Optimization of Object Query Languages

Promotiecommssie:

Prof.dr. P. M. G. Apers, promotor
Dr.ir. R. A. de By, referent
Dr. A. N. Wilschut, 2e referent
Dr. H. M. Blanken
Prof.dr.ir. N. J. I. Mars
Prof.dr. M. H. Scholl, Universität Konstanz
Prof.dr. J. Paredaens, Universiteit van Antwerpen

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Steenhagen, Hendrika Janna

Optimization of object query languages / Hendrika Janna
Steenhagen. – [S.l. : s.n.]. – Ill.
Thesis Universiteit Twente Enschede. – With index, ref.
ISBN 90–9008745–1
Subject headings: database systems / query optimization.

OPTIMIZATION OF OBJECT QUERY LANGUAGES

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. Th. J. A. Popma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 19 oktober 1995 te 13.15 uur

door

Hendrika Janna Steenhagen

geboren op 2 juli 1954
te Almelo

Dit proefschrift is goedgekeurd door:
Prof.dr. P. M. G. Apers, promotor

Contents

Summary

The task of a Database Management System (DBMS) is to safely store (usually large amounts of) consistent data, and to provide easy and fast access to these data, either for retrieval or update purposes. The data model of a DBMS lays down the possible structure of the data; to provided easy access to the user, a high-level query language is supported. The implementation of such a high-level query language requires an enormous effort; it is the task of the query optimizer to ensure fast access to the data stored in the database. A DBMS is a complex piece of software that consists of many layers. The three main layers distinguished are (1) the user interface, (2) an intermediate layer that is called the logical algebra, and (3) the bottom layer called the physical algebra, which provides mechanisms to actually access the data stored in files. In query processing, the user query is first mapped into a logical algebra expression, and this expression in turn is mapped into an expression of the physical algebra. Traditionally, query optimization consists of two phases: (1) logical optimization, which is the rewriting of a logical algebra expression into one that (hopefully) can be evaluated with less costs, and (2) cost-based optimization, which concerns the mapping of logical algebra expressions into expressions of the physical algebra. Cost-based optimization is guided by specific database characteristics such as table size (i.e. cardinality and tuple width), the presence of indices, et cetera. Logical rewriting does not make use of such knowledge about the database but instead is guided by heuristics: rules-of-thumb that are expected to have a beneficial effect.

In recent years, database research has concentrated on object-oriented data models, which allow to store highly structured data (and even operations on them). With regard to the data structuring concepts offered, an object-oriented data model can be looked upon as an extension of the nested relational model, which allows to store relations as attribute values. The nested relational model, in turn, is an extension of the relational model, which allows for flat table structure only. With growing complexity of data structuring concepts, the complexity of the accompanying query language grows as well, and thus also the complexity of query processing and optimization. In this thesis, we investigate the first phase of query processing, i.e. the translation of an SQL-like query language into a logical algebra, in the context of an advanced data model supporting complex objects. The context for the work presented here is the (slightly extended) nested relational model. We believe that an efficient implementation of a nested relational query language forms the basis for efficient implementation of query languages for object-oriented data models.

The research topic consists of two parts: (1) to define the logical algebra that is suitable to implement the user query language called OSQL, and (2) to provide the actual translation. We define a logical algebra called ADL, which is an extension of one of the algebras defined for the nested relational model. ADL is a set-oriented language, extended with constructs that allow for explicit iteration over sets, which are necessary because of the presence of set-valued attributes. We believe that as for the relational model, also in advanced DBMSs the logical algebra should be a set-oriented language, because set operators offer many optimization opportunities.

Traditionally, the translation of the user language into the logical algebra has been considered as the task of the parser—considerations with relation to efficiency do not play a role in this phase of query processing. However, we think that query optimization issues should be taken into consideration in every step of the implementation process. We revisit the translation of SQL into relational algebra, and we show that the actual translation of the query language into the logical algebra heavily influences performance, especially when arbitrary language constructs such as universal quantification and disjunction are taken into consideration. Standard translation algorithms often result in inefficient logical algebra expressions, and, moreover, these may be hard to optimize algebraically. We propose to extend relational algebra with some non-standard (join) operators and we make an attempt to provide an improved translation algorithm: we combine translation with optimization. Starting from a select-project-product expression, the main heuristic used in standard relational logical optimization is to push through selections and projections, based on the wish to reduce the size of intermediate results. However, it turns out that the translation SQL into the algebra, as well as logical optimization, should be guided by a more detailed cost model. For example, to decide between optional translation strategies, a cost model that estimates the relative processing load of algebraic operators can be useful.

For the translation of nested OSQL queries into the algebra ADL, we define the nestjoin operator, which is the complex object equivalent of the relational join. We show that, for the translation of OSQL, we can adhere to the same strategy as traditionally followed in the relational context: a transformation into a (nested) product expression, followed by a phase of pushing operators down the operator tree. However, this observation is of theoretical importance mainly—to achieve more efficient algebraic expressions we need a carefully designed translation algorithm, all the more because logical rewriting in an algebra supporting complex objects is even more difficult than in the relational model. We provide a heuristics-based framework for the translation of nested OSQL queries. We present transformation rules and propose an initial rewrite strategy. The rewrite strategy, i.e. the order of rule application, determines the form of the transformation result—we conclude that for an efficient translation of (O)SQL into the algebra, we need a logical cost model that can be used to choose between equivalent algebraic expressions. Appropriate cost models have not been the topic of our study. As lessons learned from our investigations, we present four generally valid transformation rules, and we show that specific results obtained by these rules correspond to possible definitions of more or less well-known algebraic operators.

Samenvatting

De taak van een Database Management Systeem (DBMS) is het veilig opslaan van (gewoonlijk grote hoeveelheden) consistente gegevens, en te voorzien in een gemakkelijke en snelle toegang tot deze gegevens, ofwel om ze op te vragen of om ze te wijzigen. Het gegevensmodel van een DBMS legt de mogelijke structuur van de gegevens vast; om de gebruiker makkelijk toegang te geven wordt een hoog-niveau querytaal ('vraagtaal') ondersteund. De implementatie van zo'n hoog-niveau querytaal vereist een enorme inspanning; het is de taak van de query optimizer een snelle toegang tot de gegevens te verzekeren. Een DBMS is een complex geheel van software dat uit vele lagen bestaat. De drie hoofdlagen die men onderscheidt zijn (1) het gebruikersinterface, (2) een tussenlaag welke de logische algebra genoemd wordt en (3) de fysieke algebra, die voorziet in primitieven die feitelijk toegang geven tot de gegevens die opgeslagen zijn in bestanden. Query processing ('vraag-verwerking') bestaat uit het vertalen van een vraag gesteld door de gebruiker naar een expressie in de logische algebra, die vervolgens wordt vertaald naar een expressie in de fysieke algebra. Gewoonlijk bestaat query-optimalisatie uit twee fasen: (1) logische optimalisatie, het herschrijven van een logische algebra-expressie naar een die (hopelijk) met minder kosten geëvalueerd kan worden en (2) het vertalen van logische algebra expressies naar expressies in de fysieke algebra, gebaseerd op een kostenmodel. Optimalisatie op basis van een kostenmodel wordt gestuurd door specifieke database-karakteristieken zoals tabelgrootte (het aantal records en attributen), de beschikbaarheid van indices, *et cetera*. Logische optimalisatie maakt geen gebruik van zulke kennis betreffende de database, maar wordt daarentegen gestuurd door heuristieken: vuistregels waarvan men verwacht dat ze een gunstig effect hebben.

In de afgelopen jaren heeft het database-onderzoek zich toegespitst op object-georiënteerde gegevensmodellen, die het mogelijk maken gegevens met een complexe structuur (en zelfs operaties op de gegevens) op te slaan. Met betrekking tot de structureringsconcepten die aangeboden worden kan een object-georiënteerd gegevensmodel beschouwd worden als een extensie van het geneste relationele model, dat relaties als attribuutwaarden toestaat. Het geneste relationele model kan op zijn beurt beschouwd worden als een uitbreiding van het relationele model, waarin alleen vlakke tabellen toegestaan zijn. Naarmate de structureringsconcepten complexer worden, groeit mede de complexiteit van de bijbehorende querytaal, en daarmee ook de complexiteit van query-processing en -optimalisatie. In deze dissertatie onderzoeken we de eerste fase van het query-processings traject,

de vertaling van een SQL-achtige querytaal naar een logische algebra, in de context van een geavanceerd gegevensmodel dat complexe objecten toestaat. Het werk dat hier gepresenteerd wordt kan geplaatst worden binnen de context van het geneste relationele model. Wij zijn van mening dat een effciënte implementatie van een querytaal voor het geneste relationele model de basis vormt voor een effciënte implementatie van querytalen voor object-georiënteerde gegevensmodellen.

Het onderzoek bestaat uit twee delen: (1) de definitie van een logische algebra die geschikt is voor de implementatie van de gebruikerstaal (genaamd OSQL) en (2) het maken van de feitelijke vertaling. We definiëren een logische algebra genaamd ADL, een extensie van één van de algebra's die gedefiniëerd zijn voor het geneste relationele model. ADL is een verzamelings-georiënteerde taal, uitgebreid met taalconstructies die expliciete iteratie over verzamelingen toestaan. Deze laatste zijn nodig omdat attributen verzamelingswaardig kunnen zijn. We zijn van mening dat, zoals in relationele systemen, ook in geavanceerde systemen de logische algebra zoveel mogelijk verzamelings-georiënteerd dient te zijn, omdat verzamelingsoperatoren vele mogelijkheden tot optimalisatie bieden.

De vertaling van de gebruikerstaal naar de logische algebra wordt traditioneel beschouwd als de taak van het parseeralgorithme; overwegingen met betrekking tot performance (prestatie) spelen geen rol in deze fase van query-processing. Wij denken echter dat optimalisatie een rol dient te spelen in elke stap van het implementatieproces. We besteden opnieuw aandacht aan de vertaling van SQL naar de relationele algebra, en we laten zien dat de feitelijke vertaling van de querytaal naar de logische algebra een grote invloed heeft op de uiteindelijke performance, vooral wanneer ook willekeurige taalconstructies zoals universele kwantificatie en disjunctie in de beschouwing betrokken worden. Standaard vertaalalgorithmen resulteren vaak in inefficiënte logische algebra-expressies, die daarbij vaak moeilijk algebraïsch te optimaliseren zijn. We stellen voor relationele algebra uit te breiden met enkele niet-standaard (join-) operatoren en we doen een poging te komen tot een verbeterd vertaalalgorithme: we combineren vertaling met optimalisatie. Uitgaande van een select-project-produktexpressie is de voornaamste heuristiek die gebruikt wordt in traditionele relationele optimalisatie het doorduwen van selecties en projecties. Deze heuristiek is gebaseerd op de wens de grootte van tussenresultaten zo klein mogelijk te houden. Het blijkt echter dat zowel de vertaling van SQL naar de algebra als logische optimalisatie gestuurd zou moeten worden door een meer gedetailleerd kostenmodel. Om te kunnen kiezen uit verschillende vertaalstrategieën zou een model dat een schatting maakt van de relatieve kosten van de diverse operatoren bijvoorbeeld nuttig kunnen zijn.

Voor de vertaling van geneste OSQL queries naar de algebra definiëren we de nestjoin-operator, de equivalent van de relationele join-operator voor modellen met complexe objecten. We tonen aan dat we voor de vertaling van OSQL expressies dezelfde strategie kunnen hanteren als gebruikt wordt in het relationele model: een transformatie naar expressies bestaande uit (geneste) produkt-expressies. Deze observatie is echter hoofdzakelijk van theoretisch belang; teneinde meer efficiënte algebraïsche expressies te verkrijgen hebben we een zorgvuldig ontworpen vertaalalgorithme nodig, temeer daar logische herschrijving van een taal met complexe objecten nog moeilijker is dan het herschrijven van relationele algebra-expressies. We presenteren een raamwerk voor de vertaling van geneste OSQL

queries, gebaseerd op heuristieken. We geven transformatieregels en stellen een initiële herschrijfstrategie voor. De herschrijfstrategie (de volgorde van regeltoepassing) bepaalt de vorm van het resultaat van de transformatie. We concluderen dat we voor een efficiënte vertaling van (O)SQL naar de algebra behoefte hebben aan een logisch kostenmodel dat gebruikt kan worden om te kiezen tussen verschillende algebraïsche expressies. Geschikte kostenmodellen zijn niet aan de orde geweest in ons onderzoek. Als weerslag van ons onderzoek presenteren we vier algemeen geldige transformatieregels, en we tonen aan dat resultaten die verkregen worden middels toepassing van deze regels corresponderen met mogelijke definities van meer of minder bekende algebraïsche operatoren.

Acknowledgements

In the course of doing my Ph.D.-work, I had the help of many people. I want to express my thanks to all of them. In particular, I want to name the following people. First of all, I thank my promotor Peter Apers, who gave me the opportunity to do scientific research, and who has been a friendly, non-restrictive, and patient supervisor. I thank Henk Blanken and Rolf de By, who took part in daily supervision and thus contributed much to my work. I am honoured that Marc Scholl, Jan Paredaens, Koos Mars, and Annita Wilschut kindly agreed to be members of the committee.

Gratefully, I acknowledge the anonymous referees, whose comments helped to improve the quality of my work.

My colleagues are thanked for their pleasant cooperation and friendliness. Especially, my thanks go to Rolf de By, who has been a truly helpful friend, and to Sandra Westhoff, with whom I spent many breaks for a coffee and a smoke.

I thank my family for their support and encouragement. And, above all, I thank the friends who stood close by me during these years, and in years past.

Chapter 1

Introduction

A Database Management System (DBMS) is a complex piece of software that consists of many layers. A first distinction is that between user level and internal level. The user level may be further distinguished into the conceptual level, describing the database as a whole, and the external level, describing only parts of the database that a specific user group is interested in. The internal level falls apart into the logical level and the system-specific physical level. The physical level lays down the details of data storage and access paths; the logical level serves as the intermediate between the user and the physical level. On each level, a specific language is supported, consisting of a type system and a set of operations. Operations on the data involve update and retrieval operations; in this thesis we only consider retrieval operations (queries). Mappings between the different languages lay down how to translate types and operations from one level into the other.

Query processing is the sequence of actions that takes as input a query formulated in the user language and delivers as result the data asked for. Query processing involves query transformation and query execution. *Query transformation* is the mapping of queries and query results back and forth through the different levels of the DBMS. *Query execution* is the actual data retrieval according to some access plan, i.e. a sequence of operations in the physical access language. An important task in query processing is query optimization. Usually, user languages are high-level, declarative languages allowing to state *what* data should be retrieved, not *how* to retrieve them. For each user query, many different execution plans exist, each having its own associated costs. The task of query optimization ideally is to find the best execution plan, i.e. the execution plan that costs the least, according to some performance measure. Usually, one has to accept just feasible execution plans, because the number of semantically equivalent plans is to large to allow for enumerative search.

Relational DBMSs have become a standard tool for business data processing. However, in recent years, many new application domains have emerged, putting new demands on data modelling and processing. The development of the nested relational model can be considered as the first step towards meeting these new demands, allowing the data to have a structure more complex than the flat table structure by supporting relation-valued

attributes. Currently, much research is done on object-oriented data models, which support concepts like object identity, complex objects, inheritance, etc.

The subject of this thesis concerns the first part of the query processing route: the translation of an SQL-like query language for advanced data models into an algebra supporting complex objects. To be able to clarify the research topic of this thesis, we discuss the concept of query processing in some more detail.

Query processing in relational systems

Processing a database query consists of the following steps: query parsing, query validation, view resolution, query optimization, plan compilation, and query execution [Grae93].

The user query, written in some high-level declarative query language, for example SQL, or a deductive language like Datalog, is parsed, and translated into an internal representation. Usually, relational algebra is chosen as the intermediate language; other intermediate representations are possible, though. As in [Grae93], we call the intermediate language the logical algebra. In query validation (typing), it is checked that all attribute and relation names are valid and semantically meaningful. During view resolution, and also because of integrity constraint maintenance, the user query may be expanded beyond its original form. Plan compilation is the translation of the logical algebra query into an access plan. The access language, or the physical algebra, is a system-specific language, closely corresponding to the logical algebra. For each logical algebra operator, one or more physical algebra operators exist that efficiently implement the logical operator.[1]

Query optimization is the process of finding the best, or, rather, a reasonably efficient execution plan, i.e. of translating the logical algebra expression into the physical algebra in the best possible way. Two types of query optimization are distinguished: heuristic and cost-based. Heuristic (or algebraic) optimization is the rewriting of expressions independent of some system-specific cost model. Heuristic optimization consists of the following steps [JaKo84]: standardization (providing a starting point for optimization), simplification (the removal of redundancy), and amelioration (rewriting into more efficient expressions, for example pushing down selections). Cost-based (or systematic) optimization is the rewriting or translation of expressions guided by some cost model that reflects system-specific knowledge about operator implementations, the presence of indices, relation cardinalities, the selectivity of predicates, etc. Usually, cost-based optimization is guided by heuristics as well; the complexity of the search space prohibits an exhaustive search for the best execution plan.

Traditionally, the term query optimization is reserved for the two phases of the rewriting of logical algebra expressions (algebraic optimization) and the generation of access plans. However, thinking about query processing as the *efficient implementation* of a database query, the mind broadens a little. Given the user language, the efficiency of data retrieval is determined by the choice of the logical and physical language, the two being closely related, and the quality of the mappings between the languages.

[1] This correspondence is not exactly one to many. One physical operator may implement a sequence of logical operators, and also the physical algebra may contain operators not corresponding to any one of the logical operators, e.g. the sort operator.

In the relational context, often relational algebra is chosen as the intermediate representation form. [Codd72] provides a (standard) algorithm for translating relational calculus into relational algebra; [CeGo85] gives an algorithm for translating SQL into relational algebra directly. The algebraic expressions that are the result of this translation may be very inefficient. (Algebraic) optimization is expected to reduce the inefficiency introduced in the translation. Not restricting ourselves to project-select-join queries, but considering algebraic expressions that contain arbitrary operators such as set operators and division, we are convinced that this is quite a hard task. In our opinion, there are two possibilities for improvement. First, relational algebra may be extended with non-standard operators. Examples of non-standard operators are the semi- and antijoin operator, efficiently implementing some types of queries involving (negated) quantification. Second, the translation algorithm may be improved; this observation has been made elsewhere too [Bry89, Naka90]. Algebraic expressions that contain arbitrary operators are hard to optimize. Finding the proper rewrite rules, and to control the sequence of rule application is difficult in a pure algebraic context. Calculus-like expressions are more succinct than algebraic expressions, in which, in a sense, information concerning the original query structure is scattered throughout the expression. Trying to find a good translation from calculus into the algebra right away does seem a better approach than trying to rewrite inefficient algebraic expressions afterwards [Naka90].

Query processing in nested relational and object-oriented systems

Research into query processing in nested relational and OO systems has moved into many directions. With respect to the nested relational model, much research has been done on its theoretical aspects. For example, the expressive power of nested relational algebras has been studied in depth, e.g. [PaGu92]. Also, the implementation of nested algebras has been a research topic in the past [Scholl et al. 89, Schek et al. 90]. However, no commercial nested relational DBMS has ever been brought on the marketplace. One of the missing links is a translation of an SQL-like query language for the nested relational model into some nested relational algebra; to our knowledge, no such translation has ever been published.

Also for OO data models, no such translation has been made yet. Because OO data models offer the possibility to store user-defined operations, the need for an ad-hoc query language was not immediately recognized. However, proposals for a declarative query language have been made in recent years, e.g. [BaCD92, BaBZ93]. Currently, the ODMG group is working on the standards for object-oriented database management systems [Catt93]. The proposal includes a description of an object query language named OQL, which is an SQL-like language. Only recently, [ClMo93] addressed the problem of optimization of nested SQL-like queries in object bases. Work on algebras for object-oriented data models has been published for example in [ShZd89, Vand93]. Specific features of OO data models offer many additional opportunities for optimization. Object identity can be employed to speed up join processing [ShCa90], the presence of inheritance hierarchies and path expressions allows to design new index structures [Bert93], et cetera.

Goal of this thesis

As already stated, in this thesis we investigate the mapping of an SQL-like query language for advanced data models into an algebra; the emphasis lies on the treatment of complex objects. Mapping one language into the other involves a mapping of types and a mapping of expressions. We restrict ourselves in that we only consider the mapping of expressions; mapping of types (storage design, or clustering) is an independent research area that does not fall within the scope of this thesis. The work is motivated by the observation that OO databases need a high-level, declarative interface that must be implemented efficiently. A straightforward mapping of an extended SQL language into for example C++ code does not suffice; such an approach defeats the lessons learned in the past from research into the implementation of the relational and extended relational model. Our basic assumptions are described below.

Our first assumption is that it is useful to search for analogies between the implementation of the relational model and that of advanced data models. Much research effort has been put into the implementation of the relational model; the knowledge gained should be used whenever possible. To give an example, we believe that the logical algebra should be a set-oriented language, just like relational algebra. Though it seems that navigation has been considered as the prevailing method to access OO databases in the past, recently more attention has been paid to set-oriented access methods, like pointer-based joins [ShCa90].

