalized projection operator (Silberschatz *et al.*, 2005, p. 104). In this case, the projection list may include expressions that define the computations to be performed for each input tuple (for instance, $\pi_{ID,NAME \leftarrow FIRST||' '||LAST}$).

However, this extension is still weak to express unbounded one-to-many data transformations. The limited expressive power of relational algebra expressions was addressed very early in the database literature (Paredaens, 1978). Later, Aho & Ullman (1979) proposed extensions to overcome the limitations of RA.

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

Figure 2.1: Transformation of Example 1.1.1 using an SQL union query.

Atzeni & de Antonellis (1993) have shown that RA expressions are not capable of generating new data items.

2.3 Extensions to Relational Algebra

To support the growing range of RDBMS applications, several extensions to RA have been proposed since its inception. These extensions were introduced in the form of new declarative operators and also through the introduction of language extensions to be executed by the RDBMS. The most well known extensions introduced to the original RA operators are perhaps grouping and the computation of aggregates (Klug, 1982). However, this section addresses only those extensions that are relevant for expressing one-to-many data transformations. The first extension to be analyzed is the *pivot* operator introduced in SQL Server 2005 (Cunningham *et al.*, 2004). Unfortunately, this operator only allows to express bounded data transformations. Hence, alternatives to also express unbounded data transformations like *recursive queries* and *stored procedures*, introduced by SQL:1999 (ISO-ANSI, 1999) as well as the *table functions* of SQL-2003 (Eisenberg *et al.*, 2004) will be also be examined.

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

Figure 2.2: Transformation of Example 1.1.1 using the SQL Server 2005 unpivot operator.

2.3.1 Pivoting operations

The *pivot* and *unpivot* operators constitute an important extension to RA, which where first natively supported by SQL Server 2005. The pivot operation collapses similar rows into a single wider row adding new columns on-the-fly (Cunningham *et al.*, 2004). In a sense, this operator collapses rows to columns. Thus, it can be seen as expressing a many-to-one data transformation. Its dual, the unpivot operator transposes columns into rows. Henceforth, the discussion focuses on the unpivot operator, since this operator can be used for expressing bounded one-to-many data transformations.

In what concerns expressiveness, the unpivot operator does not increase the expressive power of RA, since, as Cunningham *et al.* (2004) admit, the unpivot operator can be implemented with multiple unions. Its semantics can be emulated by employing multiple union operations as proposed above for expressing bounded one-to-many data transformations through RA (Section 2.2).

Nevertheless, expressing one-to-many data transformations using the unpivot operator brings two main benefits comparatively to using multiple unions. First, the syntax is more compact. Figure 2.2 shows how the unpivot operator can be employed to express the bounded one-to-many data transformation of Example 1.1.1. Second, data transformations expressed using the unpivot operator are more readily optimizable using the logical and physical optimizations proposed in Cunningham *et al.* (2004).

2.3.2 Recursive queries

The expressive power of RA can be considerably extended through the use of recursion (Aho & Ullman, 1979). Although the resulting setting is powerful enough

to express many useful one-to-many data transformations, this alternative has a number of drawbacks. Recursive queries are not broadly supported by RDBMSs, they are difficult to optimize and hard to understand.

Recursive processing was addressed early and gained much attention in the study of logic query languages like LDL (Chimenti *et al.*, 1989) and Datalog (Ullman, 1988). Diverse aspects concerning the optimization of recursive queries were studied by Valduriez & Boral (1986) and Shan & Neimat (1991). Several proposals for extending SQL to handle particular forms of recursion can be found in the works of Agrawal (1988) and Ahad & Yao (1993). Despite being relatively well understood at the time, recursive query processing was not supported by SQL-92 (ISO-ANSI, 1992). As a consequence, some of the leading RDBMSs (e.g., Oracle, DB2 or POSTGRES) were in the process of supporting recursive queries when the SQL:1999 standard was released (ISO-ANSI, 1999; Melton & Simon, 2002). It turns out that these systems ended up by supporting different subsets of recursive queries with different syntaxes. Presently, the broad support of recursion constitutes a subject of debate (Pieciukiewicz *et al.*, 2005).

As explained before, the semantics of a one-to-many data transformation can be emulated by using a recursive query. Figure 2.3 presents a solution for Example 1.1.2 written in SQL:1999. The recursive query is divided into three sections. The first section is the *base* of the recursion that creates the initial result set (lines 2–8). The second section, known as the *step*, is evaluated recursively on the result set obtained so far (lines 10–18). The third section specifies the *output expression* responsible for returning the final result as a query (lines 19–20). In the base step, the first parcel of each loan is created and extended with the column `REMAMNT`, whose purpose is to track the remaining amount. Then, at each step the set of resulting rows is enlarged. All rows without `REMAMNT` already constitute a valid parcel and are not expanded by recursion. Those rows with `REMAMNT` > 0 (line 18) generate a new row with a new sequence number set to `SEQNO` $+ 1$ (line 14) and with the remaining amount decreased by 100 (line 16). Finally, the `PAYMENTS` table is generated by projecting away the extra `REMAMNT` column.

Clearly, when using recursive queries to express data transformations, the logic of the data transformation becomes hard to grasp, especially if several functions

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

Figure 2.3: Transformation of Example 1.1.2 using an SQL:1999 recursive query.

are used. Even in simple examples, like Example 1.1.2, it becomes difficult to understand how the cardinality of the output tuples depends on each input tuple. Furthermore, a great deal of ingenuity is often needed for developing recursive queries.

2.3.3 Persistent stored modules

Several RDBMSs support some form of procedural construct for specifying complex computations. This feature is primarily intended for storing business logic in the RDBMS for performance reasons or to perform operations on data that cannot be handled by SQL. Several database systems support their own procedural languages, like SQL-PL in the case of DB2 (Janmohamed *et al.*, 2005), TransactSQL in the case of Microsoft SQL Server and Sybase (Kline *et al.*, 1999), or PL/SQL in the case of Oracle (Feuerstein & Pribyl, 2005). These extensions, designated as *Persistent Stored Modules* (PSMs), were introduced in the SQL:1999 standard (Garcia-Molina *et al.*, 2002, Section 8.2). A module of a PSM can be,

among others, a procedure, usually known as *stored procedure* (SP), or a function, known as a *user defined function* (UDF).

Table functions extend the expressive power of SQL because they may return a relation. Table functions allow recursion[1] and make it feasible to generate several output tuples for each input tuple. The advantages are mainly enhanced performance and re-use (Rahm & Do, 2000). Moreover, complex data transformations can be expressed by nesting UDFs within SQL statements (Rahm & Do, 2000). However, table functions are often implemented using procedural constructs that hamper the possibilities of undergoing the dynamic optimizations familiar to relational queries.

Besides table functions, other kinds of UDFS exist, like *user defined scalar functions* (UDSFs), and *user defined aggregate functions* (UDAFs) (Jaedicke & Mitschang, 1998). Still, SQL extended with UDSFs and UDAFs may not be enough for expressing one-to-many data transformations. First, calls to UDSFs need to be embedded in an extended projection operator, which, as discussed in Section 2.2, is not powerful enough for expressing one-to-many transformations. Second, UDAFs must be embedded in aggregation operations, which can only represent many-to-one data transformations.

An interesting aspect of PSMs is that they are powerful enough to specify bounded as well as unbounded data transformations. Figure 2.4 presents the implementation of the data transformation introduced in Example 1.1.2 as a *user defined table function* (TF), as proposed by the SQL 2003 (Eisenberg *et al.*, 2004). The table function implementation written in PL/SQL has two sections: a declaration section and a body section. The first one defines the set of working variables that are used in the procedure body and the cursor `CLOANS` (lines 6–7), which will be used for iterating through the `LOANS` table. The body section starts by opening the cursor. Then, a **loop** and a **fetch** statement are used for iterating over `CLOANS` (lines 10–11). The loop cycles until the **fetch** statement fails to retrieve more tuples from `CLOANS`. The value contained in `ACCTVALUE` is loaded into the working variable `REMAMNT` (line 12). The value of this variable will be later decreased in parcels of 100 (line 19). The number of parcels is controlled by the guarding condition `REMAMNT>0` (lines 14 and 22). An inner loop is used

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

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

Figure 2.4: Transformation of Example 1.1.2 using an Oracle PL/SQL table function.

to form the parcels based on the value of REMAMNT (lines 14–21). A new parcel row is inserted in the target table PAYMENTS for each iteration of the inner loop. The tuple is generated through a **pipe row** statement that is also responsible for padding the value of ACCTVALUE with zeroes (lines 16–17 and 23–24). When the inner loop ends, a last **pipe row** statement is issued to insert the parcel that contains the remainder. The details concerning the creation of the row and table types PAYMENTS_ROW_TYPE and PAYMENTS_TABLE_TYPE are not presented.

The main drawback of PSMs is that they use a number of procedural constructs that are not amenable to optimization. Moreover, there are no elegant solutions for expressing the dynamic creation of tuples using PSMs. One needs

to resort to intricate **loop** and **pipe row** statements (or **insert into** statements in the case of a stored procedure) as shown in Figure 2.4. From the description of Example 1.1.2, it is clear that a separate logic is used to compute each of the attributes. Nevertheless, in the PL/SQL code, the computation of `ACCTNO` is coupled with the computation of `AMOUNT`. Thus, the logic to calculate `ACCTNO` is duplicated in the code. This makes the code maintenance difficult and the code itself hard to optimize.

2.4 Data Restructuring Languages

The increasing adoption of semi-structured data spurred new languages that address the problem of querying, integrating and transforming semi-structured data (Suciu, 1998). In the last years, the Web has been promoting data exchange and storage using XML, a language that can be used to represent semi-structured data.

Semi-structured data objects are mapped into labeled trees to represent both data and schema, like the *object exchange model* (OEM), an intermediate model championed by the TSIMMIS data integration system (Papakonstantinou *et al.*, 1996). Data transformations in these languages consist of translating the specific source trees into appropriate target trees. A transformation specification is constituted of a set of rules, each representing a part of the translation. A rule consists of a head and a body. The rule body includes a pattern and a Boolean predicate that, together, encode a query over the nodes of the intermediate model. The pattern collects instances and the predicate filters them. The head of the rule usually encodes a translation function that specifies how the instances matched by the body are to be restructured.

2.4.1 Semi-structured data restructuring languages

Several languages that have been proposed for querying semi-structured data can be envisioned as RA extensions for handling objects represented as trees or graphs. Lorel is a query language for semi-structured data that takes the form of an extension of SQL with path expressions (Abiteboul *et al.*, 1997). UnQL

has an expressive power similar to RA, with the particularity that queries are evaluated over edge-labeled graph structures than can be cyclic (Buneman *et al.*, 1996). UnQL queries may return unions of graphs simulating bounded one-to-many transformations. However, unbounded data transformations are not possible to express, since it is not possible to return a graph whose size depends dynamically on data contained in the edges.

YATL is a language for specifying the integration and transformation of semi-structured data represented as lists of trees (forests) in the YAT prototype (Cluet *et al.*, 1998). In YATL, the generation of new object identifiers is restrained through the concept of *safe recursive* specifications to avoid potentially dangerous computations (Cluet & Siméon, 1997). Since certain classes of recursive specifications, in particular those involving recursive *oid* generation, are limited, the language is not powerful enough for expressing the dynamic creation of data instances required by unbounded one-to-many data transformations.

Strudel is a system for specifying and generating data-intensive Web sites (Fernandez *et al.*, 1998). This system comprises two languages: *(i)* STRUQL, a declarative rule-based query and transformation language for semi-structured data and *(ii)* a template language for specifying the HTML output. A query comprises two identifiable sections, one that is responsible for integrating multiple data sources and another that performs the transformations. Recursion is introduced sparingly in STRUQL through an operator for computing the transitive closure (Fernandez *et al.*, 1998). It can be argued that unbounded one-to-many transformations cannot be expressed, since there is no way of creating a web page with a number of linked pages determined by the value of a source attribute.

2.4.2 XML data transformation languages

Within the W3C XSL recommendations initiative, several languages have been proposed for transforming XML documents. Perhaps the most noteworthy is XQuery (W3C, 2006), which has become the standard for querying XML documents. XQuery is declarative language that can be used to specify transformations of data represented as XML documents. One-to-many data transformations can be represented in XQuery since the language is Turing complete.

Other two functional languages XPath have been proposed (Clark & DeRose, 1999) and XSLT (Clark, 1999). These aim at querying and transforming XML documents, respectively. XPath evaluates a regular path expression over a document tree and returns a forest as the result. Chamberlin noted that XPath can only select existing nodes (Chamberlin, 2002, p. 604). Hence, XPath can only return nodes that already exist in the document. An extension of XPath named XQL (J. Robie, 1998) was proposed as a natural extension of XPath pattern syntax for joining elements of XML documents. Its deep return operator '??' is used to flatten a node. Nevertheless, it is not possible to express transformations that generate new nodes whose quantity is based on the contents of a source node. XSLT employs sets of rules for transforming elements obtained through XPath queries into new XML documents or other output formats. Concerning its expressive power, one distinguishing feature of XSLT is that, since it allows recursion, it becomes Turing complete.

Many languages for querying XML, benefited from advances of semi-structured data querying. XML-QL is a language for querying, transforming and integrating XML data (Deutsch *et al.*, 1998, 1999). This language is based on UnQL graph patterns. However, it is not possible to create output trees dynamically since the construct clause is evaluated once for each tree element returned by the where clause. The Quilt language aims at querying and integrating heterogeneous information sources (Chamberlin *et al.*, 2000). This language incorporates concepts of Lorel (Abiteboul *et al.*, 1997), YATL (Cluet & Siméon, 1997) as well as XPath and XQL. New nodes can be generated through an *element constructor* expression. However, the number of output nodes generated is bounded by the number of nodes of the source model and by the size of the query. Quilt can be envisioned as the precursor of the XQuery language.

2.5 Schema Mapping Tools

In order to perform data transformation or data integration, it is necessary to establish a set of *schema mappings*, describing the relationships among the elements of the different schemas (Madhavan *et al.*, 2002). In data transformations, schema

mappings describe how elements of one schema are mapped to the other, while in data integration they describe how elements of source schemas are combined.

Establishing such schema mappings is often a laborious and complex task exacerbated by the frequent lack of documentation. Over the years several research tools like, Clio (Miller *et al.*, 2001), COMA (Madhavan *et al.*, 2001), Cupid (Do & Rahm, 2002), GLUE (Doan *et al.*, 2002), MOMIS (Castano & Antonellis, 1999) and TranScm (Milo & Zhoar, 1998), have been proposed that automate to some degree the discovery of schema mappings.

Schema mappings take the form of inter-schema constraints. Since they are geared toward mapping structure (i.e., schema), they are not powerful enough for deriving many useful data (i.e., instance) transformations. As noted by Koch (2001), since the semantics of data can be considerably different in two models, bridging them may involve complex data transformations that cannot be expressed or derived from schema mappings. Coarsely speaking, schemas do not describe in detail how data instances are represented. Thus, mappings established between a source and a target schema are not powerful enough to represent complex instance transformations. Consider, for example, the problem of mapping between salaries relations where each source tuple represents one year and the target uses one tuple for each month. This transformation cannot be expressed by a schema mapping alone, since neither the source nor the target schema represents how data is aggregated.

TransScm is a schema matching and transformation system (Milo & Zhoar, 1998). It is based on the idea that schema matching can be used to perform data translation. More specifically, it assumes that both source and target schemas are given as input, and suggests data translation to be based on matching rules specified among the two schemas. Although TransScm rules may specify data translations, the limits in the use of recursion do not allow the creation of new objects based on an attribute's value and, consequently, one-to-many instance transformations cannot be expressed.

Notably, in Clio schema mappings are expressive enough to induce *select-project-join* queries (Miller *et al.*, 2001). These queries are compiled to perform data transformations from schema mappings. Recent work on Clio proposed to perform the transformation of data instances from a source schema into a target

schema based on source-to-target schema dependencies (Fagin *et al.*, 2003). However, their semantics of *universal solutions* is not powerful enough to entail the class of one-to-many transformations we propose to tackle. COMA (Madhavan *et al.*, 2001), GLUE (Doan *et al.*, 2002) and TranScm (Milo & Zhoar, 1998) represent mappings through simple assertions established among schema elements. These assertions must extended by the user before being used in data transformations. For example, TransScm leaves to the user the specification of non-standard transformations.

Building on similar ideas, Rifaieh & Benharkat (2002) propose deriving data transformations queries automatically from schema mappings. They aim at using RDBMSs as transformation engines for data warehousing. However, the mapping language they propose can only represent conjunctive queries. Data transformations that consist of aggregations or one-to-many data transformations cannot be expressed.

2.6 Data Integration Tools

Data integration is realized through a virtual homogeneous system with a single integrated schema , also known as *mediated schema*, against which, queries are evaluated (Batini *et al.*, 1986). Evaluating a query against an integrated schema involves locating the data sources, possibly using different query languages, and then combining the results.

The main approaches to solve the problem of efficiently answering queries over multiple heterogeneous data sources are *federated databases* (Sheth & Larson, 1990) and *mediators* (Wiederhold, 1992). Currently, all the leading RDBMS vendors are suppling data integration solutions through federation, see, e.g. (Haas *et al.*, 2002). Concerning mediators, an initial upsurge of research prototypes, like Information Manifold (Kirk *et al.*, 1995), Squirrel (Zhou *et al.*, 1996) and TSIMMIS (Garcia-Molina *et al.*, 1997), provided the concepts that allowed the emergence of commercial data integration tools, such as Business Objects Data Integrator[1] or BEA Liquid Data[2].

[1] http://www.businessobjects.com/products/dataintegration
[2] http://www.bea.com

In those RDBMSs that support federation, the data sources can be transformed through RA queries and the RDBMS extensions discussed in Section 2.3. Hence, these systems support bounded and unbounded one-to-many transformations. The languages of mediator systems have an expressive power similar to RA with extensions that are not powerful enough to represented unbounded one-to-many data transformations. Squirrel's ISL allows view definitions that have a similar power to relational algebra extended with aggregation and user supplied algorithms are provided for addressing object fusion. The TSIMMIS MSL language has an expressive power comparable to Datalog without recursion.

2.7 ETL and Data Cleaning tools

Data cleaning and ETL are two intimately tied activities (Lomet & Sarawagi, 2000). ETL often requires data cleaning operations to enhance the quality of data loaded into data warehouses; and the implementation of data cleaning transformations requires data to be extracted and loaded into a temporary repository. This relationship is underscored by the large number of tools that handle both ETL and Data Cleaning (Barateiro & Galhardas, 2005).

Although the importance of ETL has raised in recent years (Kimball & Caserta, 2004), the Express prototype of Shu *et al.* (1977), can be considered the first ETL tool. The architecture of Express is akin to that of an ETL tool where the file layouts are compiled into file reader and loader programs for extraction and load. The data transformations are specified through data restructuring queries that are compiled into PL/1 programs. The data transformation language used by Express resembles SQL augmented with specific operators to work with hierarchical data (Shu *et al.*, 1975). The expressivity of the data transformation language is similar to that of relational query languages. Therefore, unbounded one-to-many data transformations cannot be expressed.

References to ETL as a research subject are relatively recent. Ajax (Galhardas *et al.*, 2000), Potter's Wheel (Raman & Hellerstein, 2001) and ARTKOS (Vassiliadis *et al.*, 2000) are the the first research systems to explicitly address ETL. The former two are data cleaning tools, an activity that is intimately blended with ETL.

```
1: data PAYMENTS(keep=ACCTNO AMOUNT SEQNO)
2:   set LOANS(rename=(ACCT=ACCTNO))
3:   SEQNO = 1;
4:   REMAMNT = AM;
5:   do while (REMAMT > 0);
6:     if (REMAMNT > 100) then
7:       AMOUNT = 100;
8:     else
9:       AMOUNT = REMAMNT;
10:    REMAMNT = REAMNT - 100;
11:    output;
12:    SEQNO + 1;
13:  end
```

Figure 2.5: Transformation of Example 1.1.2 using an SAS Data Step, showing the use of an ETL tool for performing one-to-many data transformations.

Both Potter's Wheel and Ajax (Galhardas *et al.*, 2001) have proposed operators for expressing one-to-many data transformations for data cleaning purposes. Potter's Wheel (Raman & Hellerstein, 2001) is a tool for discrepancy detection that allows the user to successively apply simple schema and data transformations. The authors acknowledge that one-to-many data transformations can be encoded using the fold operator. However, this operator can only express bounded one-to-many data transformations, since there has to be a bound k on the number of output tuples known a-priori (Raman & Hellerstein, 2000). Ajax proposes the map operator to express bounded and unbounded one-to-many data transformations (Galhardas, 2001).

The work of Amer-Yahia & Cluet (2004) uses a specialized middleware to perform data transformations of object-oriented database though an object-oriented extension to RA that features a specialized map operator for data transformations. However, their language is not powerful enough to express data transformations that produce a number of tuples determined by the value of an input object's attribute. Cui & Widom (2001) identify many-to-one data transformations, like aggregations, together with one-to-many data transformations to be the main classes of data transformations in ETL scenarios for Data Warehousing.

Many commercial ETL tools do not use declarative formalisms, relying instead on procedural scripting languages that lack a formal foundation. To better

understand the issue, consider the code shown in Figure 2.5, that presents an implementation of Example 1.1.2. This implementation uses the component of the SAS system which is responsible for data warehouse construction (Refaat, 2006). In SAS, iterating on the input table `LOANS` and materializing the results are implicit operations. The assignment of the account number is performed by renaming `ACCT` as `ACCTNO` (line `2`). Then, two auxiliary variables used for populating each new target tuple are declared and initialized (lines `3`–`4`). The **do while** loop is used to produce the output rows (lines `5`–`13`). The output values are loaded into the corresponding attributes (lines `6`–`10`) and a new parcel is generated through the **output** statement (line `11`). However, such procedural specification hampers the introduction of even simple optimizations. For example the push of simple projection over a defined transformation cannot be expressed through algebraic rewriting.

The languages used by many ETL tools either provide a very large number of operators —e.g., Sagent[1]—, for transforming data or only a small set of operators —e.g., FileAid/Express[2]. The first group of tools is not easy to use, given the large number of abstractions that a programmer must handle. In the second group of tools, complex transformation logics must be developed as external ad-hoc functions through programming interfaces. As a result, programs that handle rich transformation semantics become complex and difficult to debug. This situation often arises when one-to-many data transformations are required.

2.8 Conclusions

This chapter analyzed several alternatives for implementing one-to-many data transformations. First, the discussion was organized around the two sub-classes of one-to-many data transformations, bounded and unbounded data transformations. This arrangement is interesting, since it uses the expressivity of RA as a boundary. Second, the different alternatives for expressing one-to-many data transformations were studied. A starting conclusion is that RA is only capable of expressing bounded one-to-many data transformations. The extensions to

[1] http://www.sagent.com
[2] http://www.compuware.com/products/fileaid/express.htm

RA supported by RDBMSs are not general enough to support one-to-many data transformations. Although bounded data transformations can be expressed by combining unions and selections, unbounded data transformations require more advanced constructs, such as SQL:1999 recursive queries and table functions introduced in the SQL 2003 standard. However, these are not yet supported by many RDBMSs.

Third, several languages for querying, integrating and transforming semi-structured data were reviewed. Some of these languages provide some form of structural recursion to unwind the input elements represented as trees (or graphs). These features can be seen as a form of unbounded one-to-many data transformations, since the number of output elements can be determined by the depth of the input element and not by the size of the query. Nevertheless, besides XSLT and XQuery, none of these languages is powerful enough to specify a one-to-many data transformation on which the number of output elements is determined by the value of an attribute of an input element. In fact, the expressivity of some languages, like YATL, is restricted to avoid potentially dangerous specifications (that may result in diverging computations).

Several tools that support data transformations were also considered. The data transformations that can be expressed through schema mappings are less expressive than RA, being limited to bounded queries. The languages of data integration tools are geared toward expressing views over multiple data sources, mostly based on RA and Datalog without recursion. None of the covered tools supports unbounded one-to-many data transformations despite its usefulness. Research ETL tools support both bounded and unbounded one-to-many transformations and acknowledge the need for powerful data transformation operators. Commercial systems usually support natively bounded one-to-many data transformations, but unbounded transformations are often developed through external functions or proprietary procedural scripts that hamper optimization. The languages supported by these tools are often not powerful enough to represent complex data transformations. Typically, complex transformations are handled by ad-hoc programs coded outside the tools.

Another way of encoding data transformations consists of using a general purpose language, like writing a Java program that connects to the RDBMS

through JDBC or writing a Perl script. The use of a general purpose language is hindered by two factors. First, these languages have a procedural nature that contrasts with the declarative nature of query languages. This characteristic turns data transformations difficult to understand and maintain.

Thus, it can be concluded that there is no general solution for expressing one-to-many data transformations. None is declarative, optimizable and at the same time expressive enough to represent one-to-many data transformations. Most data transformation solutions are simply not expressive enough for representing one-to-many data transformations. They are either based on a procedural formalism, which difficults optimization, or require one-to-many data transformations to be entangled as external programs.

Such hindrances can be coarsely minimized by supporting one-to-many data transformations concisely through a specialized operator. This thesis proposes one such operator, the *data mapper*, which extends RA for expressing one-to-many data transformations. Since data transformations are often performed by RDBMSs, or by tools and languages that are also to based on RA to various extents, the new operator is a general solution to express one-to-many data transformations in these systems. Another advantage is that it can be embedded in expressions having standard relational operators and be logically and physically optimized.

Chapter 3

The Mapper Operator

This chapter presents the mapper operator. First, the formal definition of the operator is given and then several important properties are studied. Then, the expressive power of the mapper operator and traditional relational algebra operators are compared. It is formally demonstrated that mappers subsume the rename, projection and selection unary relational operators. In the sequel, the expressive power of relational algebra extended with the mapper operator is considered. Finally, a straightforward extension to the SQL *select* block to handle mappers is proposed.

3.1 Introduction

Currently, the frameworks used for data transformation tasks do not provide adequate support for expressing one-to-many data transformations. In most of them, one-to-many data transformations are either tedious to write or impossible to express directly. The root cause seems to lie in that one-to-many data transformations were not accounted as first class citizens.

The difficulties in handling one-to-many data transformations can be addressed by means of a specialized operator. Herein, one such operator, named *data mapper*, is proposed as an extension to Relational Algebra (RA). The mapper enables the concise expression of bounded and unbounded data transformations. A mapper μ_F is defined as a unary operator, that takes a relation instance of a

given relation schema as input (source schema) and produces a relation instance of another relation schema as output (target schema)[1].

Like generalized projection and aggregation, the mapper operator relies on arbitrary external functions. It is parameterized by a list of functions. Each function, designated as a *mapper function*, expresses part of the intended data transformation by producing a specific part of the result. When applied to a tuple, mapper functions may produce *several values* as output. The output values are then combined to produce multiple output tuples.

3.2 Formalization

This section, starts by introducing some preliminary notation used throughout the thesis following Atzeni & de Antonellis (1993) and Abiteboul *et al.* (1995). Then, mapper functions are discussed in detail and the semantics of the mapper operator is presented. Examples of how mappers can be used express one-to-many data transformations are supplied.

3.2.1 Preliminaries

A domain D is a set of atomic values. A set $\mathcal{D}$ of domains, a set $\mathcal{A}$ of attribute names, together with a function $Dom : \mathcal{A} \rightarrow \mathcal{D}$ that associates domains to attributes are assumed. The natural extension of this function to lists of attribute names: $Dom(A_1, ..., A_n) = Dom(A_1) \times ... \times Dom(A_n)$ will be represented as Dom.

A *relation schema* R consists of a list $A = A_1, ..., A_n$ of distinct attribute names represented by $R(A_1, ..., A_n)$, or simply $R(A)$. The quantity n is known as the *degree* of the relation schema. Its domain is defined by $Dom(A)$. A *relation instance* (or relation, for short) with schema $R(A_1, ..., A_n)$ is a finite set $r \subseteq Dom(A_1) \times ... \times Dom(A_n)$; represented as $r(A_1, ..., A_n)$, or simply $r(A)$. Each element t of r is called a *tuple* or *r-tuple* and can be regarded as a function that associates a value of $Dom(A_i)$ with each A_i; this value is denoted by $t[A_i]$. Given

[1]The symbol μ was also used to represent the *nest* operator of *Nested Relational Algebra* (Jaeschke & Schek, 1982; Thomas & Fischer, 1986); the mapper operator is not related to nest.

a list $B = B_1, ..., B_m$ of distinct attributes in $A_1, ..., A_n$, $t[B]$ denotes the tuple $\langle t[B_1], ..., t[B_m] \rangle$ in $Dom(B)$.

The *relational algebra* as introduced by Codd (1970) will be used. The basic operations considered are *union*, *difference*, *Cartesian product*, *projection* (π_X, where X is a list of attributes), *selection* (σ_C, where C is the selection condition) and *renaming* ($\rho_{A \to B}$, where A and B are lists of attributes).

3.2.2 Mapper functions

A *mapper function* enables the expression of part of the data transformation focused on one or more attributes of the target schema. Although the idea is to apply mapper functions to the tuples of a source relation, it may happen that some of the attributes of the source schema are irrelevant for the envisaged data transformation. The explicit identification of the attributes that are relevant is then an important part of a mapper function. Mapper functions are formally defined as follows:

DEFINITION 3.1: *A* mapper function f_A *is a triple* $\langle A, B, f \rangle$ *where* A*, a non-empty list of distinct attributes, defines the output attributes,* B*, also a list of distinct attributes, identifies the relevant input attributes, and the function* f*:*$Dom(B) \to \mathcal{P}(Dom(A))$ *is a computable function (if* B *is empty, then* f *is just a set). The function* f_A *is said to be an* $A-$*mapper function. Let* t *be a tuple of a relation instance* $s(X_1, ..., X_n)$ *s.t. all the attributes in* B *are also in* $X_1, ..., X_n$*. The notation* $f_A(t)$ *will be used to represent the application of the underlying function* f *to the tuple* t*, i.e.,* $f(t[B])$*.*

In this way, a mapper function describes how a specific part of the target data can be obtained from the source data, simultaneously defining part of the target schema. The intuition is that each mapper function establishes how the values of a group of attributes of the target schema can be obtained from the attributes of the source schema. The key point is that, when applied to a tuple, a mapper function produces a set of values, rather than a single value.

The function f_A shall be used freely use to denote both a mapper function $\langle A, B, f \rangle$ and the function f itself, omitting the list B whenever its definition

is clear from the context. Moreover, $Dom(f_A)$ will be used to refer to list B. This list should be regarded as the list of the source attributes declared to be relevant for the part of the data transformation encoded by the mapper function. Notice, however, that even if f_A is a constant function, it may be defined as being dependent on all the attributes of the source schema. The relevance of the explicit identification of these attributes will be later clarified, when the algebraic optimization rules for projections are presented (see Section 4.2).

Certain classes of mapper functions enjoy properties that enable the optimizations of algebraic expressions containing mappers (see Section 4.1). Mapper functions can be classified according to:

i) the number of output tuples they may produce;

ii) the number of output attributes.

Mapper functions that produce singleton sets, i.e., $\forall(t \in Dom(X))\ |f_A(t)| = 1$, are designated *single-valued mapper functions*. In contrast, mapper functions that produce multiple elements are said to be *multi-valued mapper functions*. Concerning the number of output attributes, mapper functions with one output attribute are called *single-attribute*, whereas functions with many output attributes are called *multi-attribute*.

The single-valued mapper functions $\langle A, A, f\rangle$ s.t. $f(t) = \{t\}$ are designated as *identity mapper functions*. Also interesting is the class of the single-valued mapper functions $\langle A, B, f\rangle$ s.t. $Dom(B) = Dom(A)$ and $f(t) = \{t\}$. These are called *renaming mapper functions*, given that they only establish a transformation of the schema. Finally, a *constant mapper function* is a mapper function $\langle A, [], f\rangle$ s.t. $f(t) = c$, for every $t \in Dom(B)$ and some $c \in \mathcal{P}(Dom(A))$.

As mentioned before, a mapper operator is parameterized by a list of mapper functions.

DEFINITION 3.2: *A list $F = f_{A_1}, ..., f_{A_m}$ of mapper functions is said to be* proper *for transforming the data of a relation $s(X_1, ..., X_n)$ iff, for $1 \leq j \leq m$, the attributes included in the A_j lists are all distinct.*

In other words, F is proper if it specifies, in a unique way, how the values of the schema $Y = A_1 \cdot ... \cdot A_m$ —the target schema— are produced ('$\cdot$' denotes polymorphic concatenation). The informal idea is that a set of mapper functions is proper for transforming the data from the source to the target schemas if it specifies unambiguously how the values of every attribute of the target schema are produced.

3.2.3 Semantics of the mapper operator

The mapper operator μ_F puts together the data transformations of the input relation defined by the mapper functions in F. Given a tuple s of the input relation, $\mu_F(s)$ consists of the tuples t of $Dom(Y)$ that, for each list of attributes A_i, associate values in $f_{A_i}(s)$. Formally, the mapper operator is defined as follows:

DEFINITION 3.3: *Given a relation $s(X_1, ..., X_n)$ and a proper list of mapper functions $F = f_{A_1}, ..., f_{A_m}$, the* mapper *of s with respect to F, denoted by $\mu_F(s)$, is the relation instance with schema $Y = A_1 \cdot ... \cdot A_m$ and the set of tuples defined by*

$$\mu_F(s) \stackrel{\text{def}}{=} \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \ t[A_i] \in f_{A_i}(u)\}$$

As mentioned before, this new operator relies on the use of arbitrary computable functions that are external to the resulting extension of the relational algebra. In this sense, the mapper operator resembles the extension to RA proposed by Klug (1982) for the computation of aggregates. The mapper may be also be defined in terms of partial functions, i.e., the underlying functions do not have to be defined for all values of their source set. It follows from Definition 3.3 that if $f_{A_i}(t)$ is undefined for some $f_{A_i} \in F$ and $t \in s$, then so is $\mu_F(s)$.

The set of admissible functions can be further constrained, if required. As it will be later explained in Section 3.4, for some specific classes of admissible functions, the integration of the mapper operator with existing query execution processors is easier.

In order to illustrate this new operator, Example 1.1.2 is revisited.

EXAMPLE 3.2.1: *The requirements presented in Example 1.1.2 can be described by the mapper $\mu_{acct,amt}$, where acct is an `[ACCTNO]`-mapper function with domain*

ACCT that returns a singleton with the account number ACCT properly left padded with zeroes and amt is the [AMOUNT,SEQNO]-mapper function with domain AM s.t., amt(am) is given by

$$\{(100, i) \mid \ 1 \leq i \leq (am/100)\} \ \cup \ \{(am\%100, (am/100)+1) \ \mid \ am\%100 \neq 0\}$$

where / and % have been used to represent the integer division and modulus operations, respectively.

For instance, if t is the source tuple (901, 250.00), the result of evaluating amt(t) is the set {(100, 1), (100, 2), (50, 3)}. Given a source relation s including t, the result of the expression $\mu_{acct,amt}(s)$ is a relation that contains the set of tuples {⟨'0901', 100, 1⟩, ⟨'0901', 100, 2⟩, ⟨'0901', 50, 3⟩}.

Example 3.2.2 describes a real world application of the mapper operator that encodes a cleaning step of a data cleaning transformations used to clean CiteSeer input data with the Ajax tool (Galhardas *et al.*, 2000).

EXAMPLE 3.2.2**:** *Consider the a source relation containing dirty data about scientific articles CITEDATA[AUTHORS, TITLE, EVENTNAME, LOCATION, PUBDATE] taken from the CiteSeer database. This information needs to be transformed into the relation EVENTS[NAME, TITLE, EVENT, COUNTRY, CITY, YEAR] that contains data about be used to support the generation of different types of reports. The attributes are mapped as follows:*

1) *The target attribute NAME is the author name. Each author's name is obtained after normalizing the source attribute AUTHORS that consists of a string with author names in different formats (e.g. with and without abbreviations, with and without salutation, using different types of separators, etc.).*

2) *The target attribute TITLE is obtained by normalizing the source attribute TITLE by performing adequate capitalization taking into account punctuation and adjusting spacing.*

3) *Attribute EVENT is obtained by normalizing the attribute EVENTNAME. The associated transformation is responsible for performing several common ab-*

breviation expansions (e.g. "Int'l" to "International" or "Proc" to "Proceedings"), detecting the different spellings for the same event (e.g. "SIGMOD" and "International Conference on Management of Data") and removing superfluous punctuation.

4) *The attributes* `CITY` *and* `COUNTRY` *are both mapped from the attribute* `LOCATION`*. Some locations are only given the city name and that the order of appearance of the the city and the the country can be different.*

5) *The attribute* `YEAR` *is derived from the source attribute* `DATE` *containing dates in a variety of formats.*

The transformation specified in Example 3.2.2 can be implemented by means of a mapper $\mu_{name,title,event,loctn,year}$, where the $authors_{\texttt{NAME}}$ is a one-to-many mapper function that produces the different author names from the character string denoted by the attribute `NAME`. The mapper function $loctn_{\texttt{CITY,COUNTRY}}$ is a single-valued and multi-attribute function. Finally, the functions $title_{\texttt{CITY}}$, $event_{\texttt{EVENTNAME}}$ and $year_{\texttt{YEAR}}$ are single-valued functions that map the attributes `TITLE`, `EVENTNAME`, and `YEAR`, respectively.

3.3 Properties of Mappers

Notice that the mapper operator admits a more intuitive definition in terms of the Cartesian product of the sets of tuples obtained by applying the underlying mapper functions to each tuple of the input relation. More concretely, the following proposition holds:

PROPOSITION 3.1: *Given a relation $s(X_1, ..., X_n)$ and a proper list of mapper functions $F = f_{A_1}, ..., f_{A_m}$,*

$$\mu_F(s) = \bigcup_{u \in s} f_{A_1}(u) \times ... \times f_{A_m}(u).$$

PROOF

$$\begin{aligned}\mu_F(s) &= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \ t[A_i] \in f_{A_i}(u)\} \\ &= \bigcup_{u \in s} \{t \in Dom(Y) \mid \forall 1 \leq i \leq m \ t[A_i] \in f_{A_i}(u)\} \\ &= \bigcup_{u \in s} f_{A_1}(u) \times ... \times f_{A_m}(u)\end{aligned}$$

This alternative way of defining $\mu_F(s)$ is also important because of its operational flavor, equipping the mapper operator with a *tuple-at-a-time* semantics. When integrating the mapper operator with existing query execution processors, this property plays an important role, because it means the mapper operator admits physical execution algorithms that favor pipelined execution (Graefe, 1993).

The algorithm that computes the data transformations through mappers just needs to compute the Cartesian product in Proposition 3.1. Obviously, this algorithm relies on the computability of the underlying mapper functions and builds on concrete algorithms for computing them. Furthermore, the fact that the calculation of $\mu_F(s)$ can be carried out tuple by tuple clearly entails the monotonicity of the mapper operator.

PROPOSITION 3.2: *The mapper operator is* monotonic*, i.e., for every pair of relations* $s_1(X)$ *and* $s_2(X)$ *if* $s_1 \subseteq s_2$*, then* $\mu_F(s_1) \subseteq \mu_F(s_2)$*.*

PROOF

$$\begin{aligned}\mu_F(s_1) &= \{t \in Dom(Y) \mid \exists u \in s_1 \ \forall 1 \leq i \leq m \quad \text{s.t. } t[A_i] \in f_{A_i}(u)\} \\ &\quad \text{by hypothesis } s_1 \subseteq s_2 \\ &\subseteq \{t \in Dom(Y) \mid \exists u \in s_2 \ \forall 1 \leq i \leq m \quad \text{s.t. } t[A_i] \in f_{A_i}(u)\} \\ &\subseteq \mu_F(s_2)\end{aligned}$$

Mapper operators whose mapper functions are all single-valued admit an equivalent mapper with only one mapper function. Applying one mapper function to each input element mimics the behavior of the map operator of functional programming languages.

PROPOSITION 3.3: *Given a set* $F = \{f_{A_1}, ..., f_{A_m}\}$ *of single-valued mapper functions proper for transforming* $S(X)$ *into* $T(Y)$*. For every mapper* μ_F*, there exists an equivalent mapper with only one* $Y-$*mapper function* g_Y*, s.t.,* $\mu_F = \mu_{\{g_Y\}}$.

PROOF It suffices to show how to obtain g_Y. Consider the mapper function $g_Y[Y_i] = f_{A_i}$, for every $1 \leq i \leq m$. The result is obtained by juxtaposition of the values produced by each function $f_{A_i} \in F$.

This proposition states that a mapper comprising only single-valued functions can be compiled to a mapper using only one single function. This definition is interesting because in can serve as the basis for simple implementations of the mapper operator.

3.4 Normal Forms

As defined in Definition 3.2, a list of mapper functions F is proper for transforming the data of a given relation only if the subsets of attributes produced by any two different mapper functions in F do not overlap.

In general, a data transformation can be achieved through different lists of functions. Consider, for instance, the `[ACCTNO,AMOUNT,SEQNO]`-mapper function named *payments* with domain `[ACCT,AM]` that yields installment amounts jointly with the transformed account numbers. Clearly, the list of proper mapper functions $F = acct, amt$ defined in Example 3.2.1 is equivalent to the single element list $G = payments$, with respect to the data transformation they specify. However, algebraic expressions containing μ_F offer more opportunities for optimization than expressions containing μ_G. Compared to G, the list F can be regarded as being reduced compared to G. In a similar way, mapper functions may use dispensable input attributes. Consider $acct'$ to be a mapper function with domain `[ACCT,AM]`. Then, the list of functions F can be compared with the list of functions $H = acct', amt$ where $acct'$ only differs from $acct$ in the domain. Given that H includes one mapper function with a domain larger than it is required, F can be regarded as being in a more reduced form than H.

In fact, the list F is what will be henceforth designated a *normal form*, because it cannot be reduced in a sense that is made precise below.

DEFINITION 3.4: *Let $S(X_1, ..., X_n)$ be a fixed relation schema. The* reduction relationship *between lists of mapper functions proper for transforming the data of relations with schema $S(X_1, ..., X_n)$, represented as $\longrightarrow$, is the greatest transitive relationship satisfying the following constraints:*

1) *if $[f_1, ..., \langle A, B_f, f\rangle, ..., f_m] \longrightarrow [f_1, ..., \langle A, B_g, g\rangle, ..., f_m]$ then the list of attributes B_g is strictly a sublist of B_f and $f(t) = g(t)$, for every $t \in Dom(X)$.*

2) *if $[f_1, ..., \langle A, B, f\rangle, ..., f_m] \longrightarrow [f_1, ..., \langle A_1, B_1, g_1\rangle, \langle A_2, B_2, g_2\rangle, ..., f_m]$ then B_1 and B_2 are sublists of B, and a permutation ϵ exists such that $A = \epsilon(A_1 \cdot A_2)$ and $f(t) = \epsilon(g_1(t) \times g_2(t))$, for every tuple $t \in Dom(X)$.*

Intuitively, a list of mapper functions can be reduced if one of its mapper functions either includes superfluous attributes in its domain or defines a transformation of data that can be decomposed, that is, expressed as a Cartesian product of two functions:

DEFINITION 3.5: *A mapper μ_F is in* normal form *if there does not exist a list of mapper functions G s.t. $F \longrightarrow G$, i.e., if F cannot be reduced.*

From a practical point of view, a mapper that is not in the normal form presents a number of limitations. To begin with, the co-existence of multiple independent functions (that produce distinct target attributes) nested within the same mapper function, limits the choice of physical execution algorithms. For instance, consider using caching for the most expensive functions. If an expensive function is implemented together with an inexpensive one in one single function, it may not be possible to apply this algorithm, as it may not be feasible to decide at compile time which is the expensive function. Another important aspect is the number of optimization opportunities that may arise in expressions involving mappers: the opportunities for applying optimizations in Section 4.1 increase as the mapper operators involved are closer to normal forms.

From a software engineering point of view, trying to maintain an implementation where the logic of several functions is bundled into fewer functions is also undesirable. It violates a desirable property of software artifacts which is *high cohesion*. The notion of normal form characterizes a principled way to verify

whether the specification of a mapper together with its functions has this property.

3.5 Expressive Power of Mappers

Concerning the expressive power of the mapper operator, two important questions are addressed. First, the expressive power of relational algebra (RA) is compared with its extension by the set of mapper operators, henceforth designated as *M-relational algebra*, or simply *MRA*. Second, the simulation of standard relational operators by a mapper operator is investigated.

MRA is more expressive than standard RA. The expressive power of mapper operators comes from being allowed to use arbitrary computable functions. In fact, the class of mapper operators of the form μ_f, where f is a single-valued function, is computationally complete. This implies that MRA is computationally complete and, hence, MRA is not a query language like standard RA.

The question that naturally arises is whether MRA is more expressive than the relational algebra with a generalized projection operator π_L where the projection list L has elements of the form $Y_i \leftarrow f(A)$, where A is a list of attributes in $X_1, ..., X_n$ and f is a function involving arithmetic operations only (Silberschatz *et al.*, 2005).

With generalized projection, it becomes possible to define arithmetic computations to derive the values of new attributes. Still, there are MRA-expressions whose effect is not expressible in when extended with the generalized projection operator, even when considering any computable function. The latter shall be designated as *RA-gp*.

The additional expressive power results from mapper operators using functions that map values into sets of values, becoming able to produce a set of tuples from a single tuple. For some multi-valued functions, the number of tuples that are produced depends on the specific data values of the source tuples and does not even admit an upper-bound.

Consider, for instance, a database schema with relation schemas `S(NUM)` and `T(NUM, IND)`, s.t. the domain of `NUM` and `IND` is the set of natural numbers. Let s

be a relation with schema S. The cardinality of $\mu_{[f]}(s)$, where f is a [NUM,IND]-mapper function s.t. $f(n) = \{\langle n, i\rangle : 1 \leq i \leq n\}$, does not (strictly) depend on the cardinality of s. Instead, it depends on the values of the concrete $s-$tuples. For instance, if s is a relation with a single tuple $\{\langle x\rangle\}$, the cardinality of $\mu_{[f]}(s)$ depends on the value of x and does not have an upper bound.

This situation is particularly interesting because it cannot happen in RA-gp.

PROPOSITION 3.4: *For every expression E of the relational algebra RA-gp, the cardinality of the set of tuples denoted by E admits an upper bound defined simply in terms of the cardinality of the atomic sub-expressions of E.*

PROOF This can be proved in a straightforward way by structural induction in the structure of relational algebra expressions. Given a relational algebra expression E, let $|E|$ denote the cardinality of E. For every non-atomic expression: $|E_1 \cup E_2| \leq |E_1| + |E_2|; |E_1 - E_2| \leq |E_1|; |E_1 \times E_2| \leq |E_1| \times |E_2|; |\pi_L(E)| \leq |E|; |\sigma_C(E)| \leq |E|; |\rho_{X_1,...,X_n \to Y_1,...,Y_n}(E)| \leq |E|$.

Hence, it follows that:

PROPOSITION 3.5: *There are expressions of the M-relational algebra that are not expressible by the relational algebra RA-gp on the same database schema.*

Another aspect of the expressive power of mappers is the ability of mappers for simulating other relational operators. It will be shown below that projection, renaming and selection operators can be seen as special cases of mappers. That is to say, there exist three classes of mappers that are equivalent, respectively, to projection, renaming and selection. From this it can be concluded that the restriction of MRA to the operators mapper, union, difference and Cartesian product is as expressive as MRA.

Projection can be obtained through mapper operators over identity mapper functions. One identity mapper function is included for each project attribute. The project attribute has to be an attribute of the source schema.

RULE 3.1: *Let $S(X_1, ..., X_n)$ be a relation schema and $Y_1, ..., Y_p$ a list of different attributes in $X_1, ..., X_n$. For every relation instance $s(X_1, ..., X_n)$, the term*

$\pi_{Y_1,...,Y_p}(s)$ is equivalent to $\mu_F(s)$, where $F = f_{Y_1}, ..., f_{Y_p}$ and f_{Y_i} is the identity mapper function, for every $1 \leq i \leq m$.

PROOF

$$\begin{aligned}
\pi_{Y_1,...,Y_p}(s) &= \{t[Y_1, ..., Y_p] \mid t \in s\} \\
&= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } u[Y_i] = t[Y_i]\} \\
&= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } t[Y_i] \in \{u[Y_i]\}\} \\
&\text{because } f_{Y_i}(t) = \{t\}, \text{ for every } t \in Dom(Y_i) \\
&= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } t[Y_i] \in f_{Y_i}(u)\} \\
&= \mu_{f_{Y_1},...,f_{Y_p}}(s)
\end{aligned}$$

Strictly speaking, a renaming *ren* is a bijective function among sets of attributes X and Y s.t. $Dom(X_i) = Dom(Y_i)$ and $ren(X_i) \neq X_i$, for every $X_i \in X$. This function is usually represented as $X_1, ..., X_n \rightarrow Y_1, ..., Y_n$. The relational renaming operator is a unary relational operator parameterized by a renaming function (Abiteboul *et al.*, 1995; Atzeni & de Antonellis, 1993). Renaming can also be expressed by a mapper parameterized by renaming mapper functions. One renaming function is included for mapping each source attribute to the corresponding target attribute.

RULE 3.2: *Let $S(X_1, ..., X_n)$ and $T(Y_1, ..., Y_n)$ be two relation schemas, such that, $Dom(X) = Dom(Y)$. For every relation instance $s(X_1, ..., X_n)$, the expression $\rho_{X_1,...,X_n \rightarrow Y_1,...,Y_n}(s)$ is equivalent to $\mu_F(s)$ where $F = f_{Y_1}, ..., f_{Y_n}$ and, for every $1 \leq i \leq n$, f_{Y_i} is the renaming mapper function $\langle Y_i, X_i, id_{Dom(Y_i)} \rangle$.*

PROOF

$$
\begin{aligned}
&\rho_{X_1,...,X_n \to Y_1,...,Y_n}(s) \\
&= \{t[Y_1, ..., Y_p] \mid t \in s\} \\
&= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } u[Y_i] = t[Y_i]\} \\
&= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } t[Y_i] \in \{u[Y_i]\}\} \\
&\text{because } f_{Y_i}(t) = \{t\}, \text{ for every } t \in Dom(Y_i) \\
&= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \quad \text{s.t. } t[Y_i] \in f_{Y_i}(u)\} \\
&= \mu_{f_{Y_1},...,f_{Y_p}}(s)
\end{aligned}
$$

Since mapper functions may map input tuples into empty sets (i.e., no output values are created), they may act as filtering conditions which enable the mapper to behave not only as a tuple producer but also as a filter. In order to illustrate this property of mappers, Example 3.5.1 presents an example of selective transformation of data.

EXAMPLE 3.5.1: *Consider the conversion of yearly salary data into quarterly salary data. Let* `EMPSAL`[`ESSN`, `ECAT`, `EYRSAL`] *be the source relation that contains yearly salary information about employees. Suppose a target relation has to be generated with schema* `EMPDATA`[`ENUM`, `QTNUM`, `QTSAL`]*, which maintains the quarterly salary for the employees with long-term contracts. In the source schema, the attribute* `EYRSAL` *maintains the yearly net salary. Furthermore, consider that the attribute* `ECAT` *holds the employee category and that code* `'S'` *specifies a short-term contract whereas* `'L'` *specifies a long-term contract.*

This transformation can be specified through the mapper $\mu_{empnum,sal}$, where *empnum* is a `[ENUM]`-mapper function with domain `[ESSN,ECAT,EYRSAL]` that makes up new employee numbers (i.e., a Skolem function (Hull & Yoshikawa, 1990)), and *sal* is the `[QTNUM,QTSAL]`-mapper function

$$sal_{\texttt{QTNUM, QTSAL}}(ecat, eyrsal)$$

with domain `[ECAT,EYRAL]` that generates quarterly salary data, defined as:

$$sal(ecat, eyrsal) = \begin{cases} \{(i, \frac{eyrsal}{4}) \mid \ 1 \leq i \leq 4\} & if\ ecat = \texttt{'L'} \\ \emptyset & if\ ecat = \texttt{'S'} \end{cases}$$

As it turns out, mappers are sufficiently expressive for encoding relational selections, as formalized by the following rule:

RULE 3.3: *Let $S(X_1, ..., X_n)$ be a relation schema, C a condition over the attributes of this schema. There exists a set F of proper mapper functions for transforming $S(X)$ s.t., for every relation instance $s(X_1, ..., X_n)$, the term $\sigma_C(s)$ is equivalent to $\mu_F(s)$.*

PROOF It suffices to show how F can be constructed from C and prove the equivalence of σ_C and μ_F. Let $F = f_{X_1}, ..., f_{X_n}$ where each mapper function f_{X_i} is the mapper function with domain X_i s.t.

$$f_{X_i}(t) = \begin{cases} \{t[X_i]\} & \text{if } C(t) \\ \emptyset & \text{if } \neg C(t) \end{cases}$$

Thus,

$$\begin{aligned} \mu_F(s) &= \{t \in Dom(X) \mid \exists u \in s\ \forall 1 \leq i \leq n\ t[X_i] \in f_{X_i}(u)\} \\ &\quad \text{by the definition of } f_{X_i} \\ &= \{t \in Dom(X) \mid \exists u \in s \text{ s.t. } (\forall 1 \leq i \leq n\ t[X_i] \in \{u[X_i]\}) \text{ and } C(u)\} \\ &= \{t \in Dom(X) \mid \exists u \in s \text{ s.t. } (\forall 1 \leq i \leq n\ t[X_i] = u[X_i]) \text{ and } C(u)\} \\ &= \{t \in Dom(X) \mid \exists u \in s \text{ s.t. } t = u \text{ and } C(u)\} \\ &= \{t \in Dom(X) \mid t \in s \text{ and } C(t)\} \\ &= \sigma_C(s) \end{aligned}$$

3.6 SQL Syntax for Mappers

The mapper operator can be easily embedded into the SQL syntax by incorporating mapper functions as expressions into the *select* block. The main change consists of replacing the standard list of columns and expressions that follow the **select** keyword by a list of mapper functions as illustrated in Figure 3.1. The

select

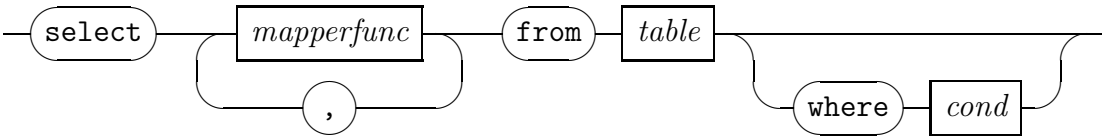


Figure 3.1: Syntax diagram of a simplified version of the select statement.

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

As illustrated in Figure 3.2, a mapper function can be a column name, an expression, a function call or an inline mapper function definition. The name of a column of the input schema denotes an identity mapper function that maps the same column onto the output schema. Alternatively, the name of the column in the output schema can be specified. In this case, the function is specified outside the select statement using a more appropriate programming language. This usage of mapper functions is aligned with the SQL syntax for the computation of aggregates in the sense that aggregate functions like `COUNT` or `SUM` are implemented elsewhere and then embedded in the select statement as parameters of the aggregation operator. An expression defines a single-valued mapper function that produces one output column. A mapper function call is identified by the **map** keyword followed by the function name. These mapper functions must have been previously declared. In order to avoid clashing of the output column names of the mapper function with the ones produced by other functions, the mapper function call can be followed by the specification of new column names. Another

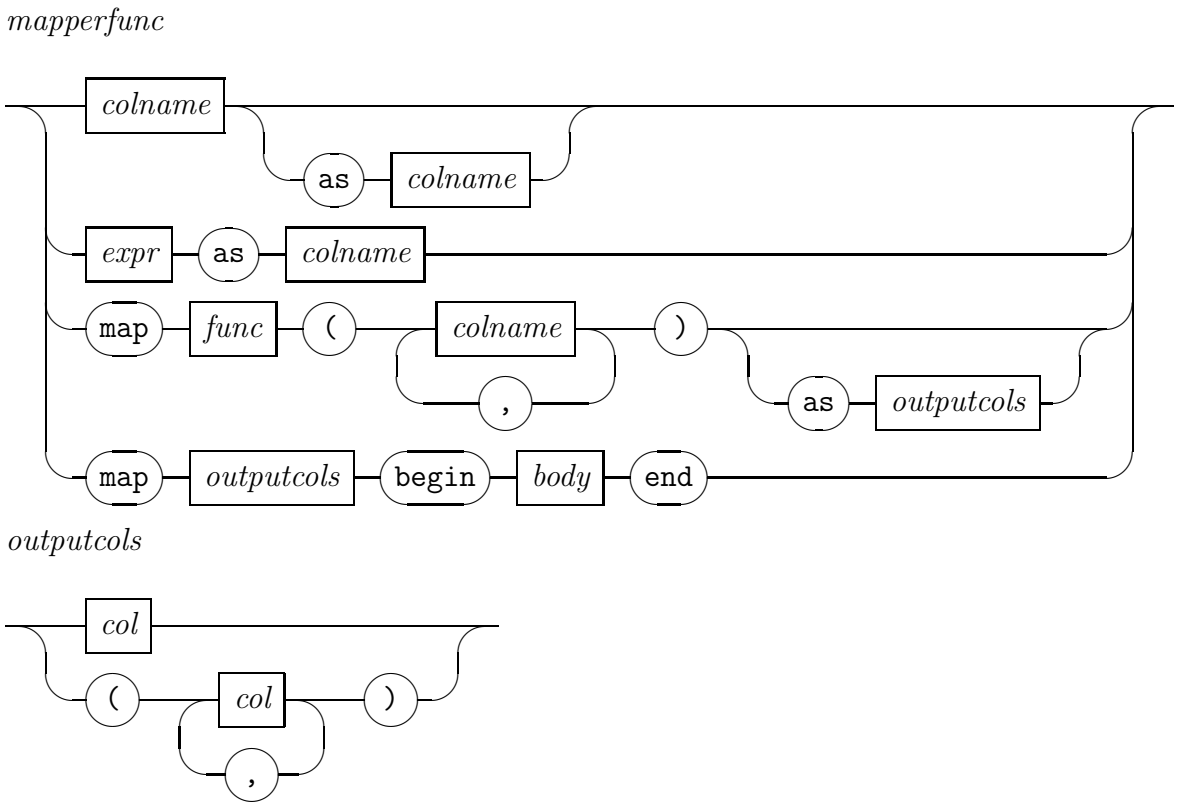


Figure 3.2: Syntax diagram of a mapper function specification.

way to define a mapper function consists of specifying inline an anonymous function. This function is specified through the **map** keyword with the output column names that will contribute to the output schema, followed by an inline specification of the function *body* within the **begin...end** block. In the case of inline function specifications, the input columns do not need to be specified. Instead, they are implicitly defined when the function implementation body accesses the columns of the input relation.

The solution for Example 1.1.2 using the proposed SQL syntax for the mapper operator is illustrated in Figure 3.3. The first mapper function, `lpad`, produces only one output value for each input value (line `2`). It is implemented as the expression that pads zeroes on the left of the source column `ACCT` to form the column `ACCTNO`. The second mapper function is specified inline and generates multiple output values (lines `3`–`16`). In this function, an auxiliary variable `rem_amnt`

```
 1: select
 2:   lpad(tostr(ACCT), 4, '0') as ACCTNO,
 3:   map AMOUNT, SEQNO
 4:     begin
 5:       var rem_amnt: numeric
 6:       var seq_no: integer  = 0
 7:       rem_amnt = AMT
 8:       loop while rem_amnt > 100 do
 9:         rem_amnt = rem_amnt - 100
10:         seq_no = seq_no + 1
11:         insert rem_amnt, seq_no
12:       end loop
13:       if rem_amnt > 0 then
14:         insert rem_amnt, seq_no + 1
15:       end if
16:     end
17: from LOANS
```

Figure 3.3: Transformation of Example 1.1.2 using inline mapper functions and the proposed syntax of the mapper operator.

is initialized with the AMT value and is used to partition the total amount into parcels of 100. The dynamic creation of output values is achieved by nesting an **insert** statement (line 11) into a **while** loop. Each time an **insert** is executed, a new output value, with two components, one for AMOUNT and another for SEQNO, is added to the set of values to be returned by the mapper function. When both functions are executed for an input tuple, the values stored in the sets of values are combined through a Cartesian product to produce the output values.

The distinguishing feature illustrated by this example is that mappers confine the mapping logic used to populate target fields in separate mapper functions. For example, by comparison with the table function implementation illustrated in Figure 2.4, the logic used to load the field ACCTNO in this example is kept outside the loop. In practice, this turns data transformations implemented using a mapper easier to read. This is especially true when dealing with target tables with tens of columns, which are common in real-world problems. Nesting all rules within the loop, like in stored procedures and table functions, compromises their readability (see Section 2.3).

```
1: select map acct(ACCT) as ACCTNO,
2:        map amt(AM) as AMOUNT
3: from LOANS, ACCOUNTS
4: where ACCOUNTS.ACCTN = LOANS.ACCT
5:   and ACCOUNTS.STATUS = 'O'
6:   and AMOUNT < 50
```

Figure 3.4: A query that selects small payments of open accounts by implementing a mapper together with a-priori and a-posteri filters.

Mapper functions used in mapper function calls can be built-in, like the aggregation functions of SQL, or defined by the user. In the current proposed syntax of the mapper operator, no syntax is supplied for declaring *user defined mapper functions*. User defined mapper functions can be defined in mostly any language as long as it provides some mechanism for returning multiple values. One example of such mechanism is the **pipe row** statement of PL/SQL (Feuerstein & Pribyl, 2005).

Specifying filters in mapper queries

Filters are specified using the *cond* block of the **where** clause. Two kinds of filters can be specified:

i) *a-priori filters*, which apply to each tuple of the input relation defined by the *table* and are evaluated before the mapper; and

ii) *a-posteriori filters*, which are evaluated over the output of the mapper and are used to limit the mapper results.

Although *cond* consists of only one Boolean expression, these different kinds of filters are identified by sub-expressions defined over particular sets of columns. Sub-expressions that are defined over the columns of the input schema expression define a-priori filters while sub-expressions that are defined over columns of the output schema define a-posteriori filters. In the query presented in Figure 3.4, the sub-expression `ACCOUNTS.STATUS = 'O'` defines an a-priori filter while the sub-expression `AMOUNT < 50` defines an a-posteriori filter.

In some situations it is not possible to clearly separate these two kinds of filters. For example, if the condition is dependent both on the input and output columns of the mapper like `AMOUNT < ACCOUNTS.WDRAWLIMIT`, the predicate can only be evaluated after the mapper, i.e., a-posteriori. The specification of a-posteriori filters in the **where** clause opens an interesting possibility of defining the condition using mapper functions. Since mapper functions return sets of values, their results can be tested with set operators like **in** or **exists**. Consider, for example, a condition for testing if an article is contained in a list of names extracted from a text description expressed as `ARTICLE_NAME` **in** `cleannames(ARTICLE_DESCRIPTION)`, where the `cleannames` is a function that returns multiple values.

3.7 Related Work

To support the growing range of applications of RDBMSs, extensions to RA have been proposed since its inception. The most widely used extension is possibly the aggregation operator, proposed by Klug (1982) for data consolidation, which relies on a set of supplied aggregation functions.

Applications requiring data transformations bring a new requirement to RA. Their focus is no longer limited to the initial idea of deriving information as suggested by Paredaens (1978) and Aho & Ullman (1979). Transformations also involve the production of new data items. Up to now some operators have been proposed for addressing the problem of expressing one to many data-transformations (Amer-Yahia & Cluet, 2004; Cunningham *et al.*, 2004; Galhardas *et al.*, 2001; Raman & Hellerstein, 2001). Although these operators show similarities with mappers, most of them are only capable of expressing bounded one to many transformations.

The unpivot operator of SQL Server 2005 transposes columns into rows and can be used for expressing one-to-many data transformations (Cunningham *et al.*, 2004). However, this operator can only be used to express bounded transformations.

Potter's Wheel fold operator is capable of producing several output tuples for each input tuple, which the authors identify as *one-to-many transforms* (Raman & Hellerstein, 2001). The main difference with respect to the mapper operator

lies in the number of output tuples generated. In the case of the fold operator, the number of output tuples is bound by number of columns of the input relation, while the mapper operator may generate an arbitrary number of output tuples.

The semantics of the Ajax map operator represents exactly a one-to-many data transformation (Galhardas *et al.*, 2001). Unlike our data mapper, the Ajax map operator allows the specification of a selection condition applied to each input tuple. Abstracting from the issue of generating rejected records, the semantics of Ajax map can be obtained by composing the mapper operator presented here with other relational algebra operators.

The work of Amer-Yahia & Cluet (2004) addresses the problem of efficiently extracting and loading data. They propose that data transformations can be expressed through RA operations extended with a grouping operator and a map operator. Likewise, this thesis defends that data transformations should be based on an extended RA featuring the mapper operator. However, unlike the mapper operator, the map operator does not perform one-to-many tuple transformations. Nevertheless, its presence validates the need for a powerful data transformation operator.

Functional programming languages like, for example, ML (Paulson, 1996) and Scheme (Abelson *et al.*, 1985), have a map function that can be regarded as an operator that applies one function to a set of elements, producing a set of transformed elements. However, there is a fundamental difference in the semantics of the functional map and the mapper operator: the map function operator only applies one function to the input elements. It can be argued that the different functions that compose a mapper can be compiled into one (for example using Proposition 3.3). However, as explained in Section 3.4, mapper operators with many different functions are preferable, since the exposition of more functions enables more optimization opportunities.

3.8 Conclusions

This chapter presented a specialized mapper operator for expressing one-to-many data transformations that extends Relational Algebra. Similarly, to other extensions to the basic RA, like generalized projection and aggregation, the mapper

operator relies on the use of external functions. These functions express part of the envisioned data transformation by producing a subset set of the the output attributes and are capable of producing multiple output values.

The study of the mapper operator proceeded by defining its formal semantics and analyzing some of its properties. A commencing result consisted of deriving an alternative semantics in terms of a Cartesian product of the function outputs. This result is important, since it endows an intuitive iterator-based physical execution algorithm for the mapper operator. It is well known that iterator-based physical operators lend themselves to simpler implementations (Graefe, 1993). Concerning the expressive power of mappers, it has been shown that RA extended with the new operator becomes more expressive than standard RA. Moreover, it was also demonstrated that mappers subsume unary relational operators, like projection, renaming and selection.

One driving concept of mappers consists of promoting the enclosing the logic to populate distinct attributes into separate functions. Since mapper functions may be user-defined, the idea of separation is undermined if the mapping logic is coupled to a few functions. To establish the desired form of a mapper, a formal definition of a mapper *normal form* was introduced. This definition can be used to decide from equivalent mappers which one is preferable. The low coupling promoted by normal forms, besides enhancing readability, is beneficial form the point of view of performance, because mapper functions can be explored for both logical and physical optimizations.

Finally, a seamless extension to the SQL syntax for representing mapper operations was proposed and then used to express some examples.

Chapter 4

Algebraic Optimization

This chapter addresses the rewriting of expressions containing standard relational operators and mappers. It presents a set of algebraic rewriting rules that enable the logical optimization of data transformation expressions combining relational operators with mappers. These rules are given with their formal proofs of correctness. This chapter also introduces a cost model for deciding which rules should be applied in query optimizations. The proposed cost model is illustrated in an example with rules for pushing selections.

4.1 Introduction

Algebraic rewriting rules are equations that specify the equivalence of two algebraic terms. Queries presented as relational expressions can be transformed, through algebraic rewriting rules, which are then evaluated more efficiently.

Consider the data transformation presented in Figure 3.4. This data transformation applies a filter to the result of a mapper operator. This mapper operator, in turn, is evaluated over the relation that results from applying a filter to the input relation denoted by a join operation. The query plan for this query is depicted in Figure 4.1. Therein, the filter $\sigma_{\texttt{AMOUNT < 50}}$ is applied to the mapper $\mu_{acct,amt}$, which takes as input the tuples of the input relation $\texttt{ACCOUNTS} \bowtie_{\texttt{ACCOUNTS.ACCTN=LOANS.ACCT}} \texttt{LOANS}$ that are not filtered by $\sigma_{\texttt{ACCOUNT.STATUS = 'O'}}$.

The plans of one-to-many data transformations may undergo two kinds of rewritings. First, the rewritings common to RA queries can be applied (Chaud-

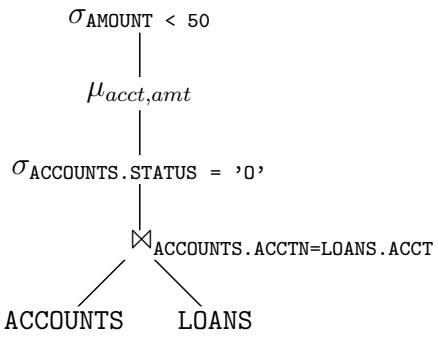


Figure 4.1: Query plan for the query presented in Figure 3.4.

huri, 1998; Ullman, 1988). For example, when the input relation is defined through join operations, some selections can be pushed through the join operators. Second, a set of rewritings specific to the proposed mappers can be introduced. In the example of Figure 4.1, the condition `AMMOUNT < 50` can be pushed down the *amt* mapper function using an optimization rule to be presented in this chapter (Rule 4.3).

One rewriting heuristic consists of deriving an equivalent algebraic expression that minimizes the amount of information transferred from operator to operator. In this spirit, two classes of algebraic rewriting rules are adapted to the mapper operator. First, rules for *pushing selections*, which attempt to reduce the cardinality of the source relations to be evaluated as early as possible. Secondly, the rules for *pushing projections*, which avoid propagating attributes that are not used by subsequent operators are presented.

4.2 Projections

A projection applied to a mapper is an expression of the form $\pi_Z(\mu_F(s))$. If $F = f_{A_1}, ..., f_{A_m}$ is a list of mapper functions, proper for transforming $S(X)$, then an attribute Y_i in $Y = A_1 \cdot ... \cdot A_m$ such that $Y_i \notin Z$, (i.e., not projected by π_Z) is said to be *projected away*. Attributes that are projected away offer optimization opportunities. Since they are not required for subsequent operations, the mapper

functions that generate them do not need to be evaluated. Rule 4.1 makes this idea precise.

RULE 4.1: *Let $F = f_{A_1}, ..., f_{A_m}$ be a list of mapper functions, proper for transforming $S(X)$ and $Y = A_1 \cdot ... \cdot A_m$. Let Z and Z' be lists of attributes in Y. For every relation instance s of $S(X)$, $\pi_Z(\mu_F(s)) = \pi_Z(\mu_{F'}(s))$, where $F' = \{f_{A_i} \in F \mid A_i$ contains at least one attribute in $Z\}$.*

PROOF In what follows, $A_i \cap Z \neq \emptyset$ is used to represent that *at least one attribute of A_i is in the list Z*. Thus,

$$\begin{aligned}
\pi_Z(\mu_F(s)) &= \{t[Z] \mid t \in Dom(Y) \text{ and } t \in \mu_F(s)\} \\
&= \{t[Z] \mid t \in Dom(Y) \text{ and } \exists u \in s \ \forall f_{A_i} \in F \text{ s.t. } t[A_i] \in f_{A_i}(u)\} \\
&\text{because only attributes in } A_i \cap Z \text{ are projected} \\
&\text{and, by hypothesis, } A_i \cap Z \neq \emptyset \Leftrightarrow f_{A_i} \in F' \\
&= \{t[Z] \mid t \in Dom(Y) \text{ and } \exists u \in s \ \forall f_{A_i} \in F' \text{ s.t. } t[A_i] \in f_{A_i}(u)\} \\
&= \pi_Z(\mu_{F'}(s))
\end{aligned}$$

Concerning Rule 4.1, it should be noted that if $Z = A_1 \cdot ... \cdot A_m$ (i.e, all attributes are projected), then $F' = F$ (i.e., no mapper function can be forgotten).

EXAMPLE 4.2.1: *Consider the mapper $\mu_{acct,amt}$ defined in Example 3.2.1. The expression $\pi_{\texttt{AMOUNT}}(\mu_{acct,amt}(\texttt{LOANS}))$ is equivalent to $\pi_{\texttt{AMOUNT}}(\mu_{amt}(\texttt{LOANS}))$. The acct mapper function is forgotten because the* `ACCOUNT` *attribute was projected away. Conversely, neither of the mapper functions can be forgotten in the expression $\pi_{\texttt{ACCTNO,SEQNO}}(\mu_{acct,amt}(\texttt{LOANS}))$.*

Attributes that are not used as input of any mapper function do not need to be retrieved from the mapper input relation. Thus, a projection that retrieves only those attributes that are relevant for the functions in F' can be introduced.