Our second assumption is that query optimization should not be considered as a separate step in query processing. In our opinion, optimization should play a role in each phase of the implementation process. This second assumption has two aspects. First, we feel that the logical algebra should be designed as a true intermediate language. The function of the intermediate language is to bridge the gap between the efficient access algorithms expressed by the physical algebra and the declarative constructs present in the user language, in the best possible way. Consequently, logical algebras usually are redundant languages. On the one hand, the logical algebra must have at least the expressive power as the user language. On the other hand, the logical algebra must contain operators that correspond closely to the operators (i.e., the access algorithms) offered by the physical algebra. For example, the join can be expressed by means of a selection on a Cartesian product, however, the very reason for adding the join to relational algebra is that the operator sequence of selection and product can be implemented much more efficiently. Second, we claim that the actual algorithm for the translation of user language expressions into the logical algebra expressions has a strong influence on performance. Generally, user language expressions can be translated into the algebra in many ways. In the relational, and also extended relational context, translation algorithms, being relatively simple (cf. the reduction algorithm of Codd [Codd72]), often result in algebraic expressions that can be considered as the Most Costly Normal Form (MCNF, [KeMo93]). With [Naka90], we feel that putting more effort in the translation algorithm pays off in achieving better results, i.e. in achieving algebraic expressions that are more efficient and less difficult to optimize than the expressions that are the result of standard translation algorithms.

Outline of this thesis

In Chapter 2, we give an overview of the work that has been done on query processing in relational, nested relational, and object-oriented systems. The emphasis lies on logical (heuristic) query optimization. In Chapter 3, we give a formal definition of the language used in this thesis. We define a language that consists of calculus-like constructs as well as pure algebraic operators. The calculus-like part of the language can be looked upon as a simplified version of an SQL-like language for the nested relational model. The algebra part of the language closely resembles nested relational algebra as defined in [ScSc86]. In the next chapter, Chapter 4, we re-examine the translation of relational calculus into relational algebra. The reasons for this re-examination are twofold. First, we want to support our claim that the translation algorithm strongly influences performance, and second, we want to show that the algebraic language must be tailored towards performance, i.e. it must contain operators that are both useful and efficient. Chapter 5 and Chapter 6 are both reprints from articles published elsewhere. The first paper was presented at the EDBT '94 conference, and treats some of the theoretical difficulties in translating nested calculus expressions in complex object models into algebraic expressions. The second paper, presented at the VLDB conference of 1994, describes a general approach to translate and optimize nested queries in complex object models. Both papers can be considered as an introduction on the following chapter, Chapter 7, in which the actual translation of the calculus into the algebra is discussed. The chapter is presented in the form of an article as well; it is accepted for the EDBT conference of 1996. In Chapter 8, we present an overview of transformation techniques and also briefly pay attention to the implementation of algebraic operators. In Chapter 9, we present our conclusions and give directions for future research.

References

[BaCD92] Bancilhon, F., S. Cluet, and C. Delobel, *A Query Language for O_2*, in: *Building an Object-Oriented Database System—The Story of O_2*, eds. F. Bancilhon, C. Delobel, and P. Kanelakis, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[BaBZ93] Balsters, H., R.A. de By, and R. Zicari, "Typed Sets as a Basis for Object-Oriented Database Schemas," *Proceedings ECOOP*, Kaiserslautern, 1993.

[Bert93] Bertino, E., "A Survey of Indexing Techniques for Object-Oriented Databases," in: *Query Processing for Advanced Database Systems*, eds. J.-C. Freytag, D. Maier, and G. Vossen, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Bry89] Bry, F., "Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited," *Proceedings ACM SIGMOD*, Portland, Oregon, June 1989, pp. 193–204.

[Catt93] R.G.G. Cattell, ed., *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[CeGo85] Ceri, S. and G. Gottlob, "Translating SQL into Relational Algebra: Optimization, Semantics, and equivalence of SQL Queries," *IEEE Transactions on Software Engineering*, Vol. SE–11, No. 4, April 1985.

[ClMo93] Cluet, S. and G. Moerkotte, "Nested Queries in Object Bases," *Proceedings Fourth International Workshop on Database Programming languages*, New York, Sept. 1993.

[Codd72] Codd, E.F., "Relational Completeness of Data Base Sublanguages," in: *Data Base Systems*, ed. R. Rustin, Prentice Hall, 1972.

[Grae93] Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2), June 1993, pp. 73–170.

[JaKo84] Jarke, M. and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, 16(2), June 1984, pp. 111–152.

[KeMo93] Kemper, A. and G. Moerkotte, "Query Optimization in Object Bases : Exploiting Relational Techniques," in: *Query Processing for Advanced Database Systems*, eds. J.-C. Freytag, D. Maier, and G. Vossen, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Naka90] Nakano, R., "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS*, 15(4), December 1990, pp. 518–557.

[PaGu92] Paredaens, J. and D. van Gucht, "Converting Nested Algebra Expressions into Flat Algebra Expressions," *ACM TODS*, 17(1), June 1992.

[Schek et al. 90] Schek, H.-J., H.-B. Paul, M.H. Scholl, and G. Weikum, "The DASDBS Project: Objectives, Experiences, and Future Prospects," *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990, pp. 25–43.

[ScSc86] Schek, H.-J., and M.H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems* , 11(2), 1986, pp. 137–147.

[Scholl et al. 89] Scholl, M., S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and D. Verroust, "VERSO: A Database Machine Based on Nested Relations," in: *Nested Relations and Complex Objects in Databases*, eds. S. Abiteboul, P.C. Fischer, H.-J. Schek, LNCS 361, Springer-Verlag, 1989, pp. 27–49.

[ShZd89] Shaw, G.M., and Zdonik, S.B., "Object-Oriented Queries: Equivalence and Optimization," *Proceedings 1st International Conference on Deductive and Object-Oriented Databases*, Kyoto, December 1989.

[ShCa90] Shekita, E.J. and M.J. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proceedings ACM SIGMOD*, Atlantic City, May 1990, pp. 300–311.

[Vand93] S.L. Vandenberg, "Algebras for Object-Oriented Query Languages," Ph.D. Thesis, University of Wisconsin-Madison, 1993.

Chapter 2

Query optimization

In this chapter, we give a more precise description of the problem addressed in this thesis, i.e., the translation of an SQL-like query language for object-oriented data models into an algebra supporting complex objects, to be defined yet. This research topic cannot be addressed in isolation; it is part of the larger problem of how to implement a database query language. Therefore, we first present a general view on the implementation of database query languages in Section 2.1. Having set the stage, we present an overview of query optimization issues as found in the literature. We discuss query optimization aspects as present in relational, nested relational, and object-oriented database systems in Section 2.2, 2.3, and 2.4, respectively. The overview is not complete; emphasis is placed on issues that we consider important with respect to the performance of query evaluation. Next, in Section 2.5, we come to a refinement of the problem as stated above. The chapter ends with a summary in Section 2.6.

2.1 Implementation of database query languages

In this section, we give a general view of database system architecture. Database management systems are complex pieces of software—layering and modularization are necessary means to describe and build such systems.

Most end-user database query languages are high-level, declarative languages, based on set theory and first-order predicate calculus. The basic construct of a block-structured query language such as SQL is the set comprehension expression $\{f(x) \mid x \in X \wedge p(x)\}$. Declarative languages need much optimization. A straightforward, naive implementation, i.e., a simple declarative language interpretor that does not make use of special implementation techniques may result in time and/or space consuming programs. At the same time, declarative languages offer many opportunities for optimization. In declarative languages the result of a computation is described, not how it is to be computed. Declarative languages therefore leave the language implementor a great deal of freedom in choosing the algorithms that will efficiently compute the desired result.

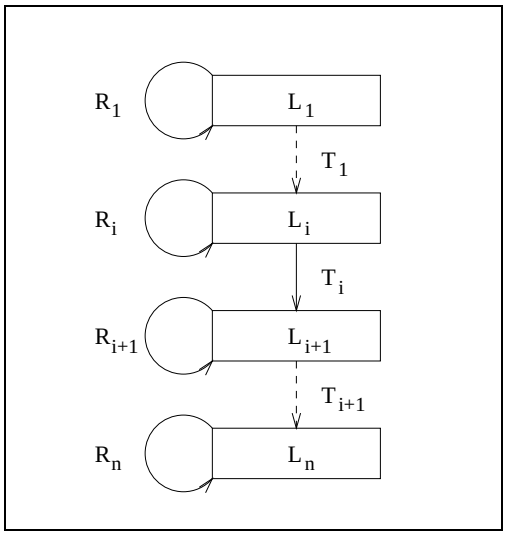


Figure 2.1: General view on the implementation process

In the implementation of a general purpose programming language, the bottom layer is the machine-dependent assembler language. In database management systems, the bottom layer is the file access language. The file access language, as far as retrieval is concerned, has three basic primitives: *open_file*, *close_file*, and *next_item*, where items usually are records. With these primitives, a collection of efficient file access algorithms is built, which constitutes the so-called physical access language, or physical algebra. Standard techniques employed to achieve good performance are sorting, hashing, and the use of additional access structures like indices. In addition, parallel query processing techniques can be used to improve performance. The physical algebra is a set-oriented language. The standard techniques mentioned above can be considered as set preprocessing techniques, that operate on sets (tables) as a whole, and enable to retrieve only tuples that are really needed, not entire sets.

The distance between a declarative query language and the physical algebra is large, so implementation is done in a step-by-step manner. In Figure 2.1, the process of implementing a database query language—or a general purpose programming language, for that matter—is depicted.

The transformation of expressions of one language into those of another is called translation (T); applying transformations within one and the same language is called rewriting (R). Rewriting expressions in the language L_1, the source language, is called preprocessing; post-processing is the rewriting of expressions in the target language L_n. Preprocessing in many cases involves the reduction of a language to a predefined kernel; postprocessing is usually applied for simplification purposes. In the implementation process, several intermediate languages L_i may be used. Translation consists of two parts: to provide a mapping of data structures and a mapping of expressions. In each phase of the implemen-

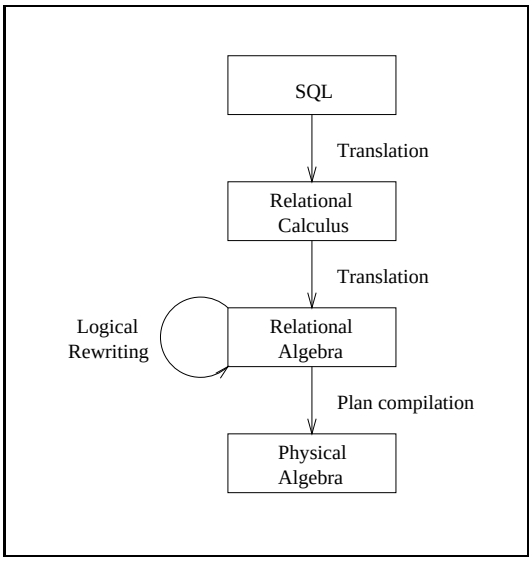


Figure 2.2: Implementation of SQL

tation process, measures can be taken to improve the performance of query evaluation; this is called optimization. Optimization may be based either on heuristics, rules of thumb that are expected to reduce the costs of query evaluation in general, or on a cost model that provides more or less accurate estimates of the costs of query execution. Optimization based on heuristics is also called logical optimization; cost-based optimization is also called systematic optimization.

In database systems, we distinguish between the user, the logical, and the physical level. The top level is the end-user query language to be implemented, the second, intermediate language is a language in which logical transformations take place, and the third, bottom level is the physical algebra that implements system-specific, efficient query processing algorithms. For example, a relational DBMS may have an architecture as depicted in Figure 2.2. The end-user query language is SQL and the intermediate language is (some extended form of) relational algebra. SQL may be translated into relational algebra directly, or indirectly by using relational calculus as a second intermediate language. The bottom layer is the physical algebra, i.e., a collection of operators that implement relational algebra. For each of the relational operators, a number of physical operators or execution methods exists. Which one to use when depends on specific database characteristics or physical properties such as sort order, the presence of indices, etc.

As already mentioned, implementation involves a mapping of data structures, and a mapping of expressions. *Physical database design* concerns the actual representation of the data in the database. A conceptual schema, expressed in the type system of the end-user query language, can in several ways be mapped to a physical database schema, expressed in the type system of the physical algebra. Given the so-called work load, i.e., a specific

collection of user queries, the goal of physical database design is to achieve a near optimal mapping of data structures including placement of additional data structures like indices.

Once the physical schema and the mapping of the conceptual schema into the physical schema have been established, the remaining task is the mapping of the end-user queries into the operations offered by the physical algebra. *Query processing*, i.e., the mapping of expressions, roughly consists of the following steps: query validation (or typing), view resolution, translation into the intermediate language, logical optimization, plan compilation, and query execution [Grae93].

After query parsing, query validation, and view resolution, the user query is translated into the intermediate language. Depending on end-user and intermediate language paradigms and the operations supported by both languages, translation may involve a simple syntactic transformation, or some more complex algorithm. In this translation phase, usually little attention is paid to performance issues. At the intermediate level, the goal is to transform expressions into semantically equivalent expressions that have better performance. Possible transformations are laid down by a collection of equivalence rules. Logical optimization is guided by heuristics; the cost model is simple. The physical level supports a number of system-specific query processing algorithms. Intermediate language expressions are translated into the physical algebra. The translation is guided by physical database characteristics; a complex cost model is used.

The above description of a possible database system architecture is an idealized description. Many systems do not make a clear distinction between user, logical, and physical level, in the sense that each level entails a clearly-defined language interface, and that well-defined algorithms exist that precisely lay down how to transform expressions from one level into the other. Therefore, often the different query processing steps are not bound to specific language levels as much as described above. For example, logical optimization can take place in the user language itself. Examples are the transformation of Datalog programs (e.g. the magic sets transformation, [BMSU86]), the rewriting of nested SQL queries of [Kim82], and the loop optimization of [LiDe92].

Usually, only in the phases of logical optimization and plan compilation (the mapping of the logical into the physical algebra) considerations with respect to performance do play a role. Logical optimization, the rewriting of intermediate language expressions, is based on heuristics, representing an implicit cost model. Plan compilation, the translation of a logical expression into a physical algebra expression, is based on an explicit cost model that incorporates specific database characteristics considered important. However, given our general view of the implementation of database query languages, one may ask whether performance considerations should be restricted to the two phases mentioned. Having to deal with a tuple-oriented user language and a set-oriented physical access language, and assuming a three-level architecture, we have to make a decision about the following three issues.

- What language to choose as intermediate language. First we have to decide about the appropriate language paradigm. For example, both an algebra and a calculus can be used as intermediate language [JaKo84]. The second question is which operations should be part of the intermediate language. The answer to these questions

depends on the functionality of the user language as well as that of the physical algebra. The intermediate language should have at least the same expressive power as the user language, but, more importantly, the intermediate language has to be such that it provides the best opportunity for mapping the user language to the physical algebra in the best possible way. In other words, the functionality of the intermediate language should allow to fully exploit the possibilities for efficient evaluation offered by the physical algebra.

- Where to use heuristics to achieve better performance. For example, it can be asked whether heuristic optimization should be restricted to the rewriting of expressions on the intermediate language level, or that heuristic rules should play a role in the translation process as well.
- Where to use an explicit cost model to improve performance. Usually, logical optimization and plan compilation are separate steps in query processing. Heuristics are used for the rewriting of intermediate language expressions and an explicit cost model guides the translation into the physical algebra. However, in [GrMc93], in which the the Volcano optimizer generator is discussed, it is suggested that rewriting of logical algebra expressions might be based on a cost model as well; the optimizer implementor can decide whether or not to do so.

In the remainder of this chapter, we give an overview of optimization in the relational, extended relational, and object-oriented context. The overview is not meant to be an exhaustive enumeration of work done in the field of query optimization. We attempt to provide a general background on the work done in relational query optimization, and then try to decide in what respect optimization in new data models ((X)NF^2 and object-oriented models) differs from relational optimization.

2.2 Implementation of relational query languages

In the survey paper of [JaKo84], a general framework for query optimization in relational systems is given. The following steps are distinguished:

1. The choice of a suitable internal representation: a language that is intermediary between the end-user query language and the operators of the low-level access language. Intermediate representation forms that have been used are relational calculus, relational algebra, query graphs, and tableaux.
2. Logical optimization: the transformation of queries into equivalent ones that can be evaluated more efficiently. The phases distinguished in logical optimization are (1) standardization, the rewriting of expressions into some canonical form, (2) simplification, to remove redundancy, and (3) amelioration, the rewriting of the simplified canonical form into the desired form.
3. The generation of candidate access plans: the mapping of the operations of the intermediate language into the operations of the physical algebra.

4. The choice for and execution of the cheapest access plan. The final task of the query optimizer is to compute the cost for each access plan and the choice for and execution of the cheapest.

Below, we discuss each of these steps in some more detail.

2.2.1 Choice of intermediate language

A key issue in the implementation c.q. optimization of a query language is the choice of intermediate language level(s). An appropriate internal representation, according to [JaKo84], is one that (1) is powerful enough to represent a large class of queries and (2) provides a well-defined basis for query representation. It goes without saying that the language has to be powerful enough to implement the user language. However, it is not immediately clear what type of language provides the best (if any) basis for query representation.

In [JaKo84], relational calculus is regarded as a better starting point for query optimization than relational algebra since "it provides an optimizer only with the basic properties of the query; optimization purposes may become hidden in a particular sequence of algebra operators (p. 118)." Below, we discuss the two language paradigms in some more detail.

Calculus versus algebra

The important distinction between relational calculus and relational algebra is that between tuple- and set-orientation. Tuple-oriented, or tuple-at-a-time languages support the concept of tuple variables, thus allowing for nesting of expressions. For example, in a calculus-like language, we may write:

$$\sigma[x : \exists y \in Y \bullet p(x, y)](X)$$

The expression above, a selection σ with a predicate that is an existential quantification $\exists$, can be regarded as a nested-loop expression. For each tuple $x \in X$, the correlated subquery $\exists y \in Y \bullet p(x, y)$ is evaluated. Set-oriented languages do not support tuple variables and therefore nesting of expressions is not possible.

SQL, relational calculus, and also logical query languages are examples of tuple-oriented languages. As remarked in [Date90], SQL is merely a calculus-based language, but it also supports algebraic operators like set (comparison) operators. Relational algebra is a purely algebraic, set-oriented language. The algebra of [RoKS88], for the nested relational model, also is an example of a set-oriented language. The nested relational algebra of [ScSc86] provides set- as well as tuple-oriented constructs.

In a set-oriented language, expressions are context free, i.e., the operators of the language embody concise, well-defined execution steps that are mutually independent. In a tuple-oriented language, expressions may occur nested within others, and variables from a higher level may occur free in lower level expressions. To further illustrate the differences between calculus and algebra, consider the following (equivalent) expressions:

$$\begin{array}{ll} \text{(calculus)} & \sigma[x : \nexists y \in Y \bullet p(x,y)](X) \\ \text{(algebra)} & X - \pi_X(X \bowtie_{x,y:p(x,y)} Y) \end{array}$$

The calculus expression is a selection σ with a predicate that is a negated existential quantifier. The algebraic expression consists of a join $\bowtie$, followed by a projection π, the result of which is the right operand of the set difference operator $-$. We note the following differences between the two formalisms:

Join processing order From an operational point of view, the calculus expression may be regarded as a nested-loop expression. In this nested-loop expression, table X necessarily is the outer loop operand—for each tuple $x \in X$ the predicate is evaluated. In contrast, the algebraic expression does not fix the join order in any way. Because the join operator is commutative and associative, we may choose to evaluate either $X \bowtie Y$, or $Y \bowtie X$. Physical algebra operators that implement the join usually are not commutative, which means that choosing either X or Y as left join operand may heavily influence performance.

Intermediate results In calculus-like languages, intermediate results are invisible. In algebraic languages, each operator creates—at least, conceptually—an intermediate result. The presence of intermediate results offers additional opportunities for optimization. For example, in a join operator sequence, each join result can be sorted on the attributes that are needed in a subsequent join operation.

In spite of today's main memory sizes, intermediate results often must be written to disk and read again later on. The creation of temporary files can be prevented for example by means of pipelining (data-driven execution) or the use of iterators (demand-driven execution) [Grae93]. The cost of pipelining involves operating system scheduling and interprocess communication. Using iterators, data items are passed from one operator to the next by means of procedure calls. Each time an operator needs an input item, it calls its input operator(s) to produce one.

Extensibility Algebraic languages are extensible. Starting off with a basic set of algebraic operators, new operators can be introduced whenever the need arises. For example, it is easy to introduce the semi- and antijoin operator in standard relational algebra. The above expressions are equivalent to the antijoin expression:

$$X \underset{x,y:p(x,y)}{\triangleright} Y$$

which can be evaluated much more efficiently than the algebraic join expression above. Calculus-based languages are rather fixed, and not easily extensible.

We remark that the notions of tuple- and set-oriented query processing appear under many different names in the literature: depth- versus breadth-first query processing [KeGM91], top-down versus bottom-up query processing (in the context of logical languages, [CeGT90]), nested-loop versus sort-domain data traversal [KKWD88], object- versus file-server (with page-server as intermediate concept) [DFMV90], etc.

In [JaKo84], query graphs and also tableaux are classified as possible internal representation forms. In our opinion, query graphs are not so much a different internal representation form, but rather a graphical representation of any language whatsoever, as opposed to a pure syntactical representation. Both relational algebra and relational calculus for example may have a graph-based representation. Tableaux, and also for example the hypergraph representation described in [Ullm89], are better considered as optimization techniques, aimed at investigating specific query properties.