RULE 4.2: *Let $F = f_{A_1}, ..., f_{A_m}$ be a list of mapper functions, proper for transforming $S(X)$ and $Y = A_1 \cdot ... \cdot A_m$. For every relation instance s of $S(X)$, $\mu_F(s) = \mu_F(\pi_N(s))$, where N is a list of attributes in X, that includes only the attributes in $Dom(f_{A_i})$, for every mapper function f_{A_i} in F.*

PROOF

$$\mu_F(s) = \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \text{ s.t. } t[A_i] \in f_{A_i}(u)\}$$

by the definition of mapper function,

$$f_{A_i}(u) = f_{A_i}(u[B]) = f_{A_i}(u[N])$$
$$= \{t \in Dom(Y) \mid \exists u \in s \ \forall 1 \leq i \leq m \text{ s.t. } t[A_i] \in f_{A_i}(u[N])\}$$
$$= \{t \in Dom(Y) \mid \exists u \in \pi_N(s) \ \forall 1 \leq i \leq m \text{ s.t. } t[A_i] \in f_{A_i}(u)\}$$
$$= \mu_F(\pi_N(s))$$

EXAMPLE 4.2.2: *Consider the relation* `LOANS`$[$`ACCT`, `AM`$]$ *of Example 1.1.2. The attribute* `AM` *is an input attribute of the mapper function amt defined in Example 3.2.1. Thus, the expression* $\mu_{amt}($`LOANS`$)$ *is equivalent to* $\mu_{amt}(\pi_{\texttt{AM}}($`LOANS`$))$.

4.3 Selections

Two algebraic rewriting rules for optimizing expressions that combine filters expressed as relational selection operators with mappers are now presented. The first rule alleviates the cost of performing the Cartesian product operations that are used to implement the mapper operator. The second rule avoids superfluous function evaluations by pushing selections to the sources, and thus reducing the number of tuples fed to the mapper as early as possible.

4.3.1 Pushing selections to mapper functions

When applying a selection to a mapper, one can take advantage of the mapper semantics to introduce an important optimization. Given a selection $\sigma_{C_{A_i}}$ applied to a mapper $\mu_{f_{A_1},...,f_{A_m}}$, this optimization consists of pushing the selection $\sigma_{C_{A_i}}$, where C_{A_i} is a condition on the attributes produced by some mapper function f_{A_i}, directly to the output of the mapper function. Rule 4.3 formalizes this notion.

RULE 4.3: *Let* $F = f_{A_1}, ..., f_{A_m}$ *be a list of multi-valued mapper functions, proper for transforming relations with schema* $S(X)$. *Consider a condition* C_{A_i} *dependent of a set of attributes* A_i *for some* $1 \leq i \leq m$. *Then, for every relation*

instance $s(X)$,

$$\sigma_{C_{A_i}}(\mu_F(s)) = \mu_{F\setminus\{f_{A_i}\}\cup\{\sigma_{C_{A_i}}\circ f_{A_i}\}}(s)$$

where

$$(\sigma_{C_{A_i}} \circ f_{A_i})(t) = \begin{cases} f_{A_i}(t) & \text{if } C(t) \\ \emptyset & \text{if } \neg C(t) \end{cases}$$

PROOF Let $Y = A_1 \cdot ... \cdot A_m$.

$$\begin{aligned}
\sigma_{C_{A_i}}(\mu_F(s)) &= \{t \in Dom(Y) \mid t \in \mu_F(s) \text{ and } C_{A_i}(t[A_i])\} \\
&= \{t \in Dom(Y) \mid \exists u \in s \\
&\quad \forall 1 \leq j \leq m \ s.t.\ t[A_j] \in f_{A_j}(u) \text{ and } C_{A_i}(t[A_i])\} \\
&= \{t \in Dom(Y) \mid \exists u \in s \\
&\quad \forall 1 \leq j \leq m,\ j \neq i \ s.t.\ t[A_j] \in f_{A_j}(u) \text{ and} \\
&\quad t[A_i] \in f_{A_i}(u) \text{ and } C_{A_i}(t[A_i])\} \\
&= \{t \in Dom(Y) \mid \exists u \in s \\
&\quad \forall 1 \leq j \leq m,\ j \neq i \ s.t.\ t[A_j] \in f_{A_j}(u) \text{ and} \\
&\quad t[A_i] \in \sigma_{C_{A_i}}(f_{A_i}(u))\} \\
&= \mu_{F\setminus\{f_{A_i}\}\cup\{\sigma_{C_{A_i}}\circ f_{A_i}\}}(s)
\end{aligned}$$

The benefits of Rule 4.3 are easier to understand when considering the alternative definition for the mapper semantics in terms of a Cartesian product presented in Section 3.3. Intuitively, if at least one of the mapper functions is multi-valued, it follows from Proposition 3.1, that the Cartesian product expansion generated by $f_{A_1}(u) \times ... \times f_{A_m}(u)$ can produce duplicate values for some set of attributes A_i, $1 \leq i \leq m$. To see how, please refer to Example 3.2.1. Hence, by pushing the condition C_{A_i} to the mapper function f_{A_i}, the condition will be evaluated fewer times, i.e., only once for each output value of $f_{A_i}(t)$ as opposed to once for each output tuple of $\mu_F(t)$. This is particularly important for expensive predicates, e.g., those involving expensive functions or sub-queries (e.g., evaluating the SQL **exists** operator). See, e.g., Hellerstein (1998) for details on optimization of queries with expensive predicates.

Furthermore, note that when $C_{A_i}(t)$ does not hold, the evaluation of $(\sigma_{C_{A_i}} \circ f_{A_i})(t)$ returns the empty set. Considering the Cartesian product semantics of the mapper operator presented in Proposition 3.1, once a function returns the empty set, no output tuples will be generated. Thus, the evaluation of all mapper functions f_{A_j}, such that $j \neq i$ can be skipped. Physical execution algorithms for the mapper operator, like the Shortcircuiting algorithm to presented in Section 5.3, can take advantage of this optimization by evaluating f_{A_i} before any other mapper function.

This optimization can be employed even in situations in which neither expensive functions nor expensive predicates are present, as it alleviates the average cost of the Cartesian product, which depends on the cardinalities of the sets of values produced by the mapper functions.

EXAMPLE 4.3.1: *Consider the relation* $\texttt{SMALLPAYMENTS}[\texttt{ACCTNO}, \texttt{AMOUNT}, \texttt{SEQNO}]$ *formed by all payments whose amount is smaller than* 5. *This relation can be obtained from the relation* PAYMENTS *presented in Example 1.1.2 by composing a selection with a mapper. According to Example 3.2.1,* $\mu_{acct,amt}(\texttt{LOANS})$ *corresponds to the relation* PAYMENTS. *Then, the expression* $\sigma_{\texttt{AMOUNT}<5}(\mu_{acct,amt}(\texttt{LOANS}))$ *denotes the relation* SMALLPAYMENTS. *By applying Rule 4.3 to the above expression, the expression* $\mu_{acct,\sigma_{\texttt{AMOUNT}<5} \circ amt}(\texttt{LOANS})$, *which is likely to be faster to evaluate, is obtained.*

4.3.2 Pushing selections through mappers

An alternative way of rewriting expressions of the form $\sigma_C(\mu_F(s))$ consists of replacing the attributes that occur in the condition C by the mapper functions that compute them. Suppose that, in the selection condition C, an attribute A is produced by the mapper function f_A. By replacing the attribute A with the mapper function f_A in condition C, an equivalent condition is obtained.

In order to formalize this notion, some further notation is required. Let $F = f_{A_1}, ..., f_{A_m}$ be a list of mapper functions proper for transforming $S(X)$ and $Y = A_1 \cdot ... \cdot A_m$. The function resulting from the restriction of f_{A_i} to an attribute $Y_j \in A_i$ is denoted by $f_{A_i}|_{Y_j}$. Moreover, given an attribute $Y_j \in Y$, $F|_{Y_j}$ represents

the function $f_{A_i}|_{Y_j}$ s.t. $Y_j \in A_i$. Note that, because F is a proper list of mapper functions, the function $F|_{Y_j}$ exists and is unique.

RULE 4.4: *Let $F = f_{A_1}, ..., f_{A_m}$ be a list of mapper functions, proper for transforming $S(X)$, $Y = A_1 \cdot ... \cdot A_m$ and $B = B_1, ..., B_p$ be a list of attributes in Y. If $H = F|_{B_1}, ..., F|_{B_p}$ is a list of single-valued functions then, for every relation instance s of $S(X)$,*

$$\sigma_{C_B}(\mu_F(s)) = \mu_F(\sigma_{C[B_1,...,B_p \leftarrow F|_{B_1},...,F|_{B_p}]}(s))$$

where C_B means that C depends on the attributes of B, and the condition that results from replacing every occurrence of each B_i by E_i is represented by the expression $C[B_1, ..., B_p \leftarrow E_1, ..., E_p]$.

This rule replaces each attribute B_i in the condition C by the expression that describes how its values are obtained. In practice, this rule is of broad application as the attributes used in the condition of a selection are often generated either by single-valued functions like:

i) identity mapper functions;

ii) constant mapper functions;

iii) arithmetic expressions.

Cases *(i)* and *(ii)* draw from attribute renaming and value assignments. Consider, for example the condition C to be $A < B$. The expression $\sigma_{A<B}(\mu_{X \to A,2 \to B,f_C}(s))$ can be re-written as $\mu_{X \to A,2 \to B,f_C}(\sigma_{X<2}(s))$. Concerning case *(iii)*, a new condition is produced by expanding attributes with arithmetic expressions. In this case, although the expression is evaluated twice —once in the condition and once in the mapper—, the number of tuples that have to be handled by the mapper operator can be drastically reduced. These tradeoffs are analyzed in detail in Section 4.6.

PROOF Rule 4.4 can be demonstrated by proceeding in two steps. First, by expanding both expressions into their corresponding sets of tuples. Second, the

equivalence of these sets is established. So, on the one hand,

$$\begin{aligned}\sigma_{C_B}(\mu_F(s)) &= \{t \in Dom(Y) \mid t \in \mu_F(s) \text{ and } C_B(t)\} \\ &= \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } t[A_i] \in f_{A_i}(u) \text{ and } C_B(t), \\ &\quad \forall 1 \leq i \leq m\}\end{aligned} \tag{4.1}$$

On the other hand,

$$\begin{aligned}&\mu_F(\sigma_{C[B_1,...,B_p \leftarrow F|_{B_1},...,F|_{B_p}]}(s)) \\ &= \{t \in Dom(Y) \mid \exists u \in \sigma_{C[B_1,...,B_p \leftarrow F|_{B_1},...,F|_{B_p}]} \text{ s.t. } t[A_i] \in f_{A_i}(u), \\ &\quad \forall 1 \leq i \leq m\} \\ &= \big\{t \in Dom(Y) \mid \exists u \in \{v \in Dom(X) \mid v \in s \text{ and} \\ &\quad C[B_1, ..., B_p \leftarrow F|_{B_1}, ..., F|_{B_p}](v)\} \text{ s.t. } t[A_i] \in f_{A_i}(u), \forall 1 \leq i \leq m\big\} \\ &= \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } t[A_i] \in f_{A_i}(u) \\ &\quad \text{and } C[B_1, ..., B_p \leftarrow F|_{B_1}, ..., F|_{B_p}](u), \forall 1 \leq i \leq m\}\end{aligned} \tag{4.2}$$

It now remains to prove that, if $t[A_i] \in f_{A_i}(u)$, for every $1 \leq i \leq m$, then

$$C[B_1, ..., B_p \leftarrow F|_{B_1}, ..., F|_{B_p}](u) \text{ iff } C_B(t)$$

This trivially follows from the definition of $F|_{B_i}$, considering that all functions in H are single-valued.

4.4 Joins

Another important binary operation is the *join*, represented as $\bowtie_C$ (see e.g., Ullman (1988) or Garcia-Molina *et al.* (2002)). Join operators can be obtained as a combination of a selection with a Cartesian product[1] (Mishra & Eich, 1992). Concretely, $r \bowtie_C s = \sigma_C(r \times s)$. Using this equivalence, it can be easily seen that the mapper operator can be distributed over the join in two steps. First, pushing the mapper over the selection σ_C and second, distributing the mapper over the

[1] To be precise, the renaming operator should also be employed when the schemas of r and s share attributes. For simplicity of presentation, disjointness of the schemas is assumed. This assumption does not interfere with the results drawn.

Cartesian product $r \times s$. For the first step, Rule 4.4 can be used. However, the second step is hindered by the fact that the set F appropriate for transforming data with the relation schema of $r \times s$ does not necessarily contain sets F_R and F_S for transforming data with the relation schema of r and s. However, if F can be partitioned into two disjoint subsets F_R and F_S, then the equivalence $\mu_F(s \times r) = \mu_{F_S}(s) \times \mu_{F_R}(r)$ holds. This notion is formalized in Rule 4.5.

RULE 4.5: *Let $F = \{f_{A_1}, ..., f_{A_m}\}$ be a set of mapper functions proper for transforming $SR(X,Y)$ into $T(Z)$. Let s and r be relation instances with schemas $S(X)$ and $R(Y)$ respectively. If there exist Z_R and Z_S, such that, $Z_R \cdot Z_S = Z$, and two disjoint subsets $F_R \subseteq F$ and $F_S \subseteq F$ of mapper functions, proper for transforming, respectively, $S(X)$ into $T_R(Z_R)$ and $R(Y)$ into $T_S(Z_S)$ then $\mu_F(s \times r) = \mu_{F_S}(s) \times \mu_{F_R}(r)$.*

PROOF

$$
\begin{aligned}
\mu_F(r \times s) &= \{t \in Dom(Z) \mid \exists u \in (r \times s) \text{ s.t. } t[A_k] \in f_{A_k}(u), \forall f_{A_k} \in F\} \\
&= \{t \in Dom(Z) \mid \exists u \in (r \times s) \text{ s.t. } \quad t[A_i] \in f_{A_i}(u), \forall f_{A_i} \in F_R \\
&\qquad \text{and } t[A_j] \in f_{A_j}(u), \forall f_{A_j} \in F_S\} \\
&\quad \text{since for every } f_{A_i} \in F_S,\ Dom(f_{A_i}) \text{ is in } X, \\
&\quad \text{and for every } f_{A_j} \in F_R,\ Dom(f_{A_j}) \text{ is in } Y, \\
&= \big\{t \in Dom(Z) \mid \exists u \in (r \times s) \text{ s.t. } \big(t[A_i] \in f_{A_i}(u[X]), \forall f_{A_i} \in F_R \\
&\qquad \text{and } t[A_j] \in f_{A_j}(u[Y]), \forall f_{A_j} \in F_S\big)\big\} \\
&\quad \text{because } \{u \mid u \in r \times s\} = \{u \mid u[R] \in r \text{ and } u[S] \ \in s\}, \\
&= \{t \in Dom(Z) \mid \exists v \in r \text{ s.t. } \quad t[A_i] \in f_{A_i}(v), \forall f_{A_i} \in F_R \\
&\qquad \text{and } \exists w \in s \text{ s.t. } t[A_j] \in f_{A_j}(w), \forall f_{A_j} \in F_S\} \\
&= \mu_{F_R}(r) \times \mu_{F_S}(s)
\end{aligned}
$$

4.5 Other Binary Operators

Unary operators of relational algebra enjoy useful distribution laws over binary operators. The mapper operator is a unary operator and it allows the following straightforward equivalence to be established:

RULE 4.6: *Let $F = \{f_{A_1}, ..., f_{A_m}\}$ be a set of mapper functions, proper for transforming $S(X)$ into $T(Y)$. Let r and s be relation instances with schema $S(X)$. Then, $\mu_F(r \cup s) = \mu_F(r) \cup \mu_F(s)$*

PROOF

$$\begin{aligned}\mu_F(r \cup s) &= \{t \in Dom(Y) \mid \exists u \in (r \cup s) \text{ s.t. } t[A_i] \in f_{A_i}(u), \forall 1 \leq i \leq m\} \\ &= \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } t[A_i] \in f_{A_i}(u) \text{ or } \exists v \in s \text{ s.t.} \\ &\quad t[A_i] \in f_{A_i}(v), \forall 1 \leq i \leq m\} \\ &= \mu_F(r) \cup \mu_F(s)\end{aligned}$$

However, the mapper operator does not distribute over intersection and difference, since these operators are not monotonic. For example, consider a mapper function f such that $f(\mathtt{a}) = 0$ and $f(x) = 1$ *if* $x \neq \mathtt{a}$. Let $A = \{\mathtt{a}, \mathtt{b}\}$ and $B = \{\mathtt{a}, \mathtt{c}\}$. In this case, $\mu_f(A \cap B) = \{0\}$, but this result is different from $\mu_f(A) \cap \mu_f(B) = \{0, 1\}$.

4.6 Cost of Expressions

This section presents the cost estimation framework for expressions that combine selections and mappers. First, the cost of applying a selection to a mapper is estimated. Second, the cost estimates for optimized expressions obtained by applying Rule 4.3 and Rule 4.4 are developed, giving particular attention to the gains obtained with the proposed optimizations.

The primary factors affecting the gain obtained when applying the proposed optimizations are *predicate selectivity* (Selinger *et al.*, 1979), the *mapper function fanout* and the *mapper function evaluation cost*. Similarly to Chaudhuri & Shim (1993), the average cardinality of the output values produced by a mapper function is designated as the *function fanout*. Analogously, the *mapper fanout* is defined as the average number of tuples produced by the mapper for each input tuple.

4.6.1 Cost of mappers

Since the evaluation of a mapper can be performed on a tuple by tuple basis, the cost of evaluating the mapper operator expression $\mu_F(r)$ can be estimated by adding up the per-tuple cost of transforming each tuple of the input relation r. For each tuple $t \in r$, the cost of producing the output tuples can be defined as the sum of the costs of evaluating all mapper functions and the cost of performing the Cartesian product of the function outputs.

The notion of cost introduced above can be formalized as follows. Consider C_f to be the estimated cost per-tuple of a mapper function f. Then, C_F is the estimated cost per-tuple of evaluating all the mapper functions $f \in F$, given by $C_F = \sum_{f \in F} C_f$. Furthermore, note that the cost of computing a Cartesian product is linear in the size of its inputs, i.e., given two sets of elements, A and B, the Cartesian product $A \times B$ can be computed in time linear to $|A| \cdot |B|$.

For a given tuple t, when evaluating an expression of the form $\mu_F(t)$, the input of the Cartesian product consists of the sets returned by the mapper functions in F. In this way, if $F = f_{A_1}, ..., f_{A_m}$, the cost of computing the Cartesian product algorithm, is $k \cdot |f_{A_1}(t)| \cdot ... \cdot |f_{A_m}(t)| + m \cdot k_0$, where k is an adjustment factor[1], m is the number of functions in F, and the constant k_0 represents the overhead incurred by the algorithm for checking the emptiness of the input sets. In practice, when $|f_{A_i}(t)| = 0$, the cost of the Cartesian product algorithm is not zero but a small amount captured by $m \cdot k_0$.

Since the exact number of elements produced by $f(t)$ can only be determined after evaluating the functions, an estimate for $|f(t)|$ for every $f \in F$ is necessary. The estimated value of $|f(t)|$ is given by the expected fanout of a mapper function f, designated by O_f.

The *mapper fanout* is represented as O_F. Assuming that the function outputs are not correlated, the value of O_F can be approximated by $\prod_{f \in F} O_f$. Therefore, if F has m mapper functions, the estimated per-tuple cost of executing the Cartesian product is $C_{prd} = k \cdot O_F + m \cdot k_0$.

[1]It is assumed that implementations of the Cartesian product handle attribute values by reference and not by value. As an effect, k is independent of the size of the inputs.

Finally, for an input relation r with cardinality n, the estimated cost of $\mu_F(r)$ is:

$$C_{\mu_F} = n \cdot (C_{prd} + C_F) = n \cdot (k \cdot \prod_{f \in F} O_f + m \cdot k_0 + \sum_{f \in F} C_f) \tag{4.3}$$

4.6.2 Cost of a filter applied to a mapper

The cost of the expression $\sigma_{C_A}(\mu_F(r))$ can be estimated as the cost of evaluating the mapper plus the cost of evaluating the selection condition on each tuple produced by the mapper. In the sequel, this expression will be referred to as the non-optimized expression.

Consider C_{sel} to be the average per-tuple cost of evaluating the selection condition C_A and let α be its corresponding selectivity, with $0 \leq \alpha \leq 1$. The cost of the non-optimized expression is:

$$C_{nonopt} = C_{\mu_F} + n \cdot O_F \cdot C_{sel} \tag{4.4}$$

Multiplying n by the fanout of the mapper O_F, results in the expected number of output tuples for the mapper operator. Since the selection condition is evaluated once for each tuple returned by the mapper, $n \cdot O_F \cdot C_{sel}$ represents the total cost of evaluating the selection condition.

4.6.3 Cost of an expression optimized with rule 4.3

The optimized expression obtained through Rule 4.3 is $\mu_{F \setminus g_{A_j} \cup \{\sigma_{C_{A_j}} \circ g_{A_j}\}}(r)$. Assuming that $g_{A_j} \in F$ is the mapper function onto which the condition is pushed, the cost corresponds to the costs of the list of mapper functions where the function $\sigma_{C_{A_j}} \circ g_{A_j}$ replaces g_{A_j}.

The estimated per-tuple cost of evaluating $\sigma_{C_{A_j}} \circ g_{A_j}$ is $C_{g_{A_j}} + O_{g_{A_j}} \cdot C_{sel}$, i.e., the cost of evaluating the mapper function g_{A_j} plus the cost of evaluating the selection condition C_{A_j} for each element produced by the function.

Obviously, the cost of the Cartesian product for the optimized expression is not the same as the cost of the Cartesian product for the non-optimized expression, since $\sigma_{C_{A_j}} \circ g_{A_j}$ and g_{A_j} have different fanouts. More precisely, since α represents the probability that C_{A_j} holds, the fanout of $\sigma_{C_{A_j}} \circ g_{A_j}$ is given by $\alpha \cdot O_{g_{A_j}}$.

This means that the cost of the Cartesian product for the optimized expression, represented as $C_{prd'}$ is given by $k \cdot O_{F \setminus g_{A_j}} \cdot \alpha \cdot O_{g_{A_j}} + m \cdot k_0$, which is equivalent to $k \cdot \alpha \cdot O_F + m \cdot k_0$.

The estimated cost of the mapper corresponding to the optimized expression is:

$$n \cdot (C_{prd'} + C_{F \setminus g_{A_j}} + C_{g_{A_j}} + O_{g_{A_j}} \cdot C_{sel}) \tag{4.5}$$

which corresponds to the cost of the Cartesian product plus the cost of computing all functions except g_{A_j}, plus the cost of computing $\sigma_{C_{A_j}} \circ g_{A_j}$. This can be simplified to

$$n \cdot (C_{prd'} + C_F + O_{g_{A_j}} \cdot C_{sel}) \tag{4.6}$$

The expected gain for this optimization, $\Delta_{4.3}$, is now computed as the difference between (4.4) and (4.6), which becomes:

$$\Delta_{4.3} = n \cdot (C_{prd} + C_F) + n \cdot O_F \cdot C_{sel} - n \cdot (C_{prd'} + C_F + O_{g_{A_j}} \cdot C_{sel}) \tag{4.7}$$

Since $C_{prd'} = k \cdot \alpha \cdot O_F + m \cdot k_0$, developing and simplifying (4.7) yields (see Appendix A.1):

$$\Delta_{4.3} = n \cdot k \cdot O_F \cdot (1 - \alpha) + n \cdot C_{sel} \cdot (O_F - O_{g_{A_j}}) \tag{4.8}$$

Notably, high gains are obtained for small selectivities. In contrast, as the selectivity α approaches 100%, the factor $n \cdot k \cdot O_F \cdot (1 - \alpha)$ in (4.8) tends to zero, thus decreasing the gain. Concerning the influence of the mapper function fanout $O_{g_{A_j}}$, it can be concluded from (4.8) that the larger is the difference between O_F and $O_{g_{A_j}}$, the higher is the gain. It is interesting to observe that if $O_{g_{A_j}} > O_F$, when the selectivity is near 100%, $\Delta_{4.3}$ will be negative. However, for this to be possible, since $g_{A_j} \in F$, some other function in F should have a fanout much smaller than 1. If the fanout $O_{g_{A_j}}$ is smaller than the mapper fanout, i.e., $O_{g_{A_j}} < O_F$, the gain will always be positive. In this situation, the higher is the value of C_{sel}, the higher is the gain $\Delta_{4.3}$ obtained.

4.6.4 Cost of an expression optimized with rule 4.4

As presented in Section 4.3.2, the optimized expression obtained by applying Rule 4.4 takes the form $\mu_F(\sigma_{C[B_1,...,B_l \leftarrow F|_{B_1},...,F|_{B_l}]}(s))$, where $H = F|_{B_1}, ..., F|_{B_l}$ is the set of mapper functions that are propagated into the selection condition.

The cost of the optimized expression is given by summing *(i)* the cost of evaluating the new selection condition $C[B_1, ..., B_l \leftarrow F|_{B_1}, ..., F|_{B_l}]$, with *(ii)* the cost of evaluating the mapper μ_F for every tuple that is not filtered by the condition. Since the new condition is obtained by inlining the mapper functions of H in the condition C, the estimated per-tuple cost of evaluating the new condition is $C_{sel} + C_H$. This corresponds to the cost of evaluating the initial selection plus the cost of evaluating the propagated functions. Therefore, when applying this rule, the Cartesian product and the rest of the mapper functions are only evaluated when $\sigma_{C[B_1,...,B_l \leftarrow F|_{B_1},...,F|_{B_l}]}$ holds. Thus, the estimated cost of the optimized expression is:

$$n \cdot (C_{sel} + C_H) + n \cdot \alpha \cdot (C_{prd} + C_F) \tag{4.9}$$

where $n \cdot (C_{sel} + C_H)$ represents the cost of evaluating the condition and $n \cdot \alpha \cdot (C_{prd} + C_F)$ represents the cost of evaluating the mapper for the tuples that are not filtered by the condition. Note that, since only single-valued functions can be pushed into the condition, the mapper functions in H have fanout equal to one.

The gain of this optimization is obtained as the difference between (4.4) and (4.9). Hence,

$$\Delta_{4.4} = n \cdot (C_{prd} + C_F) + n \cdot O_F \cdot C_{sel} - n \cdot (C_{sel} + C_H) - n \cdot \alpha \cdot (C_{prd} + C_F) \tag{4.10}$$

which becomes:

$$\Delta_{4.4} = n \cdot (1 - \alpha) \cdot (C_{prd} + C_F) + n \cdot C_{sel} \cdot (O_F - 1) - n \cdot C_H \tag{4.11}$$

The formula (4.11) above indicates that smaller selectivities α result in higher gains. The gain $\Delta_{4.4}$ increases with the fanout of the mapper O_F, with the evaluation cost of the selection condition C_{sel} and with the evaluation cost of

all mapper functions C_F. Pushing fewer functions or cheaper functions to the selection condition means lower values of C_H, which also results in higher gains.

4.6.5 Selecting the best optimization

In some situations, only one of the rewriting rules applies. Rule 4.4 can only be applied when the attributes of the condition are produced by single-valued functions, while Rule 4.3 can be employed when optimizing selections whose conditions involve attributes mapped by multi-valued or single-valued functions. Additionally, Rule 4.3 can only be applied when the attributes of the selection condition are produced by only one function, while Rule 4.4 can be applied when conditions involve multiple attributes that are produced by multiple functions.

If the attributes of the selection condition are produced by only one mapper function and, furthermore, if this mapper function is single-valued, then both rules can be applied. In this case, the rule that brings the highest gain has to be identified. This is determined by comparing the gains obtained by both rules. It is more advantageous to use Rule 4.3 instead of Rule 4.4 when $\Delta_{4.3} - \Delta_{4.4} > 0$, which is the same as:

$$n \cdot C_H + n \cdot C_{sel} \cdot (O_{g_{A_j}} - 1) - n \cdot (1 - \alpha) \cdot (C_F + m \cdot k_0) > 0 \tag{4.12}$$

Appendix A.1 gives details on deriving the expression of $\Delta_{4.3} - \Delta_{4.4}$. Equation (4.12) is developed under the assumption that, since g_{A_j} is single-valued, the fanout $O_{g_{A_j}}$ is 1. This yields:

$$C_H > (1 - \alpha) \cdot (C_F + m \cdot k_0) \tag{4.13}$$

As the selectivity α approaches 100%, $(1 - \alpha) \cdot (C_F + m \cdot k_0)$ gets smaller. For higher selectivities, Rule 4.4 is more likely to perform better than Rule 4.3. Since C_H and C_F are fixed, there is always a selectivity α_0 for which Rule 4.4 is better than Rule 4.3. Moreover, α_0 decreases as the difference between the C_F and C_H increases.

4.7 Related Work

Several extensions to RA have been proposed in the form of new operators accompanied by the corresponding logical optimizations (Bleiholder & Naumann, 2005; Börzsönyi *et al.*, 2001; Gray *et al.*, 1997; Li *et al.*, 2005).

The unpivot operator proposed by Cunningham *et al.* (2004) also addresses one-to-many data transformations. The rewriting rules proposed for the unpivot operator only consider pushing projections and selections and are not as comprehensive as the ones proposed here for mapper operator.

One-to-many data transformations can be expressed as extensions to relational queries (that can be represented as trees). In the view of Shu *et al.* (1977), rewriting rules for optimizing expressions denoting data transformations should aim at maximizing parallelism. However, this is not the primary concern of the rewriting rules proposed for the mapper.

Few of the existing Commercial ETL tools perform logical optimization of the data transformation specifications (Simitsis *et al.*, 2005). As recognized by Galhardas *et al.* (2000), the origin of this limitation lies in the lack of a clear separation between the logical and physical levels. Since ETL programs usually run for a very long time, measured in hours or even days, devoting more computational effort to their optimization is highly beneficial. More comprehensive rewriting strategies are feasible. Several important advances have been made in this direction. The problem of the logical optimization of an ETL process defined as a workflow of data transformation activities is addressed in Simitsis *et al.* (2005). The authors model the ETL optimization problem as a global state-space search problem using three classes of re-writings. This solution is useful only for logical optimization. The algebraic rewriting rules proposed for the mapper operator could be integrated into the optimization algorithm proposed.

In a parallel line of research, Amer-Yahia & Cluet (2004) address the problem of efficiently extracting and loading data. In their setting, data transformations are expressed through RA operations extended with a grouping operator and a map operator. A set of optimization rules and a cost model are developed for optimizing such algebraic representations. Their cost model is geared toward optimizing the loading of the relations obtained in the target database, while the

one proposed here tries, in a sense, to minimize the effort required to compute the relations. The contribution of Amer-Yahia & Cluet (2004) validates the usefulness of algebraic optimization in the context of data transformations.

4.8 Conclusions

This chapter presented rewriting rules for performing the logical optimization of algebraic expressions that combine mappers with standard relational operators.

The rules consist of a set algebraic rewritings, which were given together with their proofs of correctness. They describe how to commute mappers with other unary operators, like projections and selections, and to how to distribute mappers over binary operators, like unions and Cartesian products.

The proposed rewriting rules lead to expressions that are faster to evaluate, because there will be less mapper function evaluations and less I/O. The expressions obtained through the rules for pushing projections reduce the number of mapper functions. By pushing selections, the number of tuples fed to the mapper is reduced, causing a decrease on the number of function evaluations. There are two reasons for I/O reduction. First, by pushing projections less columns are required. Secondly, by pushing selections less tuples are read from the input relations. The rules for pushing mappers over binary operators become advantageous whenever mappers act as filters, since pushing the mapper to the input relation reduces the total number of tuples processed by the binary operator.

A cost model was proposed to equip a query optimizer with means to decide which rewriting rules should be applied. The decisions are based on cost estimates computed based on the standard predicate selectivity estimates and also estimates for mapper function cost and fanout. The proposed cost model is demonstrated for rewriting rules that apply selections to mappers. The cost estimates for the remaining optimization rules are likely to be simpler than those presented here.

Chapter 5

Mapper Execution Algorithms

This chapter discusses the physical execution of the mapper operator. Although the semantics of the mapper operator suggest a one-tuple-at-a time processing algorithm, this naïve execution approach is often inefficient in practice. Two alternative execution algorithms that alleviate the computation effort of mappers by sidestepping unneeded function evaluations are proposed. The first explores the idea of shortcircuiting computation by taking advantage of the semantics of the mapper operator, while the second explores the presence of duplicate values in the input relation through in-memory caching. This chapter also introduces a new cache replacement strategy, which is suited for mapper evaluations with duplicates and expensive functions.

5.1 Introduction

The formal semantics of the mapper operator presented in Section 3.3 suggests the following straightforward evaluation algorithm: for each tuple of the input relation, perform the Cartesian product of the result of evaluating all mapper functions. The output relation is obtained by unioning the obtained tuples.

However, this naïve execution algorithm can become very inefficient in many real-world settings. First, the computation of mapper functions can be expensive. In data cleaning applications this is frequently the case. Examples of common *expensive mapper functions* include check-digit computations, string pattern

matching and manipulations, and BLOB/CLOB object treatment. Second, mappers often produce many columns. In legacy data migrations, like those reported by Carreira & Galhardas (2004a), mappers are required to produce several hundreds of columns. Third, data transformations are often applied to very large relations, containing millions of tuples. Hence, finding efficient algorithms to execute mappers is of utmost importance.

This chapter focuses on algorithms that alleviate the computation effort (by contrast with I/O effort) required to evaluate a mapper by avoiding superfluous function evaluations. These algorithms explore two common situations:

Mapper functions that return empty sets. One possible outcome of a mapper function is the empty set for some input tuple t (as in Example 3.5.1). In this case, this function is acting as a filter and, as a consequence, t will not be reflected in the output. Different situations can cause a mapper function to return an empty set. First, the function may not be able to correctly process some ill-formed inputs. Is this case, an empty set is returned after the occurrence of an exception, like a division by zero. Second, the function itself encodes a constraint on the input data, resulting in an explicit rejection, which is also encoded as an empty set. In a sense, the function is encoding a predicate, stating that only a subset of the input set is processed. Third, the empty set result may follow from the definition of the function itself. Consider, for instance, a function returning restaurant addresses corresponding to a zip code: this function will return an empty set, if it is invoked with a zip code corresponding, say, to a government building. As soon as one mapper returns an empty set the evaluation of the remaining functions can be skipped because no tuple is to be generated.

Input relations that have duplicate values. Another common situation is input relations with duplicates in some columns. Duplicates in a relation can come either directly from the stored relation or arise indirectly during query evaluation, for example, as the result of *theta joins* or *outer joins*.

These observations motivate the research of hash-based algorithms for the mapper operator evaluation based on an in-memory cache of function results.

Algorithm 1 Naïve mapper evaluation

INPUT: r : the input relation
$f_{A_1}, ..., f_{A_m}$: the mapper functions
OUTPUT: s : the output relation
VARIABLES: t : an input tuple from s

1: $s \leftarrow \emptyset$;
2: **for all** $t \in r$ **do**
3: $\quad s \leftarrow s \cup (f_{A_1}(t) \times ... \times f_{A_m}(t))$
4: **end for**

Each time a mapper function is evaluated, the cache is checked by hashing the input value to find a previously computed result. If it is found, an evaluation of the function is saved. To be useful, the hash method requires a large amount of main memory. A compromise to make it useful in practice consists on managing the available memory by replacing the entries that are less likely of being requested again, making room for those that are more expensive to compute and requested more often.

5.2 Naïve Evaluation Algorithm

The mapper operator could be evaluated in a simple *tuple-at-a-time* manner as illustrated in Algorithm 1, which repeats the following steps for each input tuple t: *(i)* fetch t from the input relation; *(ii)* apply each mapper function to t; *(iii)* combine all mapper function results, through a Cartesian product, adding the produced tuples so obtained to the output relation.

The unpivot operator is implemented in a similar fashion, as described by Cunningham *et al.* (2004). It iterates over the input relation once and generates multiple output rows for each input row. However, unlike the Naïve evaluation algorithm, no Cartesian product operations are performed because the unpivot operator does not use functions.

Intuitively, the cost of evaluating the mapper operator expression $\mu_F(r)$ using the Naïve algorithm can be estimated by adding up the per-tuple cost of transforming each tuple of the input relation r. For each tuple $t \in r$, the cost of producing the output tuples can be defined as the sum of the cost of evaluating

all mapper functions and the cost of performing the Cartesian product of the function outputs.

As explained in Section 4.6.1, the cost of evaluating the mapper operator using the Naïve algorithm can be estimated by considering the estimated per-tuple cost of evaluating all the mapper functions $f \in F$, represented as C_F, and the estimated *mapper fanout* represented as O_F. The cost of evaluating a mapper $\mu_{f_1,...,f_m}$ over an input relation r with n tuples is estimated as:

$$n \cdot (k \cdot O_F + m \cdot k_0 + C_F) \tag{5.1}$$

where k is a small adjustment factor for the cost of performing the Cartesian product, and k_0 represents the overhead incurred by the algorithm for checking the emptiness of the input sets.

5.3 Shortcircuiting Evaluation Algorithm

The naïve evaluation algorithm first evaluates all the mapper functions and performs, thereafter, the Cartesian product. An interesting observation is that whenever the result of a mapper function is an empty set, the Cartesian product of the function outputs will also be an empty set. This observation motivates the development of the *shortcircuiting evaluation algorithm*. This algorithm is inspired in the shortircuiting semantics of expression evaluation in programming languages like C or Java, and reduces the expected overall function evaluation costs. It works as follows: For a given tuple $t \in r$, instead of evaluating all the mapper functions, whenever a $f_{A_i}(t)$ returns $\emptyset$, the remaining functions are not evaluated, since $f_{A_1}(t) \times ... \times f_{A_m}(t) = \emptyset$ if $\exists 1 \leq i \leq m$ s.t. $f_{A_i} = \emptyset$. This algorithm relies on the evaluation of the mapper functions according to a predefined *evaluation sequence*:

DEFINITION 5.1**:** *Let F be a set with m mapper functions, a list $\omega = f_{A_1} \cdot ... \cdot f_{A_m}$ where each $f_{A_i} \in F$ and $1 \leq i \leq m$ is called an* evaluation sequence *of F. The set of all evaluation sequences of the set F will be represented by $\Omega(F)$.*

Algorithm 2 Shortcircuiting mapper evaluation

INPUT: r : the input relation
$\omega = f_{A_1} \cdot ... \cdot f_{A_m}$: a sequence of mapper functions
OUTPUT: s : the output relation
VARIABLES: t : an input tuple from s
i : index of the current mapper function
o_i : output of $\omega[i](t)$
shortcircuit : flag that indicates that an empty set was returned

```
1: s ← ∅;
2: for all t ∈ r do
3:    shortcircuit ← false
4:    for all ω[i] where 1 ≤ i ≤ m do
5:       o_i ← ω[i](t);
6:       if o_i = ∅ then
7:          shortcircuit ← true;
8:          exit for
9:       end if
10:   end for
11:   if ¬shortcircuit then
12:      s ← s ∪ (o_1 × ... × o_m)
13:   end if
14: end for
```

Given an evaluation sequence ω, $\omega[i]$ represents the ith function in the sequence. In addition, a mapper function $\omega[i]$ is said to *precede* the evaluation of a mapper function $\omega[j]$ in the sequence ω if $i < j$. Whenever $i = j - 1$, the evaluation of the function $\omega[i]$ is immediately followed by the evaluation of $\omega[j]$. A sequence ω that meets such criteria is indicated by the notation $\omega_{i \prec j}$.

One possible implementation of the Shortcircuiting evaluation algorithm is presented in Algorithm 2. Given an input tuple t, each mapper function $\omega[i]$ is first evaluated over t individually and then, if no empty result is found, the Cartesian product is performed. During the function evaluation, the result of each function is checked for emptiness. If an empty result is found, the `shortcircuit` flag is set and the function evaluation loop is immediately abandoned. The Cartesian product is computed only if `shortcircuit` is not set, otherwise the current tuple is discarded and the next tuple is fetched from the input relation.

The Shortcircuiting algorithm reduces the evaluation cost because some functions will not be evaluated. Additionally, the Cartesian product operation is evaluated only when no function returns an empty set. To determine the ex-

pected overall cost of evaluating a mapper using this algorithm, the probability of evaluating each mapper function has to be estimated first.

The selectivity factor α_i can be seen as the probability that the function denoted by $\omega[i]$ produces an empty set. In a sequence ω, a function $\omega[i]$ is evaluated if none of its predecessors returns an empty set, i.e., if $\forall 1 \leq l < i,\ w[l](t) \neq \emptyset$. Since selectivities of the mapper functions are independent variables, the probability of evaluating the ith function, on an evaluation sequence $\omega = f_{A_1} \cdot ... \cdot f_{A_m}$, represented by $P^{\omega}(f_{A_i})$ is defined as:

$$P^{\omega}(f_{A_i}) = \begin{cases} 1 & \text{if } i = 1 \\ \prod_{1 \leq j < i}(1 - \alpha_j) & \text{if } 1 < i \leq m \end{cases}$$

The expected per-tuple cost of a mapper function for a given evaluation sequence is defined as $C^{\omega}_{f_{A_i}} = P^{\omega}(f_{A_i}) \cdot C_{f_{A_i}}$. The Cartesian product is performed only if the last function f_{A_m} is evaluated and its result is not empty. Hence, the expected cost of the Cartesian product is $P^{\omega}(f_{A_m}) \cdot C_{prd}$. Let C^{ω}_F represent the expected cost of evaluating all functions of F according to the sequence ω. Given an input relation r with cardinality n, the estimate of the cost of $\mu_F(r)$ using the Shortcircuiting evaluation algorithm is:

$$n \cdot \left(P^{\omega}(f_{A_m}) \cdot C_{prd} + C^{\omega}_F\right) \tag{5.2}$$

which expands to

$$n \cdot \left(P^{\omega}(f_{A_m}) \cdot k \cdot \prod_{f \in F} O_f + m \cdot k_0 + \sum_{f \in F} C^{\omega}_f\right) \tag{5.3}$$

Determining the cheapest evaluation sequence

The Shortcircuiting algorithm presented above applies the mapper functions using a fixed sequence given *a-priori*. However, since different evaluation orders may imply different per-tuple costs, computing the most favorable evaluation order can have a dramatic impact in performance. Thus, it is important to determine the *optimal evaluation sequence* of a set of mapper functions, defined as the evaluation sequence that minimizes the total function evaluation cost, while executing the

mapper operator using the Shortcircuiting algorithm. The notion of optimal evaluation sequence is formalized below:

DEFINITION 5.2: *An evaluation sequence ω of a set of mapper functions F is said to be* optimal *iff no evaluation sequence ω' is cheaper than ω*[1]*. In other words, ω is optimal iff, $\forall \omega' \in \Omega(F),\ C_F^{\omega} \leq C_F^{\omega'}$.*

The criterion supplied in Definition 5.2 is useful for deciding whether an evaluation sequence is optimal, but it does not provide a way to compute it. In order to construct an optimal sequence, mapper functions can be ordered according to a metric adapted from the notion of *rank order* (Hellerstein & Stonebraker, 1993). This metric is presented in Definition 5.3 and enjoys the desired property of rendering an optimal sequence for evaluating mapper functions when the provided average function costs and selectivity estimates are accurate.

DEFINITION 5.3: *The* rank *of a mapper function $f \in F$ represented as $rank(f)$ is defined as $rank(f) = C_f/(1 - \alpha_f)$, where C_f and α_f are the cost and selectivity of the function f, respectively. Furthermore, two functions $f, g \in F$ are said to be* rank ordered *if $rank(f) \leq rank(g)$.*

The idea behind such rank ordering metric is that the most selective functions that are at the same time cheaper should be evaluated first. Before proceeding to the main result, we establish that evaluation sequences where mapper functions are rank ordered are cheaper than those where functions are not rank ordered.

LEMMA 5.1: *The cheapest ordering of two mapper functions in an evaluation sequence corresponds to the ascending rank order. Formally, for any sequence $\omega_{j \prec i} \in \Omega(F)$ we have $C_F^{\omega_{i \prec j}} \leq C_F^{\omega_{j \prec i}}$ iff $rank(\omega[i]) \leq rank(\omega[j])$.*

PROOF See Appendix A.2

Using Lemma 5.1, we establish that any evaluation sequence ω that corresponds to the ascending rank order of the mapper functions is optimal for the Shortcircuiting algorithm.

[1]Given two evaluation sequences $\omega_1 \in \Omega(F)$ and $\omega_2 \in \Omega(F)$, if $C_F^{\omega_1} < C_F^{\omega_2}$, then evaluating of the Shortcircuiting algorithm with the sequence ω_1 is said to be *cheaper* than using ω_2.

THEOREM 5.1: *Given a set of mapper functions F, every evaluation sequence $\omega \in \Omega(F)$, which corresponds to an ascending rank ordering of the mapper functions is optimal for the Shortcircuiting algorithm.*

PROOF See Appendix A.3

This theorem provides a principled way to determine, beforehand, the cheapest evaluation order of the mapper functions for the Shortcircuiting algorithm.

5.4 Cache-based Evaluation Algorithm

Mappers are evaluated against input relations that usually contain duplicate values in certain columns. When evaluating a mapper, this characteristic of the source relation can be explored through *caching* of mapper function results. Whenever a duplicate input value is presented to a mapper function, the result that has been previously stored in the cache is returned. As a consequence, superfluous function evaluations are bypassed.

The implementation of the mapper operator can take advantage of caching as given in Algorithm 3. Each time a function $f_{A_i}(t)$ needs to be evaluated, a cache C is checked for a previously stored result. Each element $e \in C$ is known as a *cache entry* and takes the form of $\langle t[Dom(f_{A_i})], f_{A_i}(t)\rangle$[1]. A cache is usually encoded through a *hash table* (Cormen *et al.*, 2001) and is accessed through *lookup* operations using $t[Dom(f_{A_i})]$ as the key. Since mapper functions can have the same input domain, the function name, f_{A_i}, must also be supplied to the lookup operation (line `4`). The result of the lookup is then stored in the variable r_i (line `4`). A lookup operation that succeeds in finding a previously stored result is designated as a *cache hit*, otherwise it is designated as a *cache miss* (line `5`). In the latter case, $f_{A_i}(t)$ must be evaluated (line `6`). If the cache buffer is not full yet, a new entry is added to the buffer (line `8`).

For performance reasons, the cache buffer is maintained in main memory and has a limited size. Hence, the amount of memory devoted to caching the results of each function is limited. This will be specially true if the mapper operator is

[1] $t[Dom(f_{A_i})]$ represents the input of a mapper function f_{A_i}. See Chapter 3.

Algorithm 3 Cache-based mapper evaluation

```
INPUT:      r : the input relation
            f_{A_1}, ..., f_{A_m} : the mapper functions
OUTPUT:     s : the output relation
VARIABLES:  t : an input tuple from s
            i : index of the current mapper function
            r_i : the result of evaluating f_{A_i}(t)
 1: s ← ∅;
 2: for all t ∈ r do
 3:    for all 1 ≤ i ≤ m do
 4:       r_i ← lookup(f_{A_i}, t[Dom(f_{A_i})])
 5:       if r_i = ⊥ then
 6:          r_i ← f_{A_i}(t)
 7:          if ¬isfull() then
 8:             insert(f_{A_i}, t, r_i)
 9:          else
10:             replace(f_{A_i}, t, r_i)
11:          end if
12:       end if
13:    end for
14:    s ← s ∪ (r_1 × ... × r_k)
15: end for
```

plugged in the query processor of an RDBMS, since the memory space available for caching mappers has to be shared with other operators. As a result, the number of entries that have to be stored outgrows the buffer cache size. When the cache becomes full, existing cache entries are discarded and replaced by new ones (line `10`). In order to perform such replacement operation, the entry to be replaced (known as the *victim*) has to be identified (see Appendix B).

The cache algorithm assumes that the function results stored in the cache will be requested in the future, thus reducing the number of function evaluations. The algorithm can be easily extended to support multiple replacement strategies by substituting the implementation of the *replace* procedure (line `10`).

The performance of the Cache-based algorithm is also influenced by the cost of each mapper function and by the number of duplicates of its input attributes. In order to be subject to caching, a mapper function must meet two criteria:

i) It must not be to cheap —the average cost of storage and lookup c_0, may not exceed the average cost c of computation.

ii) Its input must contain a minimum of duplicate values —the savings produced, by eliminating the associated computations must offset the caching overhead.

Clearly, a constant mapper function is not a good candidate for caching. Mapper functions converting attributes that constitute the key of a relation are clearly not good candidates for caching, since all input values are distinct. Moreover, the caching of this function would jeopardize the caching of the remaining functions, since it would be competing for the same cache space without providing any advantage.

Although Algorithm 3 considers that all mapper functions are cached, it can be extended to consider caching a subset of the supplied mapper functions. However, to simplify its presentation as well as the presentation of the forthcoming algorithms, it is assumed that all the functions are cached.

In the following sections, three cache replacement strategies studied for Algorithm 3 are discussed in detail.

5.5 LRU Caching Strategy for Mapper Functions

The LRU (Least Recently Used) strategy explores temporal locality: it assumes that the least used entry is the least likely to be requested in the near future. The strategy of replacing the least recently used entry can be implemented very efficiently by linking the nodes of the hash table as a stack. New entries are added to the top of the stack. When the cache becomes full, the entry at the bottom of the stack is discarded to make room for the new entry. Each time an entry is referenced, it is also moved to the top of the stack. This way, the most recently used entries are maintained nearer the top of the stack, while those that have been not referenced fall to the bottom. The resulting algorithm has complexity $O(1)$.

One important property of the LRU stack is that the depth of an entry in the stack implicitly encodes the *time-to-last* reference to that entry. By replacing the entry at the bottom of the stack, LRU replaces the entry with the greatest *time-to-last* reference. In formal terms, let t_l represent the instant of the last

Algorithm 4 Cache-based mapper evaluation with LRU replacement

```
INPUT:      r : the input relation
            f_{A_1}, ..., f_{A_m} : the mapper functions
OUTPUT:     s : the output relation
VARIABLES:  t : an input tuple from s
            i : index of the current mapper function
            r_i : the result of evaluating f_{A_i}(t)
 1: s ← ∅;
 2: for all t ∈ r do
 3:    for all 1 ≤ i ≤ m do
 4:       r_i ← lookup(f_{A_i}, t[Dom(f_{A_i})])
 5:       if r_i ≠ ⊥ then
 6:          pull(r_i)
 7:       else
 8:          r_i ← f_{A_i}(t)
 9:          if isfull() then
10:             pop()
11:          end if
12:       end if
13:       push(f_{A_i}, t, r_i)
14:    end for
15:    s ← s ∪ (r_1 × ... × r_k)
16: end for
```

reference to some cache entry $e \in C$. At each instant t_0 where $t_0 > t_l$, the LRU replacement strategy minimizes the time-to-last of the reference of the entries in stored the cache by replacing the entry with greater $t_0 - t_l$.

Algorithm 4 gives the cache based evaluation algorithm that incorporates the LRU replacement strategy. Each time the lookup operation results in a *cache hit*, the entry is taken out of the stack trough the *pull* operation (line 6). Otherwise, it is a *cache miss*, and therefore the result of $f_{A_i}(t)$ is computed (line 8). If the cache is full, the bottom entry is discarded by the *pop* operation (line 10). Finally, the entry is pushed to the top of the stack by the *push* operation (line 13). Pushing an entry that was previously pulled corresponds to the *move* operation of an entry to the top of the stack.

5.5.1 Limitations

The LRU replacement strategy is prevalent in caches of operating systems and database systems. For decades, is has been empirically shown that LRU achieves

a very good performance by replacing the entry with the greatest *time-to-last* reference. However, this strategy performs badly in the following two situations: it is vulnerable to cache pollution and it does not cope with the variability of cache entry cost (Casey & Osman, 1974).

Vulnerability to cache pollution. Since LRU cannot discriminate well between frequent and infrequent entries (Lee *et al.*, 1999), its behavior is affected by two forms of pollution: singleton references and burstiness. Singleton references are entries that are referenced only once, when they are added to the cache. Due to the limited size of the cache, these entries will cause other entries that are frequently referenced to be flushed (O'Neil *et al.*, 1993; Rizzo & Vicisano, 2000) out. Bursty workloads are characterized by short intervals where a large number of hits are directed toward a limited number of entries (O'Neil *et al.*, 1993). It is often the case that very frequently accessed entries stay dead (without being referenced) in the cache for a long time. LRU moves a recently accessed entry to the top of the stack because it assumes that the references to an entry are correlated in time and hence that the entry will be reused often in the near future (Jiang & Zhuang, 2002). However, this correlation is application dependent. For example it does not apply to in the case of RDBMS page caches (Robinson & Devarakonda, 1990). Until the entry is purged, it remains occupying cache space and contributing for artificially lowering the available cache space. LRU is unable to detect and remove entries that are not likely to be referenced in the future.

Uniform cost assumption. LRU does not take into account the cost of computing an entry. In fact, it considers that the costs of all entries are homogeneous. As result, expensive entries are often replaced to the benefit of cheaper entries.

The above mentioned insufficiencies of LRU are particularly acute in the context of the mapper operator, where the cache is required to handle several functions simultaneously. The presence of multiple functions presents workloads with different characteristics competing for the same cache space. It has been recognized

that in real databases it is often the case that some values for a given attribute occur more frequently than others (Lowe, 1968). Furthermore, the distribution of values often follows the Zipfian power-law distribution. In these situations, many input values are referenced only once and a few values are reference many times (Zipf, 1949). Informally, this translates to a few cache entries being very often referenced while the majority of the cache entries are seldom referenced. Additionally, the cost of mapper functions is variable. The cost of evaluating a mapper function over distinct input values can also vary significantly. Replacing an entry without taking into account the cost of materializing it in the cache often results in the replacement of expensive entries that will be needed in the future (Casey & Osman, 1974).

5.5.2 Enhancements

LRU can be enhanced to minimize cache pollution and distinguish between cheap and expensive entries by using the following approaches:

Forcing entries to age at different speeds. Those entries which enjoyed periods of high frequency caused by bursts of references tend to stay resident in the cache for too long. One interesting mechanism used to minimize this effect is to consider as indistinguishable the bursts of references that occur in tiny intervals and treat them as one reference (O'Neil *et al.*, 1993). The problem with this approach is that the size of the interval is domain dependent and cannot be determined on-the-fly. An alternative approach consists of forcing entries referenced within small intervals to age faster than entries accessed during large periods. Since cache accesses usually represent a Zipfian distribution, there are more entries with low frequency access patterns than high frequency access patterns. Hence, given two cache entries with different average access frequencies, after some time, the most frequent one should be considered the less useful, because it has less probability of being seen again in the future. Both Lee *et al.* (1999) and Robinson & Devarakonda (1990) proposed enhancements to LRU that also take frequency into account, and result in more powerful replacement strategies.

Replacing entries with the least expected cost. In order to overcome the problem of replacing expensive entries by inexpensive entries, the *cost* of evaluating the mapper function must also be taken into account by the replacement strategy. A straightforward extension consists of replacing the entry with the *least expected cost* (LEC), which is determined by multiplying the average access frequency by the cost of materializing the entry in cache (Casey & Osman, 1974). Despite the fact that LEC is superior to LRU and LFU in contexts involving cost variation, this strategy does not try to minimize pollution.

As it turns out, any replacement strategy that optimizes a single parameter like time-to-last, frequency or cost is inherently limited. Hence, the replacement strategy for a cache that handles mapper function entries must consider all these parameters when deciding which entry to replace.

5.6 LUR Caching Strategy for Mapper Functions

As discussed, the LRU replacement strategy is limited due to performing replacement decisions based on the time-to-last reference only. As a consequence, the entries selected for replacement are frequently inadequate choices when aiming at reducing the total computation cost. Herein, this issue is addressed through a more sophisticated cache replacement strategy that maximizes the expected *utility* of the entries residing in the cache. The *utility of a cache entry* is defined as a function that takes as input the time-to-last reference together with the number of accesses, the access frequency and the cost of evaluating the mapper function to obtain each entry. This new strategy is designated as *Least Useful Replacement* (LUR), since it replaces the entry that is estimated to be the *least useful* according to the proposed metric of utility. The use of utility functions for driving cache replacement decisions has been addressed in literature related to Web proxy caching (Cao & Irani, 1997).

5.6.1 Utility metric for cache entries

The utility of a cache entry is computed from a record of its past reference information. However, saving comprehensive past reference information is demanding in terms of space and, more importantly, computation effort. Thus, only a summary of the past reference history is maintained in the cache entries. Each cache entry takes the form $\langle t, f_{A_i}(t)\rangle_h$, where $h = \langle t_a, t_l, n_h, c\rangle$ is a reference history data structure. The instant of the first reference to the entry (time of arrival), is recorded in the component t_a. The instant of the last reference to the entry (time of last reference) is recorded in t_l, where $t_l > t_a$. The number of references to the entry within the interval $[t_a, t_l]$, is kept in n_h. Finally, the cost of evaluating $f_{A_i}(t)$ is represented by c. Like in LRU, time is measured as a discrete event count, but in this case it is associated with the number of tuples processed so far.

For a given cache entry e in a cache C, at some instant t_0, its *utility*, represented as $u_{t_0}(e)$, is computed taking as input:

i) t_l – the time of the last access.

ii) θ – the average access frequency of the entry;

iii) n_h – the number of observed past references (i.e., hits) to the entry;

iv) c – the cost of evaluating the function;

These parameters can be combined to produce an utility metric that corresponds to the *expected benefit* of keeping an entry in cache. This metric aims at minimizing the cache pollution and overcoming cost heterogeneity, i.e., managing entries with different costs, by considering as the most useful those entries that are more likely to be referenced in the future and are simultaneously are more expensive to compute.

The *frequency of an entry* e, represented as θ, is computed as the number of references to the entry divided by the length of the interval $[t_a, t_l]$. Since all the references to the entry (n_h in total), occur within the interval $[t_a, t_l]$, the access frequency of the entry is defined as $\theta = n_h/(t_l - t_a)$, when $n_h \geq 2$. The *average inter-reference interval* of e, represented by p, is $1/\theta$ (Coffman Jr. & Denning, 1973, Section 7.3.1).

The expected future usage of an entry can be estimated using statistical inference. Let e be an entry with average access frequency θ. According to the literature in statistics, the variable K that models the number of cache accesses (experiments) before the next hit to e (a success) is modeled through a Geometric distribution $\mathcal{G}(\theta)$. Such Geometric distribution is accurate under the *independent reference model* (IRM) commonly used in the analysis of cache replacement strategies (Coffman Jr. & Denning, 1973, pg. 268). This model assumes that the probabilities to reference different cache entries are independent and identically distributed random variables.

Given an entry e whose instant of the last reference is t_l, the number K of cache accesses after t_l before the next hit to e, has probability function $P(K = k)$ defined as $(1-\theta)^{k-1} \cdot \theta$ for $k > 0$[1]. The probability function $P(K = k)$ defines the probability that the entry e is referenced in the last of a sequence of k cache accesses and its average is $1/\theta$, which also represents the average inter-reference interval of the entry e. The probability that the next reference to the entry e takes place in the future, after k accesses to the cache, can also be computed. This amounts to determining the probability of an hit on e after $t_l + k$ cache accesses, which is given by $P(K \geq k)$. In the case of a Geometric distribution with parameter θ, it equates to $(1-\theta)^k$. The probability of future reference, can be combined with the amount of past references and the cost of the entry into an utility metric as follows:

DEFINITION 5.4**:** *Let e be a cache entry with average access frequency θ, instant of last reference t_l, cost c and n_h recorded references within the interval $[t_a, t_l]$. The* utility of an entry e, at instant $t_0 > t_l$, *denoted by $u_{t_0}(e)$, is defined as $n_h \cdot c \cdot (1-\theta)^k$, where $k = t_0 - t_l$ represents the* time to last reference.

Besides considering the entries with the highest probability of being referenced in the future to be the most useful, this utility metric addresses cache pollution and cost heterogeneity by considering entries with bursty accesses patterns that

[1] Consider a sequence of cache accesses that reference the cache entry e with probability θ. Since, $1-\theta$ represents the probability of not accessing e, $(1-\theta)^{k-1} \cdot \theta$ represents the probability referencing the entry e in the last of a sequence of k cache accesses.

are cheaper to be less useful than entries with uniform access patterns that are expensive.

Definition 5.4 penalizes entries with high access frequencies confined in short periods, which are *bursty*, in favor of entries that exhibit more uniform access patterns in two ways. First, it considers highest frequency entries to be less useful since the utility of the entries decreases toward zero with the increase of the frequency[1]. The more intense the burst is, the faster the utility of the entry drops. Second, burstiness is penalized by considering as most useful the entries that have been in cache for a longer period. Definition 5.4 considers as more useful those entries with a greater number of past references n_h (this is demonstrated analytically in Appendix A.4). This definition captures the intuition that, given two entries with the same frequency, the entry that has been in the cache for the longest time is the less likely to be a burst.

Definition 5.4 addresses cost heterogeneity by selecting the cheapest entries for replacement and favoring the maintenance of the most expensive entries in the cache. It considers the the cost c incurred in evaluating the mapper function to create the cache entry. As a result, the most expensive entries (those with greatest values of c) are considered the most useful.

5.6.2 Complexity

At each instant t_0, the LUR strategy tries to maximize the overall cache utility by replacing the entry $e \in C$ with the smallest utility $u_{t_0}(e)$, and thus maximizing the expected utility of the whole cache. The algorithm always chooses as victim the cache entry with the smallest utility.

The cost of evaluating an entry is c, on the first time that it is referenced. Each subsequent reference to the entry saves $c - c_0$, where c_0 is a small constant access cost that represents the overhead of performing a cache lookup. In general, c is much higher than c_0. Otherwise, if $c \sim c_0$, the overhead of caching cancels its benefit. Thus, an implementation of the LUR algorithm must choose between one of the following approaches:

[1]Note that $\lim_{\theta \to 1} (1-\theta)^k = 0$.

i) Paying an overhead for each cache access to maintain the entries ordered by utility —as a result, the least useful entry can be quickly identified.

ii) Not paying an overhead for each cache access and not maintaining any ordering of the entries —each time a victim needs to be chosen, a direct search for the least useful cache entry is performed.

The first alternative, which consists of maintaining the m cache entries permanently ordered has been addressed in literature by implementing a priority queue such as e.g. Greedy Dual (Cao & Irani, 1997), resulting in a complexity of at least $O(log(m))$. In the case of LUR, the complexity of maintaining the entries ordered by priority is at least $O(m)$. According to the utility metric proposed in Definition 5.4, the ordering of entries e_1 and e_2 may change even if none of them is accessed. For some t_0, it can be the case that $u_{t_0}(e_1) < u_{t_0}(e_2)$, but $u_{t_0+1}(e_1) > u_{t_0+1}(e_2)$, because the utility of the entries changes as time elapses since the distance of t_0 to the corresponding instants of last reference t_l increases. Thus, virtually, for each cache access, the entire set of entries needs to be reordered, implying a cost per reference of $O(m \cdot log(m))$. Even if some smart strategy could be devised to keep the entries that are not accessed ordered, the complexity would at least be proportional to $log(m)$. The second alternative, not maintaining any ordering, means that each time a cache miss occurs the least useful entry must be found. This represents a complexity of $O(1)$ to handle a cache hit, but implies a complexity of $O(m)$ to handle a cache miss, since the least useful entry has to be found via direct search.

As discussed, the complexity of the LUR replacement strategy is at least $O(log(m))$. This complexity can be acceptable for managing disk pages and Web documents. Replacement strategies for disk caches, like LFU or LRU-K (O'Neil *et al.*, 1993), as well as many replacement strategies for web proxy cache management run proportionally to $log(m)$. However, in the case of a cache for mapper functions, a complexity of $O(log(m))$ can be too high for practical applications. The first argument is that, to be useful, a cache replacement strategy for mapper functions has to be lighter in terms of computation than a cache management policy for disk pages or web documents. In fact, the cost of a mapper function cache entry is often smaller than the cost of transferring a disk page, —because

evaluating a mapper function is cheaper than transferring a page from secondary storage. Moreover, in the context of the mapper operator, the same amount of memory holds many more cache entries than a disk page cache or a document cache of a Web proxy, because cache entries of mapper functions occupy less space than disk pages. As an illustration, consider an entry for caching a typical mapper function used for cleaning names that takes as input a string with 50 characters and produces 50 characters as output. Each entry occupies around 100 bytes, which is 40 times smaller than an entry comprising a 4K page. A similar reasoning applies when comparing the mapper cache performance that of Web proxy caches, where documents occupy several Kilobytes.

5.7 XLUR Caching Strategy for Mapper Functions

As explained before, determining the entry which has the absolute minimum utility has a very high computational cost. Hence, LUR is only feasible for caching mapper functions that are relatively expensive. Herein, a new replacement strategy is proposed which improves on LUR by replacing entries whose utility is an approximation of the entry with absolute minimum utility. Its goal is to replace entries with low utility and low runtime overhead. The new strategy, henceforth designated as Relaxed LUR (XLUR), has a complexity $O(1)$ since it relies on maintaining multiple LRU queues.

Because a complexity of $O(log(m))$ can be unacceptable in terms of performance, several authors have considered improving LRU in order to make smarter replacement decisions, maintaining the $O(1)$ complexity. The improvements are built on two basic ideas. The first consists of avoiding the insertion of low frequency items at the top of the stack. The second, consists of removing entries before they get to the bottom of the stack, as soon as it is known that they were used for the last time. These ideas are implemented through a mechanism that actively separates entries that are frequently accessed (*hot* entries) from those that are seldom accessed (*cold* entries) by promoting and demoting entries be-

Algorithm 5 Cache-based mapper evaluation with XLUR replacement

INPUT: r : the input relation
$f_{A_1}, ..., f_{A_m}$: the mapper functions
$L = \{l_1, ..., l_q\}$ of lru stacks
OUTPUT: s : the output relation
VARIABLES: t_0 : the current instant
t : an input tuple from s
i : index of the current mapper function
$r_i\langle t_a, t_l, n, l\rangle$: entry information for the function f_{A_i} representing a result r_i, with time of arrival t_a, time of last t_l, number of accesses n and stack l.
l^{new} : index of the new stack of an updated entry
l^{victim} : index of the stack with the less useful LRU entry

1: $s \leftarrow \emptyset$
2: $t_0 \leftarrow 0$
3: **for all** $t \in S$ **do**
4: **for all** $1 \leq i \leq m$ **do**
5: $r_i\langle t_a, t_l, n, l\rangle \leftarrow lookup(f_{A_i}, t[Dom(f_{A_i})])$
6: **if** $r_i\langle t_a, t_l, n, l\rangle \neq \perp$ **then** {update an already existing entry}
7: $pull\big(r_i\langle t_a, t_l, n, l\rangle\big)$
8: $l^{new} = min\Big\{log_2\Big(\lfloor \frac{t_0 - t_a}{n+1} \rfloor\Big), l_q\Big\}$
9: $push\big(f_{A_i}, t, r_i\langle t_a, t_0, n+1, l^{new}\rangle\big)$
10: **else** {insert a new entry}
11: **if** $isfull()$ **then**
12: $l^{victim} = l \in L$ such that $u_{t_0}\big(lru(l)\big) = \underset{l' \in L}{min}\big\{u_{t_0}(e) \mid e = lru(l')\big\}$
13: $pop(l^{victim})$
14: **end if**
15: $r_i \leftarrow f_{A_i}(t)$
16: $push(f_{A_i}, t, r_i\langle t_0, t_0 - |L|, 1, l_q\rangle)$
17: **end if**
18: **end for**
19: $t_0 \leftarrow t_0 + 1$
20: $s \leftarrow s \cup (r_1 \times ... \times r_k)$
21: **end for**

tween LRU queues and auxiliary data structures (like other LRU stacks or FIFO queues).

The driving idea of the XLUR replacement strategy proposed here is to manage cache entries through multiple LRU stacks, where each stack contains entries with different frequencies. Since the stack is a data structure that does not allow direct access, entries cannot be inserted or removed from arbitrary positions in the stack. The usage of multiple LRU stacks endows a virtual LRU stack partitioned according to access frequencies. The stacks $L = \{l_1, ..., l_q\}$ are designated as *frequency clusters* since they contain entries with approximately the same ac-

cess frequency. The stack l_1 contains entries with highest frequencies while l_q contains the lowest frequency entries.

This replacement strategy is given in Algorithm 5 and works as follows: Each time a function needs to be evaluated, the algorithm checks if a value for that function has already been computed. On a cache hit, the entry is pulled from its current stack l and pushed onto the top of a new stack l^{new} that better reflects the new frequency of the entry $\theta = (n+1)/(t_0 - t_a)$ (lines `7-9`). The computation of the new stack is performed by taking $log_2(\theta^{-1})$ where θ^{-1} represents the average inter-reference interval of the entry (line `8`). The function log_2 was chosen, because it can be very efficiently implemented over an integer input value through bitwise operations.

On a cache miss, the entry with the least utility (from the bottom of all the stacks) is selected as the victim to throw away. The new entry is inserted into the last stack (lines `11-16`). The new entry is placed at the top of the last stack l_q with t_l set to the current instant t_0 and t_a set to $t_0 - |L|$ where $|L|$ represents the current size of the cache measured in number of entries. The newly installed entry is awaiting to be referenced again in the near future. The rational for placing it in the last stack is as to do with the fact that when an entry is referenced for the first time, no information about its frequency is available. Hence, it is placed in the stack l_q, which serves as a quarantine area for this entry: either the entry is referenced again, and as a result of having its frequency updated, it is moved into another stack, or reaches the end of stack and eventually gets selected as victim.

Analysis of the XLUR replacement strategy

The XLUR strategy continuously adapts the number of entries contained in each stack by moving referenced entries to stacks that better reflect their frequency or by selecting them as victims for replacement.

By arranging entries according to log_2 of their average access frequency, the stack l_i will tend to have 2^i entries. Each of the entries in l_i is referenced in average one out of 2^i times, otherwise the strategy will eventually move it to another stack or evict it. As it turns out, if $i > j$ then the stack l_i will hold, in average, more entries than l_j. Another way to look at it is to realize that

higher frequency stacks (those with smaller values of i) tend to have less entries. Moreover, the inter-reference interval p of the entries in the stack l_i is such that $2^{i-1} < p \leq 2^i$. Thus, the entries in the stack l_3, for instance, are expected to be accessed one out of every $2^2 + 1$ to 2^3 cache references. The next stack, l_4 will hold more entries, 16, and so on.

This mechanism addresses the problems of cache pollution and cost heterogeneity due to the following characteristics:

i) It prefers least recently used entries of each stack for victims, since the victim for replacement is always selected from a set consisting of the least recently used entry of each stack. The victim that will be chosen by LRU is always in the set of victims considered, i.e., the entry with the absolute greatest time to last reference. XLUR performs a better choice in terms of utility.

ii) It is more aggressive for high frequency entries, since XLUR checks the various stacks to determine the entry to replace. Hence, in relative terms, the entries of the highest frequency stacks are considered for replacement more often. This feature detects burstiness by guaranteeing that higher frequency entries are removed more aggressively.

iii) It prefers cheaper entries for replacement, because it takes cost into account when comparing the utilities of the bottom entries of the distinct LRU stacks with one another.

The appropriate number of stacks q should be the largest integer such that

$$\sum_{i=0}^{q} 2^i \;\leq\; m \tag{5.4}$$

For a cache that holds a total of m entries, q will be $\lfloor log_2(m+1) \rfloor - 1$. For a large cache, the number of stacks can be selected by considering log_4 or log_8. This results in less stacks. A smaller number of stacks means less comparisons when selecting a victim. Nevertheless, as it will be shown in the next chapter, eight stacks are often enough in practice to achieve good results.

5.8 Related Work

The mapper execution algorithms proposed require different subjects to be discussed in terms of related work. First, since mappers deal with expensive functions, the literature related to enhancing a query processor for handling expensive functions needs to be addressed. Second, the usage of caching to speed up query evaluation also needs to considered, since a cache-based evaluation algorithm is proposed for the mapper. Finally, since two new cache replacement strategies (LUR and XLUR) are proposed, other cache replacement strategies presented in literature have to be reviewed.

Query evaluation with expensive functions

The traditional System/R query optimization algorithm described by Selinger *et al.* (1979) is built on the simplifying assumption that the predicate evaluation cost is neglectable when compared with the I/O cost of the join algorithm. Hence, predicates that are estimated to be the most selective are evaluated as soon as possible in an attempt to reduce the number of tuples early on.

The work of Hellerstein & Stonebraker (1993) generalizes the criterion of predicate selectivity with that of predicate *rank*, where rank is a metric derived from the expected evaluation cost and selectivity. The authors prove that ordering expensive predicates on the join tree according to their rank result in an overall reduction of the evaluation cost. Informally, queries whose predicates are rank-ordered discard tuples earlier and at a lower average cost. The Shortcircuiting algorithm proposed in Section 5.3 draws on ideas similar to those described by Hellerstein & Stonebraker (1993): if mapper functions are rank-ordered, the tuples are discarded at a lower average cost because those functions that are cheaper and more selective are evaluated first. The Shortcircuiting algorithm also builds on the idea proposed by Hanani (1977) of ordering the Boolean factors in a *conjunctive normal form* according to their estimated selectivity, and take advantage of Boolean simplification laws to minimize the query evaluation time. Since the most selective predicates are evaluated first, even if their cost per tuple is very high, this approach fails to correctly optimize queries involving expensive predicates (Boolean expressions with expensive functions).

The work of Porto *et al.* (2003) takes the idea of Boolean simplification further. They propose to use an abstraction of the input relation modeled as hyper-graphs that guide the query processor, allowing it to adapt on-the-fly the query processing order and to skip unnecessary predicate evaluations. Optimization of queries with expensive predicates is studied in further detail in Hellerstein (1998) and Chaudhuri & Shim (1999).