2.2.2 Logical optimization

Logical optimization is defined as the heuristics-based rewriting of expressions in the intermediate representation form chosen. Expressions are rewritten into expressions that have better performance. Even for a moderately complex language like relational algebra, the number of equivalent candidate access plans, i.e., physical algebra expressions for a complex query is too large to handle. Instead of exhaustively enumerating candidate access plans and choosing the cheapest among them directly, the search space is limited in advance by means of logical rewriting. The output of the phase of logical optimization then is the starting point for a next phase of cost-based, or systematic optimization. Logical optimization is guided by so-called heuristic rewrite rules: rules that are assumed to be generally beneficial.

As mentioned before, the phases distinguished in logical optimization by [JaKo84] are standardization, simplification, and amelioration.

Standardization

Transformation into a canonical form is the starting point of logical query optimization. In calculus-based languages, most often the query is first transformed into Disjunctive or Conjunctive Prenex Normal Form (DPNF or CPNF). In algebraic rewriting, the canonical form typically is an expression involving a sequence of (conjunctive) selections and projections on the Cartesian product of the base relations involved in the query. Transformation into canonical form facilitates the rewrite process and offers additional opportunities for optimization.

Both a Cartesian product expression and an expression in PNF are highly inefficient. A Cartesian product expression can be thought to ‘materialize’ the nested-loop computation expressed by an equivalent calculus expression in PNF. In relational optimization, pushing through selections is considered one of the most important heuristic rules, however, the importance of the rule is mainly due to the method of translation c.q. optimization.

Simplification

The goal of simplification is to avoid unnecessary computations. Simplification may involve the application of idempotency rules, (run-time) application of rules that concern empty sets, and removal of common subexpressions. Other simplification techniques mentioned in [JaKo84] are constant propagation (by applying the rule of transitivity), the incor-

poration of integrity constraints (semantic query optimization), and the detection of predicate unsatisfiability.

Amelioration

The phase of amelioration aims at the reduction of the number, the size (the number of attributes), and the cardinality (the number of tuples) of intermediate results. The number of intermediate results is determined by the number of algebraic operators present in the expression. Heuristics applied are:

- The combination of operations, for example rewriting a cascade of projections into one.
- To perform selections as early possible (pushing through selections).
- To perform projections as early as possible (pushing through projections).
- To perform operations that deliver the smallest intermediate result first, for example executing the most restrictive join before others.

A simple algebraic rewrite algorithm based on heuristics can be found in [Ullm89]. Another example of a heuristic optimization algorithm is the Wong-Youssefi algorithm for the optimization of QUEL select-project-join queries [WoYo76]. In [Ullm89], a description of this algorithm, based on the hypergraph representation, is given.

2.2.3 Generation and choice of access plans

The generation of candidate access plans and choice of the cheapest may be called physical optimization, as opposed to the preceding phase of logical optimization. The generation of an access plan involves mapping intermediate language expressions into expressions of the physical access language. In the survey paper of [Grae93], an overview of query processing algorithms for relational DBMSs is given. Physical optimization is cost-based. Cost-based or systematic optimization is based on knowledge of physical database characteristics: statistical information, the presence of indices, the storage structures that have been chosen, etc. An example of cost-based optimization is the determination of the best join order, together with the best join method for each join by means of combinatorial optimization techniques [SwGu88].

2.2.4 Discussion

Despite all the work done on relational query optimization, the performance of relational query processing can still be improved. Improvement can be achieved by (1) extending relational algebra with non-standard relational algebra operators such as the semijoin, antijoin, et cetera, and (2) by improving the algorithm for the translation of the user language into the algebra.

Standard algorithms for translating relational user languages into relational algebra often result in expressions that are unnecessarily inefficient. We give a simple example. Consider the SQL expression:

```
SELECT *
FROM X
WHERE NOT EXISTS (SELECT * FROM Y WHERE X.a=Y.a)
```

The SQL expression given above is in our formalism denoted as:

$$\sigma[x : \nexists y \in Y \bullet x.a = y.a](X)$$

and this expression can be mapped to the antijoin expression:

$$X \underset{x,y:x.a=y.a}{\triangleright} Y$$

Assume that each SQL-construct has a naive execution plan, then naive execution of the antijoin operation by means of nested-loop processing has performance equivalent to naive execution of the SQL expression. Performance of the antijoin can be improved by the use of techniques such as sorting and hashing. Translation of the SQL expression into a standard relational algebra expression:

$$(X \underset{x,y:x.a\neq y.a}{\bowtie} Y) \div Y$$

in accordance to the reduction algorithm of [Codd72], or, equivalently, into:

$$X - \pi_X(X \underset{x,y:x.a=y.a}{\bowtie} Y)$$

as is done in [CeGo85], results in expressions that have performance most probably worse than that of naive execution, regardless of the implementation of the two algebraic expressions. Unless we have at our disposal logical equivalence rules that can be applied to transform the join expressions into the antijoin expression, translation into the logical algebra has a negative effect that cannot be undone.

This example shows that to handle the SQL expression properly, either the antijoin must be included into relational algebra, which requires to adapt the translation algorithm or the logical rewrite process, or that the logical algebra and the physical algebra have to be extended such that the quantification can be dealt with properly. With respect to this example, the set of standard relational algebra operators and query processing algorithms does not suffice to achieve good performance.

We continue our overview considering query optimization in the (extended) NF^2 data models.

2.3 Implementation of nested relational query languages

Dropping the First Normal Form constraint from the relational data model has led to the development of the NF^2 (Non-First Normal Form) data model in which attributes are allowed to be relation-valued. Because the NF^2 data structures are an extension of relational data structures, query languages for the NF^2 model usually are an extension of relational query languages. Several SQL-like query languages, calculi, and algebras for the NF^2 data model have been developed. An SQL-like language for the NF^2 data model is described for example in [RoKB88]; below we discuss some algebraic languages for the NF^2 model.

2.3.1 NF^2 algebras

Two types of nested algebras can be distinguished: the nest/unnest algebra, and the nested algebra, with and without explicit nesting. The respective algebras differ in the provisions that are taken for accessing attributes of relations nested within relations (subrelations). In the nest/unnest algebra, tuples of relation-valued attributes are brought to the top level by unnesting these attributes. In the nested algebras, the operations to be applied to subrelations are brought to the subrelations concerned, either by using one or more algebraic operators as a navigator, or by employing path expressions. For a complete overview of algebras for the nested relational model, we refer to [Vand93]. Below, we discuss the different types of algebras in more detail.

Nest/unnest algebras

The most simple solution to the task of defining an algebra for the nested relational model is to extend relational algebra with operators nest (ν) and unnest (μ) [ThFi86]. The simplicity of this approach is attractive, however, theoretical as well as practical problems arise.

As is well-known, theoretical problems arise because the nest operator is not the inverse of the unnest operator. Unnesting can be undone by nesting only if (1) the nested relation is in Partitioned Normal Form (PNF), and (2) no tuple in the relation being unnested does have the value $\emptyset$ (the empty set) for the attribute being unnested. A nested relation is in PNF if the atomic attributes form a key for the relation, and if, recursively, also the subrelations are in PNF. Unnesting/nesting a table that is not in PNF will result in a table in which grouping is different from that in the original table; unnesting/nesting a table with empty set-valued attributes will cause a loss of tuples. The above consideration has led to the study of PNF relations and investigation of keying methods to allow unnesting of arbitrary nested relations [ÖzWa92]. In [RoKS88], a calculus and algebra for PNF relations is defined. The algebra supports the nest and unnest operators, and the standard relational operators have been extended such that the property of PNF is preserved (the algebra is closed under PNF). Theoretical results concerning the expressive power of the nest/unnest algebra are given in [PaGu88, PaGu92]. When applied to flat relations, having flat relations as output as well, the nest/unnest algebra has the same expressive power as relational algebra.

From a practical point of view, having to unnest and later on to nest relations to be able to access nested attributes may cause a major performance problem. First, unnesting causes data redundancy, replicating the values of other attributes for each element of the set-valued attribute. Second, restructuring just to allow access to relation-valued attributes can be considered as pure computational overhead if a navigator is included in the language (see below). It is not clear, though, whether unnesting/nesting should be avoided in any case. For example, if the nesting phase can be skipped, unnesting may be a feasible strategy.

Explicit nesting

Another approach to support access to attributes of subrelations is to provide a navigator, i.e., an operator that can be used to apply operations to subrelations. In the algebra of [ScSc86], the projection operator can be used as navigator; arbitrary algebraic expressions may occur in the projection list. In addition, the select and project operator may be used in selection predicates, as well as set comparison operators and set constants. Nesting of operations is explicit, i.e., visible in the syntactic form of expressions.

Implicit nesting

In the algebra of [Colb89, Colb90], explicit navigation is avoided by using path expressions. Algebraic operators like selection, projection, and join, but also nest and unnest can be applied to subrelations by writing down the path through the hierarchical structure leading to the subrelation(s) of interest. Two kinds of set operator are supported–the standard version and an extended version in which the operation is propagated to subrelations. The extended versions are variations of the extended operators defined in [RoKS88].

Actually, the notation employing path expressions is a convenient abbreviation for explicit nesting of expressions. For example, assume we have a nested relation R with two attributes A and X. Attribute X is relation-valued, with only one attribute B. The application of some condition p to restrict the relation-valued attribute X of R is in the algebra of [Colb89, Colb90] expressed as $\sigma(R(X_p))$. In the algebra of [ScSc86], with its explicit nesting of operators, the query is formulated as $\pi[A, \sigma[p](X)](R)$, using the projection as navigator. In our algebra, as we will see, the map operator α is used as navigator, and we write $\alpha[r : r \ \mathbf{except}\ (X = \sigma[c : p](r.X))](R)$. (The **except** construct can be used to modify one or more attribute values.)

From a theoretical point of view, it does not matter whether projection is used as navigator, or a special-purpose navigation operator like the map. From a practical point of view, relational projection is a special case of the map operator for which many special-purpose implementations exist, so it may be inconvenient to modify the definition of projection. In addition, it must be noted that to use projection as a navigator, and also to shorten notation by means of path expressions is possible in the NF^2 model only, because NF^2 relations are orderly tree structures in which each attribute can be reached from the

root by a unique path. In extended NF^2 data models, in which the types of the elements of collections (sets, lists, etc.) are not necessarily tuple types, it is not possible to reach all values by means of path expressions.

2.3.2 The extended NF^2 data model

The main difference between the NF^2 and eXtended NF^2 (XNF^2) data models such as that of the AIM DBMS [SüLi90], is that XNF^2 data models (1) support additional data types such as lists and (2) allow for arbitrary nesting of type constructors.

From the viewpoint of query optimization, these additional features are not essential. For each additional data type, type-specific operators have to be supported and implemented efficiently, and also equivalence rules must be added. The first concern in query optimization, however, is to deal with iteration over bulk or collection types, and to reduce the number of (nested) iterations that has to be carried out. The type of the collection that is iterated over, e.g. set or list, is important, but how to deal with for example ordering may be considered as a secondary problem.

2.3.3 Query optimization

While there do exist proposals for SQL-like query languages for the (extended) NF^2 data model, direct translations of any of these languages into any one of the nested algebras have not been presented in the literature, to our knowledge. Translations from nested calculi to nested algebras do exist though, e.g. [RoKS88]. However, in these translations usually little emphasis is placed on performance aspects; often the major goal is to prove equivalence of calculus and algebra.

With regard to logical optimization of NF^2 algebras, algebraic equivalence rules for the nested relational algebra are given in [Scho88]. In [Buss91], logical optimization of single-pass selections is studied. It is shown that complex object selections do not always commute. However, this is due to the definition of a complex object selection used in that paper, which is a combination of one-level restriction and navigation.

In the Verso [Scholl et al. 89] and DASDBS [Schek et al. 90] projects, a great deal of work has been done on the use of nested relations as storage structures for the relational model. In this approach, the main task in logical optimization is to remove redundant joins from flat queries. With respect to the implementation of algebraic operators, emphasis is placed on single-pass queries, i.e., on queries that can be evaluated in a single hierarchical scan over the data. In the DASDBS-project, it was decided to include only operators needed for processing single-pass queries into the kernel (the physical algebra).

An example of a new query processing algorithm is given by [DeLa92], that presents a hash-based join processing method for a join between a relation and a subrelation. In the nested algebra of [ScSc86], any operation that occurs nested with the projection list, having relation-valued and/or base tables as operands, can be executed as it is. For example, a possible evaluation strategy for the nested join expression:

$$\alpha[x : x.c \underset{z,y:p(z,y)}{\bowtie} Y](X)$$

joining set-valued attribute c of X with base table Y, is to evaluate the nested operation for each of the tuples $x \in X$, using standard relational processing algorithms for the join. Another approach is to come up with a new algorithm (implying a new logical operator as well) for the evaluation of such an expression in which a subrelation is joined with a base table, as is done in [DeLa92].

2.3.4 Discussion

In the past, a world of knowledge has been gained with respect to optimization of relational query languages. Comparing the work done on the relational model with that done on the NF^2 model, we feel that the latter is rather incomplete. The main part of the work concerns the definition of algebras, not their function, which is to facilitate efficient query evaluation. A large number of algebras for the nested relational model has been developed. The algebras differ in the formalisms used, in assumptions about the structure of the data, in the facilities offered to access nested values, and in expressive power. In our opinion, the algebra of [ScSc86] is the most orthogonal language proposal, allowing to apply algebraic operators to relations wherever they may occur, using the projection operator as a navigator. To abbreviate expressions by making use of path expressions as in the algebra of [Colb90] may be very convenient, but is not essential.

Research into NF^2 database management systems has paid little attention to the following issues:

- The translation from nested relational SQL into an algebra.
- A logical rewrite algorithm.
- The implementation of arbitrary logical algebra operations, e.g. nested algebra expressions unlike single-pass queries.

To our knowledge, it has not been investigated how to translate an SQL-like query language for the NF^2 model (say, XSQL) into an algebra for the nested relational model. Our explanation is the following. Translating XSQL into a nest/unnest algebra is a task comparable to that of translating a calculus for nested relations into a nest/unnest algebra. The latter exercise has been carried out in the past, and is worthwhile for theoretical reasons mainly, as explained. Translating XSQL into a nested algebra like that of [ScSc86] is rather trivial, just because the algebra, a quite orthogonal language, supports nesting of expressions, and therefore allows for an almost one-to-one mapping. However, such a one-to-one mapping that leaves nested expressions as they are does not solve everything—part of the problem of how to implement (relational) SQL efficiently is essentially solved in the step of translation into the (relational) algebra, which means a transition from a tuple-oriented query processing model, in which nesting of expressions (subqueries) may occur, to a set-oriented one. Subqueries are replaced by joins, thus allowing to employ set-preprocessing techniques like sorting and hashing.

Because the nested relational model often is considered as the basis for object-oriented models, the state of affairs as described above is not exactly promising. In the following

section we discuss some important features of object-oriented data models, and try to discover the implications for query processing.

2.4 Implementation of object-oriented query languages

The relational model is well-suited for use in traditional, administrative environments with large amounts of data having relatively little structure. Set-oriented query processing is the best way to handle large amounts of data that have similar structure. Object-Oriented DBMSs (OODBMSs) are being designed and built to support new application domains like GIS and CAD/CAM. In new applications, the characteristics of data collections may be very different from those present in the traditional administrative environment. For example, in a CAD/CAM application, classes may contain relatively few objects, but with a relatively deep structure.

2.4.1 Object-oriented features

Some important features of object-oriented models not present in extended relational models are inheritance, object identity, the use of classes as attribute domains, and the presence of methods.

Inheritance

Inheritance is a key concept of object-oriented data models. In [Atkinson et al. 89], four of the possible forms of inheritance are described: substitution, inclusion, constraint, and specialization inheritance.

The implementation of specialization (attribute) inheritance, and hence of substitution (method) and constraint inheritance is mainly a matter of physical database design. In [ElNa89] some options for mapping the EER concept of generalization/specialization (i.e., attribute inheritance) to the relational model are described. Attribute inheritance may be implemented by means of oid equality, by means of clustering of sub- and supertype attributes, or by using flags together with clustering of (possibly NULL-valued) sub- and supertype attributes. Each way of implementing attribute inheritance has its own performance penalties for query and constraint evaluation.

Implicit joins

In object-oriented data models, classes may be used as attribute domains. In the sequel, these kinds of domains are called class reference types. Implementation of class reference types is also an issue of physical database design.

Typically, class references will be implemented by means of object identifiers (instead of fully materializing the attribute domain instance). In the implementation, the object-oriented schema is mapped to a schema in which class reference types are replaced by the object identifier type. In the object-oriented query language, attribute values of domains

that are classes are accessible by means of attribute selection; in the literature this is often called an implicit join. In our view, implicit joins will often be mapped to explicit joins; depending on the implementation of the type object identifier type, these joins may be pointer-based or value-based. (In the literature sometimes explicit joins are defined as value-based joins).

In many object-oriented algebras, attribute selection is supported as an algebraic primitive.

Object identity

Object identity is a property that can be useful in several ways. Conceptually, an object identifier is just a unique system-generated identifier. In actual implementations, object identifiers are used as (typed) pointers. Using object identifiers as pointers allows for recursive schema definitions, enables sharing, facilitates to check referential integrity, and may offer special advantages with respect to query optimization.

Object identifiers may be implemented as physical pointers, in some way or the other reflecting the object location on disk, or as logical pointers.

Implementing object identifiers as physical pointers means that the relationships between objects laid down in the schema by class references can be used as (user-defined) fast access paths. Join execution methods making use of these predefined fast access paths are called pointer-based join methods. In [ShCa90] a performance comparison is made between some of the traditional value-based join methods and their pointer-based counterparts. The comparison has been made for full joins and joins with a selection on the join operand referencing the other join operand (forward traversal). To speed up backward traversal, backward references (that also may greatly facilitate checking referential integrity) can be maintained. Pointer-based join methods following the predefined access paths (forward traversal) are generally faster than the traditional relational join methods. However, the pointer-based nested loop method (naive pointer traversal or pointer chasing) was shown to perform poorly in almost all cases. For small joins, though, the pointer-based nested loop method proved to be the best. Implementing object identifiers as logical pointers means that no special advantage can be gained; in that case object-oriented joins are standard value-based joins.

In the literature, it is sometimes stated that in object-oriented database systems value-based joins are no longer necessary [Bert93]. However, we believe that there do exist meaningful join queries that relate objects in other ways than by means of the user-defined relationships laid down in the schema.

Methods

Optimization of methods is often considered a problem, mainly due to the fact that in many existing object-oriented DBMSs methods for the most part are written in a third generation general purpose programming language like C++. Optimization of such a general purpose programming language in the context of database applications seems a hard problem indeed. Writing methods in a language more amenable to optimization, for example the

(declarative) query language, will partially solve the problem. For example, in the object-oriented data model TM, methods are written in a high-level, declarative language of expressions. Method calls in a query can then be textually substituted by their definition, which allows for optimization. TM retrieval methods can simply be looked upon as parameterized view definitions. As with view definitions, method code may be optimized locally, or the code may be substituted into the expression containing the method call.

2.4.2 Optimization

A number of object-oriented algebras have been proposed in the literature, e.g. [ShZd89, ClMo93]. See also [Vand93] for an overview. In most of the proposals, a special navigator (map, image, etc.) is included in the language, which allows for nesting of expressions.

2.4.3 Discussion

In our view, an object-oriented data model that supports a high-level declarative query language is implemented as follows. The target language (the language the model is mapped to, for example an algebra) does not support inheritance, object identity, implicit joins, methods, and views. It is assumed that data type constructors supported by the source and target language are the same; the target language has an additional basic type **oid**. In the sequel we describe the issues involved in mapping an object-oriented data model to lower levels.

From the viewpoint of optimization, the most significant new feature of object-oriented data models is object-identity. Complex objects are already present in (X)NF^2 models, inheritance is a matter of physical database design, and implicit joins, if not materialized, will be transformed into explicit joins at the logical level.

On the other hand, OODBMSs are tailored towards technical applications in which data collections may have statistical properties differing from those in traditional administrative environments. It is not sure whether set-oriented query processing will have the same benefits that it has in traditional application domains.

2.5 Our view on optimization

In database systems, performance is of crucial importance. Data volume may be large and complex, and used by many applications at the same time. Performance may be measured in terms of response time, CPU costs, I/O costs, storage costs, communication costs, etc., or combinations thereof. Naive, i.e. interpretative evaluation of user queries will result in unacceptable performance.

The subject of this thesis concerns the translation of an SQL language for object-oriented database systems, say, OSQL, into an algebra for complex objects. The ultimate goal is to achieve an efficient implementation of OSQL. We assume a three-level database system architecture. The top layer is OSQL. The bottom layer provides us with at least the

efficient query processing techniques that are available in relational systems. The intermediate language is an algebra supporting complex objects. Naive query processing can be improved in two different ways:

1. By providing efficient access algorithms. Efficient access algorithms form the basis for efficient query processing. The basic techniques employed to speed up query evaluation in comparison to naive execution are sorting, hashing, and the use of indices.