Another approach for optimizing of queries involving expensive function calls consists of modeling expensive function calls as joins. This approach was initially proposed for extending the LDL system (Chimenti *et al.*, 1989). In LDL, the evaluation of an external predicate over a relation is processed as a join with an infinite virtual relation induced by the predicate. Expensive functions calls are modeled by Chaudhuri & Shim (1993) as joins with expensive *foreign tables*, which represent functions. This idea is extended by Mayr & Seshadri (1999) to handle client-side expensive functions in the context of distributed query execution (Kossmann, 2000). Their approach adapts the semi-joins of Bernstein & Chiu (1981) with external tables that represent functions to handle expensive function calls.

An important issue when modeling external functions as virtual relations is the fact that some the attributes must be bound to values prior to obtaining the actual relation tuples. This constraint turns invalid certain query plans. The requirement of binding the attributes prior to a function invocation was captured for the first time by the idea of *safety constraints* by Chaudhuri & Shim (1993). In the context of data integration, Florescu *et al.* (1999) proposed *binding patterns* and *directional joins* to address the issues of plan generation in the presence of the aforementioned constraints. Hergula & Härder (2001) propose several query rewriting strategies for accessing foreign functions.

Caching of function results was adapted to the context of query optimization of expensive functions in the predicate migration algorithm of Hellerstein & Stonebraker (1993). After pulling up a predicate to above the join, in order to guarantee the duplicates produced do not result in extra predicate evaluations, the algorithm relies on caching the results of expensive functions.

The idea of avoiding redundant calls to functions was taken one step further in the context of ORDBMSs by Hellerstein & Naughton (1996). However, the au-

thors do not consider cache replacement. They optimize the evaluation of a single expensive function by staging the input tuples to disk when the cache memory becomes full. Instead, the Cache-based algorithm proposed in this chapter relies on performing replacing cache entries once the cache becomes full. One difficulty of the proposal of Hellerstein & Naughton (1996) when applied to mappers, is that there is no clear way to extend the presented hybrid-hash algorithm to handle more than one expensive function.

Caching on RDBMSs

Literature on database caching focuses essentially page caching (Chou & DeWitt, 1985; Effelsberg & Haerder, 1984; Johnson & Shasha, 1994; O'Neil *et al.*, 1993). Some approaches to RDBMS buffer management like those presented by Chou & DeWitt (1985) and Sacco & Schkolnick (1986) propose to use separate sets of cache entries. Similarly to XLUR, each set of cache entries is managed separately as an LRU or MRU stack. These buffer management strategies explore the fact that the sequences of references to cache entries (pages) on an RDBMS can be predicted taking from the request pattern of the physical operators. This technique cannot be directly applied to mapper evaluation, since the mapper operator does not endow any specific cache access pattern. The cache access pattern of a mapper is determined by the sequence of values read from the input relation, and there is no simple a way to foretell which input values will be fed to the mapper functions, unless the input relation is analyzed first.

Database buffer cache management explores an access pattern known as *spatial locality*. Spatial locality means that, when an object is referenced, others nearby are also referenced. For example, if a page of a table is needed, other pages nearby, which contain related tuples, are also likely to be needed. Spatial locality is explored through *prefetching* (Smith, 1978). Prefetching can be used for evaluating mappers: whenever the tuples contained on a page are being transformed, tuples contained in contiguous pages are likely to be requested to be transformed.

The evaluation of queries involving expensive functions can be enhanced by auxiliary data structures. One such data structure is the *function index*, which

can be regarded as a pre-computed function or as a cache materialized in secondary storage. Function indexes have been proposed for extending RDBMS with user defined operators and abstract data types (Lynch & Stonebraker, 1988), and for supporting queries on Object Oriented DBMSs with method calls (Hwang, 1995; Maier & Stein, 1986).

Cache replacement strategies

Several enhancements have been proposed to LRU that try to overcome the cache pollution problem by actively separating frequently used entries from entries that are seldom used. Strategies like FBR (Robinson & Devarakonda, 1990), 2Q (Johnson & Shasha, 1994), LIRS (Jiang & Zhuang, 2002), MQ (Zhou *et al.*, 2001) and ARC (Megiddo & Modha, 2004) are based on this concept. These replacement techniques maintain $O(1)$ complexity and have been shown to outperform LRU in a number of situations. Broadly speaking, these replacement strategies partition the LRU stack into multiple regions (a quantity q) according to recency or frequency.

The XLUR strategy proposed in this chapter uses multiple LRU stacks. When the number of stacks q is one, the XLUR replacement strategy behaves like LRU. If, instead, q is a small number like two or three it can be compared to the above strategies FBR, 2Q, LIRS, and ARC. The FBR strategy can be implemented using three LRU stacks, ARC uses two LRU stacks, while 2Q and LIRS use an LRU stack and a FIFO queue. In a sense, these strategies make a discrete distinction between hot and cold entries by keeping them in in distinct data structures: hot entries in an LRU stack and cold entries either in a second LRU stack or in a FIFO. For small values of q, these strategies are likely to be more effective at distinguishing hot from cold entries based on the time-to-last-reference and on the access frequency. Hence, if the cost variation is small, they are likely to achieve a better performance than XLUR. However, as q increases, XLUR is presumably better, since it considers more victim candidates when performing a replacement decision instead of only two (one of each data structure).

When q is three or more, XLUR can be compared with the MQ strategy. Like the XLUR, on each cache hit, the MQ strategy adjusts the entries according to

$log_2(\theta)$ where θ is the access frequency. However, unlike XLUR, in MQ, entries at the bottom of a queue are demoted to the next queue as their frequency drops. Instead, XLUR never demotes entries: the bottom entries are only considered for replacement. The demotion mechanism has two problems:

i) Increases the per-reference overhead of MQ. Each time the cache is accessed all the queues have to be adjusted incurring in on a complexity $O(q)$ even on a cache hit. In contrast, XLUR performs no demotion and has complexity $O(1)$ on a cache hit.

ii) Contributes to cache pollution. Each time a frequent entry is demoted, it enters the top of the next LRU stack. Hence, it will only be considered for replacement after traversing to the bottom of the next stack, which has more entries. In contrast, XLUR does not demote it. Either the entry is referenced again, deserving to live in the current frequency stack, or is replaced.

The behavior of MQ also differs from XLUR on a cache miss: In MQ, the entry to be replaced is the last entry of the first non-empty stack. Hence, on a cache miss, a number of queues, at least one and at most $q-1$, have to be checked and each of the LRU stacks has to be adjusted afterwards. In XLUR, all non-empty queues, at most q, have to be checked. However, unlike MQ, no adjustments to LRU stacks take place in XLUR. Finally, XLUR considers the cost of evaluating the mapper functions, which is not taken into account by MQ.

However, unlike the LUR and XLUR strategies, the strategies considered above do not take the cost of computing an entry into account. The idea of exploring an utility function, like the one proposed for the LUR and XLUR algorithms, to perform cache replacement has been widely employed in cache replacement strategies for Web Proxies. In fact, these caches have to deal with similar problems to those involved in caching mapper functions. The requests handled by Web proxy caches are Zipfian distributed (Breslau *et al.*, 1999), and the size and cost of the document caches is variable (Wang, 1999). Several replacement strategies for Web Proxy caching that have been proposed achieve results better than traditional replacement strategies. Notably, Greedy Dual (Cao & Irani,

1997), Hybrid (Wooster & Abrams, 1997), LNC (Scheuermann *et al.*, 1997) and LRV (Rizzo & Vicisano, 2000) employ utility functions defined over several parameters.

5.9 Conclusions

This chapter proposed several execution algorithms for the mapper operator. The first was the Naïve algorithm induced by the formal semantics of the mapper operator. Then, alternative algorithms were explored, aiming at reducing the overall computation cost of the mapper operator. The proposed algorithms are built on the idea of avoiding superfluous mapper function evaluations by exploring the selectivity of mapper functions and the existence of duplicate values in attributes of the input relation.

The Naïve algorithm should be used when very few duplicates are present and when the functions are not likely to return empty sets. The Shortcircuiting algorithm is to be used when we have costly mapper functions mixed with functions that may return empty sets. Finally, cache based algorithms should be used when expensive functions that operate over inputs with duplicates are present.

The performance of the Shortcircuiting Algorithm is influenced by the order by which mapper functions are evaluated. It is possible to compute the optimal evaluation order based on statistics of the cost and selectivity of mapper functions. One limitation is that the algorithm does not react to skewed data. However, the algorithm can be enhanced to adjust on-line the function evaluation sequence in order to react to changes of statistics of cost and selectivity. The enhancement consists of keeping the list of functions ordered according to rank by moving a function that returns an empty set to the head of the function evaluation list. Although this strategy does not result in an optimally rank-ordered list, it consists of an approximation that can be implemented with a small computation effort.

To take advantage of duplicates, this chapter proposed a Cache-based algorithm that hashes the results of mapper functions using the function input as key. However, evaluating a mapper using this technique requires disproportionate amounts of main memory. Thus, cache entries that are not likely to be used are replaced to provide room for the newer ones. The entry replacement policy

has a major impact on the performance of the cache based evaluation algorithms. Thus, the study of the Cache-based algorithm proceeded by analyzing different possibilities to perform entry replacement.

The first cache replacement strategy considered was the least recently used (LRU). This strategy bases its replacement decisions solely on the time to last reference. Unfortunately, when duplicates have a high variation, as it is the case of Zipfian distributed data (of database relations), or when the cost of materializing the entries is not uniform, this strategy performs very poorly. Hence, the reference frequency and the cost of computing an entry also have to be taken into account.

This chapter presents two new cache replacement strategies, that attempt to circumvent the shortcomings of LRU, designated as LUR (Least Useful Replacement) and XLUR (relaXed Least Useful Replacement). The LUR strategy replaces the least useful entry and is built on a generalized notion of utility metric, which accounts not only for the time to last reference, but also for computation cost and access frequency. To perform replacements based on utility, the LUR algorithm requires the ordering of entries to be maintained using a priority queue. Thus, this algorithm has a complexity of at least $O(log(m))$ per-reference, where m is the number of cache entries. Since m is very high in the case of mapper functions, it is unlikely that this algorithm can be useful in practice, except perhaps in the cases where the mapper functions are expensive. XLUR, a modification to the LUR strategy, was proposed to overcome this problem. XLUR uses multiple LRU stacks to organize entries according to their access frequencies. Contrarily to LUR, this replacement strategy is scalable: as the cache size grows, it maintains a complexity of $O(1)$ for every cache hit and a complexity $O(q)$ for cache misses, where q is the number of LRU stacks of the algorithm.

Chapter 6

Experimental Validation

This chapter reports on a number of experiments aimed at validating the feasibility of the mapper operator, including the logic optimizations and physical execution algorithms proposed. First, the adequacy of RDBMSs to execute and optimize one-to-many data transformations is studied and compared with a mapper implementation. Second, the logical optimizations are validated by contrasting the response time required to evaluate expressions involving the mapper operator with its optimized equivalents. Finally, this chapter presents the improvements obtained by the Shortcircuiting and Cache-based algorithms introduced in the previous chapter.

6.1 Introduction

The first part of the thesis studies the problem of expressing one-to-many data transformations, starting by comparing several alternatives for implementing one-to-many data transformations (Chapter 2), and then proposing the mapper operator (Chapter 3). It became clear that none of the alternatives studied is at the same time declarative and sufficiently expressive for tackling one-to-many data transformations. In addition, these alternatives are not the most adequate in terms of performance when executing one-to-many transformations.

The second part of the thesis handled the problem of executing one-to-many data transformations efficiently. This issue was addressed by proposing a set of algebraic re-writing rules (Chapter 4), together with different physical execution

algorithms for executing the mapper operator (Chapter 5). These proposals enable the logical and physical optimization of expressions that combine mappers with standard relational operators to express one-to-many transformations.

The claim that the current solutions for implementing data transformations have difficulties in handling one-to-many data transformations is validated by comparing the performance of different RDBMS implementations of one-to-many data transformations. Additionally, these implementations are also compared with an implementation of the mapper operator to validate its usefulness from a performance standpoint. These experiments are presented in Section 6.2.

The logical optimization rules are compared by contrasting the original expressions with their optimized equivalents. The results obtained are described in Section 6.3. The usefulness of the different physical algorithms is validated through a set of experiments that compare their performance in different situations, described in Section 6.4. Both sets of experiments present the factors that influence the execution performance and optimization gains.

Finally, Section 6.5 reports on the usefulness of the implementation of the mapper operator in Data Fusion, a commercial product that has been selected for several real-word legacy data migration projects.

6.2 Performance of One-to-many Data Transformations

This section studies the performance of alternative implementations of one-to-many data transformations. The factors that influence the performance of one-to-many data transformations are identified and the optimization opportunities of each solution are examined.

The performance study is based on implementations of the one-to-many data transformations that correspond to Examples 1.1.1 and 1.1.2 developed using RDBMS solutions, namely relational queries, recursive queries, table functions, stored procedures and also using the mapper operator. For conciseness, this chapter uses the acronyms and abbreviation B for bounded, U for unbounded, TF for table function, SP for stored procedure, and Rec for recursive query.

Mechanisms for implementing one-to-many data transformations								
	Bounded				Unbounded			
	Union	Table Function	Stored Procedure	Mapper	Recursive Query	Stored Procedure	Table Function	Mapper
DBX	yes	no	yes	no	yes	yes	no	no
OEX	yes	yes	yes	no	no	yes	yes	no
XXL	no	no	no	yes	no	no	no	yes

Table 6.1: Mechanisms for implementing the one-to-many data transformations performed for the experiments.

Table 6.1 shows the entire set of implementations that were considered. The experiments compare the results of RDBMS solutions with the results obtained with the implementation of the mapper operator. RDBMS implementations are executed on top of two industry leading commercial systems henceforth designated as DBX and OEX[1]. The mapper operator is implemented as a relational operator, using the Naïve algorithm presented in Chapter 5, on top of the XXL library which provides database query processing and optimization functionalities (van den Bercken *et al.*, 2000, 2001).

Due to the limitations of the RDBMSs used, some of the mechanisms available to implement one-to-many data transformations could not be used. Table functions are not available in DBX. Furthermore, unbounded data transformations cannot be expressed as recursive queries in OEX, since the class of recursive queries supported by OEX is not powerful enough to represent an unbounded data transformation. Pivoting operations are not considered, since they are not supported by any of the RDBMS systems considered.

The performance of data transformations is expressed in terms of *throughput*, i.e., the number of source records transformed per second. Throughput is computed by dividing the number of tuples of the input relation by the *response time* needed to transform the entire input relation. The response time is measured as the time interval that mediates the submission of the data transformation implementation from the command line prompt and its conclusion. All time

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

measurements were obtained using the Unix time command. The interval that mediates the submission of the request and the execution by the system, known as *reaction time*, is considered neglectable.

6.2.1 Setup

The experiments were conducted on a computer with an Intel Pentium IV CPU at 3.4 GHz, 1GB of RAM, and a Samsung SP1614C hard disk with 160GB and 16MB of cache. The operating system installed is Linux (kernel version 2.4.2). A number of configuration parameters of the different systems were carefully aligned to ensure the fairness of the experiments. The main aspects of this configuration are discussed below.

I/O conditions. An important aspect regarding I/O is that all experiments use the same region of the hard-disk. To induce the use of the same area of the disk, I/O was forced through raw devices. The hard-disk is partitioned in cylinder boundaries as illustrated in Figure 6.1. The first partition is a primary partition formatted with *Ext3* file system and journaling enabled and is used for the operating system and RDBMS installations as well as for the database control files. The second partition is used as swap space. The remaining partitions are the logical partitions accessed as raw devices. These partitions handle data and log files. Each RDBMS accesses tablespaces created in distinct raw devices. The first logical partition (`/dev/hda5`) handles the tablespace named `RAWSRC` for input data; the second logical partition (`/dev/hda6`) handles the tablespace named `RAWTGT` for output data. The partition (`/dev/hda7`) is used for raw logging. Finally, (`/dev/hda8`) is used as the temporary tablespace. The implementation of the mapper accesses only `RAWSRC` and `RAWTGT` raw devices. To minimize the I/O overhead, both input and output tables were created with `PCTFREE` set to 0.

Buffers. To improve performance, RDBMSs cache frequently accessed pages in independent memory areas. One such area is the *buffer pool*, which caches disk pages (Effelsberg & Haerder, 1984). The configuration of buffer pools in DBX differs from that of the OEX system. For the purpose of the

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

Figure 6.1: Hard-disk partitioning for the experiment.

experiments, the main difference lies in the fact that, in DBX, individual buffer pools can be assigned to each tablespace, while OEX uses one global buffer pool for all tablespaces. Except for the experiments that vary the size of the cache buffer, DBX assigns a buffer pool of 4MB to the `RAWSRC` tablespace, which contains the source data; the cache size of OEX is set to 4MB. The implementation of the mapper operator is not influenced by the buffer size.

Logging. Both DBX and OEX use *write-ahead* logging mechanisms that produce undo and redo log (Gray *et al.*, 1981; Mohan & Levine, 1992). Mappers do not generate a log, since the implementation of the operator, for the time being, does not deal with concurrency or recovery issues[1]. Logging activity is disabled on both DBX and OEX. However, logging cannot be disabled in the case of stored procedures, because **insert into** statements executed within stored procedures always append data to the log.

6.2.2 Workload characterization

The tests were executed on synthetic versions of the input relations used in Examples 1.1.1 and 1.1.2, respectively for bounded and unbounded data transformations.

Since the representation of data types is not the same across all RDBMS, the record length was equalized. A dummy column was added to the input table table `LOANS`. The size of this column was chosen so that each record matches the record size of the table `LOANEVT`. The record length of both `LOANS` and `LOANEVT` was monitored to be approximately 29 bytes for every experiment.

[1]It is assumed that while performing data transformations, the target table can be entirely reconstructed from the source table in case of a crash.

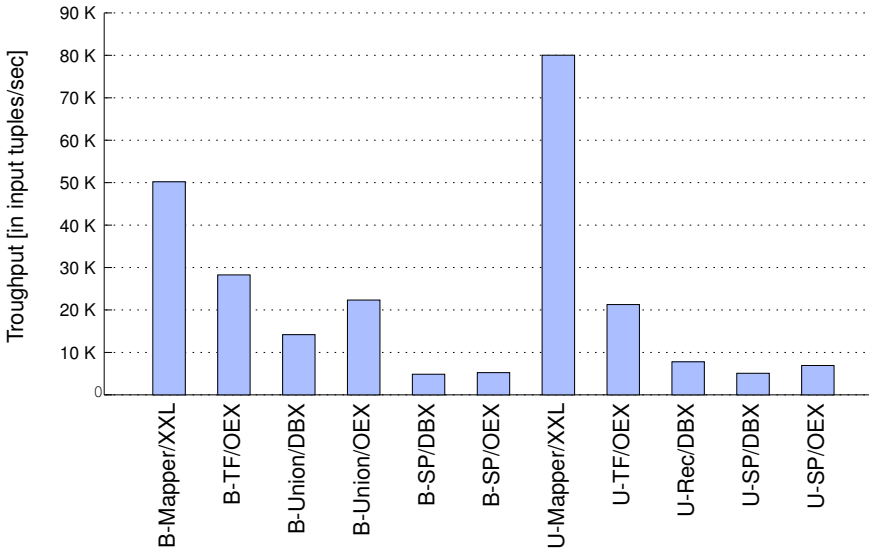
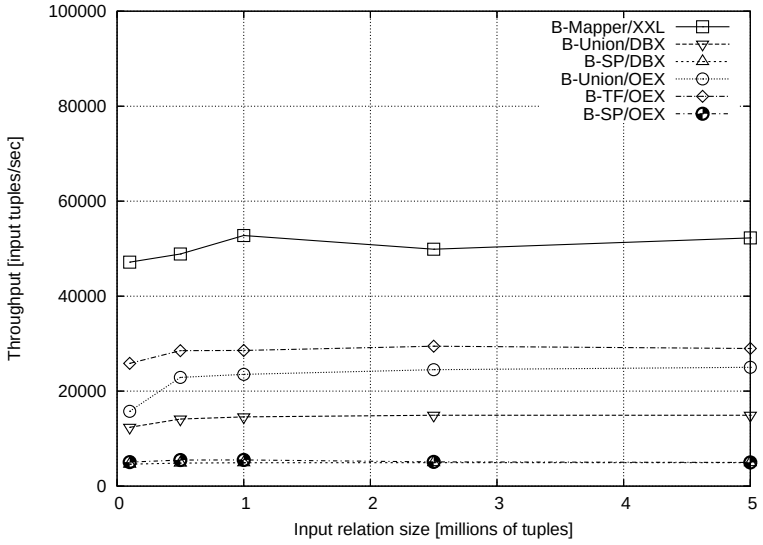


Figure 6.2: Throughput of data transformation implementations for the studied mechanisms. The results reflect the average of several runs of each implementation over input relations with different sizes. Fanout is fixed to 2.0, selectivity fixed to 0.5, and cache size set to 4MB.

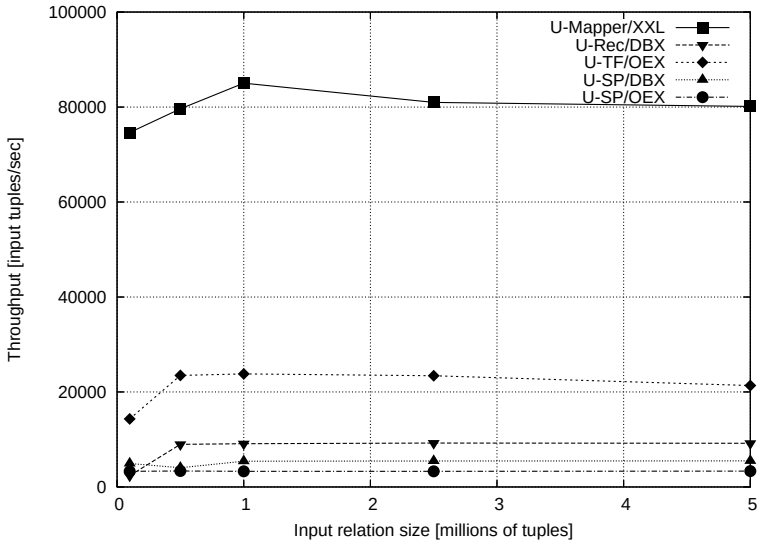
6.2.3 Throughput comparison

To compare the throughput of the evaluated alternatives, their implementations are executed over input relations with increasing sizes. The average throughput results are shown in Figure 6.2. The throughput of the mapper implementation is in average better than any alternative RDBMS implementation. In addition, table functions are more efficient than unions and recursive queries. Finally, stored procedures are the least performing alternative. As shown in Figure 6.3, the throughput is mostly constant with the increase of the input relation size. The average results presented in Figure 6.2 have a small standard deviation.

To gain further insight on the results presented above, the I/O activity of each solution was analyzed considering the amounts of read operations, write and logging activity. Figure 6.4 depicts the distribution of I/O activity in terms of the input relation size for the alternatives considered. An interesting observation is that bounded data transformations implemented as unions read an amount of



(a) Bounded transformations



(b) Unbounded transformations

Figure 6.3: Throughput as a function of relation sizes for bounded (a) and unbounded (b) data transformations. Fanout is fixed to 2.0, selectivity fixed to 0.5, and cache size set to 4MB.

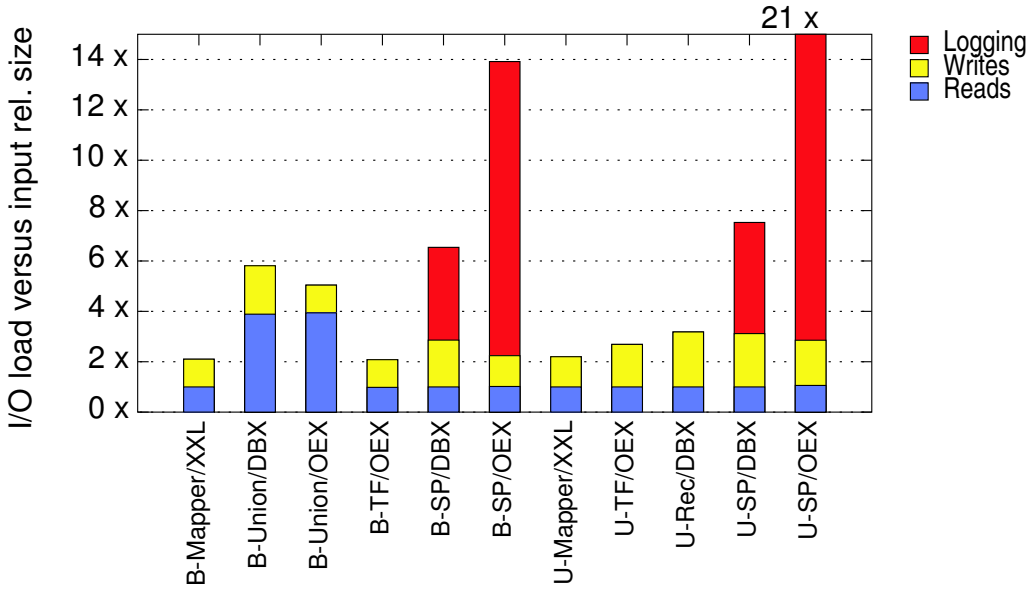


Figure 6.4: Distribution of I/O load for bounded and unbounded data transformation implementations as a function of input relation size. Fanout is fixed to 2.0, selectivity fixed to 0.5, and cache size set to 4MB. I/O is reported as the number of transferred bytes normalized by the input relation size.

data that corresponds to 4 times the size of the input relation. The analysis of the query plans of union queries, shows clearly that union queries scan the input relation multiple times. In contrast, the remaining implementations only scan the input relation once. The differences in the write activity are mainly due to the record sizes of the output relations being bigger than the record size of the input relation. Furthermore, recursive queries perform multiple joins with intermediate relations. This may imply writes to temporary tables.

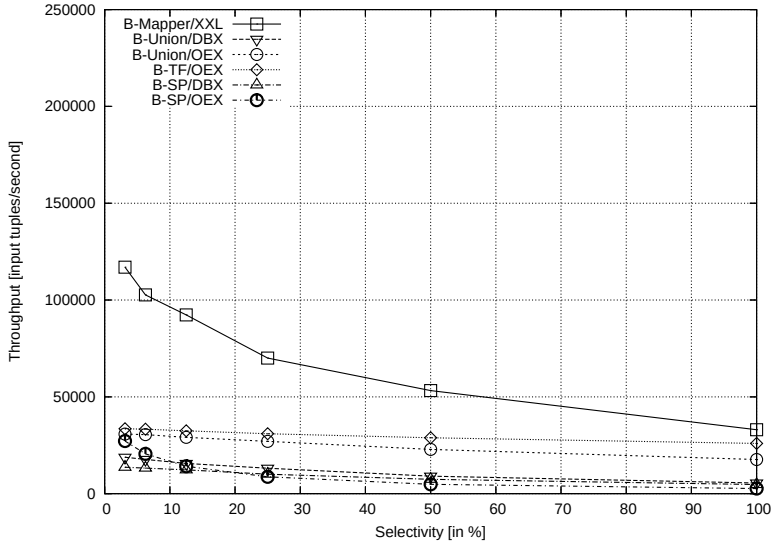
The low throughput observed in stored procedures by comparison with the other solutions is due to the huge amount of logging activity incurred during their execution. Logging cannot be disabled for stored procedures. In the experiments, the logging overhead monitored for stored procedures experiments is $\approx$ 118.9 blocks per second in the case of DBX and $\approx$ 189.2 blocks per second in the case of OEX. Taking into account the measured log overhead, stored procedures with logging disabled would execute with a performance comparable to that of table functions.

6.2.4 Influence of selectivity and fanout factors

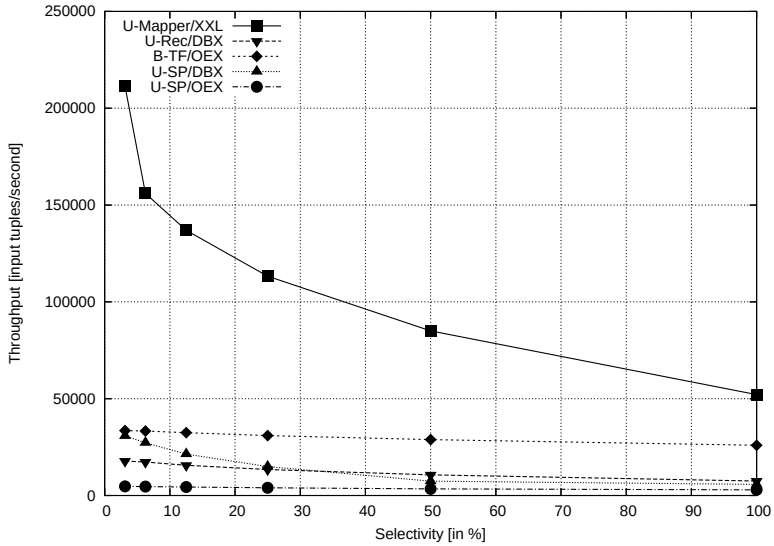
The I/O activity depends directly on two important factors: the selectivity and the fanout of data transformations. To help understand the impact of these factors on the performance of data transformations, a set of experiments varying selectivity and fanout factors was put into place.

Concerning selectivity, Figure 6.5 shows that higher throughputs are obtained for smaller selectivities. This stems from having less output tuples created when the selectivity is smaller. Union-based implementations perform several table scans independently of the selectivity of the data transformation. Thus, smaller selectivities produce minor performance improvements by comparison with the mapper operator. In contrast, stored procedures (for DBX) show an interesting performance for small selectivities, because less tuples are generated and therefore less log records are written. The throughput of the mapper operator also decreases with the selectivity, since higher selectivities imply more output tuples, which increase the cost of materializing the result and computing the Cartesian product.

To observe the impact of the fanout factor, the throughput of one-to-many data transformations was analyzed increasing the fanout factor from 1 to 32. Figure 6.6 illustrates the evolution of the throughput with increasing fanout factors. A degradation is observed when the fanout is increased. This situation is explained by the generation of more output tuples for higher fanouts. In the case of RDBMSs implementations, increasing the fanout factor also implies extra costs besides materializing more output tuples. As explained in Section 2.2, in the case of bounded transformations implemented as SQL queries, the query length increases with the fanout. Hence, longer queries are required for expressing data transformations with greater fanouts, which translates into performing more table scans. In the case of recursive queries, more I/O is incurred because higher fanouts increase the size of the intermediate relation. Finally, for stored procedures, the more tuples are written, the more log data is generated. The mapper operator performs better than other implementations but its throughput also decreases with the increase of the fanout factor mainly due to two factors: *(i)* the cost of computing the Cartesian product for producing the output tuples and *(ii)* the cost of materializing the output tuples. Since neither the implementation of

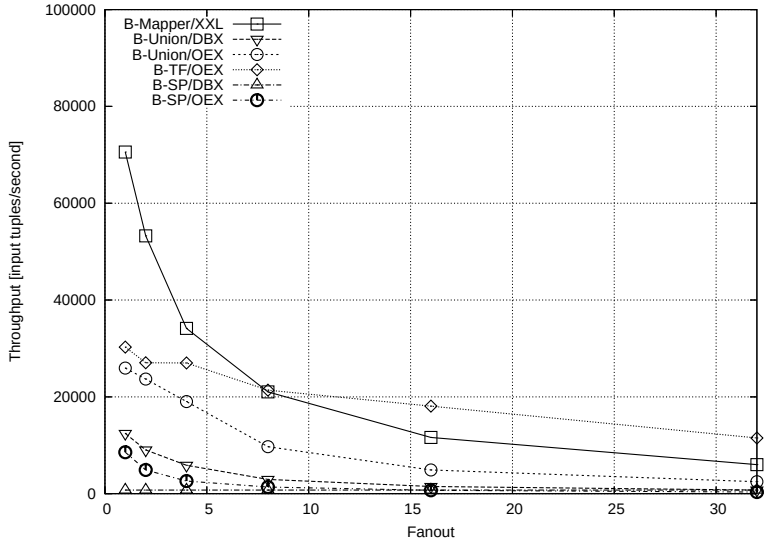


(a) Bounded transformations

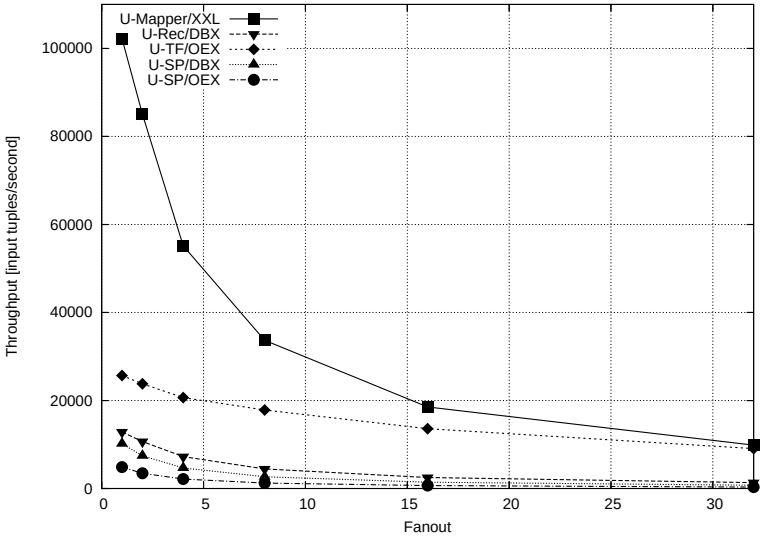


(b) Unbounded transformations

Figure 6.5: Throughput of bounded (a) and unbounded (b) data transformations with varying selectivities. Experiments conducted with an input relation of 1M tuples with fanout set to 2.0 and a cache with 4MB.



(a) Bounded transformations



(b) Unbounded transformations

Figure 6.6: Throughput of bounded (a) and unbounded (b) data transformations implementations with varying fanout factors. Experiments conducted with an input relation with 1M tuples with selectivity set to 0.5 and 4MB of cache.

the Cartesian product used by the mapper not its I/O operations were optimized, the performance of the mapper drops quickly. In the case of bounded transformations, the performance of the mapper operator eventually becomes worse than the performance of table functions (see Figure 6.6a).

6.2.5 Query optimization and execution issues

The analysis of the query plans shows that the systems used in this evaluation are not always capable of optimizing queries involving one-to-many data transformations. To validate this hypothesis, the execution of a simple selection applied to a one-to-many data transformation, represented as $\sigma_{\texttt{ACCTNO}>p}(T(s))$, is contrasted with its corresponding optimized equivalent $T(\sigma_{\texttt{ACCT}>p}(s))$. T represents the data transformation specified in Example 1.1.2, except that the column `LOANS` is directly mapped, and p is a constant used only to induce a specific selectivity. The optimized versions of the several implementations are obtained manually, by pushing down the selection condition.

Figure 6.7 presents the response times of the original and optimized versions implemented as table functions, recursive queries, and mappers. Clearly, the RDBMS optimized versions are considerably more efficient than their corresponding non-optimized versions. The improvement observed in the case of the mapper operator is small because most of the time necessary to complete the transformation is spent on I/O. Section 4.6 explained how selections applied to mappers can be advantageously optimized.

The insufficiencies of RDBMSs to optimize one-to-many data transformations are presumably a consequence of the intrinsic difficulties of optimizing queries using recursive functions and table functions, as explained in Section 2.3.2. In turn, table functions are implemented using procedural constructs that hamper optimizability. Once the table function makes use of procedural constructs, it is not possible to perform the kind of optimizations that relational queries undergo.