2. By providing a good translation. The user query has to be mapped to the physical level in the best possible way.

The problem studied in this thesis is how to translate a **subset** of OSQL into **an algebra** such that the result of the transformation is **efficient**.

Why a subset?

In our opinion, the two features of OSQL that are the most important with respect to efficient query processing are the orthogonality of the language and the presence of set-valued attributes. Language orthogonality allows for arbitrary nesting of OSQL constructs. While nesting in the WHERE-clause has been studied extensively in the relational model, it is not known how to handle queries with nesting in the SELECT-clause, that, in general, expresses the formation of complex results.

The presence of set-valued attributes is a second important language feature. In an orthogonal query language for a data model that supports set-valued attributes, such attributes can be used in any place where base tables can be used (except at the top level of course). Set-valued attributes may occur as operand of a SELECT-FROM-WHERE query block, as the operands of set (comparison) operators, as quantifier ranges, and even in join predicates. Operations on set-valued attributes can be intermingled with operations on base tables in arbitrary ways. How to deal with queries involving set-valued attributes and base tables is an open problem. As explained before, one option is to unnest set-valued attributes to enable to access the elements, however, such an approach may be inefficient and causes problems of a theoretical nature.

As described, some work has been done on logical optimization for the NF^2 data model. Comparatively little work has been done on the design of physical algebra operators for the implementation of arbitrary queries involving complex objects. Most of the work done on access algorithms for systems supporting complex objects is focussed on resolution of references, or the evaluation of path expressions, i.e., the retrieval of complex objects according to user-defined relationships. The assembly operator of [KeGM91], is designed to assemble complex objects using a strategy that is a combination of object-at-a-time (depth-first) and set-oriented (breath-first) retrieval. Index structures for evaluating path expressions have been surveyed in [Bert93]. For queries that establish arbitrary relationships between complex objects, little work has been done yet. A notable exception is the paper of [DeLa92], that studies how to implement a join between a set-valued attribute of one nested relation and a second flat relation. An example of an arbitrary query concerning

other than user-defined relationships (or pre-defined access paths) is the following: select all suppliers together with the set of suppliers that supply at least the same parts. In OSQL this is:

select s **except** sps = **select** s'
from s' **in** SUPPLIER
where s'.parts $\subseteq$ s.parts
from s **in** SUPPLIER

The above query is a join between two tables (or class instances) of which the join predicate is complex, testing the subset relationship between two set-valued attributes.

Why an algebra?

To bridge the gap between the tuple-oriented user language and the set-oriented physical algebra, we propose to use an algebra. The set-oriented query evaluation paradigm seems appropriate at the logical level too. In Section 2.2 we have shown that an algebraic language is easily extensible, and does not fix the processing order like a calculus does. Usually, the logical algebra is used to perform logical optimization. However, as will be discussed in more detail below, logical optimization of an algebraic expression involving arbitrary operators (not only select-project-join) can be very difficult. Therefore, we try to perform optimization during the translation of OSQL into the algebra.

Which logical algebra?

The logical algebra has to support operators such that an OSQL expression can be mapped to a logical expression that, when mapped to a physical algebra expression, has performance at least equal to, but preferably better than that achieved by means of naive query evaluation. The goal is to map OSQL expressions to that logical algebra expression that can be mapped to the best physical algebra expression, given the database characteristics. In other words, the logical algebra must contain operators such that the cost of evaluation of algebraic expressions is smaller than the cost of naive evaluation. For example, a simple nest/unnest algebra is not suited for implementing a nested relational query language.

Why efficient?

Traditionally, the term query optimization is restricted to the phases of logical rewriting and plan compilation. As explained, logical optimization is the heuristics-based rewriting of a logical algebra expression into an expression that can be evaluated more efficiently. For each logical algebra expression, a large number of possible query evaluation plans exist; the task of plan compilation is to pick the plan that is the best, given the database state. Usually, the phases of logical optimization and plan compilation are viewed as separate stages in query processing. However, in [GrMc93], it is proposed to combine both in one optimizer.

In the framework sketched above, translation of the user query into the algebra does not have any performance related aspects. Even stronger, translation into the algebra generally results in an expression that can be considered as the most costly normal form [KeMo93]. A naive, standard translation may result into an algebraic expression that has performance far worse than the original query when evaluated by means of nested-loop processing, and rewriting such an expression into the desired form may be a complex task [Naka90]. Instead of performing logical optimization after translation, we propose, as is also done in [Naka90], to perform some logical optimization during translation of OSQL into the logical algebra. Pure algebraic rewriting, i.e., rewriting using algebraic equivalence rules only, becomes more complicated as expressions grow larger and the algorithms expressed by source and target expression differ more from each other.

2.6 Summary

The general approach of this thesis is as follows. The task is to achieve an efficient implementation of a subset of the object-oriented declarative query language OSQL. We assume a three-level architecture consisting of end-user query language, the intermediate language in which logical optimization takes place and a physical access language implementing the intermediate language. It is assumed that a phase of preprocessing deals with the object-oriented features of inheritance, class references, and methods; in this thesis we will discuss neither clustering algorithms nor optimization of methods. The OSQL subset that is the output of the preprocessing phase is further restricted in that we do not consider update queries, and that the type system of the OSQL subset, and that of the intermediate language, is the NF^2 type system. The intermediate language is a combination of a calculus-like language and a pure algebraic language. The translation of the restricted OSQL subset into the intermediate language is a naive, almost one-to-one translation; transformation of the calculus-like expressions that are the result of the translation will be done guided by equivalence or rewrite rules and will be based on a simple cost model. The access language is a so-called physical algebra, a language that supports one or more operator execution methods for each logical algebra operator. The emphasis in this thesis will be on the *efficient* transformation of calculus into algebraic expressions, which we call logical transformation.

References

[Atkinson et al. 89] Atkinson, M., F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, "The Object-Oriented Database System Manifesto," *Proceedings 1st DOOD*, Kyoto, Japan, December 1989.

[BMSU86] Bancilhon, F., D. Maier, Y. Sagiv, and J.D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proceedings 5th ACM PODS*, 1986, pp. 1–15.

[Bert93] Bertino, E., "A Survey of Indexing Techniques for Object-Oriented Databases," in: *Query Processing for Advanced Database Systems*, eds. J.-C. Freytag, D. Maier, and G. Vossen, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Buss91] van den Bussche, J., "Evaluation and Optimization of Complex Object Selections," *Proceedings DOOD*, 1991.

[CeGo85] Ceri, S. and G. Gottlob, "Translating SQL into Relational Algebra: Optimization, Semantics, and equivalence of SQL Queries," *IEEE Transactions on Software Engineering*, Vol. SE–11, No. 4, April 1985.

[CeGT90] Ceri, S., G. Gottlob, and L. Tanca, **Logic Programming and Databases**, Surveys in Computer Science, Springer-Verlag, Berlin Heidelberg, 1990.

[ClMo93] Cluet, S. and G. Moerkotte, "Nested Queries in Object Bases," *Proceedings Fourth International Workshop on Database Programming languages*, New York, Sept. 1993.

[Codd72] Codd, E.F., "Relational Completeness of Data Base Sublanguages," in: *Data Base Systems*, ed. R. Rustin, Prentice Hall, 1972.

[Colb89] Colby, L.S., "A Recursive Algebra and Query Optimization for Nested Relations," *Proceedings ACM SIGMOD*, Portland, June 1989, pp. 273–283.

[Colb90] Colby, L.S., "A Recursive Algebra for Nested Relations," *Information Systems*, 15(5), 1990, pp. 567–582.

[DeLa92] Desphande, V. and P.-Å. Larson, "The Design and Implementation of a Parallel Join Algorithm for Nested Relations on Shared-Memory Multiprocessors," *Proceedings IEEE Conference on Data Engineering*, Tempe, Arizona, February 1992, pp. 68–77.

[Date90] Date, C.J., **An Introduction to Database Systems**, Addison-Wesley Publishing Company Inc., USA, 1990.

[DFMV90] DeWitt, D., P. Futtersack, D. Maier, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," *Proceedings VLDB*, Brisbane, Australia 1990.

[ElNa89] Elmasri, R. and S.B. Navathe, **Fundamentals of Database Systems**, Benjamin/Cummings Publishing Company Inc., 1989.

[Grae93] Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2), June 1993, pp. 73–170.

[GrMc93] Graefe, G. and W.J. McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search," *Proceedings of the IEEE Conference on Data Engineering*, 1993, pp. 209–218.

[JaKo84] Jarke, M. and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, 16(2), June 1984, pp. 111–152.

[KeGM91] Keller, T., G. Graefe, and D. Maier, "Efficient Assembly of Complex Objects," *Proceedings ACM SIGMOD*, Denver, Colorado, May 1991, pp. 148–157.

[KeMo93] Kemper, A. and G. Moerkotte, "Query Optimization in Object Bases : Exploiting Relational Techniques," in: *Query Processing for Advanced Database Systems*, eds. J.-C. Freytag, D. Maier, and G. Vossen, Morgan Kaufmann Publishers, San Mateo, California, 1993

[Kim82] Kim, W., "On Optimizing an SQL-like Nested Query," *ACM TODS*, 7(3), September 1982, pp. 443–469.

[KKWD88] Kim, K.C., W. Kim, D. Woelk, and A. Dale, "Acyclic Query Processing in Object-Oriented Databases," MCC Technical Report ACA-ST-287-88, Austin, Texas, September 1988.

[LiDe92] Lieuwen, D.F. and D.J. DeWitt, "A Transformation-Based Approach to Optimizing Loops in Database Programming Languages," *Proceedings ACM SIGMOD*, San Diego, California, 1992, pp. 91–100.

[Naka90] Nakano, R., "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS*, 15(4), December 1990, pp. 518–557.

[ÖzWa92] Özsoyoğlu, Z.M. and J. Wang, "A Keying Method for a Nested Relational Database Management System," *Proceedings of the IEEE Conference on Data Engineering*, Tempe, Arizona, February 1992.

[PaGu88] Paredaens, J. and D. van Gucht, "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions," *Proceedings 7th ACM PODS*, Austin, Texas, March 1988, pp. 29–38.

[PaGu92] Paredaens, J. and D. van Gucht, "Converting Nested Algebra Expressions into Flat Algebra Expressions," *ACM TODS*, 17(1), June 1992.

[RoKB88] Roth, M.A., H.F. Korth, and D.S. Batory , "SQL/NF: A Query Language for ¬ 1NF Relational Databases," *Information Systems*, 12(1), March 1987, pp. 99–114.

[RoKS88] Roth, M.A., H.F. Korth, and A. Silberschatz, "Extended Algebra and Calculus for Nested Relational Databases," *ACM TODS*, 13(4), December 1988, pp. 389–417.

[Schek et al. 90] Schek, H.-J., H.-B. Paul, M.H. Scholl, and G. Weikum, "The DASDBS Project: Objectives, Experiences, and Future Prospects," *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990, pp. 25–43.

[ScSc86] Schek, H.-J., and M.H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems* , 11(2), 1986, pp. 137–147.

[Scho88] Scholl, M.H., "The Nested Relational Model—Efficient Support for a Relational Database Interface," Ph.D. Thesis, Technical University of Darmstadt (in German), 1988.

[Scholl et al. 89] Scholl, M., S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and D. Verroust, "VERSO: A Database Machine Based on Nested Relations," in: *Nested Relations and Complex Objects in Databases*, eds. S. Abiteboul, P.C. Fischer, H.-J. Schek, LNCS 361, Springer-Verlag, 1989, pp. 27–49.

[ShZd89] Shaw, G.M., and Zdonik, S.B., "Object-Oriented Queries: Equivalence and Optimization," *Proceedings 1st International Conference on Deductive and Object-Oriented Databases*, Kyoto, December 1989.

[ShCa90] Shekita, E.J. and M.J. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proceedings ACM SIGMOD*, Atlantic City, May 1990, pp. 300–311.

[SwGu88] Swami, A. and A. Gupta, "Optimization of Large Join Queries," *Proceedings SIGMOD*, Chicago, June 1988, pp. 8–17.

[SüLi90] Südkamp, N. and V. Linnemann, "Elimination of Views and Redundant Variables in an SQL-like Database Language for extended NF^2 Structures," *Proc VLDB*, Brisbane, 1990.

[ThFi86] Thomas, S.J. and Fischer, P.C., "Nested Relational Structures," *Advances in Computing Research III, The Theory of Databases*, P.C. Kanellakis, ed., JAIpress, 1986, pp. 269–307.

[Ullm89] Ullman, J.D., **Database and Knowledge-base Systems**, Computer Science Press, Rockville, USA, 1989.

[Vand93] S.L. Vandenberg, "Algebras for Object-Oriented Query Languages," Ph.D. Thesis, University of Wisconsin-Madison, 1993.

[WoYo76] Wong, E. and K. Youssefi, "Decomposition–A Strategy for Query Processing," *ACM TODS*, 1(3), September 1976, pp. 223–241.

Chapter 3

Languages and transformation goal

In this thesis, we investigate query processing in the context of a data model that supports complex objects. Starting from an SQL-like query language, the task is (1) to define an algebra for complex objects, (2) to provide a translation of the end-user query language into the algebra, and (3) to study logical optimization in the algebra. In this chapter, we describe the end-user query language OSQL and the language ADL, which is used as intermediate language between OSQL and the system-specific physical algebra.

Various SQL-like languages for data models that support complex objects exist, e.g. [RoKB88, PiAn86, BaCD92, Catt93]. In this chapter, we define a language called OSQL. OSQL is not a new language; it is a prototype of block-structured SQL-like query languages for advanced data models. OSQL can be considered as a subset of the object-oriented database specification language TM [BaBZ93, BaVr91]. In the definition of OSQL, the emphasis is placed on the language constructs that we consider important with respect to optimization, i.e. orthogonality and the presence of set-valued attributes. As explained in the previous chapter, we do not consider specific object-oriented features like inheritance, methods, etcetera. Structurally, our data model is NF^2, extended in the sense that attributes may be arbitrary sets and tuples, not only relations. This extension is not essential but merely simplifies the presentation. The main language construct of OSQL is the **select-from-where** (**sfw**-) expression, comparable to the SELECT-FROM-WHERE construct of SQL, and also HDBL [PiAn86].

The language ADL is an extension of OSQL; the extension consists of a number of pure algebraic or set operators. ADL is defined as an extension of OSQL because it is uncertain whether each and every OSQL language construct can (and should) be mapped into a set-oriented expression that consists of algebraic operators that are part of the extension only. Calculus-based concepts such as quantifiers and the support for nesting of operations are part of the intermediate language as well because they are needed to deal with set-valued attributes. The algebraic part of the language ADL is an extension of the alge-

bra of [ScSc86] that was defined for the NF^2 data model. Most importantly, the extension involves a number of non-standard join operators.

The aim is to transform OSQL expressions into algebraic expressions in the best possible way. Possible transformations are laid down in equivalence rules. We take a step-by-step approach to translation and optimization. In Chapter 4, we investigate translation and optimization of an OSQL subset that is equivalent to relational calculus. In subsequent chapters, we consider more advanced issues, the most important of them being the presence of set-valued attributes and arbitrary nesting of operators in the **select-from-where** query block.

The outline of this chapter is as follows. In Section 3.1 we describe the data types that we use. In Section 3.2 and Section 3.3, we present the languages OSQL and ADL, respectively. In Section 3.4, the semantics of the operators of OSQL and ADL is given. (Because both OSQL and ADL are rather simple languages, we not pay much attention to aspects of formality.) Because ADL is defined as a superset of OSQL, the transformation goal has to be clearly stated. In Section 3.5, we present a classification of expressions, and an initial proposal how to handle the respective classes in the translation. In Section 3.6, some considerations with respect to a cost model are presented.

3.1 Data structures

OSQL and ADL are typed languages; below we describe the data types used. We do not introduce function types. Function types are needed to give typing rules for the operators present in OSQL and ADL, but we refrain from giving these rules for reasons of simplicity.

It is assumed that a number of basic or atomic types is supported; one of them is the type **oid**, used to represent object identity. The type constructors supported are the tuple constructor $\langle\cdot\rangle$ and the set constructor $\{\cdot\}$, which may be alternated in arbitrary ways. Assume we have a set of labels or attribute names a, then the types are defined as follows:

Definition 3.1 Types

1. β is a type, whenever $\beta \in \{\mathbf{oid}, \mathbf{bool}, \mathbf{string}, \mathbf{num}\}$ (basic types)
2. $\langle a_1 : \tau_1, \cdots, a_n : \tau_n \rangle$ is a type, (tuple types)
   whenever all a_i's are distinct labels, all τ_i's are types, $1 \leq i \leq n$
3. $\{\tau\}$ is a type, whenever τ is a type (set types)

□

The types of the base tables, i.e., of the values stored in the database are assumed to be set-of-tuple types, of which the attributes may be arbitrarily typed. Whenever a value v is of type τ, we write $v : \tau$. We give some examples.

Example 3.1 Table types

- DEPT : $\{\langle doid : \mathbf{oid},$
  $dname : \mathbf{string},$
  $address : \langle street : \mathbf{string}, city : \mathbf{string}\rangle,$
  $emps : \{oid\}\rangle\}$
- EMP : $\{\langle eoid : \mathbf{oid},$
  $ename : \mathbf{string},$
  $sal : \mathbf{num}\rangle\}$

Both DEPT and EMP are sets of tuples. Attribute $address$ of DEPT has a tuple type that consists of attributes $street$ and $city$ that are of atomic type; attribute $emps$ represents a set of object identifiers that refer to employees in EMP, which is a standard relation.

3.2 OSQL

Below, we present the expression forms that are part of the language OSQL. The language is simple in that fairly basic constructs are supported. The language is split into a kernel and an extension. The OSQL kernel consists of those constructs that we consider necessary with respect to expressive power. The extension consists of constructs that usually are included into SQL languages, but that can be expressed in the OSQL kernel. We stress that, although the extension is not strictly needed with regard to expressive power, it is necessary from the viewpoint of translation and optimization. A reduction of the extension to the kernel will result into a translation that in some cases is unnecessarily inefficient. For example, set intersection can be expressed by means of set difference:

$$X \cap Y = X - (X - Y)$$

However, because it is possible to provide an efficient implementation of intersection as well as difference, the performance of the difference expression probably is much worse than that of the intersection, so it would not be wise to exclude that operator from the language.

We note that OSQL as presented in this section is not a complete query language—we have tried to find a compromise between expressive power and language constructs that we consider important with respect to translation and optimization.

Below, we first present the syntax of the OSQL kernel and extension. Next, we give the syntax of the full language in BNF notation, and then we describe the meaning of the various language constructs. In Section 3.4, the semantics of some of the language constructs is given in a less (but still) informal manner. Finally, we briefly describe how to express the extension in the kernel.

Definition 3.2 OSQL kernel

Assume we have constants c, variables x, and labels a, then the following are OSQL expressions e:

c	(constants)
x	(variables)
(e)	(brackets)
$\mathbf{select}\ e_1\ \mathbf{from}\ x\ \mathbf{in}\ e_2\ \mathbf{where}\ e_3$	(**sfw**-query block)
$e_1 \cup e_2$	(set operators)
$\mathbf{el}(e)$	(element pick)
$\bigcup(e)$	(generalized union, or flatten)
$e.a$	(attribute selection)
$\langle a_1 = e_1, \ldots, a_n = e_n \rangle$ (distinct a_i)	(tuple construction)
$\neg e$	(negation)
$e_1 \wedge e_2$	(conjunction)
$\exists x \in e_1 \bullet e_2$	(existential quantification)

□

Definition 3.3 OSQL extension

The following expressions are added to the OSQL kernel as given in Definition 3.2.

$e\ \mathbf{with}\ (x_1 = e_1, \ldots, x_n = e_n)$	(local definitions)
$e_1\, \theta\, e_2$ for $\theta \in \{\cap, -, \times\}$	(set operators)
$e[a_1, \ldots, a_n]$ (distinct a_i)	(tuple projection)
e_1 ++ e_2	(tuple concatenation)
$e\ \mathbf{except}\ (a_1 = e_1, \ldots, a_n = e_n)$ (distinct a_i)	(record modification)
$e_1\, \theta\, e_2$ for $\theta \in \{\in, \subset, \subseteq, =, \supseteq, \supset\}$	(set comparison operations)
$e_1 \vee e_2$	(disjunction)
$\forall x \in e_1 \bullet e_2$	(universal quantification)
$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	(conditional)

□

Note that we did not include arithmetical operations in our language; it can easily be extended. Below, we present the syntax of full OSQL in BNF notation.