Manual optimization is not necessary in the case of a union, since applying a filter to a union is readily optimized. The examination of the query plans showed that one-to-many data transformations implemented through a union statement

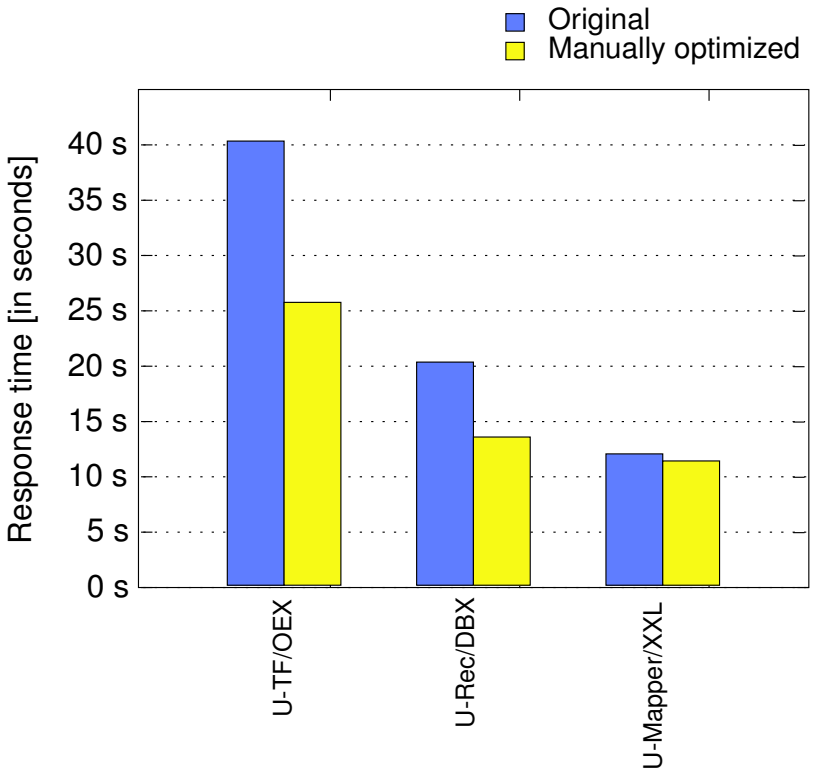


Figure 6.7: Sensivity of data transformation implementations to optimization. 1M tuples. Fanout is fixed to 2.0, selectivity of the predicate `ACCTNO` $> p$ is fixed to 0.5, the input relation has 1M tuples and cache size is set to 4MB.

take advantage of RDBMS built-in logical optimizations. The response times of the RDBMS-optimized version of a selection applied to a union is ≈ 35 seconds.

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

To evaluate the impact of the buffer pool cache size on one-to-many trans-

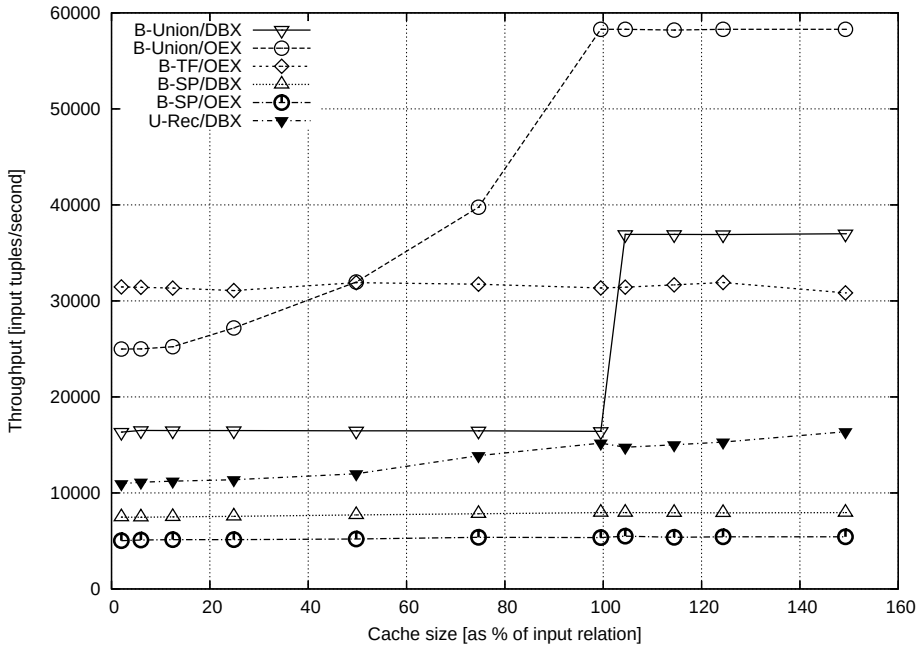


Figure 6.8: Sensivity of data transformation implementations to cache size (buffer pool) variations. The input relation has 1M tuples, selectivity is fixed to 0.5 and fanout set to 2.0.

formations, a set of experiments varying the buffer pool size were executed. The results, depicted in Figure 6.8, show that a larger buffer pool cache is most beneficial for bounded data transformations implemented as unions. This is explained because larger buffer pool caches reduce the number of physical reads that are required when scanning the input relations multiple times. A distinct behavior of the RDBMSs used in the evaluation as the cache size increases is to be remarked: the throughput in OEX increases smoothly, while in DBX there is a sharp increase. This has to do with the differences in the cache replacement strategies used by these systems while performing table scans (Effelsberg & Haerder, 1984). To select the next page to be replaced, DBX uses a variant of the *least recently used* (LRU) strategy (O'Neil *et al.*, 1993). In contrast, according to its documentation, the OEX system, uses a *most recently used* (MRU) replacement strategy (Denning, 1968). The LRU replacement strategy is affected by the problem of *sequential flooding* (Jiang & Zhuang, 2002; O'Neil *et al.*, 1993; Rizzo & Vicisano, 2000). When the size of the cache buffer is smaller than the size of the input re-

lation, full table scans purge all entries out of the cache. As a result, queries that scan the input relation multiple times perform quite poorly (Smaragdakis *et al.*, 1999). On the contrary, when input tables are small enough to fit in the cache buffer, using multiple unions is the most advantageous alternative for bounded data transformations. However, in the presence of large input relations, table functions are the best alternative since they are insensitive to cache size. This is due to the fact that the input relation is being scanned only once. Stored procedure implementations also scan the input relation only once, but are less performant due to logging. The same argument applies to the mapper operator. Finally, it is worth noting that the best relational implementation is still worse than the mapper implementation (80K tuples/sec for this configuration), even with a cache size equal to the input relation.

6.3 Algebraic Optimization

To validate the logical optimizations for the mapper operator developed in Chapter 4, together with the cost formulas proposed for expressions involving selections, a number of experiments were conducted. The experiments compared expressions combining selections with mappers to their optimized equivalents based on a physical implementation of the mapper operator using the Naïve algorithm (see Section 5.2). The experiments address the influence of predicate selectivity, the mapper function fanout and the mapper function cost on the optimizations, as proposed in Rule 4.3 and in Rule 4.4.

6.3.1 Setup

To ensure the same conditions for both rules, the setup was as follows. The expression $\sigma_{p_i}(\mu_{f_1,f_2,f_3,f_4}(r))$ was compared with the optimized variants $\mu_{f_1,\sigma_{p_i}\circ f_2,f_3,f_4}(r)$ for Rule 4.3, and $\mu_{f_1,f_2,f_3,f_4}(\sigma_{p_i[f_2]}(r))$ for Rule 4.4. The mapper function f_1, unless otherwise stated, has a fanout of 2.0, f_2 always has a fanout of 1.0 and the remaining functions, f_3 and f_4, have a fanout of 2.0. The input relation r is an input relation with synthetic data. The predicate p_i corresponds to a condition with a predefined selectivity. Furthermore, the predicate $p_i[f_2]$ represents

a new predicate that results from expanding the function f_2 in the condition corresponding to p_i, as presented in Section 4.3.2.

For the sake of accuracy, the predicates applied were tuned to guarantee predefined selectivity values. Likewise, when varying the fanout or the cost factors of a function, functions specifically tuned to guarantee predefined fanout factors and per-tuple costs were used. Each experiment measured the response time corresponding to the sum of the time taken to read the input tuples, plus the time taken to compute the output tuples, plus the time taken to write them.

To ascertain that the differences in performance were caused by improvements brought by one optimized expression over the original, the amount of I/O performed on both expressions was verified to be the same and, furthermore, that it was performed on the same regions of the disk. To that end, raw devices were used instead of regular files.

6.3.2 Real-world example

This experiment simulated a real-world scenario that consists of populating the relation SMALLPAYMENTS[ACCTNO, AMOUNT, SEQNO] formed by all payments whose amount is smaller than 50. This relation can be obtained from the relation PAYMENTS presented in Example 1.1.2. According to Example 3.2.1, since the expression $\mu_{acct,amt}(\texttt{LOANS})$ corresponds to the relation PAYMENTS, the expression $\sigma_{\texttt{AMOUNT}<50}(\mu_{acct,amt}(\texttt{LOANS}))$ denotes the relation SMALLPAYMENTS.

The original expression $\sigma_{\texttt{AMOUNT}<50}(\mu_{acct,amt}(\texttt{LOANS}))$ and its equivalent optimized expression $\mu_{acct,\sigma_{\texttt{AMOUNT}<50}\circ amt}(\texttt{LOANS})$, obtained via Rule 4.3 were evaluated over input relations with sizes varying from 1K to 10M tuples. The results, presented in Figure 6.9, show a remarkable improvement on the response time of the original expression over the optimized expression. On this first experiment, it can be observed that the optimized expression is evaluated more than 5 times faster than the original expression. The average selectivity of the predicate $\texttt{AMOUNT} < 50$ was 0.0049, and the observed fanout factor for the *amt* function was 101.6.

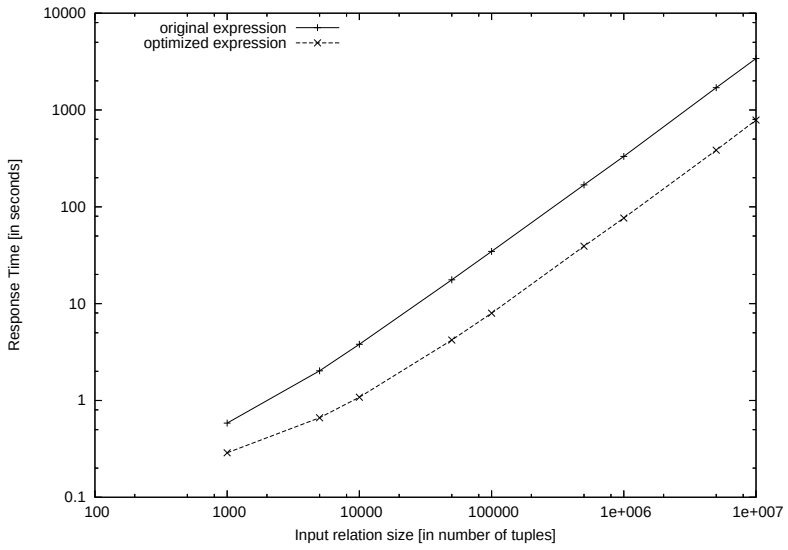


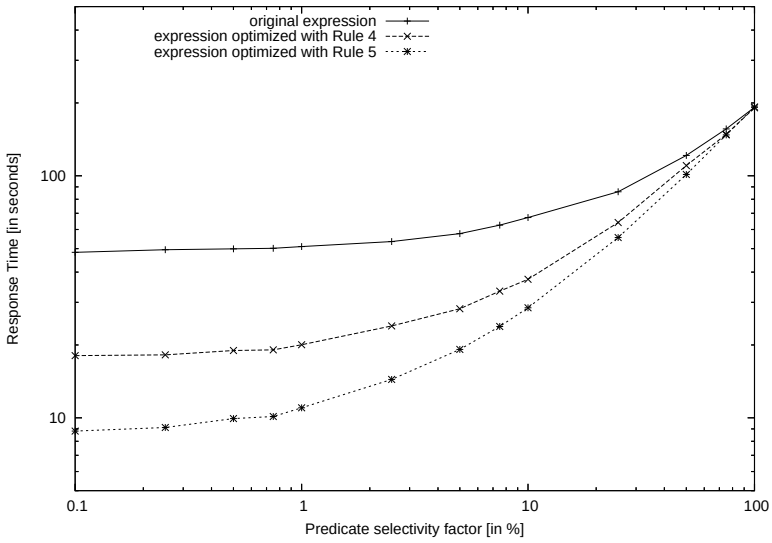
Figure 6.9: Response time for producing the `SMALLPAYMENTS` relation as a function of the number of tuples.

6.3.3 Influence of the predicate selectivity factor

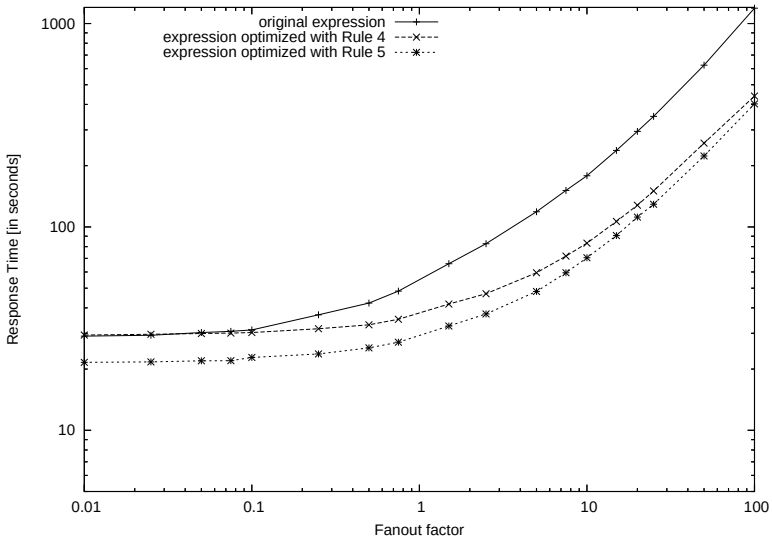
Seeking to validate the effect of the predicate selectivity, a set of experiments was carried out using a different p_i predicate with selectivities ranging from 0.1% to 100%. The tests were executed over an input relation with 1 million input tuples. Figure 6.10a shows the evolution of the response time for different selectivities, using cheap functions with their default fanouts.

As expected, for both rules, the highest gains brought by the optimization were obtained for small selectivities. For Rule 4.3, more concretely, for a selectivity of 0.1%, the optimized expression was 2.7 times faster than the original one. For Rule 4.4, with the same selectivity, the optimized expression was 5.5 times faster.

With respect to Rule 4.3, as the selectivity decreases, more results are filtered out from function f_2 by the predicate p_i and, therefore, the cost of computing the Cartesian product involved in the mapper is smaller. As the selectivity approaches 100%, the gain drops since the number of tuples filtered out from f_2 tend to zero. These results validate the gain formula (4.8). This rule also reduces the number of times the condition is evaluated. Even for a selectivity of 100%, the non-optimized expression evaluates the condition more often that the optimized



(a) Influence of selectivity



(b) Influence of fanout

Figure 6.10: Response time for the original and optimized expressions as a function of selectivity and fanout. The figure shows the effect of applying predicates p_i with increasing selectivities (a), and the effect of increasing the fanout of the mapper function f_1, maintaining the predicate selectivity fixed to 2.5% (b).

expression. However, since in these experiments the predicate evaluation is very cheap, the small gain obtained is not visible in the figure.

With respect to Rule 4.4, the mapper is evaluated over fewer tuples, as a direct effect of pushing the condition through the mapper. As a result, many Cartesian product computations and function evaluations are saved. As the selectivity of the condition approaches 100%, the number of tuples fed into the mapper grows. Therefore, the cost of the non-optimized expression is approximately the same as the cost of the optimized expression.

6.3.4 Influence of the function fanout factor

In order to experimentally check how the function fanout affects the proposed optimizations, the evolution of response time for the original and optimized expressions when the fanout factor varies was observed. Function f_1 was replaced by a function that guarantees a predefined fanout factor ranging from 0.01 (unusually small) to 100. To isolate the effect of the fanout, the selectivity of the predicate was kept constant at 2.5%. The results are depicted in Figure 6.10b.

For small values of the fanout, Rule 4.3 presents a slight degradation of $\approx 1\%$ in performance with respect to the performance of the original expression, while Rule 4.4, displays an improvement of $\approx 35\%$. The modest improvement brought by Rule 4.4 is explained by the fact that, for small values of the fanout, the Cartesian product is rarely performed, so no gain is introduced. Additionally, in the case of Rule 4.3, for small values of the mapper fanout, the expression $O_F - O_{g_{A_j}}$ is negative. As a consequence, by formula (4.8), the gain is also negative.

As explained in Section 4.6.1, the cost of the Cartesian product increases with the fanout, since the higher the fanout, the more tuples have to be produced by the Cartesian product for each input tuple. For high values of fanout, the cost of performing the Cartesian product becomes the dominant factor. Thus, the gain obtained by both rules increases with the fanout since both optimizations reduce the cost of the Cartesian product. For a fanout of 100, it can be observed that Rule 4.3 was 2.7 times faster than the original and Rule 4.4 was 2.95 times faster (see Figure 6.10a and Figure 6.10b).

In this experiment, Rule 4.4 is consistently cheaper than Rule 4.3. Since the selectivity for this experiment is 2.5%, according to (4.13), Rule 4.4 is cheaper than Rule 4.3 whenever $C_{f_2} < 97.5\% \cdot (C_F + m \cdot k_0)$. Trivially, this inequality holds because the cost of all functions in F is the same.

6.3.5 Influence of the function evaluation cost

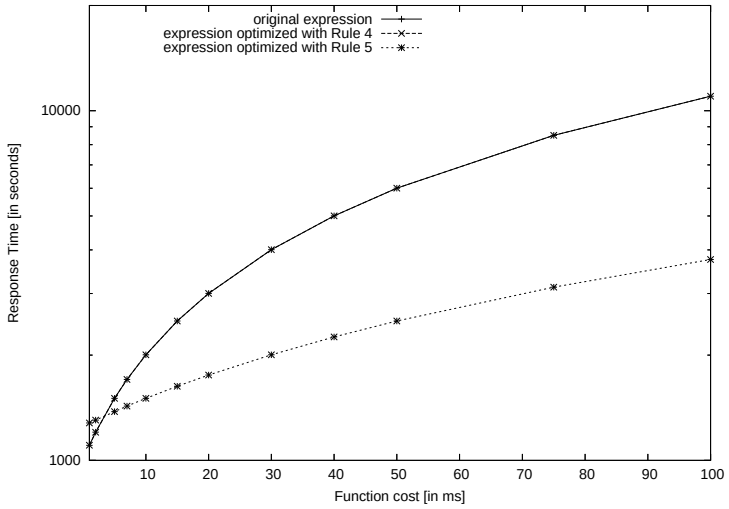
To validate how the function cost influences the optimization gains, two sets of experiments were put in place. The first experiments increased the cost of an expensive function, while the second experiments varied the selectivity of the condition in the presence of expensive functions. The function f_3 was selected to be the expensive mapper function. In the first set of experiments, shown in Figure 6.11a, the cost of f_3 varied from 1ms per call to 100ms per call. In the second set of experiments, shown in Figure 6.11b, the cost of f_3 was fixed to 25ms. In both sets of experiments the function being optimized, which is f_2, had a fixed cost of 10ms per call.

captionResponse time of the mapper expression in the presence of expensive functions for the original and optimized expressions.

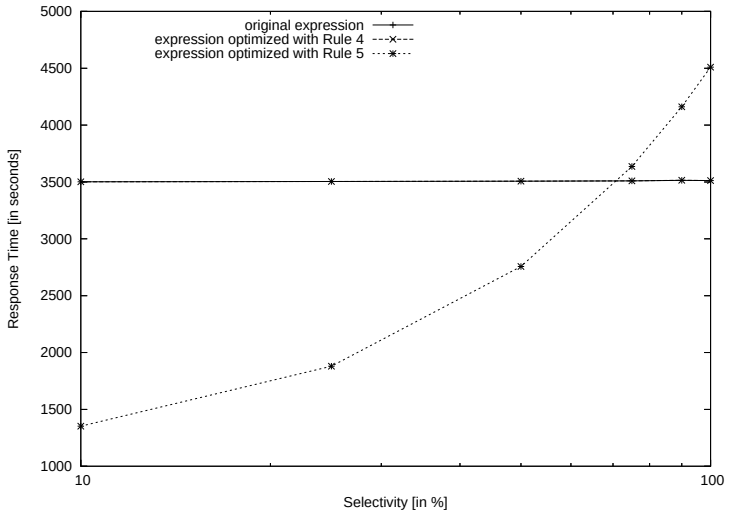
In Section 4.6.3, it has been remarked that the gain for Rule 4.3 is independent of the mapper function cost. Although there is a gain resulting from savings in the Cartesian product computation, as show by formula (4.8), this gain is very small in comparison with the mapper execution cost in the presence of expensive functions. The outcome of the experiments is aligned with the cost estimates. Notice that in Figure 6.11a and Figure 6.11b, the line plots of the optimized expressions for Rule 4.3 overlap the line for the original expression.

With respect to Rule 4.4, it can be observed that both the cost of the mapper functions and the predicate selectivity directly influence the gain. These observations validate the gain formula (4.11), in that small selectivities and a high function cost result in high gains.

In Figure 6.11a, the cost of the optimized expression for Rule 4.4 is initially higher than the cost of the original expression. This happens because for lower function costs, the mapper function f_2, which is the only function pushed into the selection condition, is more expensive than the function f_3. This means that,



(a) Influence of the cost of f_3



(b) Influence of the selectivity of p_i

Figure 6.11: Evolution of response time for the original and optimized expressions in the presence of expensive functions. The effect of increasing the cost of f_3 with a constant selectivity factor of 2.5% for p_i (a). The effect of increasing the selectivity factor of p_i, maintaining the cost of f_3 constant at 25ms per call (b). The experiments process 100K tuples with the cost of f_2 set to 10ms per call.

in the gain formula (4.11), $n \cdot C_H$ is higher than the other factors of the formula, which results in a negative gain. As f_3 gets more expensive, the value of C_F grows. This causes $n \cdot (1 - \alpha) \cdot (C_{prd} + C_F)$ to increase, eventually leading to a positive gain.

Figure 6.11b, shows that the cost of the optimized expression for Rule 4.4 eventually becomes more expensive than the cost of the original expression. In fact, as the selectivity factor α increases, $n \cdot (1 - \alpha) \cdot (C_{prd} + C_F)$ decreases, and since C_H is high, the gain eventually becomes negative.

These two experiments highlight the limitation of Rule 4.3. This rule does not optimize the cost of evaluating the functions. Thus, when the cost of evaluating the mapper functions increases, both the original and the optimized expressions increase by the same amount. By contrast, Rule 4.4 reports important gains.

Nevertheless, Rule 4.3 is quite successful if the cost of applying the predicate is high. In the optimized version for Rule 4.3, the predicate is applied for each output value of the mapper function. In the non-optimized version, the predicate is applied for each tuple of the result Cartesian product. The number of tuples produced by the Cartesian product, for each input tuple, is given by multiplying the fanout factors of *all* mapper functions. In the presence of expensive predicates, for functions with high fanout, high gains can be achieved.

6.4 Mapper Execution Algorithms

This section compares the performance of the physical execution algorithms proposed for the mapper operator. The algorithms considered are the Naïve, Shortcircuiting and Cache-based algorithms proposed in Chapter 5. The performance of each algorithm is obtained by measuring the response time required for performing one-to-many data transformations.

This section reports two groups of experiments. The first group aims at validating the performance benefits of the Shortcircuiting and Cache-based algorithms over the Naïve algorithm in the presence of selective mapper functions and duplicate function input values, respectively. In this first group of experiments of the Shortcircuiting algorithm, the experiments compare its performance with the Naïve algorithm varying the selectivity and cost per call of the mapper functions.

Function name	Input parameters	Output parameters	Avg cost per call (in μs)	Duplicates ratio (in %)
name	AUTHOR	NAME	469	58.52
title	TITLE	TITLE	113	74.04
event	EVENTNAME	EVENT	1612	46.54
loctn	LOCATION	CITY, COUNTRY	4	99.998
year	DATE	YEAR	25	99.999

Table 6.2: Details of the mapper functions used in Example 3.2.2.

The Cache-based Algorithm is compared with the Naïve algorithm varying the number of duplicates of the input relation.

The second group of experiments studies the performance of different cache replacement strategies for the Cache-based algorithm. These experiments aim at validating the performance and behavior of the XLUR replacement strategy proposed in Section 5.7. In particular the experiments compare XLUR cache replacement policies with the well-known LRU and RND (which replaces a random entry) strategies, by varying the parameters that influence their performance, such as the number of duplicates and the size of the cache.

6.4.1 Setup

Example 3.2.2 is used throughout the experiments. This example was implemented through the mapper $\mu_{name,title,event,loctn,year}$ that encodes a data cleaning transformation that takes as input the relation CITEDATA and produces the relation EVENTS. The transformation employs three expensive functions *name*, *title* and *event* and two cheap functions *loctn* and *year* (the details concerning cost and duplicate ratios for each function are given in Table 6.2).

Most experiments use a real-world version of the CITEDATA input relation. The values of the CITEDATA input relation follow a Zipfian distribution in the columns AUTHOR, TITLE and EVENTNAME (see Appendix C). The variation in the number of duplicates is obtained by using specially prepared versions of the CITEDATA input relations with different skewness parameters, obtained by selecting records from the original CITEDATA relation until the desired skewness is observed.

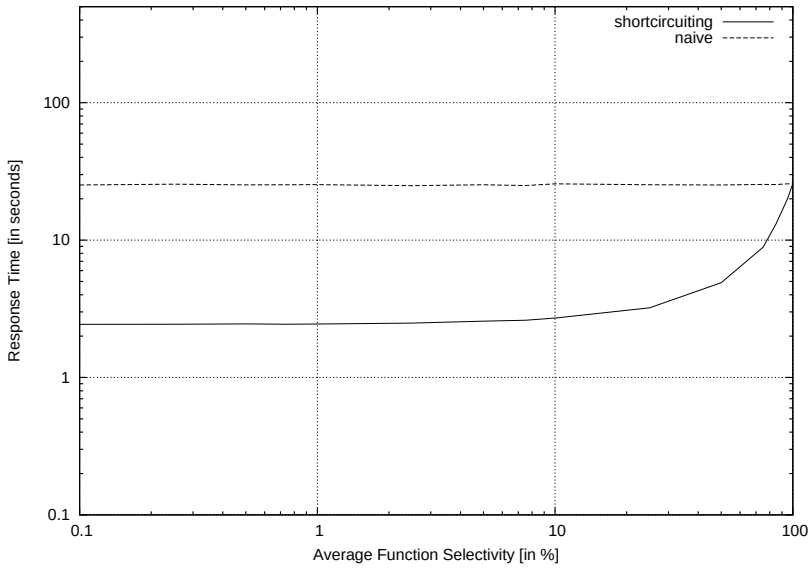


Figure 6.12: Comparison of the response time required by the Shortcircuiting and Naïve algorithms for transforming 10K tuples with increasing total mapper function selectivity varying from 0.1% to 95%. The results displayed correspond to averages of several runs, where the costs of the mapper function are random and uniformly distributed totaling 250ms. The fanout is set to 1.0.

In order to analyze the effect of the variation of the function cost and selectivity parameters, the experiments employ modified versions of the mapper functions. These mapper functions are specifically tunned to have predefined evaluation costs and selectivities.

6.4.2 Performance of the Shortcircuiting algorithm

The behavior of the Shortcircuiting algorithm was compared with the Naïve algorithm through a set of experiments that vary the average selectivity and the cost of the mapper functions.

Figure 6.12 and Figure 6.13 show the time required for transforming an input relation, varying the selectivity and cost parameters, respectively. When the selectivity is 100%, both algorithms have the same behavior. In this situation, the shortcircuiting optimization does not bring any benefit. The superiority of the Shortcircuiting algorithm becomes clear as the selectivity decreases, since

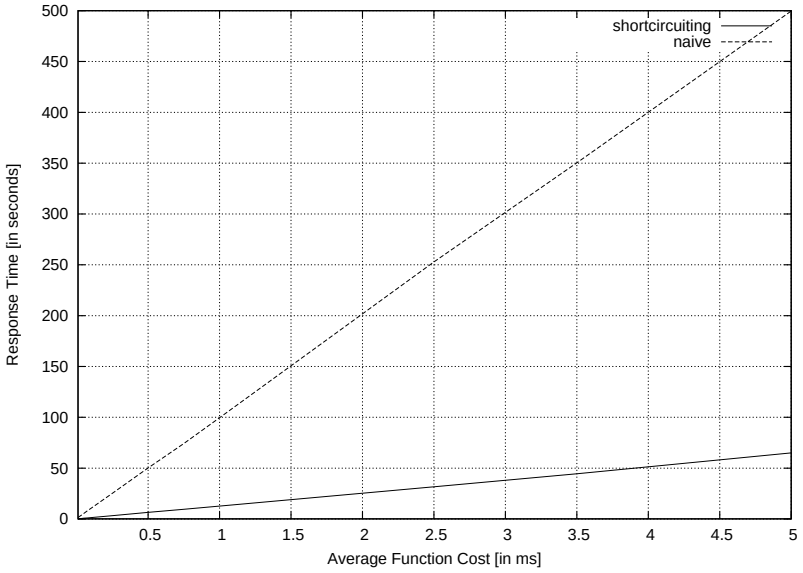


Figure 6.13: Comparison of the response time required by the Shortcircuiting and Naïve algorithms for transforming 10K tuples with increasing average function cost varying from 0.01ms to 5.0ms in total. Average selectivity is 25% and the fanout is 1.0.

the difference in the response time required to perform the data transformation using the Shortcircuiting algorithm decreases. Recall that, when executing the Shortcircuiting algorithm, when the result of executing a mapper function is the empty set, the remaining mapper functions are not evaluated. In contrast, the Naïve algorithm requires always the same amount of work to evaluate the mapper operator for a given input tuple. In fact, according to the cost model introduced in Section 4.6, the Naïve algorithm is insensitive to the variation of selectivity. Concerning the variation in terms of the mapper function cost, and taking into account the plot of Figure 6.13, although the response time of both algorithms increases with the average function cost, the performance gap between the two algorithms increases as the mapper functions become more expensive. In this experiment, the difference in the number of mapper function calls between the Shortcircuiting and Naïve algorithms is always the same since the selectivity is kept constant. Therefore, when the cost of the mapper functions increases, the difference in response time between the two algorithms also increases.

	Duplicate ratios			Cache hit ratios		
Skewness	*name*	*title*	*event*	RND	LRU	XLUR
$z = 0.01$	36.84%	36.68%	36.72%	1.64%	1.64%	1.64%
$z = 0.25$	38.36%	38.39%	38.27%	1.84%	1.84%	1.86%
$z = 0.50$	44.28%	44.27%	44.35%	3.89%	4.24%	4.87%
$z = 0.75$	56.70%	56.84%	56.94%	17.00%	19.24%	21.75%
$z = 0.99$	75.30%	75.16%	75.21%	50.89%	54.75%	56.66%

Table 6.3: Ratios of duplicate input values for each cached mapper function and the corresponding cache hit ratios of the different cache replacement strategies. The values concern processing 100K tuples on a Zipfian distributed `CITEDATA` input relation with different values of z using a cache with 5K entries.

6.4.3 Performance of the Cache-based algorithm

To validate the potential benefits of the Cache-based algorithm over the Naïve algorithm, the effect of the number of duplicates in the input relation in the response time was evaluated. The results depicted in Figure 6.14 show the performance of the cache-based implementations using different replacement strategies over input relations with increasing quantities of duplicates (as detailed in Table 6.3). The changes in the number of duplicates are obtained by varying the skewness parameter z of the Zipfian distribution used for preparing different versions of the `CITEDATA` input relation.

The Naïve algorithm takes the same time to perform the transformation, independently of the number of duplicates. In the case of the Cache-based algorithm, the response time required for transforming an equal-sized sample of the `CITEDATA` relation decreases as the number of duplicates increases. This is explained as follows: an increase in the number of duplicates eventually results in a higher cache-hit ratio. As the cache becomes more effective, the total run time of the Cache-based algorithm decreases.

Another interesting aspect of the Cache-based algorithm is that it can be implemented with a very small overhead in terms of computation cost when compared with the Naïve algorithm. The breakdown of response time in terms of the function evaluation cost *vs.* the I/O cost plus algorithm cost shown in Figure 6.14, indicates that the overhead incurred is roughly the same in all implementations.

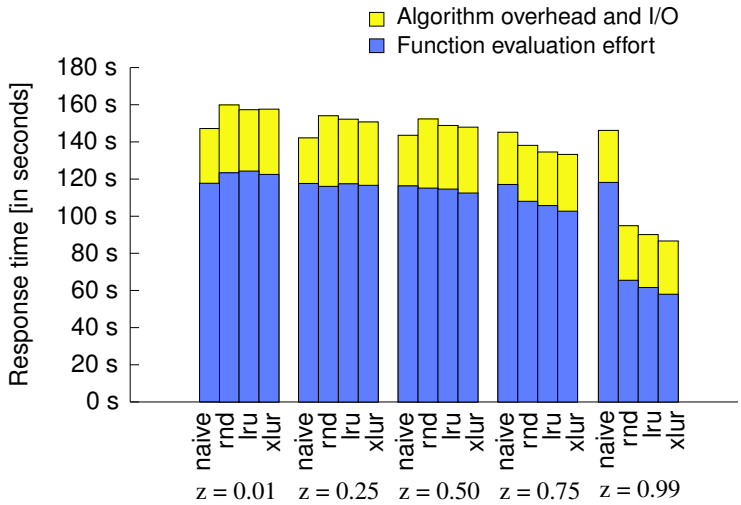


Figure 6.14: Throughput comparison of implementations of the Naïve with Cache-based mapper implementations of Example 3.2.2 with different replacement policies using a cache with 5K entries. The response times refer to transformations of 100K tuples of `CITEDATA` input relations prepared with Zipfian distribution obeying increasing skewness parameters z on the columns `AUTHOR`, `TITLE` and `EVENTNAME`. Higher values of z correspond to more duplicates.

For smaller values of z, the total cost of evaluating the mapper functions incurred by the cache-based implementations is slightly greater than the total function evaluation cost of the Naïve algorithm. This is due to the overhead of using a cache: smaller values of z mean less duplicates and hence more cache misses.

6.4.4 Performance of the cache replacement policies

The performance of the Cache-based algorithm is influenced by the parameters that affect the performance of the cache, e.g., size and access pattern characteristics, like the number of duplicates of the input and the inter-reference intervals of the cache entries. Additionally, as other mapper evaluation algorithms, the Cache-based algorithm is influenced by the cost of the mapper functions. This section focuses on the influence of these parameters on the considered cache replacement strategies: LRU, XLUR and RND. The RND replacement strategy

replaces a random entry and it is interesting because it serves as a lower bound for the performance of cache replacement strategies (Belady, 1966).

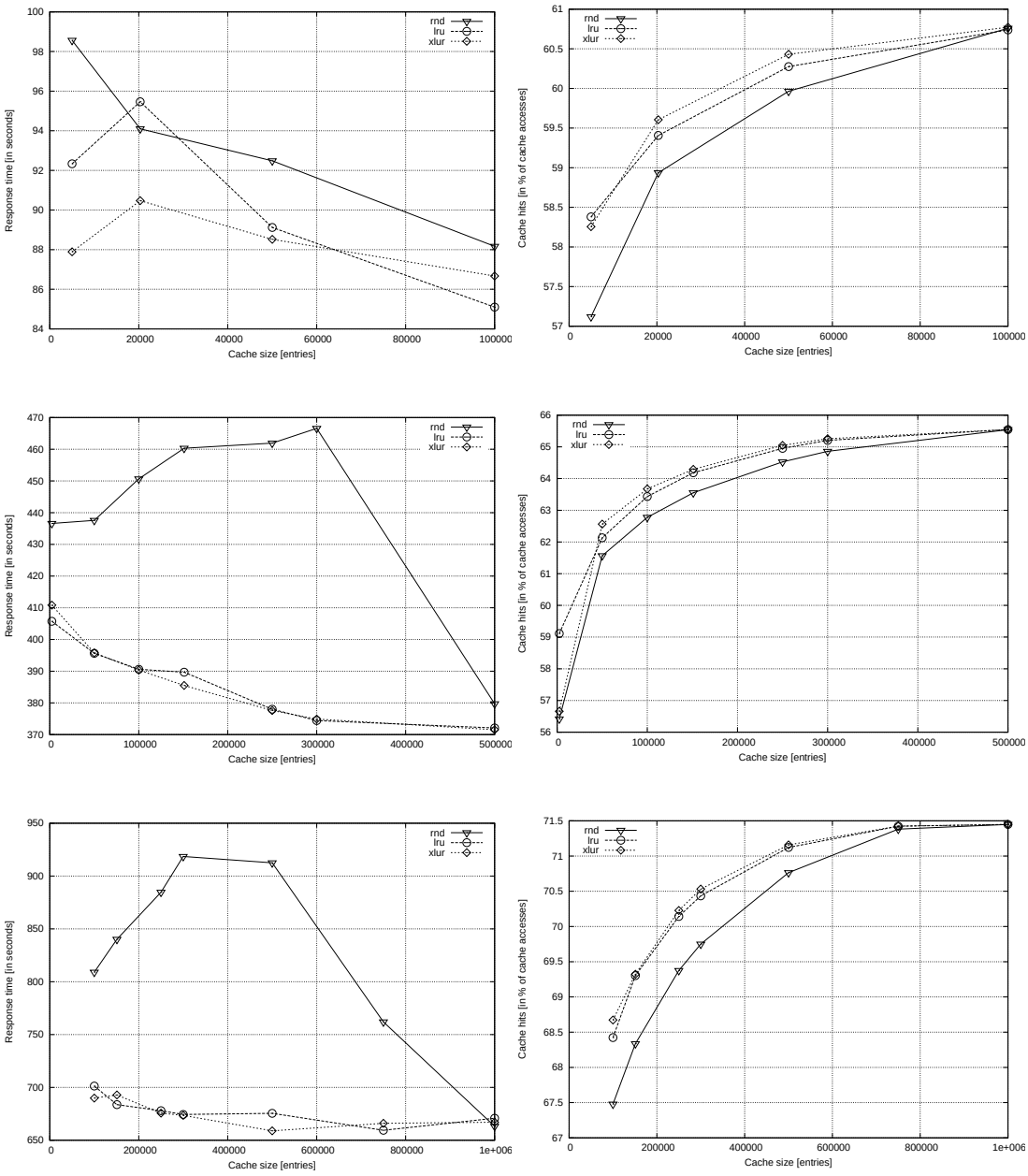
Influence of cache size. Increasing the cache size leads to higher cache-hit ratios, which results in fewer function evaluations and less total effort required for evaluating the Cache-based algorithm for any of the three replacement strategies considered.