OSQL	::=	i	(top level expressions)
i	::=	$\mathbf{select}\ e_1\ \mathbf{from}\ x\ \mathbf{in}\ e_2\ \mathbf{where}\ p$	
e	::=	$c \mid x \mid t \mid i \mid s \mid$	(expressions)
		$\mathbf{if}\ p\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{el}(e) \mid$	
		$(e) \mid e\ \mathbf{with}\ (x_1 = e_1, \ldots, x_n = e_n)$	
s	::=	$\bigcup(e) \mid e_1\ sop\ e_2$	(set expressions)
sop	::=	$\cup \mid \cap \mid - \mid \times$	(set operators)
t	::=	$e.a \mid e[a_1, \ldots, a_n] \mid e_1 +\!+ e_2 \mid$	(tuple expressions)
		$\langle a_1 = e_1, \ldots, a_n = e_n \rangle \mid$	
		$e\ \mathbf{except}\ (a_1 = e_1, \ldots, a_n = e_n)$	
p	::=	$e_1\ cop\ e_2 \mid$	(predicates)
		$\neg p \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid$	
		$\exists x \in e \bullet p \mid\ \forall x \in e \bullet p$	
cop	::=	$\in \mid \subset \mid \subseteq \mid = \mid \supseteq \mid \supset$	(set comparison operators)

The main construct of the language is the expression is the **sfw**-query block:

$$\mathbf{select}\ e_1\ \mathbf{from}\ x\ \mathbf{in}\ e_2\ \mathbf{where}\ e_3$$

which is comparable to the SELECT-FROM-WHERE query block from SQL, and many extensions of SQL such as HDBL [PiAn86] and the query language OQL that is part of the ODMG proposal [Catt93]. Expressions e_i in the expression above may be arbitrary OSQL expressions (provided they are well-typed), in which also other **sfw**-blocks may occur. The **where**-clause is optional. In the **select**-clause expression e_1 and in the predicate e_3, x may occur as a free variable. The expression e_2 is called the operand of the **sfw**-block. Informally, the meaning of the **sfw**-expression is as follows. The expression e_1, the **select**-clause function, is evaluated for those tuples x in the collection e_2 that satisfy the predicate e_3. Below, we give two examples.

Example 3.2 Sfw-query block

- Select the names of the departments located in Amsterdam, together with the names of the employees working for the department.

$$\begin{array}{l}\mathbf{select}\ \langle dname = d.dname, enames = (\mathbf{select}\ e.ename \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{from}\ e\ \mathbf{in}\ \text{EMP} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{where}\ e.eoid \in d.emps)\rangle \\ \mathbf{from}\ d\ \mathbf{in}\ \text{DEPT} \\ \mathbf{where}\ d.address.city = Amsterdam\end{array}$$

- Select the departments that have employees that earn more than 100K.

$$\begin{array}{l}\mathbf{select}\ d \\ \mathbf{from}\ d\ \mathbf{in}\ \text{DEPT} \\ \mathbf{where}\ \exists s \in (\mathbf{select}\ e.sal \\ \qquad\qquad\qquad \mathbf{from}\ e\ \mathbf{in}\ \text{EMP} \\ \qquad\qquad\qquad \mathbf{where}\ e.eoid \in d.emps) \bullet s > 100K\end{array}$$

Besides the **sfw**-expression, the language OSQL supports the standard set operators $\cup, \cap$, and $-$ (set difference), the extended Cartesian product $\times$ of the relational model, the operands of which are sets of tuples, the operators **el**, to retrieve the element of a singleton set, and flatten $\bigcup$, to 'collapse' a set of sets into one. Set comparison operators $\in, \subset, \subseteq, \supseteq$, and $\supset$ are included, and equality is defined for all types. Existential and universal quantification are important language constructs; the conditional **if - then - else** - is included for reasons of convenience. Furthermore, a number of tuple (or record) operations is supported. Tuple operations are tuple construction, tuple concatenation ++ , tuple projection, and attribute (or field) selection, as usual expressed by the dot operator.

The **except** construct is included for reasons of convenience. In TM, **except** is used for record modification, enabling to change the value of record fields without having to use tuple construction, which may become quite elaborate for tuples that have many attributes. The types of tuple fields may not be altered. In ADL, considering retrieval only, we use a more general form of **except** in that attribute types may be modified, and in that it is also allowed to add fields—attribute names that do not occur in the original tuple are assumed to represent new fields in the 'updated' tuple. We give an example of **except**. Let t denote the tuple $\langle a = 1, b = true \rangle$, then the expression $t\ \mathbf{except}\ (a = 2)$ evaluates to $\langle a = 2, b = true \rangle$, and the expression $t\ \mathbf{except}\ (c = \ 'Joe')$ evaluates to $\langle a = 1, b = true, c = \ 'Joe' \rangle$.

The **with**-construct is used for local definitions as in:

$$x.a + x.b\ \mathbf{with}\ x = \langle a = 1, b = 2 \rangle$$

The **with**-construct may be used to handle common subexpressions.

Besides the **sfw**-expression, the language TM supports set comprehension. The general format of set comprehension is $\{x : \delta \mid p\}$. In this expression, δ is a domain that is built up from regular types and/or Class names by using the type constructors of the language. Due to this general format of set comprehension, problems of safety and constructiveness have to be dealt with. As this lies outside of the scope of this thesis, we restrict ourselves to the **sfw**-expression, which is equivalent to the restricted and safe form of set comprehension $\{f(x) \mid x \in X \wedge p(x)\}$.

The expressions forms that belong to the extension can be easily expressed in the kernel. For example, set difference can be expressed using a **sfw**-block and negated existential quantification, set intersection can be expressed with the use of set difference, and set comparison operators can be rewritten by means of quantification. We show the transformation for the Cartesian product that can be expressed by means of the **sfw**-expression, attribute selection, tuple construction, and the operator flatten, under the condition that the top-level attribute names of the two operands are known. Let $X : \{\langle a_1 : \sigma_1, \ldots, a_n : \sigma_n \rangle\}$, and $Y : \{\langle b_1 : \tau_1, \ldots, b_n : \tau_m \rangle\}$, then we have:

$$\begin{aligned} X \times Y \equiv \bigcup(&\mathbf{select\ select}\ \langle a_1 = x.a_1, \ldots, a_n = x.a_n, b_1 = y.b_1 \ldots, b_m = y.b_m \rangle \\ &\quad \mathbf{from}\ y\ \mathbf{in}\ Y \\ &\mathbf{from}\ x \in X) \end{aligned}$$

3.3 ADL

The language ADL is a hybrid language: a mixture of a tuple calculus and set algebraic expressions. ADL is defined as a superset of the language OSQL; in Section 3.5, we motivate the ideas that underly the definition the language ADL, and also the goal in the transformation/optimization of OSQL expressions. The calculus features of ADL are the support for arbitrary nesting of expressions and the presence of quantifiers. The algebraic part of the language, the operators listed below together with the standard set operators, is an extension of the algebra for nested relations of [ScSc86].

Definition 3.4 ADL expressions

The set of ADL expressions e is the set of OSQL expressions according to Definition 3.2, extended with the following algebraic expressions:

$\sigma[x : p](e)$	(selection)
$\alpha[x : f](e)$	(function application or map)
$\pi_A(e)$	(projection)
$e_1 \underset{x_1,x_2:p}{\theta} e_2$ for $\theta \in \{\bowtie, \ltimes, \rhd\}$	(join operators)
$e_1 \underset{x_1,x_2:f\|p;a}{\Delta} e_2$	(nestjoin)
$e_1 \div e_2$	(division)
$\nu_{A;a}(e)$	(nest)
$\mu_a(e)$	(unnest)

□

Roughly, the algebraic operators of the language ADL are the standard set operators, the extended Cartesian product, in which operand tuples are concatenated, and division, the map operator α, selection σ, projection π, and the restructuring operators nest (ν) and unnest (μ). The map operator, which is a construct well-known from functional programming languages, is used to apply a function to every element of a set. Furthermore, a number of join operators is supported: the regular join $\bowtie$, the semijoin $\ltimes$, and the antijoin $\rhd$. The semijoin $\ltimes$ (a regular join followed by the projection on the attributes of left-hand operand) is a join operator that is useful in processing certain kinds of queries, namely tree queries [Kamb85]. The antijoin $\rhd$ is defined as a semijoin followed by a set difference of the left-hand join operand and the semijoin result. The antijoin operator is less known than the semijoin operator; it can be usefully employed to process tree queries involving universal quantification. The nestjoin operator Δ is a new join operator that is useful for the translation of non-relational queries; it will be discussed extensively throughout this thesis. In principle, in selection predicates, map functions, and join predicates, arbitrarily complex expressions can be used, including expressions that contain quantifiers, set (comparison) operators, and base tables.

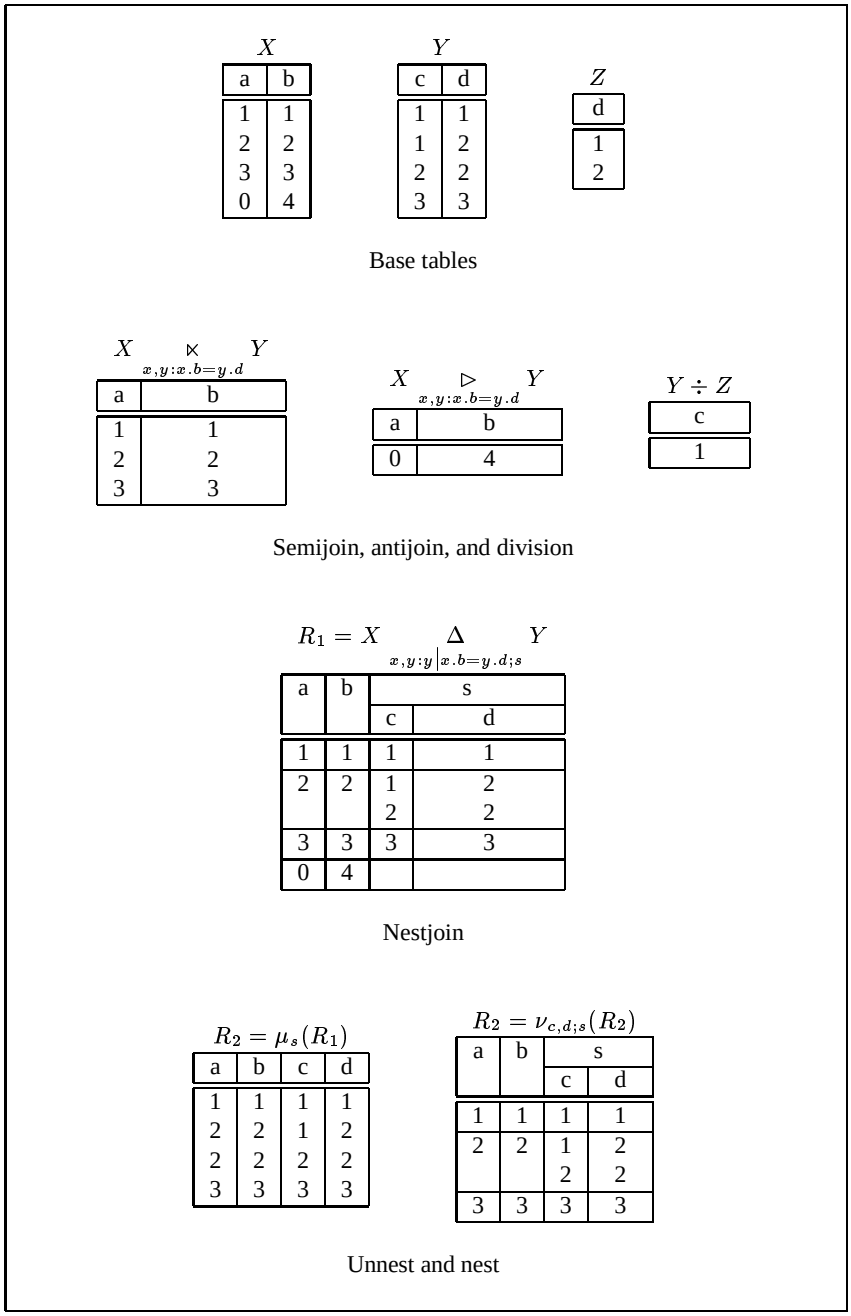
3.4 Semantics

In this section, we give the semantics of some of the ADL operators. We first list some conventions:

- We use the following as synonyms: type and domain; tuple and record; attribute and field; label and attribute name; query and expression.
- A **table type** is a set-of-tuple type. A **table** is a value that is of some table type. Values stored in the database are tables, and are called **base tables**. Base table names are denoted by capitals R, S, for relations, or X, Y, Z, in case tables are of a more complex type. Attributes may be atomic, tuple-valued, or set-valued. In examples, attribute names a, b often denote atomic or tuple-valued attributes; attribute names c, d, or s are often used to denote set-valued attributes.
- A list of attribute names is denoted as a label sequence of the form $a_1, \ldots, a_n$. Capitals A and B are used as abbreviations of attribute lists $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, respectively. Concatenation of attribute lists A and B is denoted as AB, and also as A, B. The concatenation AB denotes the attribute list $a_1, \ldots, a_n, b_1, \ldots, b_m$. An expression aB or a, B denotes the concatenation of label a and attribute list B, resulting in $a, b_1, \ldots, b_m$.
- For a table expression (an expression of some table type), the **schema** is the set of top-level attribute names. In algebraic expressions, often base table names are used to denote the corresponding schema. For example, in the expression $\pi_X(X \bowtie Y)$, the projection list X denotes the list of top-level attribute names of table X. The expression $Sch(T)$ delivers the schema of table T.
- For predicates p, the expression $Attr(p)$ delivers the set of attribute names referenced in p.
- The expression $\mathbf{select}\ f\ \mathbf{from}\ x\ \mathbf{in}\ e\ \mathbf{where}\ p$ is also denoted as:

  $$\Gamma[x : f|p](e)$$

  The above operator Γ is called the **collect** operator. The symbol $|$ serves to separate predicate p from function f.
- The symbol $\equiv$ is used to denote equivalence of expressions.
- The expression $FV(e)$ denotes the set of variable names that occur free in e.
- Whenever predicate and function names are parameterized with variable names, it holds that the actual set of variables that occur free in the predicate is a subset of the set of variable names listed. For example, predicate $p(x, y)$ denotes a predicate p is which variables x and/or y occur free, but no others.



X

a	b
1	1
2	2
3	3
0	4

Y

c	d
1	1
1	2
2	2
3	3

Z

d
1
2

Base tables

$X \underset{x,y:x.b=y.d}{\ltimes} Y$

a	b
1	1
2	2
3	3

$X \underset{x,y:x.b=y.d}{\rhd} Y$

a	b
0	4

$Y \div Z$

c
1

Semijoin, antijoin, and division

$R_1 = X \underset{x,y:y\,|\,x.b=y.d;s}{\Delta} Y$

a	b	s	
		c	d
1	1	1	1
2	2	1	2
		2	2
3	3	3	3
0	4		

Nestjoin

$R_2 = \mu_s(R_1)$

a	b	c	d
1	1	1	1
2	2	1	2
2	2	2	2
3	3	3	3

$R_2 = \nu_{c,d;s}(R_2)$

a	b	s	
		c	d
1	1	1	1
2	2	1	2
		2	2
3	3	3	3

Unnest and nest

Figure 3.1: Example operations

Below, we give the semantics of some of the ADL operators. In Figure 3.1, we give some examples that illustrate the effect of the semijoin, antijoin, division, nestjoin, unnest, and nest.

Definition 3.5 Semantics

1. (Element)

   $\mathbf{el}(\{e\}) = e$

   The operand of **el** is a singleton set, the element of which is delivered as result.

2. (Flatten)

   $\bigcup(e) = \{x \mid \exists s \in e \bullet x \in s\}$

   Expression e denotes a set of sets; the operator flatten delivers the set that consists of elements of elements of e.

3. (Product)

   $e_1 \times e_2 = \{x_1 +\!+ x_2 \mid x_1 \in e_1 \wedge x_2 \in e_2\}$

   The product operator is the extended Cartesian product operator of the relational model. Both e_1 and e_2 are tables; it is required that the schemas of e_1 and e_2 are disjoint.

4. (Collect)

   $\Gamma[x : f(x)|p(x)](e) = \{f(x) \mid x \in e \wedge p(x)\}$

   The collect operator delivers elements of operand e that satisfy predicate p, modified according to function f.

5. (Selection)

   $\sigma[x : p(x)](e) = \{x \mid x \in e \wedge p(x)\}$

   Selection restricts operand e according to predicate p.

6. (Application)

   $\alpha[x : f(x)](e) = \{f(x) \mid x \in e\}$

   Operator map applies function f to every element of operand e.

7. (Projection)

   $\pi_A(e) = \{x[A] \mid x \in e\}$

   The operand e of the projection operator π is a table; it is required that attribute list A is a (not necessarily consecutive) subsequence of the schema of operand e.

8. (Join)

   $$e_1 \underset{x_1,x_2:p(x_1,x_2)}{\bowtie} e_2 = \{x_1 \mathbin{++} x_2 \mid x_1 \in e_1 \wedge x_2 \in e_2 \wedge p(x_1, x_2)\}$$

   The join corresponds to a selection on the Cartesian product; the schemas of the operands have to be disjoint.

9. (Semijoin)

   $$e_1 \underset{x_1,x_2:p(x_1,x_2)}{\ltimes} e_2 = \{x_1 \mid x_1 \in e_1 \wedge \exists x_2 \in e_2 \bullet p(x_1, x_2)\}$$

   The semijoin delivers those left operand tuples for which there is some tuple in the right operand such that the join predicate holds.

10. (Antijoin)

    $$e_1 \underset{x_1,x_2:p(x_1,x_2)}{\rhd} e_2 = \{x_1 \mid x_1 \in e_1 \wedge \nexists x_2 \in e_2 \bullet p(x_1, x_2)\}$$

    The antijoin is the complement of the semijoin operator. It delivers those left operand tuples for which there is no tuple in the right operand such that the join predicate holds.

11. (Nestjoin)

    $$e_1 \underset{x_1,x_2:f(x_1,x_2)\,|\,p(x_1,x_2);a}{\Delta} e_2 =$$

    $$\{x_1 \mathbin{++} \langle a = X\rangle \mid x_1 \in e_1 \wedge X = \{f(x_1, x_2) \mid x_2 \in e_2 \wedge p(x_1, x_2)\}\}$$

    The nestjoin is a new operator that combines grouping, join, and function application. For each left operand tuple x_1, the set of right operand tuples that satisfy predicate p is determined. For each element x_2 in this set, the expression $f(x_1, x_2)$ is evaluated, giving the set X, and in the result x_1 is concatenated with the unary tuple $\langle a = X\rangle$. Attribute name a may not occur in the schema of e_1. The left nestjoin operand must be a table, however, the right operand may be a set of arbitrary type.

12. (Division)

    $$e_1 \div e_2 = \{x_1[A] \mid x_1 \in e_1 \wedge e_2 \subseteq \{x_1'[B] \mid x_1' \in e_1 \wedge x_1[A] = x_1'[A]\}\}$$

    Division is an operation that is difficult to understand. Both operands are tables. The schema of the dividend e_1 is AB, that of the divisor is B (the underlying types of B attributes being compatible). In division, the tuples of the left operand e_1 are grouped according to the A attribute values, and then it is checked whether the corresponding set of B attribute values is a superset of the divisor e_2. The result consists of those left operand (dividend) tuples such that the latter condition holds, restricted to their A attribute values.

13. (Nest)

    $$\nu_{A;a}(e) = \{x[B] \mathbin{++} \langle a = W\rangle \mid x \in e \wedge W = \{x'[A] \mid x' \in e \wedge x'[B] = x[B]\}\}$$

The schema of operand e is AB; a is a label that does not occur in attribute list B. Nest groups the operand tuples according to the B attribute values, i.e., according to the attributes that do not occur in attribute list A, and then, for each partition, concatenates the tuple $x[B]$ with the unary tuple $\langle a = W \rangle$, in which W denotes the set of $x[A]$ tuples belonging to each partition.

14. (Unnest)

    $\mu_a(e) = \{x' +\!\!+ x[B] \mid x \in e \wedge x' \in x.a\}$

    The schema of operand e, which is a table, is aB; attribute a is of a table type as well. Unnest concatenates each tuple in the operand, restricted to its B attribute values, with the tuples that are present in its a attribute.

□

Definitions of, for example, set (comparison) operators and predicates are straightforward, and not given here. Below, we list some obvious equivalences for the purpose illustration.

Rule 3.1 Equivalences

- $\Gamma[x : x|p(x)](X) \equiv \sigma[x : p(x)](X)$
- $\Gamma[x : f(x)|true](X) \equiv \alpha[x : f(x)](X)$
- $X \underset{x,y:p(x,y)}{\bowtie} Y \equiv \sigma[v : p(v[X], v[Y])](X \times Y)$
- $X \underset{x,y:p(x,y)}{\ltimes} Y \equiv \pi_X(X \underset{x,y:p(x,y)}{\bowtie} Y)$
- $X \underset{x,y:p(x,y)}{\triangleright} Y \equiv X - (X \underset{x,y:p(x,y)}{\ltimes} Y)$

3.5 Transformation goal

In this section, we state the goal in the transformation of OSQL expressions. We first give some definitions concerning operators and expressions, and present a classification of expressions. Next, we describe the transformation goal.

3.5.1 Classification of expressions

Definition 3.6 Operators

- The collection of **set operators** includes the standard set (comparison) operators, and also flatten, nest, unnest, and projection. Set operators are operators that have set-valued operands, and do *not* allow for nesting of expressions (see below).
- The operator Γ and its simplifications α and σ, quantifiers $\exists$ and $\forall$, and also the various join operators are called **iterators**. Iterators are those operators that allow for nesting of expressions. Note that projection π is not an iterator.

- Expression e in $\Gamma[x : e](o)$, where e abbreviates some expression $e_1|e_2$, in $\sigma[x : e](o)$ and $\alpha[x : e](o)$, in $\exists x \in o \bullet e$ and $\forall x \in o \bullet e$, and in $o_1\ \theta_{x_1,x_2:e}\ o_2$, where θ denotes an arbitrary join operator, is called the **parameter expression** of the iterator concerned. A variable x, x_i is called a **loop variable**; an expression o, o_i is called an **iterator operand**.