As shown on Figure 6.15, for very small cache sizes, LRU is usually better than XLUR in terms of *cache hit ratio*. However, in terms of response time, XLUR is still more efficient than LRU. XLUR makes better replacement decisions in terms of cost when the cache is small. As the size of the cache increases, the difference between the two algorithms vanishes, because there are less cache misses and consequently less replacement decisions to be made with a larger cache.

Moreover, despite the fact that XLUR aims at optimizing the total evaluation cost, in some situations XLUR attains higher cache hit ratios than LRU, as seen in Figure 6.15. However, in general, XLUR is less effective than LRU in terms of cache hit ratio. As described in Section 5.5.2, XLUR divides the cache into multiple stacks to force entries with distinct frequencies to age at different speeds. Hence, the size of the larger XLUR stack is smaller than a single LRU stack, turning XLUR less effective than LRU in situations where cache accesses have large inter-reference intervals.

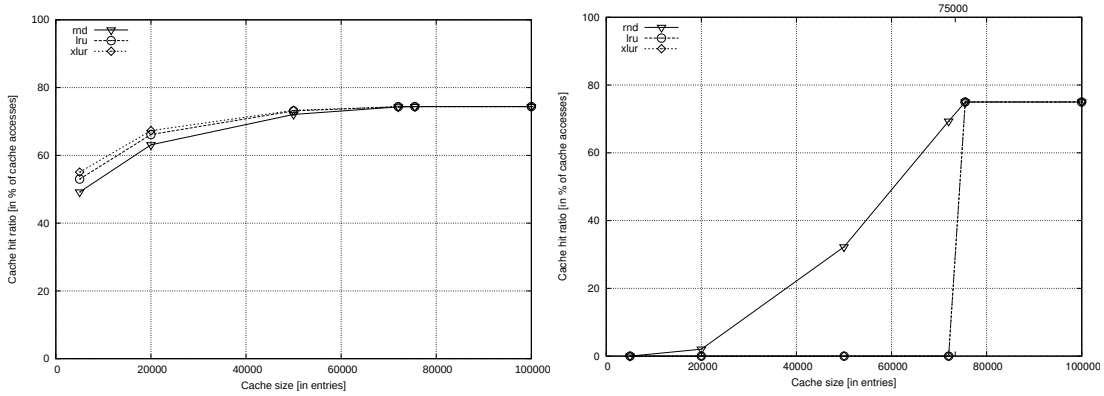
Influence of the number of duplicates. In general, a greater number of duplicates in the input leads to a higher cache hit ratio, resulting in less function evaluations. Figure 6.14 shows that the total work required to perform the data transformation decreases as the number of duplicates increases, because more duplicates usually correspond to higher cache-hit ratios.

Table 6.3 validates the hypothesis that higher ratios of duplicates correspond to higher cache hit ratios. However, by increasing the number of duplicates does not lead to a proportional increase in the cache-hit ratio. The conclusion that can be drawn is that in every replacement strategy the cache hit ratio can increase non-linearly with the number of duplicates.



(a) Evolution of Response Time (b) Evolution of Cache Hit Ratio

Figure 6.15: Evolution of response time (a) and cache hit ratios (b) as a function of cache size. From top to bottom, the graphics depict the evolution for transforming 100K, 500K and 1M records the `CITEDATA` relation, respectively.



(a) Varying inter-reference intervals (b) Uniform inter-reference intervals

Figure 6.16: Evolution of cache hit ratio for transforming 100K tuples of modified versions of the `CITEDATA` relation using increasingly large caches. The input relation was prepared with roughly the same number of duplicates ($\approx 75\%$) but with different inter-reference intervals. One relation is generated following a Zipfian distribution that produces inter-reference intervals of various sizes (a), while in the other, duplicate references to the same entry are separated by 25000 records (b).

Influence of the inter-reference interval. The inter-reference interval is an important characteristic of a given sequence of cache accesses (see Section 5.6.1), since it dictates the performance of a cache replacement strategy. LRU and XLUR, which is based on LRU, are particularly sensitive to situations where the average inter-reference interval is greater than the number of entries in the cache. For example, the modest cache hit ratios reported in Table 6.3 are explained by the the fact that a cache of 5K entries is not capable of detecting the duplicate input values whose inter-reference interval corresponds to more than 5K distinct references.

Figure 6.16 displays the evolution in terms of cache hit ratio for transforming two versions of the `CITEDATA` input relation where the cache entries have different inter-reference intervals. This experiment aims at validating the influence of inter-reference intervals. Although the number of duplicates is the same, there is a noticeable difference in the cache hit ratio. The Zipfian distributed version (left graphic of Figure 6.16) responds smoothly to increases of the cache size, while in the second version of input data, both

LRU and XLUR, whose lines overlap in the graphic, display a step increase of the cache-hit ratio from 0 to 75% as soon as the cache size crosses 75000. Since each input value is repeated twice, separated by 25000 distinct values and 3 functions are being cached, 75000 is the number of distinct references before the same entry is referenced again. When the cache is smaller than 75000, each new entry installed in the cache will force the oldest cache entry out of the cache in a cyclic way, resulting in zero cache hits. Once the cache becomes large enough to hold 75000 distinct references, all the duplicate references are detected resulting in an increase of the high cache hit ratio.

The distribution reported in Figure 6.16b explores the difficulty of LRU to deal with cyclic references to entries separated by intervals larger than the cache size. This problem also affects XLUR since its implementation is based on LRU. In contrast, RND is not affected. Distinct cache replacement strategies may react differently to the distribution of the inter-reference intervals.

6.5 Data Fusion

Data Fusion is a commercial data transformation tool developed and marketed by Oblog Consulting to address the requirements of legacy-data migration scenarios (Carreira & Galhardas, 2004a). Data Fusion incorporates the mapper operator implemented using the Naïve Algorithm.

The tool has evolved from the experience of the company in deploying several large scale legacy-data migration projects. The mapper operator was included in Data Fusion since data migrations from legacy data model into a new model require the use of the inverse of the SQL group by/aggregate primitive. Moreover, in the context of legacy-data migrations, the cost of developing such transformations is frequently very high, since there is no easy solution for expressing them.

```
 1: mapper LoanToPayments
 2:   import master LOANS
 3:   export PAYMENTS
 4:   ACCTNO = lpad(tostr(ACCT), 4, '0')
 5:   AMOUNT, SEQNO = rule
 6:     var rem_amnt: numeric
 7:     var seq_no: integer  = 0
 8:     rem_amnt = AMT
 9:     while rem_amnt > 100 do
10:       rem_amnt = rem_amnt - 100
11:       seq_no = seq_no - 100
12:       AMOUNT = rem_amnt
13:       SEQNO = seq_no
14:       insert
15:     end while
16:     AMOUNT = rem_amnt
17:     SEQNO = seq_no
18:     insert
19:   end rule
20: end mapper
```

Figure 6.17: Implementation of Example 1.1.2 as a DTL mapper in Data Fusion

6.5.1 Overview

Data Fusion offers a domain-specific language for data transformations named DTL (*Data Transformation Language*) for writing concise and short programs. It also provides an Interactive Development Environment (IDE) for efficiently producing and maintaining code.

In DTL, data transformations are organized as modules that consist of two blocks. The first block establishes a view over the source data and the second block encodes how the data of the view is mapped into one or more target relations. The first block is specified as an SQL query that contains joins and aggregations, while the second step is expressed using a modified version of the mapper operator. In the DTL implementation, a mapper operator comprises several *rules* that enclose transformations with similar logics, e.g., populate fields with the *null* value as exemplified in Figure 6.17. Rules represent mapper functions and can be re-used and arranged into libraries.

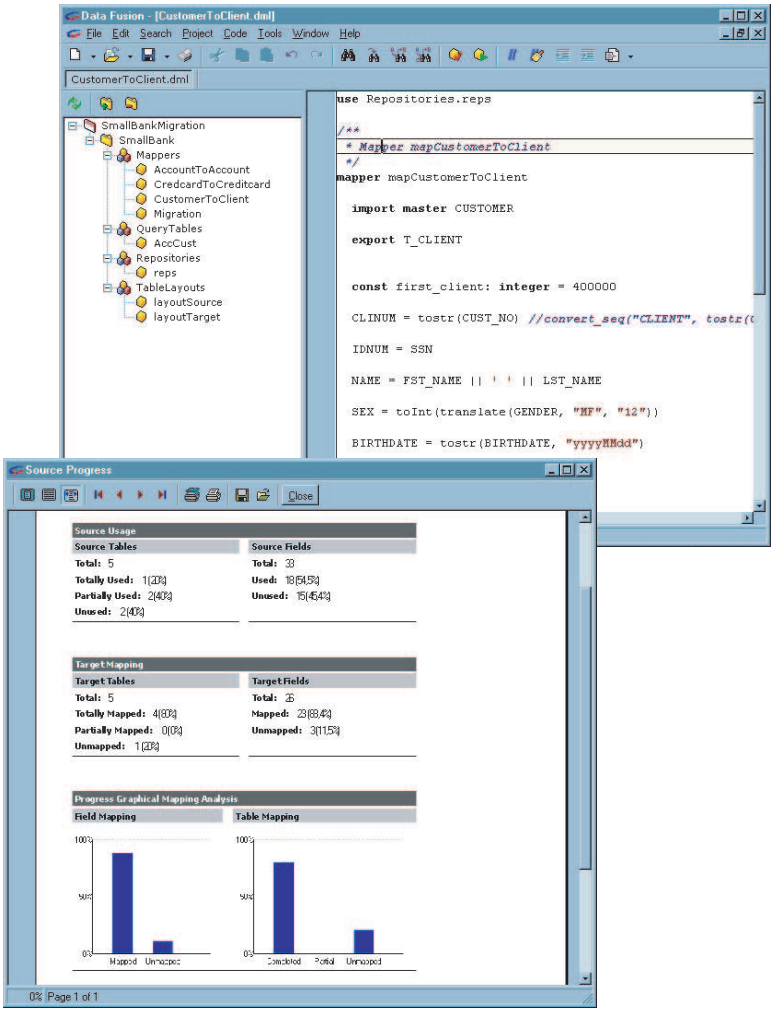


Figure 6.18: A snapshot of Data Fusion IDE displaying the editor (top) together with the mapping statistics (bottom).

DTL brings several advantages. First, migration transformations can be expressed in a language close to the problem domain. Second, very large data transformation projects can be decomposed and arranged into packages according to the functionality provided. This is an essential feature for the success of real-word projects. Third, the compiler can check if the specific vocabulary is correctly used. In DTL, for example, a target attribute cannot be assigned twice.

The Data Fusion IDE supports the development of data migration projects. It follows the trend of modern environments for software development (e.g., Eclipse

or Visual Studio). The IDE illustrated in Figure 6.18, includes a text editor that supports known functionalities, such as syntax highlighting and code templates. Moreover, the DTL compiler is integrated within the IDE and provides helpful hints when compilation errors occur. The user can configure the IDE in order to differentiate among production and development modes. The types of errors that are allowed when writing and testing a migration application are not the same as the ones that may occur when migrating real data. A debugger facility was proposed to be integrated with the IDE for tracking errors in data migration specifications.

The IDE also supports project management through a project tracking facility that shows to be very useful in real data migration applications. One feature provided is impact analysis reports that display how source and target fields depend on each another. Auditing a data migration project is a very common activity. For this purpose, a special auditing report that shows the logic underlying each source field is generated. Since the information to migrate is precious, in the sense that every source record must be migrated and every slot of the target schema must be filled in, the project stakeholders ask for periodically checking the coverage of the data migration process. The IDE also provides a set of progress reports that display the state of all source and target fields, i.e., the association between all target and source fields, the percentage of source and target tables already covered, etc.

6.5.2 Architecture

The Data Fusion platform follows the client-server architecture depicted in Figure 6.19. On the client side, the *Integrated Development Environment* (IDE) allows users to work in multiple data migration projects. On the server side, the *Run-Time Environment* (RTE) is responsible for compiling and parallelizing the data migration requests submitted from IDE instances. This client-server architecture attains scalability. An instance of the IDE may submit requests to multiple RTE instances and an instance of the RTE may run, in parallel, accepted submissions from multiple IDE instances. The IDE is constituted by:

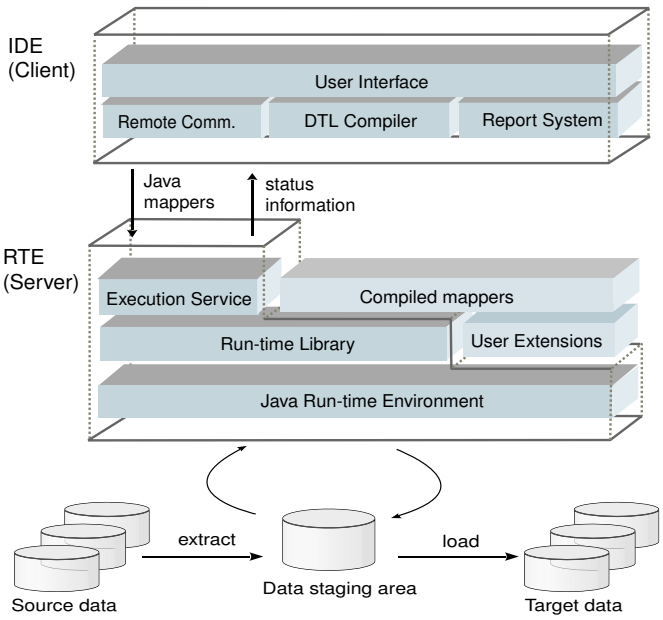


Figure 6.19: Architecture of Data Fusion.

i) the Graphical User Interface, which is a development environment for DTL specifications;

ii) the Remote Communication Subsystem, in charge of submitting the compiled mappers and receiving the migration progress information;

iii) the DTL compiler, which generates Java code from DTL mappers;

iv) the Report System, which is responsible for displaying project tracking and auditing information.

The RTE is consists of:

i) an Execution Service, responsible for processing submission requests by compiling, launching and monitoring the execution of mappers;

ii) a Run-time Library that implements the semantic concepts of DTL;

iii) the Java Run-time Environment, which is responsible for executing the Java code.

The transformations are executed by the RTE on a data staging area that can be supported by any RDBMS with a JDBC connection. Data extraction and loading are performed by third-party tools (e.g., Oracle SQL*Loader).

6.5.3 Real-world experience

Data Fusion has been used in several real data migration projects in the Portuguese banking industry. It was also selected by the Spanish software house INDRA[1] to migrate financial data, and by Siemens to integrate three databases storing Portuguese public administration information.

In each of these projects, Data Fusion handled the migration of entire information systems, each comprising around 1000 tables. Concerning the complexity of data transformation rules, most of the rules (about 90%) were simple, with a small set of rules (about 10%) accounting for most of the complexity. Simple rules consist of constant and attribute assignments, mathematic expressions, code conversions and simple conditional assignments. Complex rules are those that involve complex computations like check-digit computations or one-to-many data transformations.

Most one-to-many to many data transformations implemented consisted of bounded one-to-many data transformations requiring only one multi-valued function. Mappers proved useful when one-to-many data transformations had to be handled, drastically reducing the time taken to develop these data transformations. In DTL, mappers were also helpful as an abstraction for organizing the very large amounts of mapping rules.

6.6 Conclusions

This chapter presented the conducted experiments aimed at validating the feasibility of the mapper operator, including its logical and physical optimization. The chapter starts by analyzing several alternatives for implementing one-to-many data transformations. The analysis compare different implementations of

[1] http://www.indra.es

bounded and unbounded one-to-many data transformations using two RDBMSs and mapper operators implementing the Naïve physical execution algorithm.

The experiments show that the naïve implementation of the mapper operator outperforms the RDBMS solutions, except on the cases where the cache size may become bigger than the size of the input relation (often infeasible in practice). In that case, the implementation of bounded one-to-many data transformations using an SQL query is faster by $\approx 10\%$ than the mapper approach. The influence of selectivity and fanout factors on the throughput of one-to-many data transformations was also reported. As it turns out, the highest selectivities and highest fanout factors correspond to the lowest throughputs for all implementations. From the analysis of the query plans generated by the RDBMSs for unions and recursive queries, it was clear that RDBMSs do not, in general, perform logical optimization of queries that access the same input relation multiple times. Instead, they simply rely on the cache to save physical reads. This implies that the performance of one-to-many data transformations implemented using currently available RDBMS solutions can be very sensitive to cache size. In contrast, the performance of the mapper operator remains very good even when not using cache resources.

The logical optimization rules for the mapper operator proposed in Chapter 4 were also validated. The validation compared original unoptimized expressions with their optimized equivalents. The experiments highlight the influence of predicate selectivity, mapper function fanout, and mapper function cost on the gains obtained by the optimizations for the algebraic optimization rules that combine selections and mappers. High gains are obtained for expressions involving predicates with small selectivity factors and mapper functions with high fanouts and expensive functions. Moreover, the experiments also validated the accuracy of the cost model proposed in Section 4.6.

The behavior of the alternative execution algorithms proposed in Chapter 5 for the mapper operator was also assessed. The Shortcircuiting algorithm is clearly advantageous in cases where selective and expensive mapper functions are used. In turn, cache-based algorithms seem to be successful when the input relation has duplicate values. Despite the modest results, the XLUR Cache-based algorithm proposed in this thesis was capable of improving the performance of the mapper

implementation. This approach validates the choice of approximating an utility metric for performing cache replacement decisions.

The usefulness of the mapper was validated by implementing the mapper operator, with the Naïve algorithm, in the Data Fusion tool. The support for one-to-many mappings the large commercial legacy-data migration projects was found to greatly reduce the cost involved in deploying one-to-many data transformations.

In summary, one-to-many data transformations can be executed with promising performance using the mapper operator. A first set of experiments showed that even the naïve implementation of the mapper operator is capable of outperforming RDBMS based solutions in most situations. In addition, since expressions involving standard relational operators and the mapper operator can be logically and physically optimized, the performance of one-to-many data transformations expressed using the mapper operator can be greatly enhanced.

Chapter 7

Conclusions

This chapter starts reviewing the main objectives of the thesis, followed by a discussion of the limitations of the conducted work. Then, it presents directions for future work. The thesis closes with a discussion about the broadness of application of its contributions.

7.1 Summary

This thesis proposed a specialized operator for expressing one-to-many data transformations in a way that is declarative, expressive and optimizable. It contains the following contributions:

Evaluation of how RDBMSs handle one-to-many transformations. The alternative RDBMS implementations for expressing one-to-many data transformations (Sections 2.2 and 2.3) were compared experimentally (Section 6.2). It was concluded that bounded one-to-many transformations can be expressed as SQL queries and are optimizable by the query optimizer. However, their performance is very sensitive to cache size variations. Unbounded one-to-many data transformations can only be expressed as table functions, stored procedures or recursive queries. Table functions and stored procedures are, in general, not optimized. Recursive one-to-many transformations, expressed as relational queries, do not provide an efficient alternative. These findings support the claim that no comprehensive solution exists for

tackling one-to-many data transformations, which should be at the same time, declarative, optimizable and capable of expressing all conceivable one-to-many data transformations.

A specialized operator for expressing one-to-many transformations. In order to provide an adequate solution for the problem of expressing one-to-many data transformations, Chapter 3 introduced the new specialized *data mapper* operator, as an extension to RA.

The mapper operator was formalized as a unary operator capable of producing multiple output tuples for each input tuple (Section 3.2). Like other extensions to RA, such as the generalized projection and aggregation operator, the mapper operator also relies on the use of external functions. In order to better understand the mapper operator, some of its properties were studied (Section 3.3). Among these, the demonstration that the mapper semantics can be implemented as a Cartesian product of the function's output values leads to a simple naïve physical execution algorithm. Another noteworthy property is that the RA extended with the mapper operator is more expressive than standard RA.

A straightforward extension to the SQL syntax to handle mappers was proposed in Section 3.6. One-to-many data transformations can be denoted by expressions that combine standard relational operators with mappers, which can then be logically and physically optimized.

The mapper operator was incorporated in Data Fusion, a data transformation tool used in real world settings, in order to validate the relevance of one-to-many data transformations (Section 6.5). The tool has been selected for several legacy-data migration projects of banking information systems. The support for one-to-many data transformations through a specialized operator had a positive impact on the effort required to develop complex data transformations.

A set of provably correct algebraic optimization rules. A set of algebraic rewriting rules for generating logical query plans involving mappers and

standard relational operators were proposed. Sections 4.2 to 4.5 introduce these rules, together with their corresponding formal proofs of correctness.

The proposed logical optimization rules were validated through multiple experiments contrasting unoptimized one-to-many data transformations that apply selections to mappers, with their algebraically optimized equivalents (Section 6.3).

A cost model for the Naïve algorithm was proposed for studying the cost-based optimization of expressions involving mappers (Section 4.6). The experiments, reported in Section 6.3, confirmed the accuracy of the proposed cost-model and showed that the introduction of algebraic optimizations imparts high gains.

Optimized mapper execution algorithms. The semantics of the mapper operator (Section 3.2) suggests a simple iterator-based algorithm implementation (Section 5.2). This algorithm, designated as Naïve, despite its simplicity, may turn out to be very inefficient, especially when expensive mapper functions are present.

To overcome this difficulty, two new evaluation algorithms were proposed for reducing the overall mapper evaluation cost. Both algorithms rely on avoiding superfluous function evaluations. The Shortcircuiting algorithm, takes advantage of the semantics of the mapper operator, skipping the evaluation of the remaining functions, as soon as the result of a mapper function is an empty set (Section 5.3). The Cache-based algorithm explores the presence of duplicated values in the input relation through an in-memory cache of mapper function results (presented in Section 5.4).

To overcome the limitations of an in-memory cache, strategies for cache replacement were also considered. Section 5.5 introduced a cache-based mapper evaluation algorithm with an LRU replacement strategy, commonly implemented in databases and operating systems. Two new replacement strategies, specific for mapper evaluation, were proposed. The Least Useful Replacement (LUR) bases its replacement decisions on the maximization of an utility function. It considers the number of references to an entry as

well as the function evaluation cost, besides the time-to-last reference (see Section 5.6). Because LUR cannot be widely applied in practice due to its big overhead, a lightweight approximation to LUR based on multiple LRU stacks, designated as Relaxed LUR (XLUR) was also proposed (Section 5.7).

The main finding of the analysis of the proposed algorithms showed that both the Shortcircuiting and the Cache-based techniques are quite successful in reducing the overall evaluation cost of the mapper execution. They showed important improvements both on synthetic and real-world data sets. In addition, the experiments demonstrated that the gains are highly dependent on factors such as the mapper function cost, the mapper function selectivity and the distribution of duplicates on the input relation (Section 6.4).

7.2 Limitations

The generalizability of the results presented in this thesis is bounded by the following issues.

Lack of performance comparison with data transformation tools. The performance of one-to-many data transformations was tested on several implementations that use RDBMSs. However, a comparison of one-to-many data transformations using ETL tools has not been conducted. The main factor that hindered such comparison was the difficulty in assessing the internals of data transformation tools, which are often not documented. In contrast, RDBMSs have a well-understood architecture and comprehensive documentation. Without a precise description of the internals of data transformation tools, the conclusions drawn from the experiments could be misleading.

Nevertheless, the lack of experimentation with ETL tools has a small impact on the conclusions regarding the implementation of one-to-many data transformation, since the underlying technology is progressively becoming more and more RDBMSs-like (Amer-Yahia & Cluet, 2004; Galhardas *et al.*, 2000; Simitsis *et al.*, 2005).

Assumption of the completeness of MRA. The extension of RA with the mapper operator (MRA) was shown to be powerful enough for expressing all one-to-many data transformations. Using formal terms, it is assumed that MRA is *complete* with respect to one-to-many data transformations. Unfortunately, this has not been formally demonstrated. Hence, the claim of expressiveness of the mapper operator is undermined, because it can be questioned whether a one-to-many data transformation can be conceived, that is not expressible as a combination of RA operators and mappers.

From a theoretical standpoint, data transformations can be envisioned as functions from databases to relations. The class of functions that denote one-to-many data transformations, represented by $\mathcal{M}$, can be defined as *all data transformations that produce multiple output tuples for each input tuple*. Presumably, the set $\mathcal{M}$ can be formalized and enable the demonstration that MRA is powerful enough to express all one-to-many data transformations.

Lack of a cost model for cache-based evaluation algorithms. There is no principled way to estimate the cost of a data transformation that uses a cache based algorithm. Some of the factors that influence the performance of the cache based algorithm, such as the cache size, the number of duplicates, the mapper function cost or the average inter-reference interval, have been identified and validated in Sections 5.4 and 6.4.4, respectively. However, the accuracy of the cost estimates of a cache-based algorithm are limited by the difficulties in determining in advance the cache access patterns. In the case of the cache-based implementations of the mapper operator, such patterns are induced by the different distributions of duplicate values within the attributes of the input relations. Forecasting such cache access patterns is an interesting research problem in itself.

However, in this thesis, the lack of a cost model for a Cache-based algorithm does not impact the dynamic selection of a different execution algorithm for mappers. In Section 5.9, a straightforward heuristic is proposed for selecting the most appropriate algorithm. The remaining ambiguity of deciding

between a cache-based evaluation with LRU or one with XLUR can be resolved by always selecting XLUR as a replacement strategy. Section 6.4.4 validates that XLUR performs better or at least as good as LRU.

7.3 Future Work

During the development of this thesis, several interesting lines for future work concerning the implementation of the mapper operator were identified. The bulk of these lines of work is on extending the proposed logical and physical optimizations by incorporating the mapper operator in a query processor of an RDBMS or, alternatively, in the transformation engine of a data-transformation tool.

7.3.1 Further rewriting rules

The rules stated in Chapter 4 can be further extended, enabling the logical optimization of a broader class of queries. One way to do this is to consider further re-writing for joins, grouping and duplicate removal. Concerning join operations, one possible rewriting adopts the form $\mu_F(r) \bowtie \mu_F(s) = \mu_F(r \bowtie s)$, if none of the mapper functions in F produces duplicate values. This rewriting pushes the mappers to the sources, potentially resulting in fewer evaluations than over the joined relation. Moreover, we can introduce rules to take advantage of outer-joins, similar to those proposed by Amer-Yahia & Cluet (2004), for the map operator. The grouping operator $\gamma_{G,L}$, where G represents the grouping attributes and L is the set of aggregation functions, can be pushed through a mapper resulting in potentially huge reductions on the number of tuples passed to the mapper. For example, consider the expression $\gamma_{B,\texttt{COUNT(A)}}(\mu_{X\to A,Y\to B,f_C}(r))$, where f_C is a potentially expensive mapper function, and r is a relation instance. The grouping operator in this expression can only be computed after mapping all tuples of r. In contrast, the equivalent expression $\mu_{X\to A,Y\to B}(\gamma_{Y,\texttt{COUNT(X)}}(r))$ is more efficient to compute because the mapper is evaluated only once for each distinct value of Y. Likewise, the duplicate removal operator $\delta_X(\mu_F(r))$ can be commuted with the mapper operator, resulting in $\mu_F(\delta_Y(r))$, whenever the attributes of X are directly mapped from the attributes of Y.

7.3.2 Cost-based optimizer for one-to-many transformations

A cost-based optimizer capable of optimizing data transformations expressed as queries involving the mapper operator can be implemented. This optimizer must incorporate the logical re-writing rules, the mapper execution algorithms, and the cost model presented in this thesis. Two key extensions to the traditional System/R algorithm need to be implemented to handle mappers. First, a mechanism for maintaining accurate statistics of selectivity, fanout and cost of the mapper functions, must be available. This is necessary for estimating the expected cardinality of the mapper operator and its expected cost, using the various physical execution algorithms. Second, the optimizer must be extended to handle expensive mapper functions. Expensive mapper functions introduce a new requirement to a cost-based optimizer: when determining the join orderings, besides minimizing I/O cost, the optimizer must minimize the cost of evaluating mappers functions.

In addition, since one-to-many data transformations usually arise in the context of data management activities (e.g., ETL, data integration and cleaning), a cost-based optimizer tunned for one-to-many data transformations should take into account the characteristics of the workload of data transformations. This kind of workload differs from the traditional RDBMSs workload in aspects such as:

Long running queries. The workload resulting from data management activities is characterized by a small number of complex and long-running queries (that only change if the source or target schemas change) with a small degree of concurrency. In this context, the optimizer can devote more time and resources to finding better plans in the prospect of achieving greater savings during the execution of the queries. The heuristics of a traditional optimizer must be revised. For instance, bushy join trees can be considered instead of only left-deep join trees as considered by most RDBMS optimizers.

Heterogeneous data sources. Data is often read from heterogeneous sources. These sources are often database systems that use distinct representations of

data, may have diverse query capabilities and different data transfer rates. The challenges of optimizing queries against heterogeneous data sources have been considered in the literature on query evaluation in heterogeneous sources and distributed databases (Kossmann, 2000).

Heterogeneous data targets. Instead of being returned to the user, data produced by a query implementing a data transformation is often loaded into another database system. It is frequently required that this load be performed in multiple target systems simultaneously. As a result, the optimizer must also optimize the data-load process. One way to improve the load process consists of loading the target data ordered according to the indexes defined on the target relations, because this avoids random I/O seeks in the target system.

From a technical standpoint, the problem corresponds to that of optimizing the access plan for a set of *interesting orders* associated with the target relations. It can, in principle, be obtained by enhancing the interesting orders mechanism of the traditional optimizers. However, computing interesting orders when mappers are present is a complex task, since it is not always possible to automatically decide whether the result of a mapper function is ordered if the input of the function is ordered. The query optimizer can identify many trivial cases. Complex functions could be annotated with a flag indicating that the function is *order-preserving*.

7.4 Closing Notes

This thesis introduced the new mapper operator and proposed a solution for the problem of expressing and executing queries with it. The conducted research impacts the technology used for performing data transformations, since it shows that another class of data transformations can be expressed and optimized using the best practices of logical and physical independence granted by RDBMSs. Nowadays, RDBMSs perform increasingly complex roles in many data management activities as data staging areas and as data transformation engines. This thesis also contributes to broadening the span of application of RDBMS by enlarging

the class of data transformations that they can effectively handle. The mapper operator is also a valuable addition to a data transformation tool by uncovering one-to-many data transformations in scripts, turning data management software easier to understand and maintain.

Appendix A

Mathematical Proofs

A.1 Cost Formulas

Development of Equation (4.7) into Equation (4.8):

$$\begin{aligned}
\Delta_{4.3} &= n \cdot (C_{prd} + C_F) + n \cdot (O_F \cdot C_{sel}) - n \cdot (C_{prd'} + C_F + O_{g_{A_j}} \cdot C_{sel}) \\
&= n \cdot (C_{prd} + O_F \cdot C_{sel}) - n \cdot (C_{prd'} + O_{g_{A_j}} \cdot C_{sel}) \\
&= n \cdot (k \cdot O_F + m \cdot k_0 + O_F \cdot C_{sel}) - n \cdot (k \cdot \alpha \cdot O_F + m \cdot k_0 + O_{g_{A_j}} \cdot C_{sel}) \\
&= n \cdot (k \cdot O_F + O_F \cdot C_{sel}) - n \cdot (k \cdot \alpha \cdot O_F + O_{g_{A_j}} \cdot C_{sel}) \\
&= n \cdot k \cdot O_F \cdot (1 - \alpha) + n \cdot C_{sel} \cdot (O_F - O_{g_{A_j}})
\end{aligned}$$

Development of $\Delta_{4.3} > 0$:

$$\begin{aligned}
\Delta_{4.3} &> 0 \\
n \cdot k \cdot O_F \cdot (1 - \alpha) - n \cdot C_{sel} \cdot (O_{g_{A_j}} - O_F) &> 0 \\
k \cdot O_F \cdot (1 - \alpha) &> C_{sel}(O_{g_{A_j}} - O_F)
\end{aligned}$$

Development of Equation (4.10) into Equation (4.11):

$$\begin{aligned}
\Delta_{4.4} &= n \cdot (C_{prd} + C_F) + n \cdot O_F \cdot C_{sel} - n \cdot (C_{sel} + C_H) - n \cdot \alpha \cdot (C_{prd} + C_F) \\
&= n \cdot C_{prd} \cdot (1 - \alpha) + n \cdot C_F \cdot (1 - \alpha) + n \cdot C_{sel} \cdot (O_F - 1) - n \cdot C_H \\
&= n \cdot (1 - \alpha) \cdot (C_{prd} + C_F) + n \cdot C_{sel} \cdot (O_F - 1) - n \cdot C_H
\end{aligned}$$

Development of $\Delta_{4.4} > 0$:

$$\begin{aligned} n \cdot (1-\alpha) \cdot (C_{prd} + C_F) + n \cdot (O_F - 1) \cdot C_{sel} - n \cdot C_H > 0 \\ (1-\alpha) \cdot (C_{prd} + C_F) + (O_F - 1) \cdot C_{sel} - C_H > 0 \\ C_H < (1-\alpha) \cdot (C_{prd} + C_F) \\ + (O_F - 1) \cdot C_{sel} \end{aligned}$$

Development of $\Delta_{4.3} - \Delta_{4.4}$:

$$\begin{aligned} \Delta_{4.3} - \Delta_{4.4} &= n \cdot k \cdot O_F \cdot (1-\alpha) + n \cdot C_{sel} \cdot (O_F - O_{g_{A_j}}) \\ &\quad - \big(n \cdot (1-\alpha) \cdot (C_{prd} + C_F) + n \cdot C_{sel} \cdot (O_F - 1) - n \cdot C_H\big) \\ &= n \cdot (1-\alpha) \cdot k \cdot O_F + n \cdot C_{sel} \cdot (O_F - O_{g_{A_j}}) \\ &\quad - n \cdot (1-\alpha) \cdot (C_{prd} + C_F) - n \cdot C_{sel} \cdot (O_F - 1) + n \cdot C_H \\ &= n \cdot (1-\alpha) \cdot (k \cdot O_F - C_{prd} - C_F) + n \cdot C_{sel} \cdot (O_F - O_{g_{A_j}} - O_F + 1) + n \cdot C_H \\ &= n \cdot (1-\alpha) \cdot (k \cdot O_F - k \cdot O_F - m \cdot k_0 - C_F) + n \cdot C_{sel} \cdot (1 - O_{g_{A_j}}) + n \cdot C_H \\ &= -n \cdot (1-\alpha) \cdot (C_F + m \cdot k_0) + n \cdot C_{sel} \cdot (1 - O_{g_{A_j}}) + n \cdot C_H \\ &= n \cdot C_H + n \cdot C_{sel} \cdot (1 - O_{g_{A_j}}) - n \cdot (1-\alpha) \cdot (C_F + m \cdot k_0) \end{aligned}$$

Development of $\Delta_{4.4} - \Delta_{4.3}$:

$$\begin{aligned} \Delta_{4.4} - \Delta_{4.3} &= n \cdot (1-\alpha) \cdot (C_{prd} + C_F) + n \cdot C_{sel} \cdot (O_F - 1) - n \cdot C_H \\ &\quad - \big(n \cdot k \cdot O_F \cdot (1-\alpha) + n \cdot C_{sel} \cdot (O_F - O_{g_{A_j}})\big) \\ &= n \cdot (1-\alpha) \cdot (C_{prd} + C_F) + n \cdot C_{sel} \cdot (O_F - 1) - n \cdot C_H \\ &\quad - n \cdot (1-\alpha) \cdot k \cdot O_F - n \cdot C_{sel} \cdot (O_F - O_{g_{A_j}}) \\ &= n \cdot (1-\alpha) \cdot (C_{prd} + C_F - k \cdot O_F) + n \cdot C_{sel} \cdot (O_F - 1 - O_F + O_{g_{A_j}}) - n \cdot C_H \\ &= n \cdot (1-\alpha) \cdot (k \cdot O_F + m \cdot k_0 + C_F - k \cdot O_F) + n \cdot C_{sel} \cdot (O_{g_{A_j}} - 1) - n \cdot C_H \\ &= n \cdot (1-\alpha) \cdot (C_F + m \cdot k_0) + n \cdot C_{sel} \cdot (O_{g_{A_j}} - 1) - n \cdot C_H \end{aligned}$$

A.2 Binary Rank Ordering Lemma

Herein we present the proof of Lemma 5.1. Consider a set of mapper functions F and a sequence $\omega_{i \prec j} \in \Omega(F)$. We will prove that $\omega_{i \prec j}$ is more economic than $\omega_{j \prec i}$, whenever $rank(\omega[i]) \leq rank(\omega[j])$. To simplify the notation we set $f = \omega[i]$ and

$g = \omega[i]$. We will use C_f and C_g to represent the expected cost of evaluating f and g. The selectivity factor of the function $\omega[i]$ is represented by α_i. Whenever clearer, we will also denote the selectivity factor of f by α_f. If, by hypothesis, $\omega_{i \prec j}$ is the optimal order, we want to prove that

$$C_F^{\omega_{i \prec j}} \leq C_F^{\omega_{j \prec i}} \tag{A.1}$$

is equivalent to

$$rank(f) \leq rank(g) \tag{A.2}$$

mutatis mutandis for the case $\omega_{j \prec i}$. Taking into account the meaning of C_F^{ω}, Equation (A.1) can be rewritten as

$$\sum_{f \in F} P^{\omega_{i \prec j}}(f) \cdot C_f \leq \sum_{f \in F} P^{\omega_{j \prec i}}(f) \cdot C_f \tag{A.3}$$

which simplifies to

$$P^{\omega_{i \prec j}}(f) \cdot C_f + P^{\omega_{i \prec j}}(g) \cdot C_g \leq P^{\omega_{j \prec i}}(f) \cdot C_f + P^{\omega_{j \prec i}}(g) \cdot C_g \tag{A.4}$$

Since the function $P^{\omega_{i \prec j}}$ is defined by cases, the rest proof development will follow by cases. We start by considering the case where the probability of evaluating f is 0. Whenever the probability of evaluating f is not 0, we consider two more cases, first the sub-case when f is the first function on the sequence $\omega_{i \prec j}$, i.e., when $i = 1$, and then the sub-case when $i > 1$.