- For a quantification ($\exists x \in o \bullet e$) or ($\forall x \in o \bullet e$), operand o is also called the **quantifier range**, and e the **quantifier scope**.

□

Definition 3.7 Expressions

- An expression is said to **occur nested** if it is contained in some iterator parameter expression.

  Expressions that occur nested are called embedded expressions, nested occurrences, and also nested expressions; the latter are not to be confused with nested expressions as defined below.

- A **nested expression** is an expression that contains nested occurrences of operators with set-valued operand(s); otherwise it is called **flat**.

  Operators that have set-valued operands are the set operators and iterators as defined above in Definition 3.6.

  A flat expression corresponds to a expression of relational algebra in the sense that both do not involve nesting. However, a flat expression may contain non-relational operators like flatten, nest, and unnest, and also tuple operators that are not present in relational algebra.

- A **subquery** is an iterator expression that occurs nested and that has a base table or a subquery operand. A nested base table occurrence is *not* by definition considered a subquery.

  A **correlated subquery** is a subquery that contains one or more free variables.

  An expression that contains a subquery is also called a **subquery expression**.

- An expression that does not contain any nested base table occurrences is called a **single-table expression**.

  Single-table expressions may contain nested iterators and nested set operators; the operands of these are set-valued attributes however.

- A **set expression** is an expression that contains nested set operator occurrences with base table operand(s). □

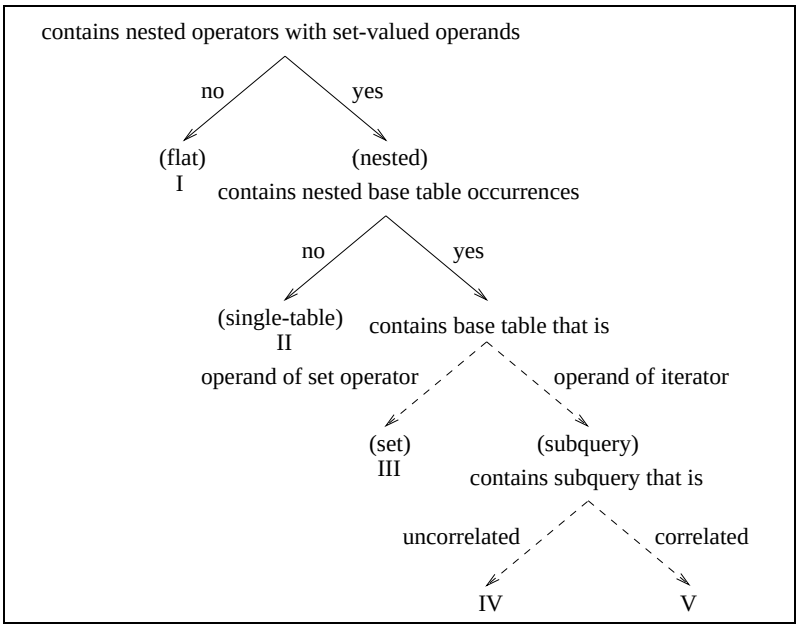


Figure 3.2: Query classification

In Figure 3.2, we have depicted the various types of queries. Five classes are distinguished. The first distinction is that between flat and nested expressions. Nested expressions may contain nested base table occurrences or not. In the latter case, the nested expressions are called single-table expressions. Single-table expressions may still contain nested iterators or set operators, but the operands are set-valued attributes instead of base tables. Nested expressions that contain base table occurrences are classified as set and/or subquery expressions. In a set expression, a nested base table is the operand of a set operator; in a subquery expression, a nested base is the operand of an iterator. The two classes are not disjoint. A final distinction is that between expressions that involve either correlated or uncorrelated subqueries. Below, we give some examples.

Example 3.3 Classification of expressions

Type I: flat $\sigma[x : x.a = 1](X)$

The above expression is a simple flat (relational) expression that does not contain any nested set operators or iterators.

Type II: single-table

- $\sigma[x : \exists z \in x.c \bullet z = 1](X)$
  This expression, which does not contain any nested base table occurrences, contains a nested existential quantification of which the operand is a set-valued attribute.
- $\sigma[x : \exists z \in x.c \bullet x.a = z.a](X)$
  The above expression contains a nested, correlated quantifier expression.

- $\alpha[x : x \textbf{ except } (c = \sigma[z : z.name = \ 'Joe'](x.c))](X)$
  The set-valued attribute c of X is restricted by means of a nested selection.
- $\sigma[x : x.c \subseteq x.d](X)$
  The selection predicate of the above expression is not atomic, but complex, involving a set comparator with operands that are set-valued attributes.

Type III: set

- $\sigma[x : x.c = Y](X)$
  Note that this set expression corresponds to the query that selects suppliers (X) that supply all parts (Y). Parts supplied are represented as a set-valued attribute (c).
- $\alpha[x : x.c - Y](X)$
  This query might be considered as one that checks referential integrity, for each $x \in X$ delivering the references in c that do not occur in table Y.

Type IV: uncorrelated subquery $\alpha[x : x.c \cup \sigma[y : y.a = 1](Y)](X)$
The nested selection does not contain x as a free variable.

Type V: correlated subquery

- $\sigma[x : \exists y \in Y \bullet x.a = y.a](X)$
  The above expression is a simple (relational) subquery expression that can be mapped to the semijoin operator. However, attributes a need not be atomic.
- $\sigma[x : \exists z \in x.c \bullet \nexists y \in Y \bullet z.a = y.a](X)$
  In this query, iteration over a base table is nested within iteration over a set-valued attribute.

Note that the five classes are not disjoint. The classes of flat and nested expressions, and the classes of single-table and subquery expressions are disjoint, though.

3.5.2 Transformation

We assume that each and every OSQL expression can be evaluated as it is, i.e., as written by the user, using a naive, nested-loop execution strategy for nested expressions. The goal is to improve over naive query processing, in which a nested-loop strategy is followed and each operator is evaluated naively. This can be done in two ways:

- Improve the efficiency of single operators. This is the task carried out at the physical level, that provides efficient implementations. One logical operator can be mapped to several physical operators; a physical operator may implement several logical operators.
- Improve the efficiency of operator sequences. This is done at the logical level by changing the order within operator sequences (algebraic rewriting), or by replacing (combinations of) operators by others.

The goal in translation is to transform nested expressions into flat or single-table expressions. Explicit iteration is to be transformed into set operators, and nested base table occurrences are removed from parameter expressions by rewriting into join expressions, as much as possible. In addition, in the transformation the overall goal is to keep the number, size, and cardinality of intermediate results as small as possible. The result of the translation should be as efficient as possible—we combine translation with optimization, as also was proposed in [Naka90]. Below, we briefly, and inevitably vaguely, describe some options for handling the various types of expressions in the transformation.

Type I: flat Flat OSQL queries are left as they are, and optimized by means of algebraic rewriting, as much as possible.

Type II: single-table Single-table queries, expressions that do not contain nested base table occurrences, may contain nested iterators or nested set operators that have set-valued operands. For example, consider the expression:

$$\sigma[x : \exists z \in x.c \bullet z = 1](X)$$

The above expression can be evaluated by means of a straightforward nested-loop execution strategy. However, it is also possible to transform the query into an expression with a nested set operator:

$$\sigma[x : 1 \in x.c](X)$$

The replacement of iteration by set operators allows for different execution strategies. In case the comparison operator is $<$ instead of $=$, such a transformation is not possible, which means that quantification must be included in the algebra. Alternatively, type II queries may be handled by the unnesting of set-valued attributes.

Type III: set Type III concerns expressions that contain some nested base table occurrence that is the operand of a nested set (comparison) operator. Consider the expression:

$$\sigma[x : x \in Y](X)$$

The query may be executed as it is, (efficiently) evaluating the set difference operator for each $x \in X$. Also, the query may be rewritten into:

$$\sigma[x : \exists y \in Y \bullet x = y](X)$$

and then into a join operation:

$$X \underset{x,y:x=y}{\ltimes} Y$$

The above transformation is standard in the relational context, but it is not certain whether all nested expressions of type III can be handled this way; this is one of the research topics of this thesis.

As with uncorrelated subqueries (see below), we may need the **with**-clause to ensure that constant expressions are evaluated only once. For example, the expression:

$$\alpha[x : x.c \cup \bigcup(Y)](X)$$

is rewritten into:

$$\alpha[x : x.c \cup Y'](X) \textbf{ with } Y' = \bigcup(Y)$$

so that the expression $\bigcup(Y)$ is not evaluated for each $x \in X$, but only once.

Type IV: uncorrelated subquery Uncorrelated subqueries are constants, which can be evaluated independently. Independent evaluation can be expressed by means of a local definition facility. For example, the expression:

$$\alpha[x : x.c \cup \sigma[y : y.a = 1](Y)](X)$$

is rewritten into:

$$\alpha[x : x.c \cup Y'](X) \textbf{ with } Y' = \sigma[y : y.a = 1](Y)$$

so that Y' is evaluated only once.

Type V: correlated subquery Correlated subqueries are removed whenever possible. The removal of correlated subqueries means a shift from tuple- to set-oriented query processing. In a naive evaluation strategy, queries that have subqueries are evaluated by means of a nested-loop execution, in which the subquery is evaluated for each of the tuples in the surrounding iterator operand(s). After the transformation of queries with subqueries into join queries, other execution strategies can be employed. It is uncertain whether in complex object models removal of all subqueries is possible; again, this issue is one of the research topics of this thesis. See for instance the second expression of type V given in Example 3.3 above:

$$\sigma[x : \exists z \in x.c \bullet \nexists y \in Y \bullet z.a < y.a](X)$$

The iteration over Y is embedded within iteration over the set-valued attribute *c* of X. The explicit iteration can be removed by the introduction of a nested antijoin operation:

$$\sigma[x : \exists z \in (x.c \underset{z,y:z.a<y.a}{\triangleright} Y) \bullet true](X)$$

but the result still is a subquery expression.

3.6 Cost model

In this thesis, we will not, and cannot provide a detailed cost model. As explained in the previous chapter, traditionally only in the phase of plan compilation, i.e., during translation of a logical algebra expression into the physical algebra, a detailed cost model incorporating physical database characteristics is used. Logical optimization is based on heuristics, using rewrite rules that are assumed to be generally beneficial. In logical rewriting, the cost model is implicit; simple performance measures are the number of times base tables are accessed, the number and size of intermediate result tables, etc. Generally speaking, in translation/optimization, we strive to:

1. Avoid *duplicate* computations, which are computations that have been carried out before.

2. Avoid *unnecessary* computations, which are those that do not contribute to the result, in the end.

On the logical level, we try to reduce the number, size, and cardinality of intermediate results. We give some examples:

- Common subexpressions are evaluated only once.

- Preference is given to cheap operators. For instance, the expression $\sigma[x : \nexists y \in Y \bullet x.a = y.a](X)$ is translated into the antijoin $X \triangleright_{x,y:x.a=y.a} Y$, and not into $(X \bowtie_{x,y:x.a \neq y.a} Y) \div Y$, an expression that contains the expensive operator division and also two occurrences of base table Y.

- It is avoided to compute results thrown away later on anyway. Consider for example the expression $\sigma[x : false](E)$, or $E \bowtie_{x,y:x.a \neq y.a} \emptyset$. To retrieve or evaluate subexpression E is useless, because the result is empty anyway.

- Non-qualifying data are eliminated as soon as possible, i.e., projections and selections are pushed through, reducing the cardinality and size (tuple width) of operands.

On the physical level, it is tried to reduce the number and the size (number of attributes) of tuples actually retrieved and/or compared. This can be done by the use of access methods like indices, by caching of results, etc.

3.7 Summary

We have described the end-user query language OSQL and the algebraic language ADL that serves as the intermediate language between OSQL and the physical access language. OSQL is a prototype of SQL-like query languages for data models supporting complex objects. Emphasis in the definition of OSQL is on aspects we consider important with respect to optimization. ADL is a hybrid language: it contains pure algebraic operators in the style of the NF^2 algebra of [ScSc86], as well as calculus-like constructs such as quantifiers. In

complex object models, the presence of set-valued attributes places specific demands on the functionality of the intermediate language. In addition to well-known algebraic operators such as selection and join, the intermediate language has to support nesting of operations, set-comparison operators, and also quantifiers.

We have set the main goal for this thesis: the transformation of nested OSQL expressions into efficient set-oriented expressions. We consider an OSQL expression nested if it contains nested operations having sets (base tables or set-valued attributes) as operands; otherwise it is considered flat. Transformation of nested expressions into flat (join) expressions is in line with the idea that underlies the classical relational calculus-to-algebra translation, i.e., to move from tuple- to set-oriented query processing. Tuple-oriented query processing generally comes down to nested-loop processing; performance can be much improved by the use of clever access algorithms that operate on sets as a whole.

The language ADL is defined as an extension of OSQL; possible transformations can be laid down in the form of equivalence rules. This approach to translation/optimization enables to prove the correctness of transformation steps, and also gives more insight into the decisions taken.

In the next chapter, as a starting point, we investigate the transformation of (a restricted form of) relational calculus into relational algebra. In traditional translation algorithms [Codd72, CeGo85], little attention is paid to the performance of the resulting algebraic expressions; the task of obtaining acceptable performance has been left to the optimizer. To enable efficient transformation of arbitrary relational calculus expressions, relational algebra, which is a subset of the language ADL, is extended with some non-standard join operators. In the chapters thereafter, more advanced features are discussed.

References

[BaBZ93] Balsters, H., R.A. de By, and R. Zicari, "Typed Sets as a Basis for Object-Oriented Database Schemas," *Proceedings ECOOP*, Kaiserslautern, 1993.

[BaVr91] Balsters, H. and C.C. de Vreeze, "A Semantics of Object-Oriented Sets", *Proceedings 3rd International Workshop on Database Programming Languages"* , Nafplion, Greece, 1991.

[BaCD92] Bancilhon, F., S. Cluet, and C. Delobel, *A Query Language for O_2*, in: *Building an Object-Oriented Database System—The Story of O_2*, eds. F. Bancilhon, C. Delobel, and P. Kanellakis, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[Catt93] R.G.G. Cattell, ed., *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[CeGo85] Ceri, S. and G. Gottlob, "Translating SQL into Relational Algebra: Optimization, Semantics, and equivalence of SQL Queries," *IEEE Transactions on Software Engineering*, Vol. SE–11, No. 4, April 1985.

[Codd72] Codd, E.F., "Relational Completeness of Data Base Sublanguages," in: *Data Base Systems*, ed. R. Rustin, Prentice Hall, 1972.

[Kamb85] Kambayashi, Y, "Cyclic Query Processing," in: *Query Processing in Database Systems*, eds. W. Kim, D.S. Reiner, and D.S. Batory, Springer Verlag, 1985, pp. 62–78.

[Naka90] Nakano, R., "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS*, 15(4), December 1990, pp. 518–557.

[PiAn86] Pistor, P., and Andersen, F., "Designing a Generalized NF^2 Model with an SQL-Type Language Interface," *Proceedings VLDB*, Kyoto, August 1986.

[RoKB88] Roth, M.A., H.F. Korth, and D.S. Batory , "SQL/NF: A Query Language for ¬ 1NF Relational Databases," *Information Systems*, 12(1), March 1987, pp. 99–114.

[ScSc86] Schek, H.-J., and M.H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems* , 11(2), 1986, pp. 137–147.

Chapter 4

Translation of relational calculus to relational algebra

The language OSQL as defined in the previous chapter can be considered as an extension of relational query languages. Therefore, in this chapter, we first discuss translation and optimization of a relational calculus language. In the chapters to follow, the more advanced language features are discussed. Motivation of the work presented in this chapter is that, in general, in relational translation algorithms, e.g. [Codd72, CeGo85], little attention is paid to efficiency.

In the literature, several algorithms for the translation of SQL into relational algebra can be found. SQL may be translated into relational algebra directly [CeGo85], or using relational calculus as intermediate language [Ullm89, PBGG89]. The reduction algorithm of [Codd72] provides a way to translate relational calculus expressions into algebraic expressions. Calculus constructs that are hard to translate efficiently are universal quantification and disjunction. In [CeGo85], universal quantification and disjunction are handled by means of the set difference and the union operator, respectively. In [Codd72], disjunction is handled by means of set union also; for universal quantification the division operator is used. A disadvantage of the above-mentioned approaches to universal quantification and disjunction is that in the resulting algebraic expressions (base) tables are accessed multiple times. In [Bry89], an improvement in translating relational calculus expressions into algebraic expressions is proposed. In this proposal, as far as possible, (base) tables are accessed only once; however, universal quantification and disjunction are handled separately, and no complete translation algorithm is given. In this chapter, we follow a systematic approach to provide a complete and efficient translation of relational calculus into relational algebra.

To provide a point of reference for further discussions, in Section 4.1 we first describe the reduction algorithm of [Codd72]. To avoid problems with safety, the calculus language we use is somewhat restricted. Next, we discuss some alternative ways of handling universal quantification and disjunction in Section 4.2. In Section 4.3, we give a description

of an algorithm to transform relational calculus into relational algebra extended with some non-standard join operators; the goal is to obtain an efficient translation, i.e. a translation that results into efficient relational algebra expressions. Roughly, the transformation consists of a phase of preprocessing, described in Section 4.4, a phase of translation, discussed in Section 4.5, and a phase of postprocessing, to which we do not pay much attention. The algorithm for the actual translation of the calculus into the algebra is presented in the form of a set of rewrite rules. As proposed in [Naka90], we strive to perform optimization *during* translation; expressions are rewritten in a language that is a mixture of calculus and algebra. In Section 4.6, we discuss some of the decisions taken in the translation, and we describe additional rewrite techniques and alternative translation rules that might give better results. The algebraic expressions that are the result of translation sometimes can be rewritten further; in Section 4.7, we give some algebraic equivalence rules for the transformation of join expressions. Next, in Section 4.8 we evaluate the results on the basis of some extensive rewrite examples that are given in Appendix B. A summary is given in Section 4.9.

4.1 Codd's reduction algorithm rephrased

In this section, we describe in a simple but formal way the algorithm of Codd for the translation of relational calculus into relational algebra [Codd72]; this description serves as a point of reference for further discussions. To explain the essence of the algorithm, a quite restricted form of relational calculus suffices.

Definition 4.1 Relational calculus The language RC is given by the following syntax:

RC	::=	i	(top level expressions)
i	::=	$\sigma[x : p](e) \mid \pi_A(e)$	(iterators)
e	::=	$c \mid i \mid s$	(expressions)
s	::=	$e_1 \cup e_2 \mid e_1 \times e_2$	(set expressions)
t	::=	$x.a$	(tuple expressions)
p	::=	$c\ cop\ t \mid t_1\ cop\ t_2 \mid$	(predicates)
		$\neg p \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid$	
		$\exists x \in e \bullet p \mid \forall x \in e \bullet p$	
cop	::=	$< \mid \leq \mid = \mid \geq \mid >$	(comparison operators)

□

In the syntax above, c denotes base tables and constants of atomic type also. However, in comparisons $c\ cop\ t$, c is of atomic type only.

An expression $\pi_A(\sigma[x : p](e))$ corresponds to the so-called alpha expression of [Codd72], which denotes a projection on a selection. For example, the expression:

$$\pi_a(\sigma[x : x.b = 2](X))$$

corresponds to the alpha expression:

$$(x[a]) : x \in X \wedge x[b] = 2$$

The language defined by the syntax above differs from standard relational calculus in that all variables are range-restricted, i.e. the language is safe. In the calculus of [Codd72], the basic construct is the alpha expression (that is, set comprehension), in which tuple variables are bound by means of range-coupled quantifiers or by means of range terms of the form $x \in X$, where X is a relation. Formulas of the language have to satisfy certain restrictions to ensure that all variables have clearly defined ranges. To simplify our presentation, we do not use set comprehension, but the selection operator instead.[1] As a consequence, we have to include the Cartesian product and set union in our language; these operators cannot be expressed by means of selection. Furthermore, in our language projections are made explicit. In the calculus of [Codd72], the target list (a list of tuple variables or indexed tuple variables, e.g. the expression $(x[a])$ in the alpha expression above) indicates the attribute values needed in the result; we use the projection operator.

For reasons of convenience, we allow to apply the equality operator to tuples ($x = y$) and also to subtuples ($x[A] = y[A]$), which is not allowed in standard relational languages. Recall that an expression $x[A]$ denotes tuple projection. In the sequel, a tuple projection $x[A]$ will also be denoted as x_A.

Relational algebra RA, as usual, consists of operators π, σ, $\bowtie$, and $\div$ together with the set operators $\times$, $\cup$, $\cap$, and $-$. The proof of equivalence of RA and RC consists of two parts: providing a translation of RA into RC, and vice versa. The translations are given below.