Case 1. When $P^{\omega_{i \prec j}}(f) = 0$. In this case, since we assume that f always precedes g, the Shortcircuiting algorithm also does not evaluate g, and thus necessarily $P^{\omega_{i \prec j}}(g) = 0$. Thus, Equation (A.4) holds trivially.

Case 2. When $P^{\omega_{i \prec j}}(f) > 0$ and $i = 1$. If we take into account that $j = i+1$, then

$$P^{1 \prec 2}(f) = 1 \text{ and } P^{1 \prec 2}(g) = \alpha_f$$

Conversely, if g was evaluated before f then we would have

$$P^{2 \prec 1}(f) = \alpha_g \text{ and } P^{2 \prec 1}(g) = 1$$

Thus, taking into account the former case case, the inequality of Equation (A.4) can rewritten as

$$C_f + \alpha_f \cdot C_g \leq \alpha_g \cdot C_f + C_g \tag{A.5}$$

After switching sides and factorizing, becomes equivalent to

$$(1 - \alpha_g) \cdot C_f \leq (1 - \alpha_f) \cdot C_g \tag{A.6}$$

Which simplifies to

$$\frac{C_f}{(1 - \alpha_f)} \leq \frac{C_g}{(1 - \alpha_g)} \tag{A.7}$$

This is the same as $rank(f) \leq rank(g)$.

Case 3. When $P(f) > 0$ and $i > 1$. If we assume that f is evaluated before g then

$$P^{\omega_{i \prec j}}(f) = \prod_{k \leq i} \alpha_k \text{ and } P^{\omega_{i \prec j}}(g) = \alpha_g \cdot \prod_{k \leq i} \alpha_k \tag{A.8}$$

Conversely, if g is evaluated before f then

$$P^{\omega_{j \prec i}}(f) = \alpha_g \cdot \prod_{k \leq i} \alpha_k \text{ and } P^{\omega_{j \prec i}}(g) = \prod_{k \leq i} \alpha_k \tag{A.9}$$

Making

$$p = \prod_{k \leq i} \alpha_{f_{A_p}} \tag{A.10}$$

This inequality expands to

$$p \cdot C_f + \alpha_f \cdot p \cdot C_g \leq \alpha_g \cdot p \cdot C_f + p \cdot C_g \tag{A.11}$$

Switching sides, we have

$$p \cdot C_f - \alpha_g \cdot p \cdot C_f \leq p \cdot C_g - \alpha_f \cdot p \cdot C_g \tag{A.12}$$

Factorizing

$$(1 - \alpha_g) \cdot p \cdot C_f \leq (1 - \alpha_f) \cdot p \cdot C_g \tag{A.13}$$

Simplifying we get

$$(1 - \alpha_g) \cdot C_f \leq (1 - \alpha_f) \cdot C_g \tag{A.14}$$

Which is the same as

$$\frac{C_f}{(1 - \alpha_f)} \leq \frac{C_g}{(1 - \alpha_g)} \tag{A.15}$$

Which denotes $rank(f) \leq rank(g)$.

A.3 Optimality of the Ascending Rank Ordering

This section demonstrates the claim of Theorem 5.1, which is that the evaluation sequence that corresponds to the ascending rank order is an optimal strategy for the Shortcircuiting algorithm. Given a set F of mapper functions, we start by observing that whatever sequence $\omega \in \Omega(F)$, the probability that an empty set is returned after evaluating all the mapper functions, i.e., $P^{\omega}(f_{A_m})$, is the same. This makes invariant

$$P^{\omega}(f_{A_m}) \cdot k \cdot \prod_{f \in F} O_f + m \cdot k_0 \tag{A.16}$$

in the cost formula of the Shortcircuiting algorithm given in Equation (5.3). Thus, the optimal evaluation sequence is the one that minimizes

$$\sum_{f \in F} P^{\omega}(f) \cdot C_f \tag{A.17}$$

which is equivalent to C_F^{ω}. It remains to be shown that the sequence that minimizes C_F^{ω} must be an ascending rank order of the mapper functions. The proof is developed by reductio ad absurdum. Suppose that there exists an evaluation sequence $\omega_{i \prec j}$, that is optimal but that is not in ascending rank order. Moreover,

consider that i and j are the indices of two adjacent functions in that evaluation sequence that are not rank ordered, i.e., such that $rank(\omega[i]) > rank(\omega[j])$. By exchanging i with j we get a sub-sequence $\omega_{j \prec i}$, which, by Lemma 5.1 is more economic than $\omega_{i \prec j}$. Thus, the sequence $\omega_{j \prec i}$ obtained from $\omega_{i \prec j}$ by exchanging i with j is cheaper than $\omega_{i \prec j}$, which is a contradiction. Hence, any optimal sequence ω corresponds necessarily to an ascending rank order of the mapper functions.

A.4 More Past References Imply Greater Utility

This section demonstrates that more past references lead to greater utility. Let C be a cache with entries e_1 and e_2 to have the same cost c and the same frequency θ. It must be proved that, after an equal k without referencing e_1 or e_2, $n_{h_1} > n_{h_2}$ implies $u_{t_0}(e_1) > u_{t_0}(e_2)$.

The equality $\theta_1 = \theta = \theta_2$ can be written as:

$$\frac{n_{h1}}{t_l - t_{a1}} = \theta = \frac{n_{h2}}{t_l - t_{a2}} \tag{A.18}$$

This is true if $n_{h1} = n_{h2}$ and $t_{a1} = t_{a2}$. Otherwise, to maintain the frequencies equal, either $n_{h1} > n_{h2}$ (which implies that $t_{a_1} < t_{a_2}$), or $n_{h1} < n_{h2}$, (which implies that $t_{a_2} < t_{a_1}$). Whenever $n_{h1} > n_{h2}$, the entry e_1 has been referenced more often in the past than e_2, implying that e_1 must have been seen for the first time before e_2. Hence, from Definition 5.4 it follows that $u_{t_0}(e_1) > u_{t_0}(e_2)$. Hence, the entry that has the oldest arrival time is the most useful. A similar reasoning applies to the case $n_{h2} > n_{h1}$.

Appendix B

Overview of Cache Replacement Strategies

A cache is a mechanism that enables algorithms to trade space for time by storing the results of costly operations in memory. Due to space constraints, some entries have to be discarded to make room for newer ones. The selection of the entry to be discarded is governed by a *cache replacement strategy*.

The cache replacement strategy directly influences the performance of cache-based algorithms, since different strategies may have a distinct capabilities to correctly predict which entries will be needed in the future. Since, it is impossible for many algorithms to predict accurately in practice the future references to a cache entry, the replacement decisions are usually based on heuristics that exploit patterns of references to the cache. These are known as *cache access patterns*.

Different algorithms result in different cache access patterns. Hence, a cache replacement strategy should be tunned to the specific cache access pattern of the algorithm in order to increase the chances of finding previously computed results in the cache. One well-known cache access pattern is *temporal locality of references*, which postulates that once an entry is referenced, then it will be referenced again in the near future (Coffman Jr. & Denning, 1973, Section 7.2). This pattern has been exploited by heuristics such as replacing the least recently used entry (LRU), or replacing the least frequently used entry (LFU).

Cache replacement strategies that base their replacement decisions on a single metric, like time-to-last reference in the case of LRU or access frequency in the

case of LFU, perform sub-optimally whenever the cost of creating the cache entries is not uniform. For example, LFU may discard a very expensive entry instead of a cheaper entry, even if the frequency of the expensive entry is marginally lower than the frequency of the cheaper one.

Cache replacement strategies can be enhanced to consider compound metrics that also include cost. In this way, the entries kept in cache are not only those that are more likely to be referenced in the future, but also those that are more expensive to compute, thus reducing the overall computation cost of the cache-based algorithm.

In the context of a cache, *time* does not correspond to the elapsed wall-clock time but rather to the number of times the cache was accessed so far.

Appendix C

Overview of the Zipfian Distribution

A Zipfian distribution is characterized as follows: Let r be a relation with n tuples. The domain of a mapper function f evaluated over r, is a relation $r[Dom(f)]$ with cardinality n and a number $d \leq n$ of distinct values. Sort the distinct values in decreasing order of frequency (also referred to as *popularity*). The position j such that $1 \leq j \leq d$ is known as the *rank*; lower ranks correspond to most frequent values. The frequency the jth distinct input of f is given by $p_j = a \cdot (1/j^z)$, where

$$a = \sum_{j=1}^{d} 1/j^z$$

is a normalization constant and z is the *skewness* parameter. Skewness reflects the asymmetry of the frequency distribution around its mean. If z is set to zero, the distinct values are distributed uniformly. As z increases, more skewed patterns are produced.

The plot of rank versus frequency on a log-log scale of data following a Zipfian distribution displays a straight line trend as exemplified in Schwartz (1963). Similarly, the plot of rank versus frequency of the input values of the relation `CITEDATA` for the functions *name*, *title* and *event* of Example 3.2.2 shown in Figure C.1.

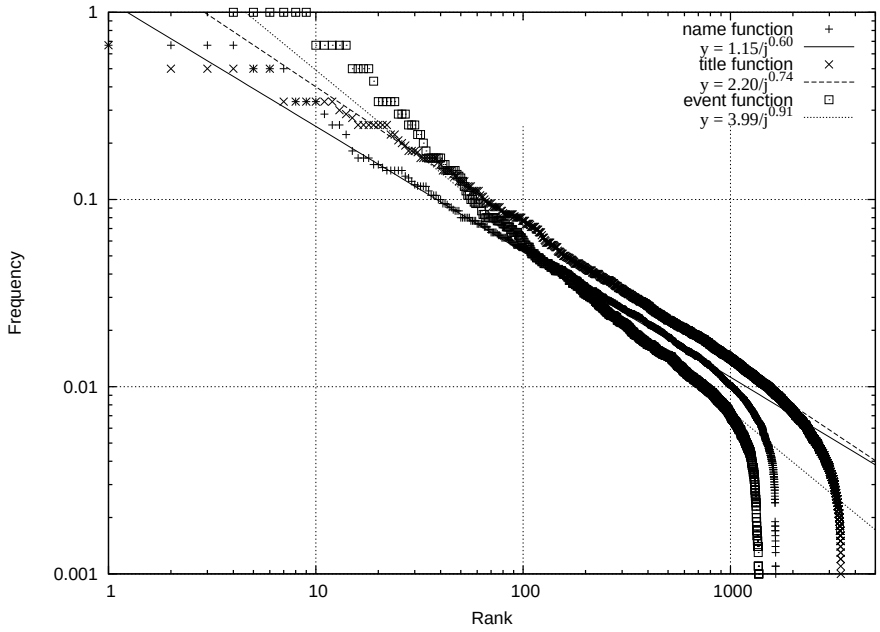


Figure C.1: Rank versus frequency characteristics of the input data for the three most expensive functions of Example 3.2.2 read from a sample with 10K tuples of the `CITEDATA` relation. The plots is rendered on a log-log scale with corresponding fitting functions in the form $y = a/j^z$, where j is the rank (x-axis).

Zipfian data distribution in literature

It as been widely acknowledged that several phenomena in computer science follow powerlaw distributions (Knuth, 1998, p. 399) like the Zipfian distribution (Zipf, 1949). In particular the values in real-world relational databases often follow a Zipfian distributions (Christodoulakis, 1984; Ioannidis & Christodoulakis, 1991; Lowe, 1968; Lynch, 1988; Motwani & Vassilvitskii, 2006; Siler, 1976). Zipfian distributed data is known to influence diverse aspects of query processing, like undermining the execution of certain relational operators (Taniar & Leung, 2003; Wolf *et al.*, 1993) and making plan selection less accurate (Ioannidis & Christodoulakis, 1991; Lynch, 1988). Synthetically generated data for evaluating the performance of database technology also follows Zipfian distributions. This was initially proposed by Siler (1976) and later Gray *et al.* (1994) employed Zipfian distribution in the data generator of the TPC benchmarks (TPC, 1999).

References

Abelson, H., Sussman, G.J. & Sussman, J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press.

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

Abiteboul, S., Quass, D., McHugh, J., Widom, J. & Wiener, J.L. (1997). The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, **1**, 68–88.

Agrawal, R. (1988). Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries. *IEEE Transactions on Software Engineering*, **14**, 879–885.

Ahad, R. & Yao, S.B. (1993). RQL: A Recursive Query Language. *IEEE Transactions on Knowledge and Data Engineering*, **5**, 451–461.

Aho, A.V. & Ullman, J.D. (1979). Universality of Data Retrieval Languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 110–119, ACM Press.

Amer-Yahia, S. & Cluet, S. (2004). A Declarative Approach to Optimize Bulk Loading into Databases. *ACM Transactions of Database Systems*, **29**, 233–281.

Atzeni, P. & de Antonellis, V. (1993). *Relational Database Theory*. The Benjamin/Cummings Publishing Company, Inc.

Barateiro, J. & Galhardas, H. (2005). A Survey of Data Quality Tools. *Datenbank-Spektrum*, **14**, 15–21.

Batini, C., Lenzerini, M. & Navathe, S.B. (1986). A Comparative Analysis of Methodologies for Database Schema Integration. In *ACM Computing Surveys*, vol. 18, 323–364.

Belady, L.A. (1966). A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal*, **5**, 78–101.

Bernstein, P.A. & Chiu, D.M.W. (1981). Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, **28**, 25–40.

Bleiholder, J. & Naumann, F. (2005). Declarative Data Fusion - Syntax, Semantics, and Implementation. In J. Eder, H.M. Haav, A. Kalja & J. Penjam, eds., *9th East European Conference on Advances in Databases and Information Systems (ADBIS 2005)*, vol. 3631 of *Lecture Notes in Computer Science*, 58–73, Springer.

Börzsönyi, S., Kossmann, D. & Stocker, K. (2001). The Skyline Operator. In *Proceedings of the 7th International Conference on Data Engineering (ICDE'01)*, 421–430.

Breslau, L., Cao, P., Fan, L., Phillips, G. & Shenker, S. (1999). Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the IEEE INFOCOM Conference*, 126–134.

Buneman, P., Davidson, S., Hillebrand, G. & Suciu, D. (1996). A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 505–516.

Cao, P. & Irani, S. (1997). Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, 193–206.

Carreira, P. & Galhardas, H. (2003). Efficient Development of Data Migration Transformations. In *Proceedings of the Semantic Integration Workshop (The Second International Semantic Web Conference)*.

Carreira, P. & Galhardas, H. (2004a). Efficient Development of Data Migration Transformations. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*.

Carreira, P. & Galhardas, H. (2004b). Execution of Data Mappers. In *International Workshop on Information Quality in Information Systems (IQIS'04)*, ACM.

Carreira, P., Galhardas, H., Lopes, A. & Pereira, J. (2005a). Extending Relational Algebra to Express One-to-many Data Transformations. In *Proceedings of the 20th Brazilian Symposium on Databases (SBBD'05)*.

Carreira, P., Galhardas, H., Pereira, J. & Lopes, A. (2005b). Data Mapper: An Operator for Expressing One-to-many Data Transformations. In *7th International Conference on Data Warehousing and Knowledge Discovery, DaWaK '05*, vol. 3589 of *LNCS*, Springer-Verlag.

Carreira, P., Galhardas, H., Lopes, A. & Pereira, J. (2007). One-to-many Transformation Through Data Mappers. *Data and Knowledge Engineering Journal*, **62**, 483–503.

Casey, R.G. & Osman, I.M. (1974). Generalized Page Replacement Algorithms in a Relational Data Base. In R. Rustin, ed., *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control*, 101–124, ACM.

Castano, S. & Antonellis, V.D. (1999). A Schema Analysis and Reconciliation Tool Environment. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS'99)*.

Chamberlin, D., Robie, J. & Florescu, D. (2000). Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB (Informal Proceedings)*, 53–62.

CHAMBERLIN, D.D. (2002). XQuery: An XML query language. *IBM Systems Journal*, **41**, 597–615.

CHAUDHURI, S. (1998). An Overview of Query Optimization in Relational Systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98)*, 34–43, ACM Press.

CHAUDHURI, S. & SHIM, K. (1993). Query Optimization in the Presence of Foreign Functions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'93)*, 529–542.

CHAUDHURI, S. & SHIM, K. (1999). Optimization of Queries with User-defined Predicates. *ACM Transactions on Database Systems*, **24**, 177–228.

CHIMENTI, D., GAMBOA, R. & KRISHNAMURTHY, R. (1989). Toward an Open Architecture for LDL. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'89)*, 195–203, Morgan Kaufmann Publishers Inc.

CHOU, H.T. & DEWITT, D.J. (1985). An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of 11th International Conference on Very Large Data Bases (VLDB'85)*, 127–141, Morgan Kaufmann.

CHRISTODOULAKIS, S. (1984). Implications of Certain Assumptions in Database Performance Evaluation. *ACM Transactions on Database Systems*, **9**, 163–186.

CLARK, J. (1999). *XSL Transformations (XSLT) Version 1.0. W3C Recomendation.*. World Wide Web Consortium.

CLARK, J. & DEROSE, S. (1999). *XML Path Language (XPath) Version 1.0. W3C Recomendation.*. World Wide Web Consortium.

CLUET, S. & SIMÉON, J. (1997). Data Integration Based on Data Conversion and Restructuring. Extended version of Cluet *et al.* (1998).

CLUET, S., DELOBEL, C., SIMÉON, J. & SMAGA, K. (1998). Your Mediators Need Data Conversion! In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 177–188.

Codd, E.F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, **13**, 377–387.

Coffman Jr., E.G. & Denning, P.J. (1973). *Operating Systems Theory*. Prentice-Hall Series in Automatic Computation, Prentice-Hall.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. & Stein, C. (2001). *Introduction to Algorithms, 2nd Edition*. MIT Press.

Cui, Y. & Widom, J. (2001). Lineage Tracing for General Data Warehouse Transformations. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01)*.

Cunningham, C., Graefe, G. & Galindo-Legaria, C.A. (2004). PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'04)*, 998–1009, Morgan Kaufmann.

Denning, P.J. (1968). The Working Set Model for Program Behavior. *Communications of the ACM*, **11**, 323–333.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A.Y. & Suciu, D. (1998). XML-QL. In *QL'98 The Query Languages Workshop (W3C Workshop)*.

Deutsch, A., Fernandez, M.F., Florescu, D., Levy, A.Y. & Suciu, D. (1999). A Query Language for XML. *Computer Networks*, **31**, 1155–1169.

Do, H.H. & Rahm, E. (2002). COMA – A System for Flexible Combination of Schema Matching Approaches. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'02)*.

Doan, A.H., , Madhavan, J. & Domingos, P. (2002). Learning to Map Between Ontologies on the Semantic Web. In *Proceedings of the 11th International WWW Conference*.

Effelsberg, W. & Haerder, T. (1984). Principles of Database Buffer Management. *ACM Transactions on Database Systems (TODS'84)*, **9**, 560–595.

Eisenberg, A., Melton, J., Michels, K.K.J.E. & Zemke, F. (2004). SQL:2003 Has Been Published. *Proceedings of the ACM SIGMOD Record*, **33**, 119–126.

Fagin, R., Kolaitis, P.G., Miller, R.J. & Popa, L. (2003). Data Exchange: Semantics and Query Answering. In *Proceedings 8th International Conference on Database Theory (ICDT)*, IEEE Computer Society.

Fernandez, M.F., Florescu, D., Kang, J., Levy, A.Y. & Suciu, D. (1998). Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 414–425.

Feuerstein, S. & Pribyl, B. (2005). *Oracle PL/SQL Programming*. O'Reilly & Associates, 4th edn.

Florescu, D., Levy, A.Y., Manolescu, I. & Suciu, D. (1999). Query Optimization in the Presence of Limited Access Patterns. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 311–322.

Galhardas, H. (2001). *Data Cleaning: Model, Declarative Language and Algorithms*. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelines.

Galhardas, H., Florescu, D., Shasha, D. & Simon, E. (2000). AJAX: An Extensible Data Cleaning Tool. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, **2**.

Galhardas, H., Florescu, D., Shasha, D., Simon, E. & Saita, C.A. (2001). Declarative Data Cleaning: Language, Model, and Algorithms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01)*.

Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J. & Widom, J. (1997). The TSIMMIS Approach to Mediation: Data Models and languages. *Journal of Intelligent Information Systems*, **8**, 117–132.

Garcia-Molina, H., Ullman, J.D. & Widom, J. (2002). *Database Systems – The Complete Book*. Prentice-Hall.

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2005). *The Java Language Specification*. Addison-Wesley, 3rd edn.

Graefe, G. (1993). Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, **2**.

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

Gray, J., Sundaresan, P., Englert, S., Baclawski, K. & Weinberger, P.J. (1994). Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*, 243–252, ACM Press.

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F. & Pirahesh, H. (1997). Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *J. Data Mining and Knowledge Discovery*, **1**, 29–53.

Haas, L.M., Lin, E.T. & Roth, M.T. (2002). Data Integration Through Database Federation. *IBM Systems Journal*, **41**, 578–596.

Halevy, A.Y., Ashish, N., Bitton, D., Carey, M.J., Draper, D., Pollock, J., Rosenthal, A. & Sikka, V. (2005). Enterprise Information Integration: Successes, Challenges and Controversies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 778–787, ACM.

Han, J. & Kamber, M. (2001). *Data Mining: Concepts and Techniques*. Morgan-Kaufmann.

Hanani, M.Z. (1977). An Optimal Evaluation of Boolean Expressions in an Online Query System. *ACM Transactions on Database Systems*, **20**, 344–347.

HELLERSTEIN, J.M. (1998). Optimization Techniques for Queries with Expensive Methods. *ACM Transactions on Database Systems*, **22**, 113–157.

HELLERSTEIN, J.M. & NAUGHTON, J.F. (1996). Query Execution Techniques for Caching Expensive Methods. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 423–434.

HELLERSTEIN, J.M. & STONEBRAKER, M. (1993). Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 267–276, ACM Press.

HERGULA, K. & HÄRDER, T. (2001). How Foreign Function Integration Conquers Heterogeneous Query Processing. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, 215–222, ACM.

HULL, R. & YOSHIKAWA, M. (1990). ILOG: Declarative Creation and Manipulation of Object Identifiers. In *Proceedings of the International Conference on Very Large Databases (VLDB'90)*, 455–468.

HWANG, D.J.H. (1995). *Function-based Indexing for Object-Oriented Databases*. Ph.D. thesis, Massachusetts Institute of Technology.

IOANNIDIS, Y.E. & CHRISTODOULAKIS, S. (1991). On the Propagation of Errors in the Size of Join Results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data (SIGMOD '91)*, 268–277, ACM Press.

ISO-ANSI (1992). *Database Language SQL*. ANSI, ISO/IEC 9075:1992 edn.

ISO-ANSI (1999). *Database Language SQL-Part 2: SQL/Foundation*. ANSI, ISO 9075-2 edn.

J. ROBIE, D.S., J. LAPP (1998). XQL. In *QL'98 The Query Languages Workshop (W3C Workshop)*.

Jaedicke, M. & Mitschang, B. (1998). On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 379–389, ACM Press.

Jaeschke, G. & Schek, H.J. (1982). Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '82)*, 124–138, ACM Press.

Janmohamed, Z., Liu, C., Bradstock, D., Chong, R., Gao, M., McArthur, F. & Yip, P. (2005). *DB2 SQL PL. Essential Guide for DB2 UDB*. Prentice-Hall.

Jiang, S. & Zhuang, X. (2002). LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*.

Johnson, T. & Shasha, D. (1994). 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, 439–450, Morgan Kaufmann Publishers Inc.

Kernighan, B. & Ritchie, D. (1988). *The C Programming Language*. Prentice-Hall, 2nd edn.

Kim, W., Choi, B.J., Hong, E.K., Kim, S.K. & Lee, D. (2003). A Taxonomy of Dirty Data. *Data Mining and Knowledge Discovery*, **7**, 81–99.

Kimball, R. & Caserta, J. (2004). *The Data Warehouse ETL Toolkit*. Willey.

Kirk, T., Levy, A.Y., Sagiv, Y. & Srivastava, D. (1995). The Information Manifold. In C. Knoblock & A. Levy, eds., *Information Gathering from Heterogeneous, Distributed Environments*, Stanford University, Stanford, California.

Kline, K., Gould, L. & Zanevsky, A. (1999). *TransactSQL Programming*. O'Reilly & Associates, 1st edn.

Klug, A. (1982). Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, **29**, 699–717.

Knuth, D. (1998). *The Art of Computer Programing*, vol. 3. Addison-Wesley, Reading, MA.

Koch, C. (2001). *Data Integration against Multiple Evolving Autonomous Schemata*. Ph.D. thesis, Technische Universität Wien, Austria.

Kossmann, D. (2000). The State of the Art in Distributed Query Processing. *ACM Computer Surveys*, **32**, 422–469.

Labio, W., Wiener, J.L., Garcia-Molina, H. & Gorelik, V. (2000). Efficient Resumption of Interrupted Warehouse Loads. *SIGMOD Record*, **29**, 46–57.

Lakshmanan, L.V.S., Sadri, F. & Subramanian, I.N. (1996). SchemaSQL - A Language for Querying and Restructuring Database Systems. In *Proceedings International Conference on Very Large Databases (VLDB'96)*, 239–250.

Lee, D., Choi, J., Kim, J.H., Noh, S.H., Min, S.L., Cho, Y. & Kim, C.S. (1999). On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 134–143, ACM Press.

Li, C., Chang, K., Ilyas, I. & Song, S. (2005). RankSQL: Query Algebra and Optimization for Relational Top-K Queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of Data (SIGMOD '05)*, 131–142, ACM Press.

Lomet, D. & Sarawagi, S., eds. (2000). *Special Issue on Data Cleaning*, vol. 23 of *IEEE Data Engineering Bulletin*, IEEE.

Lowe, T.C. (1968). The Influence of Data Base Characteristics and Usage on Direct Access File Organization. *Journal of the ACM*, **15**, 535–548.

Lynch, C.A. (1988). Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distribution of Column Values. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB '88)*, 240–251, Morgan Kaufmann Publishers Inc.

Lynch, C.A. & Stonebraker, M. (1988). Extended User-Defined Indexing with Application to Textual Databases. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, 306–317, Morgan Kaufmann Publishers Inc.

Madhavan, J., Bernstein, P.A. & Rahm, E. (2001). Generic Schema Matching with Cupid. In *The VLDB Journal*, 49–58.

Madhavan, J., Bernstein, P.A., Domingos, P. & Halevy, A.Y. (2002). Representing and Reasoning about Mappings between Domain Models. In *AAAI/IAAI*, 80–86.

Maier, D. & Stein, J. (1986). Indexing in an Object-Oriented DBMS. In *Proceedings on the International Workshop on Object-oriented database systems (OODS)*, 171–182, IEEE Computer Society Press.

Mayr, T. & Seshadri, P. (1999). Client-Site Query Extensions. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 347–358.

Megiddo, N. & Modha, D. (2004). Outperforming LRU with an Adaptive Replacement Cache. *IEEE Computer*, **37**, 58–65.

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

Miller, R.J. (1998). Using Schematically Heterogeneous Structures. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, **2**, 189–200.

Miller, R.J., Haas, L.M., Hernandéz, M., Ho, C.T.H., Fagin, R. & Popa, L. (2001). The Clio Project: Managing Heterogeneity. *SIGMOD Record*, **1**.

Milo, T. & Zhoar, S. (1998). Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'98)*.

Mishra, P. & Eich, M.H. (1992). Join Processing in Relational Databases. *ACM Computer Surveys*, **24**, 63–113.

Mohan, C. & Levine, F. (1992). ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 371–380, ACM Press.

Motwani, R. & Vassilvitskii, S. (2006). Distinct Value Estimators for Power Law Distributions. In *Proceedings of the Third Workshop on Analytic Algorithmics and Combinatorics (ANALCO'06)*.

O'Neil, E.J., O'Neil, P.E. & Weikum, G. (1993). The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 297–306, ACM Press.

Papakonstantinou, Y., Garcia-Molina, H. & Ullman, J. (1996). MedMaker: A Mediator System Based on Declarative Specifications. In *Proceedings of the International Conference on Data Engineering (ICDE'96)*.

Paredaens, J. (1978). On the Expressive Power of the Relational Algebra. *Information Processing Letters*, **7**, 107–111.

Paulson, L.C. (1996). *ML for the Working Programmer, 2nd Edition*. Cambridge University Press.

Pieciukiewicz, T., Stencel, K. & Subieta, K. (2005). Usable Recursive Queries. In *Proceedings of the 9th East European Conference, Advances in Databases and Information Systems (ADBIS)*, vol. 3631 of *Lecture Notes in Computer Science*, 17–28, Springer-Verlag.

Porto, F., Laber, E. & Valduriez, P. (2003). Cherry Picking: A Semantic Query Processing Strategy for the Evaluation of Expensive Predicates. In *Proceedings of the 18th Brazilian Symposium on Databases (SBBD'03)*, 356–370, UFAM.

Rahm, E. & Do, H.H. (2000). Data Cleaning: Problems and Current Approaches. *IEEE Bulletin of the Technical Committee on Data Engineering*, **24**.

Raman, V. & Hellerstein, J.M. (2000). An Interactive Framework for Data Cleaning. Tech. Rep. UCB/CSD-0-1110, Computer Science Division (EECS), University of California, Berkeley, California 94720.

Raman, V. & Hellerstein, J.M. (2001). Potter's Wheel: An Interactive Data Cleaning System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01)*.

Refaat, M. (2006). *Data Preparation for Data Mining Using SAS*. The Morgan Kaufmann Series in Data Management Systems.

Rifaieh, R. & Benharkat, A.N. (2002). Query-based Data Warehousing Tool. In D. Theodoratos, ed., *Proceedings of the 5th ACM International Workshop on Data Warehousing and OLAP (DOLAP 2002)*, 35–42, ACM.

Rizzo, L. & Vicisano, L. (2000). Replacement Policies for a Proxy Cache. *IEEE/ACM Transactions on Networking*, **8**, 158–170.

Robinson, J.T. & Devarakonda, M.V. (1990). Data Cache Management using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 134–142, ACM Press.

Rundensteiner, E.A. (1999). Letter from the Special Issue Editor. *IEEE Data Engineering Bulletin*, **22**, 2.

Sacco, G.M. & Schkolnick, M. (1986). Buffer Management in Relational Database Systems. *ACM Transactions on Database Systems*, **11**, 473–498.

Scheuermann, P., Shim, J. & Vingralek, R. (1997). A Case for Delay-conscious Caching of Web Documents. *Computer Networks and ISDN Systems*, **29**, 997–1005.

Schwartz, E.S. (1963). A Dictionary for Minimum Redundancy Encoding. *Journal of the ACM*, **10**, 413–439.

Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A. & Price, T.G. (1979). Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 23–34.

Shan, M.C. & Neimat, M.A. (1991). Optimization of Relational Algebra Expressions containing Recursion Operators. In *Proceedings of the 19th annual conference on Computer Science (CSC '91)*, 332–341, ACM Press.

Sheth, A.P. & Larson, J.A. (1990). Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, **22**, 183–236.

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

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

Silberschatz, A., Korth, H.F. & Sudarshan, S. (2005). *Database Systems Concepts*. MacGraw-Hill, 5th edn.

Siler, K.F. (1976). A Stochastic Evaluation Model for Database Organizations in Data Retrieval Systems. *Communications of the ACM*, **19**, 84–95.

Simitsis, A., Vassiliadis, P. & Sellis, T.K. (2005). Optimizing ETL processes in data warehouses. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*.

Smaragdakis, Y., Kaplan, S. & Wilson, P. (1999). EELRU: Simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, **27**, 122–133.

Smith, A.J. (1978). Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, **3**, 223–247.

Suciu, D. (1998). An Overview of Semistructured Data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, **29**, 28–38.

Taniar, D. & Leung, C.H. (2003). The Impact of Load Balancing to Object-Oriented Query Execution Scheduling in pArallel Machine Environment. *Information Sciences*, **157**, 33–71.

Thomas, S.J. & Fischer, P.C. (1986). Nested Relational Structures. *Advances in Computing Research*, **3**, 269–307.

TPC (1999). Benchmark H Standard Specification. http://www.tpc.org.

Ullman, J.D. (1988). *Principles of Database and Knowledge-Base Systems*, vol. I. Computer Science Press. New York.

Valduriez, P. & Boral, H. (1986). Evaluation of Recursive Queries Using Join Indices. In *1st International Conference of Expert Databases*, 271–293.

van den Bercken, J., Dittrich, J.P. & Seeger, B. (2000). XXL: A Prototype for a Library of Query Processing Algorithms. In W. Chen, J.F. Naughton & P.A. Bernstein, eds., *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM Press.

van den Bercken, J., Dittrich, J.P., Kräamer, J., Schäafer, T., Schneider, M. & Seeger, B. (2001). XXL A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01)*.

van Deursen, A., Klint, P. & Visser, J. (2000). Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, **35**, 26–36.

Vassiliadis, P., Vagena, Z., Skiadopoulos, S. & Karayannidis, N. (2000). ARKTOS: A Tool For Data Cleaning and Transformation in Data Warehouse Environments. *IEEE Data Engineering Bulletin*, **23**, 42–47.

W3C (2006). *XQuery 1.0: An XML Query Language. W3C Candidate Recomendation*. World Wide Web Consortium.

Wall, L., Christiansen, T. & Orwant, J. (2000). *Programming Perl*. O'Reilly & Associates, 3rd edn.

Wang, J. (1999). A Survey of Web Caching Schemes for the Internet. *ACM SIGCOMM Computer Communication Review*, **29**, 36–46.

Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *IEEE Computer*, **25**, 38–49.

Wolf, J.L., Yu, P.S., Turek, J. & Dias, D.M. (1993). A Parallel Hash Join Algorithm for Managing Data Skew. *IEEE Transactions on Parallel and Distributed Systems*, **4**, 1355–1371.

Wooster, R.P. & Abrams, M. (1997). Proxy Caching that Estimates Page Load Delays. In *Selected Papers from the Sixth International Conference on World Wide Web*, 977–986, Elsevier Science Publishers Ltd.

Zhou, G., Hull, R. & King, R. (1996). Generating Data Integration Mediators That Use Materialization. *Journal of Intelligent Information Systems*, **6**, 199–221.

Zhou, Y., Philbin, J.F. & Li, K. (2001). The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 Usenix Technical Conference*.

Ziegler, P. & Dittrich, K.R. (2004). Three Decades of Data Integration – All Problems Solved? In R. Jacquart, ed., *Building the Information Society, IFIP 18th World Computer Congress*, 3–12, Kluwer.

Zipf, G.K. (1949). *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Reading, MA.