4.1.1 Reduction of RA into RC

The reduction of RA to RC is rather trivial. We only show the translation of set difference, set intersection, and division:

- Let X and Y be relations that have identical schemas, then:
  $X - Y \equiv \sigma[x : \forall y \in Y \bullet x \neq y](X)$
  $X \cap Y \equiv \sigma[x : \exists y \in Y \bullet x = y](X)$
- Let $X(A, B)$ and $Y(B)$ be relations, which means that the schema of Y is a subschema of that of X, then:
  $X \div Y \equiv \pi_A(\sigma[x : \forall y \in Y \bullet \exists z \in X \bullet x[A] = z[A] \wedge z[B] = y](X))$

4.1.2 Reduction of RC to RA

Reduction of the calculus to the algebra, following the ideas of [Codd72], consists of the following steps:

[1] A selection $\sigma[x : p](X)$ is equivalent to the safe set comprehension expression $\{x \in X \mid p\}$.

1. Remove nested selections and projections. In Section 4.4.1, it is explained how this is done.

2. Transform predicates into disjunctive Prenex Normal Form (PNF). A predicate is in PNF if it is of the form:

   $$\exists/\forall x_1 \in X_1 \bullet \exists/\forall x_2 \in X_2 \bullet \cdots \exists/\forall x_n \in X_n \bullet p$$

   and p is quantifier-free expression. If p is a conjunction of disjunctions, the predicate is said to be in Conjunctive Prenex Normal Form (CPNF); if p is a disjunction of conjunctions, the predicate is in Disjunctive Prenex Normal Form (DPNF). In the transformation, negation is pushed through as far as possible.

3. Apply the following rules, exhaustively, from left to right:

   (a) $\sigma[x : \exists y \in Y \bullet p(x,y)](X) \equiv \pi_X(\sigma[v : p(v_X, v_Y)](X \times Y))$

   (b) $\sigma[x : \forall y \in Y \bullet p(x,y)](X) \equiv (\sigma[v : p(v_X, v_Y)](X \times Y)) \div Y$

   (c) $\sigma[x : p \vee q](X) \equiv \sigma[x : p](X) \cup \sigma[x : q](X)$

   (d) $\sigma[x : p \wedge q](X) \equiv \sigma[x : p](\sigma[x : q](X))$

For reasons of simplicity, it is assumed that no name clashes occur in the formation of Cartesian products in the rules presented above. In the rules for existential and universal quantification, predicates $p(x, y)$ have to be adapted syntactically to take into account the tuple concatenation that is implicit in the product. In the predicate, each occurrence of variables x and y is replaced by variable v, projected on the attributes of left- and right-hand operand, denoted as v_X and v_Y, respectively.

With respect to the reduction algorithm given above, we note the following.

- As pointed out elsewhere [JaKo84, Klug82], empty ranges must be taken into account in the transformation of predicates expressions with range-coupled quantifiers into PNF. For example, the transformation of $p \wedge \forall x \in X \bullet q$ into $\forall x \in X \bullet p \wedge q$ is incorrect; a correct transformation is:

  $$\textbf{if } X = \emptyset \textbf{ then } p \textbf{ else } \forall x \in X \bullet p \wedge q$$

  (it is assumed that x is not free in p). This type of expression clearly is not part of standard relational algebra, but can easily be included.

- In the transformation of select expressions of the form $\sigma[x : \forall y \in Y \bullet p(x,y)](X)$, empty ranges must be dealt with as well. The correct transformation is:

  $$\textbf{if } Y = \emptyset \textbf{ then } X \textbf{ else } (\sigma[v : p(v_X, v_Y)](X \times Y)) \div Y$$

- In [Codd72], conjunctive queries are handled by set intersection:

$$\sigma[x : p(x) \wedge q(x)](X) \equiv \sigma[x : p(x)](X) \cap \sigma[x : q(x)](X)$$

  The translation of conjunctive queries by means of composition of selections as indicated in Rule 3(d) above is an easy improvement from the viewpoint of performance.

So far, we have sketched the basic translation algorithm. Below, we give an example translation, presented in the form of a sequence of rewrite steps (refraining from the necessary substitution of variable names).

Rewriting example 4.1 Translation of calculus into algebra

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z)](X) \\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet p(x,y) \wedge q(x,z)](X) \textbf{ (moving } \exists \textbf{ out)} \\
&\quad\equiv\quad \sigma[x : \forall y \in Y \bullet \nexists z \in Z \bullet p(x,y) \wedge q(x,z)](X) \textbf{ (moving } \neg \textbf{ in)} \\
&\quad\equiv\quad \sigma[x : \forall y \in Y \bullet \forall z \in Z \bullet \neg p(x,y) \vee \neg q(x,z)](X) \textbf{ (moving } \neg \textbf{ in)} \\
&\quad\equiv\quad (\sigma[x : \forall z \in Z \bullet \neg p(x,y) \vee \neg q(x,z)](X \times Y)) \div Y \textbf{ (translating outer } \forall\textbf{)} \\
&\quad\equiv\quad ((\sigma[x : \neg p(x,y) \vee \neg q(x,z)]((X \times Y) \times Z)) \div Z) \div Y \textbf{ (translating inner } \forall\textbf{)} \\
&\quad\equiv\quad ((\sigma[x : \neg p(x,y)]((X \times Y) \times Z) \cup \\
&\qquad\qquad \sigma[x : \neg q(x,z)]((X \times Y) \times Z)) \div Z) \div Y \textbf{ (handling } \vee \textbf{ by } \cup\textbf{)}
\end{aligned}$$

The resulting algebraic expression is inefficient due to:

- the creation of the full Cartesian product of the base relations that occur in the query,
- the use of the division in a standard way, and
- the use of the union operator to handle disjunction.

These issues will be further discussed in the following section. The inefficiency introduced in the translation phase is expected to be reduced in the optimization phase. If we realize that the above expression is equivalent to, for example, the much simpler and more efficient algebraic expression:

$$X - ((X \underset{x,y:p(x,y)}{\ltimes} Y) \underset{x,z:q(x,z)}{\ltimes} Z)$$

this seems quite a hard task. In this chapter, the goal is to investigate possibilities to improve the translation of [Codd72] as described and illustrated above.

4.2 Discussion

As explained in the previous section, the reduction algorithm of Codd may result in inefficient algebraic expressions. Efficiency can, and must be improved by means of logical and/or cost-based optimization. Logical optimization is defined as the attempt to improve efficiency by rewriting an expression into an equivalent expression that is less costly, without making use of specific database characteristics. Logical optimization generally involves pushing through selections and projections; determination of the join order usually is done cost-based. With regard to logical optimization, three approaches can be distinguished, in which optimization is performed either before, during, or after translation; these are described below.

In the first place, a naive translation may be followed by a phase of algebraic optimization (optimization *after* translation). Using the algorithm given in the previous section, translation of the calculus language into the algebra is straightforward—only a few rewrite rules are needed. However, the result of the translation is inefficient in general; in the phase of algebraic optimization the task is reduce the inefficiency as much as possible. Logical optimization of algebraic expressions produced by a naive translation algorithm may involve complicated algorithms [Ullm89].

Secondly, optimization can be carried out *during* translation, which implies the rewriting of expressions in a mixture of calculus and algebra. In [Naka90] it is claimed that "translation with optimization is more effective and more promising than optimization after translation, because it seems difficult to optimize complicated algebraic expressions (p. 519)." Optimization of algebraic expressions is said to be difficult because complicated algebraic expressions may lose essential structural information about a query, and also because it is difficult to control the sequence of rewrite rule application, i.e. to find the meta-rules to guide the rewrite process. In translation with optimization, many more rewrite rules are needed than in a naive translation. In [Naka90], a distinction has been made between a set of basic rewrite rules and a set of so-called heuristic rewrite rules; the heuristic rewrite rules are given higher priority than the basic rewrite rules.

Thirdly, it is possible to perform logical optimization *before* translation. In [Bry89], it is proposed to transform predicates into the Miniscope Normal Form (MNF) instead of the Prenex Normal Form (PNF). Bry claims that MNF is important because it gives rise to an improvement of the algebraic translation. A formula is in MNF if, and only if, none of its quantified subformulas F contains an atom in which only variables quantified outside F occur. In the approach of [Bry89], before translation predicates are rewritten into MNF.

Another approach to improve efficiency is to extend relational algebra with new operators, as for example in [Daya87A, Bry89]. In the past, relational algebra as defined in [Codd72] was extended for two reasons: to enhance expressive power, and to improve performance. To enhance expressive power, for example, the outerjoin has been added to overcome problems with dangling tuples; aggregate functions have been included for statistical computations. To improve performance, several non-standard join operators have been proposed. The semi- and antijoin can be used for existentially and universally quantified chain or tree queries, i.e. join queries in which the right-hand join operand tuple values are not needed in the result. In [Bry89], an operator called the constrained outerjoin

is proposed to allow for more efficient processing of disjunctions. The addition of such non-standard join operators, that usually are not commutative and do not associate easily, complicates logical optimization; for each operator new rules have to be discovered and added to the set of rewrite rules.

We give an example that illustrates some of the points made in the discussion above. Recall the calculus expression:

$$\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z)](X)$$

As shown in Rewriting example 4.1, in which the algorithm of Codd is followed, this expression is translated into:

$$((\sigma[x : \neg p(x,y)]((X \times Y) \times Z) \cup \sigma[x : \neg q(x,z)]((X \times Y) \times Z)) \div Z) \div Y$$

To run ahead at things, the calculus expression is easily translated into a more efficient algebraic expression as follows:

Rewriting example 4.2

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z)](X) \\
&\quad \equiv \quad X - \sigma[x : \exists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z)](X) \textbf{ (negation to difference)} \\
&\quad \equiv \quad X - \sigma[x : (\exists z \in Z \bullet q(x,z)) \wedge (\exists y \in Y \bullet p(x,y))](X) \textbf{ (descope)} \\
&\quad \equiv \quad X - \sigma[x : \exists y \in Y \bullet p(x,y)](X \underset{x,z:q(x,z)}{\ltimes} Z) \textbf{ (unnest)} \\
&\quad \equiv \quad X - (X \underset{x,z:q(x,z)}{\ltimes} Z) \underset{x,y:p(x,y)}{\ltimes} Y \textbf{ (unnest)}
\end{aligned}$$

We use set difference for the negation, transform the predicate into MNF, and employ the semijoin operator whenever possible (the rewrite techniques used will be extensively discussed in the sequel). To find the pure algebraic equivalence rules that are needed to translate the former algebraic expression into the latter will be difficult.

In the remainder of this section, we discuss the possibilities to improve the standard translation in some more detail. The discussion focuses on the following issues: (1) the (dis)advantages of combining the phases of translation and optimization, (2) the (dis)advantages of using the Prenex Normal Form or the Miniscope Normal Form, (3) the use of the division operator, and (4) the treatment of disjunctions.

4.2.1 Point of optimization

As explained, logical optimization may take place before, during, or after translation. The (dis)advantages of combining the phases of translation and optimization are not immediately clear. Algebraic optimization of relatively simple expressions that consist of joins (or products), selections, and projections only has been studied extensively in the past. However, as we have seen above, whenever other operators than the regular join or semijoin, for example division (see also Section 4.2.3), come into play, pure algebraic rewriting becomes more and more complicated. In our approach, as in [Naka90], we combine optimization with translation, in the attempt to reach a conclusion about the pros and cons of this approach.

4.2.2 PNF versus MNF

Translation of relational calculus into relational algebra usually begins with the transformation of the calculus expression into conjunctive or disjunctive PNF. The rules used in this transformation are the laws of De Morgan (for pushing negation inwards), the laws for distribution of Boolean connectives, the laws that concern negated quantifiers, and the law of double negation. Application of the laws of De Morgan and the laws for distribution of Boolean connectives may turn a disjunctive query into a conjunctive one, and vice versa; application of the laws for distribution of connectives creates common subexpressions. In [JaKo84], the phase following standardization is called simplification. The Prenex Normal Form offers some specific opportunities for query transformation in that logical equivalence rules may be applied to the matrix of the PNF expression. Rules that are used in simplification are for example the rules of transitivity, for constant propagation and breaking cycles (see Section 4.4.3) and of idempotency, for the removal of redundant computations.

In [Bry89], a different canonical form, called the *Miniscope Normal Form* (MNF) is proposed. A formula is in MNF if and only if none of its quantified subformulas F contain atoms in which only variables quantified outside F occur. In MNF, quantifiers are pushed inwards, reducing variable scopes as much as possible. Also, in [Bry89], universal quantification is rewritten into negated existential quantification, and negation is pushed through. MNF allegedly provides opportunities to improve the translation of the calculus into the algebra. In this translation, the initial Cartesian product is avoided as much as possible. Whenever possible, universal quantification is handled by an operator called complement-join (which is equal to the antijoin), instead of division. In addition, in [Bry89] it is proposed to handle disjunction by means of an operator called the constrained outerjoin. Both operators will be discussed further in the sequel. In [Bry89], no complete translation algorithm is given. Instead, for the translation of calculus expressions into relational algebra, a limited set of rewrite rules concerning nested expressions with quantifiers is given that is claimed to be easily extensible.

4.2.3 Universal quantification

The reduction algorithm of [Codd72] handles universal quantification by means of division. Algebraic rewriting of expressions that contain division is difficult. In addition, division is an expensive operator. Therefore, whenever possible, it is better to avoid division, for example by using the antijoin operator. Below, we further discuss these issues.

Some rules for division

Very few algebraic rewrite rules for expressions that contain division have appeared in the literature. As an exception to the rule, some important equivalence rules are given in [Naka90]; we present them below (in their pure algebraic version; in [Naka90], equivalence rules are presented in the form of rewrite rules that transform calculus expressions into algebraic expressions.) We remark that the rules as given below are difficult to under-

stand; they are merely given for illustrative purposes. The main effect of the rules is the removal of redundant joins from the dividend expression. Let X_r and Y_r denote the list of attributes of X and Y needed in the join $X \bowtie_{x,y:r(x,y)} Y$, respectively.

Rule 4.1 Division

1. $\pi_{XY}((X \underset{p(x,z)}{\bowtie} Z) \underset{q(y,z)}{\bowtie} Y) \div Y \equiv \pi_{XZ_q}(X \underset{p(x,z)}{\bowtie} Z) \div \pi_{Y_q}(Y)$
   if q is a conjunctive equijoin predicate
2. $\pi_{XY}((X \underset{p(x,z)}{\bowtie} Z) \underset{q(y,z)}{\bowtie} Y) \div Y \equiv X \underset{p(x,z)}{\ltimes} (\pi_{Z_p,Z_q}(Z) \div \pi_{Y_q}(Y))$
   if both p and q are conjunctive equijoin predicates
3. $\pi_A((\pi_{XY}((X \underset{p(x,z)}{\bowtie} Z) \underset{q(y,z)}{\bowtie} Y)) \div Y) \equiv \pi_A(\pi_{Z_p,Z_q}(X) \div \pi_{Y_q}(Y))$
   if both p and q are conjunctive equijoin predicates, Z is equal to X or some selection on X, and attribute list A is contained in Z_p (or, equivalently, X_p)

Let us give an example that concerns the classical supplier-supplies-part database. Let $S(sno)$, $SP(sno, pno)$, and $P(pno)$ be the schemas of the supplier, supplies, and part relations. For simplicity, in the example given below we use attribute numbers instead of attribute names so that we do not have to take into account name clashes, and use a slightly adapted form of division in which the attributes concerned are explicitly indicated.

Example 4.1 Select the supplier numbers of the suppliers that supply all parts In the calculus, the query may be expressed as follows:

$$\pi_1(\sigma[x : \forall y \in P \bullet \exists z \in SP \bullet x[1] = z[1] \wedge y[1] = z[2]](SP))$$

The reduction algorithm of [Codd72] as described above results in the expression:

$$\pi_1(\pi_{1,2,3}(\sigma[x : x[1] = x[4] \wedge x[3] = x[5]]((SP \times P) \times SP)) \div [2/1]P)$$

Now, by the application of the third alternative of Rule 4.1 as stated above, and some other algebraic equivalence rules, this expression can be rewritten into the much simpler expression:

$$SP \div [2/1]P$$

As an aside, we remark that in a complex object model the supplier-supplies-part database may be modelled by means of a set-valued attribute of the supplier or of the part relation. Assume we have schemas $S(sno, parts(pno))$ and $P(pno)$ for the (nested) supplier and (flat) part relation, respectively, then the example query may be expressed as $\pi_{sno}(\sigma[x : P \subseteq x.parts](S))$.

Antijoin

To handle universal quantification, different solutions have been sought. One proposal to deal with universal quantification is described in [Daya87A]. As this proposal does not lie within an algebraic framework, it will not be considered here. Another solution is to rewrite universal quantification into negated existential quantification, and to handle the

negation by means of set difference. Both ways of dealing with the universal quantifier, i.e. using the division operator and using set difference, may result in inefficient expressions that are hard to optimize. Below, we show that the antijoin operator can be of good use to handle universal quantification.

The semijoin operator can be looked upon as the natural algebraic equivalent of a nested query that involves an existential quantifier:

$$X \underset{x,y:p}{\ltimes} Y \equiv \sigma[x : \exists y \in Y \bullet p](X)$$

Using a semijoin operation whenever possible in general improves performance; much research has been done on the replacement of join operators by semijoins [Kamb85, Ullm89].

The antijoin [RoGa90], also known as complement-join [Bry89] or as the anti-semijoin [Grae93], can be used to express a query that contains a negated existential quantifier:

$$X \underset{x,y:p}{\triangleright} Y \equiv \sigma[x : \nexists y \in Y \bullet p](X)$$

The antijoin is less known than the semijoin operator, however, it is very useful for processing queries with universal quantifiers. Assume X and Y are tables with attributes a, b, and c, respectively, and consider the simple expression:

$$\sigma[x : \forall y \in Y \bullet x.a \neq y.b](X)$$

Translation of this calculus expression into the algebra may be done by means of the set difference operator, division, or the antijoin operator. We show the result of each:

Set difference Assume that the universal quantifier is rewritten into a negated existential, then the result of rewriting, in which the negation is handled by set difference (we have the equivalence $\sigma[x : \neg p](X) \equiv X - \sigma[x : p](X)$), is:

$$X - \pi_X(X \underset{x,y:x.a=y.b}{\bowtie} Y)$$

Division Handling universal quantification by means of division, as in [Codd72], results in the expression:

$$(X \underset{x,y:x.a\neq y.b}{\bowtie} Y) \div Y$$

Antijoin We simply write:

$$X \underset{x,y:x.a=y.b}{\triangleright} Y$$

Of the solutions above, clearly the latter is the most efficient—tables X and Y are accessed only once. However, as we shall see, to use the antijoin operator to deal with universal (negated existential) quantification is not possible in all cases. Whenever queries are cyclic, i.e. whenever the right-hand join operand tuple values are needed in subsequent computations, instead of the semijoin operator, the regular join operator can be used. For the antijoin there is no such natural extension which preserves the right-hand tuples, just because the antijoin delivers the *non*-matching (dangling) left operand tuples. So, for certain types of queries, we have to resort to either division or the set difference operator.

4.2.4 Disjunction

In [Bry89], it is proposed to use an operator called the *constrained outerjoin* to solve disjunctive queries. Consider the expression:

$$\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(y,z)](X)$$

The standard way of handling this query is by using set union:

Rewriting example 4.3

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(y,z)](X) \\
&\quad\equiv\quad \pi_X(\sigma[x : p(v_X, v_Y) \vee \exists z \in Z \bullet q(v_Y, z)](X \times Y)) \\
&\quad\equiv\quad \pi_X(\sigma[x : p(v_X, v_Y)](X \times Y) \cup \sigma[x : \exists z \in Z \bullet q(v_Y, z)](X \times Y)) \\
&\quad\equiv\quad (X \underset{x,y:p(x,y)}{\ltimes} Y) \cup (X \underset{x,y:true}{\ltimes} (Y \underset{y,z:q(y,z)}{\ltimes} Z))
\end{aligned}$$

Clearly, the disadvantage of this approach is that tables are accessed multiple times, and also that duplicates have to be removed from the result of the union operator. In [Bry89], an operator called the constrained outerjoin is defined. Each left operand tuple in the constrained outerjoin is marked with a value that indicates whether the tuple is matched by some right operand tuple or not. The marker is used in subsequent operations. In fact, the constrained outerjoin is a combination of the semijoin that preserves *only* the left-hand operand attributes, and the outerjoin that preserves *all* left operand tuples, either matched or not. Using the constrained outerjoin, which we call the markjoin, denoted by the symbol $-$ (to be defined in Section 4.5), the above calculus expression can be expressed as:

$$X \underset{x,y:p(x,y) \vee y.m}{\ltimes} (Y \underset{y,z:q(y,z);m}{-} Z)$$

The above result is linear, i.e. each of the base table occurrences of the calculus expression occur in the algebraic expression once. The markjoin does not suffice to handle cyclic queries with disjunction because of the loss of the actual right-hand operand tuple values.

4.2.5 Summary

As we have shown in Section 4.1, the standard reduction algorithm of [Codd72] for the translation of relational calculus into relational algebra may result in algebraic expressions that are inefficient and hard to optimize. Calculus constructs that cause problems with regard to efficiency are universal quantification and disjunction. In [Naka90], it is proposed to perform logical optimization during translation. In addition to a set of basic rewrite rules, a number of heuristic rewrite rules is given for rewriting expressions in a mixture of calculus and algebra, among which some rules for efficient translation of universal quantification. No special attention is given to disjunctive queries. In [Bry89] it is proposed to rewrite predicates into MNF, and to employ the antijoin and constrained outerjoin operators for universal quantification and disjunction, respectively. A disadvantage of [Bry89] is that no complete transformation algorithm is given.

In this chapter, we study the transformation of relational calculus into relational algebra. We combine the ideas of [Naka90] and [Bry89], and try to give a complete and efficient transformation algorithm. In the next section, we give an outline of our proposal.

4.3 Outline of the transformation

Our goal is to achieve a rewrite algorithm for transformation of arbitrary calculus expressions into efficient algebraic expressions. We combine optimization with translation, rewriting expressions into a mixture of calculus and algebra. Our algebra is extended: we use the semi- and antijoin operators as well as the markjoin. In rewriting, we take a systematic, top-down approach. Nested calculus expressions are standardized, next predicates are rewritten into MNF, and then join operations are introduced. We first present a basic set of rewrite rules, and next, in Section 4.6, we discuss some techniques that can be employed to further improve the translation. In the remainder of this section, we give an overview of the transformation algorithm. We first describe the desired output, and then briefly list the main steps involved. In sections to follow, each of these steps is discussed in more detail.

The output of the translation/optimization algorithm is an algebraic expression such that, as much as possible:

- The transformation is *linear*: each of the base tables in the calculus expression occurs in the algebraic expression at most once.
- Selections and projections are *pushed down* the operator tree.
- Cartesian products are avoided, and semijoin and antijoin operators are given preference to the regular join operator.

The above three translation/optimization objectives are based on the wish to:

- reduce the number of operations,
- reduce the cardinality (the number of tuples) and size (the number of attributes) of operands, and
- give preference to cheaper operators.

The primary goal in the transformation of calculus expressions into algebraic expressions is to achieve a so-called linear translation, i.e., a translation in which the algebraic expression that is the result of the transformation process contains at most one occurrence of each base table that occurs in the calculus expression. Trying to achieve a linear translation means that a nested calculus expression of nesting depth n, should be translated into an algebraic expression containing n join operators. As we shall see, it is not always possible to obtain a linear translation.

The input of the algorithm is an expression $\sigma[x : p](e)$ or $\pi_A(e)$ of RC. According to the syntax given in Section 4.1, iterator operand e may be a base table, another selection or projection, or a Cartesian product or union of the above. In selection predicates quantifier

expressions may occur that possibly contain other selections, projections, and/or quantifier expressions. In the sequel, selections, projections, and quantifications are called *iterators*. The goal in the transformation is to remove nested iterator occurrences. The main steps in the transformation of a calculus expression into an expression of the extended relational algebra are the following.

1. **Preprocessing**

   (a) **Composition** Iterator operands are transformed into base tables: operators σ and π are removed from iterator operands e.

   Composition is necessary because the user-determined order of processing is not necessarily the most efficient.

   (b) **Standardization** Selection predicates are transformed into Prenex Normal Form (PNF). Universal quantification is rewritten into negated existential quantification to enable the use of the antijoin operator; negation is pushed through.

   Transformation into PNF is done because PNF offers some specific opportunities for optimization (see below).

   (c) **Global transformation** 'Global' transformation rules that are considered appropriate are applied to the matrix of the PNF expression. In [JaKo84], this simplification phase is described extensively. An appropriate rule is the rule of transitivity; also important is the exchange of quantifiers, whenever possible and desired, to be able to push quantifiers inwards as much as possible in the next step.

   (d) **Transformation into MNF** In composition and standardization, expressions have been brought under iterator scopes unnecessarily. In the transformation into MNF, subformulas that do not depend on quantifier variables are removed from quantifier scopes.

2. **Translation** Here, the goal is to actually translate calculus or mixed expressions that possibly contain nested quantifiers into pure algebraic expressions. This phase is discussed in detail in Section 4.5.

3. **Postprocessing**

   Postprocessing may involve combining sequences of projections and selections into one, pushing through projections, simplifying joins with constant predicates, etc. Depending on the attributes used in join predicates, projections can be inserted in the algebraic expression.

4.4 Preprocessing

In this section, we present the rules used in the preprocessing phase of the rewrite algorithm described above. Rules are applied from left to right, unless stated otherwise. The set of rules is not complete. For instance, we do not include rules for the transformation of

Boolean expressions, e.g. the rules for commutativity, associativity, etc. (some are listed in Appendix A though). Our goal is to enlighten the core of the transformation algorithm by presenting the most important rules and to describe when they are applied. In this chapter, no proofs are given; some of them can be found in Appendix A, together with some additional rules that are not needed to get a global understanding of the algorithm, but are used often, or are uncommon.

4.4.1 Composition

The input to this phase is an expression e of RC that is a selection or a projection. Within the selection predicate, other nested selections and/or projections may occur, and also quantifier expressions. The operand of a selection, a projection, or a quantifier may be a base table, a set union or a Cartesian product expression, or, recursively, another expression e of RC. In this thesis, the emphasis lies on the transformation of nested expressions, therefore, we do not consider optimization of expressions that contain iterators with operands that are set unions or Cartesian products.

In the first step of the transformation algorithm, (1) projections, except for the top-level one, are removed, and (2) operands of selections and quantifiers that are iterator expressions are transformed into base tables. Composition is necessary because the user-defined order of operations is not necessarily the most efficient. Projection operators simply can be omitted, except for the outermost one that specifies the attributes that are needed in the result. In step (2), the following rules are used:

Rule 4.2 Composition Let x not occur in p, then:

1. $\sigma[x : q](\sigma[y : p](X)) \equiv \sigma[x : p[x/y] \wedge q](X)$
2. $\exists x \in (\sigma[y : p](X)) \bullet q \equiv \exists x \in X \bullet p[x/y] \wedge q$
3. $\forall x \in (\sigma[y : p](X)) \bullet q \equiv \forall x \in X \bullet \neg p[x/y] \vee q$

(In the expression $p[x/y]$, variable y is substituted by x.)

Composition is applied without exception. However, if the operands of selections and quantifiers are expressions that do not contain free variables, composition is not strictly necessary. For example, the expression:

$$\sigma[x : \exists y \in Y \bullet x.b = y.b](\sigma[x : x.a = 1](X))$$

can in one step be translated into the algebraic expression:

$$\sigma[x : x.a = 1](X) \underset{x,y:x.b=y.b}{\ltimes} Y$$

in which the selection on table X is evaluated before the join, as it should be. However, assume that the join predicate is $x.a = y.b$, then by omitting composition we would miss the opportunity for constant propagation (see Section 4.4.3).

4.4.2 Transformation into PNF

Predicates are put into PNF, by means of the standard transformation rules (see for example [Zhon89]), and universal quantification is rewritten into negated existential quantification, pushing through negation. We may choose to rewrite the quantifier matrix into conjunctive or disjunctive normal form, or leave it as it is. We choose for the latter, as we want to investigate in what way the specific form of the matrix influences the rewrite process, and hence the result of the transformation.

The fact that quantifier variables are bound causes a problem in the case the scope of a universal quantifier is a conjunction, and in case the scope of an existential quantifier is a disjunction: empty ranges have to be taken into account. The rules for moving quantifiers out are:

Rule 4.3 Scoping Let $x \notin FV(p)$, then:

1. $p \wedge \exists x \in X \bullet q \equiv \exists x \in X \bullet p \wedge q$
2. $p \vee \exists x \in X \bullet q \equiv \begin{cases} p & \text{if } X = \emptyset \\ \exists x \in X \bullet p \vee q & \text{otherwise} \end{cases}$
3. $p \wedge \nexists x \in X \bullet q \equiv \begin{cases} p & \text{if } X = \emptyset \\ \nexists x \in X \bullet \neg p \vee q & \text{otherwise} \end{cases}$
4. $p \vee \nexists x \in X \bullet q \equiv \nexists x \in X \bullet \neg p \wedge q$

Of course the latter two rules can be easily derived; they are added for reasons of convenience. Cases (2) and (3) of Rule 4.3 listed above do pose a problem. Consider the expression:

Example 4.2 Scoping

$$\sigma[x : p(x) \vee \exists y \in Y \bullet q(x,y)](X) \equiv \begin{cases} \sigma[x : p(x)](X) & \text{if } Y = \emptyset \\ \sigma[x : \exists y \in Y \bullet p(x) \vee q(x,y)](X) & \text{otherwise} \end{cases}$$

The outcome of this phase of rewriting, and hence the outcome of subsequent rewrite phases, depends on the value of Y, to be determined at run-time. The process of translation and optimization (as a compile-time activity) is complicated by rules such as the above: both alternatives have to be dealt with separately. For reasons of simplicity, we do not take into account the possibility of empty set-valued operands in our rewritings (we always choose the "otherwise" alternative).

4.4.3 Global transformation

After transformation into PNF, some global transformations may be applied to the matrix of the predicate. In [JaKo84], the phase following standardization (transformation into PNF) is called simplification. The goal of simplification is the removal of redundancy, for example by the application of the idempotency rules, or of rules that deal with empty

relations. Another important rule is the rule of transitivity, that can be used for constant propagation:

$$x.a = y.b \wedge y.b = 3 \Rightarrow x.a = 3$$

and for breaking cycles by the addition of join predicates and thereby transforming cyclic queries into tree queries, allowing for (generalized) semijoin processing [Kamb85]. For example, the query:

$$\pi_X((X \underset{x,y:x.a=y.a}{\bowtie} Y) \underset{v,z:v_Y.a=z.a \wedge v_X.b=z.b}{\ltimes} Z)$$

is equivalent to the expression:

$$\pi_X((X \underset{x,y:x.a=y.a}{\bowtie} Y) \underset{v,z:v_X.a=z.a \wedge v_X.b=z.b}{\ltimes} Z)$$

(note that the semijoin predicate is changed so that instead of testing $v_Y.a = z.a$ it is tested whether $v_X.a = z.a$) so that the join can be replaced by a semijoin:

$$(X \underset{x,y:x.a=y.a}{\ltimes} Y) \underset{v,z:v_X.a=z.a \wedge v_X.b=z.b}{\ltimes} Z$$

Another technique that can be applied in this phase is the exchange of quantifiers. Determination of a favourable join order by the exchange of quantifiers in a separate step of the transformation algorithm is profitable, as it can help to avoid Cartesian products. Also, the number of rewrite steps is reduced. Adjacent existential or universal quantifiers may be exchanged:

Rule 4.4 Exchanging quantifiers

1. $\exists x \in X \bullet \exists y \in Y \bullet p \equiv \exists y \in Y \bullet \exists x \in X \bullet p$
2. $\forall x \in X \bullet \forall y \in Y \bullet p \equiv \forall y \in Y \bullet \forall x \in X \bullet p$

We give an example. Consider the expression:

$$\sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \wedge r(y,z)](X)$$

that, using the rules of Section 4.5, is translated into:

$$\pi_X((X \times Y) \underset{v,z:q(v_X,z) \wedge r(v_Y,z)}{\ltimes} Z)$$

The above expression needs further algebraic optimization. The exchange of quantifiers and descoping (see Rule 4.5) results in:

$$\sigma[x : \exists z \in Z \bullet q(x,z) \wedge \exists y \in Y \bullet r(y,z)](X)$$

that, by means of the same set of rewrite rules, is translated into:

$$X \underset{x,z:q(x,z)}{\ltimes} (Z \underset{z,y:r(y,z)}{\ltimes} Y)$$

We see that the exchange of quantifiers enables descoping, and hence an easy translation into a semijoin expression. However, we suspect that the problem of determination of the best quantifier order, given that there is a choice, is not easily solved in general.

4.4.4 Transformation into MNF

If selection predicates are left in PNF, then the application of the rewrite rules of Section 4.5, for the transformation of nested calculus expressions into algebraic ones, will result in inefficient expressions containing Cartesian products, not very different from the result achieved by the algorithm of Codd. Following translation, we still have to perform algebraic optimization to push through selections and projections. Therefore, we transform predicates into MNF. In the transformation into PNF, expressions have been moved under the scope of quantifiers unnecessarily, i.e., quantifier scopes include expressions in which the quantifier variable does not occur free. In the transformation into MNF, this process is reversed: quantifiers are pushed inwards as far as possible. Subformulas of quantifier expressions in which the quantifier variable does not occur free are removed from the quantifier scope. The rules that are needed for a full transformation into MNF are given in [Bry89]. In [Bry89], however, quantifier variables are not range-restricted; the rules for moving subformulas out of quantifier scopes must be adapted as follows.

Rule 4.5 Descoping Let $x \notin FV(p)$, then:

1. $\exists x \in X \bullet p \wedge q(x) \equiv p \wedge \exists x \in X \bullet q(x)$
2. $\exists x \in X \bullet p \vee q(x) \equiv \begin{cases} p \vee \exists x \in X \bullet q & \text{if } X \neq \emptyset \\ false & \text{otherwise} \end{cases}$
3. $\forall x \in X \bullet p \wedge q(x) \equiv \begin{cases} p \wedge \forall x \in X \bullet q(x) & \text{if } X \neq \emptyset \\ true & \text{otherwise} \end{cases}$
4. $\forall x \in X \bullet p \vee q(x) \equiv p \vee \forall x \in X \bullet q(x)$

In our system, universal quantifiers are rewritten into negated existential quantifiers; the rules concerning universal quantification are added for completeness. Again, we have to take into account the possibility of empty set-valued operands, but we will not do so for reasons of simplicity.

In descoping, we search for the largest subexpression p of the quantifier scope such that $x \notin FV(p)$. For example,

$$\sigma[x : \exists y \in Y \bullet (p(x) \vee q(x)) \wedge r(x,y)](X)$$

is transformed into:

$$\sigma[x : (p(x) \vee q(x)) \wedge \exists y \in Y \bullet r(x,y)](X)$$

However, the expression:

$$\sigma[x : \exists y \in Y \bullet p(y) \vee (q(x) \wedge r(x,y))](X)$$

cannot be rewritten into MNF by means of the descoping rules only. A complete transformation into MNF, i.e., the removal of *all* independent subexpressions from quantifier scopes, may require distribution of the existential quantifier:

Rewriting example 4.4

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(y) \vee (q(x) \wedge r(x,y))](X) \\
&\quad \equiv \quad \sigma[x : (\exists y \in Y \bullet p(y)) \vee (\exists y \in Y \bullet q(x) \wedge r(x,y))](X) \\
&\quad \equiv \quad \sigma[x : (\exists y \in Y \bullet p(y)) \vee (q(x) \wedge \exists y \in Y \bullet r(x,y))](X)
\end{aligned}$$

Also, a complete transformation into MNF may require distribution of Boolean connectives:

$$\sigma[x : \exists y \in Y \bullet p(x,y) \wedge (q(x) \vee r(x,y))](X)$$

It is not clear whether distribution of quantifiers, which is a non-linear transformation technique, and distribution of Boolean connectives, which introduces common subexpressions, are profitable strategies. In some cases it might be better not to transform a predicate into MNF. Distribution of quantifiers is further discussed in Section 4.6.5.

The output of the phase of transformation into MNF is a (projection on a) selection of which the predicate consists of atomic predicates, quantifier expressions, or both. Quantifier scopes also have this format. For each quantifier, it holds that the subformulas of the quantifier scope depend on the quantifier variable.

4.5 Translation

To enable efficient translation of relational calculus expressions, we extend standard relational algebra that consists of operators $\cup, -, \cap, \sigma, \pi, \times, \bowtie$, and $\div$, with the semijoin $\ltimes$, the antijoin $\rhd$ and the markjoin $\overline{-}$. As explained in Section 4.2.4, the markjoin, which is defined below, corresponds to the constrained outerjoin of [Bry89].

Definition 4.2 Let m be a label, then:

$$T_m = \{\langle m = true \rangle\} \text{ and } F_m = \{\langle m = false \rangle\} \qquad \square$$

Definition 4.3 Markjoin $(\overline{-})$ Marks matched and unmatched left operand tuples with values $true$ and $false$, respectively.

$$X \underset{x,y:p;m}{\overline{-}} Y = ((X \underset{x,y:p}{\ltimes} Y) \times T_m) \cup ((X \underset{x,y:p}{\rhd} Y) \times F_m) \qquad \square$$

The markjoin can be implemented by a simple modification of any semijoin algorithm—all that is needed is to extend the left-hand operand tuples with the marker attribute.

In the algebra defined so far, we have a number of join operators that do not preserve right-hand join operand tuples, i.e. the semi-, anti-, and markjoin, and some operators that do preserve these values, i.e. the regular join and the Cartesian product. We call the latter *full*, the former *partial* joins. In Table 4.5, we have listed the join operators used in this thesis, indicating which tuples of the left and right operand are present in the result. For illustrative purposes, we have also included the left outerjoin.

operator	left operand	right operand
product	all	all
left outerjoin	matched	matching
	unmatched	NULL
markjoin	matched	true
	unmatched	false
regular join	matched	matching
semijoin	matched	none
antijoin	unmatched	none

Table 4.1: Join operators and tuples present in the result

4.5.1 Transformation rules

For the translation of quantifier expressions that do not contain quantification in their scope, partial join operators can be used. A quantifier expression that does not contain quantifiers in its scope is called a *basic quantification*. The first step in the translation is to remove nested basic quantifications from calculus expressions. First, we need a definition of two classes of formulas: conjunctive and disjunctive path formulas. Path formulas are used in transformation rules, to allow for some concision.

Definition 4.4 Con- and disjunctive path formulas

- $P_{con}(t)$ is defined as a formula such that t is equal to P, or t is a subformula of P, and the path from the root of the operator tree corresponding to P to t contains conjunctions only. In BNF this is:

  $$P_{con}(t) ::= t \mid p \wedge P_{con}(t)$$

- $P_{dis}(t)$ is defined as a formula such that t is a subformula of P, and the path from the root of the operator tree corresponding to P to t contains one or more disjunctions, and possibly conjunctions as well. In BNF:

  $$P_{dis}(t) ::= p \vee t \mid p \vee P_{dis}(t) \mid p \wedge P_{dis}(t)$$ □

The first step in the transformation is to remove basic quantification from selection predicates, if possible, i.e. if the quantification is contained in a conjunctive or disjunctive path formula as defined above. Consider a selection of which the predicate is in MNF. We check whether the predicate contains a basic quantification that is part of a con- or disjunctive path formula, i.e. a quantification that does not occur within the scope of another quantification. If so, we apply one of the following transformation rules:

Rule 4.6 Introduction of partial joins (unnest) Let b be a predicate that does not contain quantification, let $FV(b) = \{x, y\}$, and let x not occur free in Y, then:

1. $\sigma[x : P_{con}(\exists y \in Y \bullet b(x,y))](X) \equiv \sigma[x : P_{con}(true)](X \underset{x,y:b(x,y)}{\ltimes} Y)$
2. $\sigma[x : P_{con}(\nexists y \in Y \bullet b(x,y))](X) \equiv \sigma[x : P_{con}(true)](X \underset{x,y:b(x,y)}{\triangleright} Y)$
3. $\sigma[x : P_{dis}(\exists y \in Y \bullet b(x,y))](X) \equiv \pi_X(\sigma[x : P_{dis}(x.m)](X \underset{x,y:b(x,y);m}{-} Y))$

Note that we do not need a rule for negated existential quantification in a disjunctive context. In the transformation, the subquery expression is removed from the selection predicate by the introduction of a partial join operation. It is required that both join variables occur free in the join predicate—a join with a monadic or constant join predicate does not make much sense. The join predicate is closed—we do not allow join expressions with free variables. In the original predicate, the subquery expression is substituted, either by $true$, if the subquery is contained in a conjunctive path formula, or by a term testing the markjoin mark attribute, if the subquery is contained in a disjunctive path formula. The proof of the above equivalences is by induction on the length of the path. We give the proof of the third rule:

Proof Rule 4.6 Introduction markjoin

$$\sigma[x : P_{dis}(\exists y \in Y \bullet b(x,y))](X) \equiv \pi_X(\sigma[x : P_{dis}(x.m)](X \underset{x,y:b(x,y);m}{-} Y))$$

We abbreviate the expression $b(x,y)$ to b.

(Base case)

$$\begin{aligned}
&\sigma[x : p \vee \exists y \in Y \bullet b](X)\\
&\quad\equiv\quad \sigma[x : (p \wedge \nexists y \in Y \bullet b) \vee \exists y \in Y \bullet b](X)\\
&\quad\equiv\quad \sigma[x : p](\sigma[x : \nexists y \in Y \bullet b](X)) \cup \sigma[x : \exists y \in Y \bullet b](X)\\
&\quad\equiv\quad \sigma[x : p](X \underset{x,y:b}{\triangleright} Y) \cup (X \underset{x,y:p}{\ltimes} Y)
\end{aligned}$$

Note that, in the first rewrite step, we use the logical equivalence $a \vee b \equiv a \vee (\neg a \wedge b)$. Next, disjunction is transformed into union, and then the definition of the semi- and antijoin is used. The right-hand side of the equivalence is rewritten by expanding the markjoin definition:

$$\begin{aligned}
&\pi_X(\sigma[x : p \vee x.m](X \underset{x,y:b;m}{-} Y))\\
&\quad\equiv\quad \pi_X(\sigma[x : p \vee x.m](((X \underset{x,y:b}{\ltimes} Y) \times T_m) \cup ((X \underset{x,y:b}{\triangleright} Y) \times F_m)))\\
&\quad\equiv\quad \pi_X(\sigma[x : p \vee x.m]((X \underset{x,y:b}{\ltimes} Y) \times T_m) \cup \sigma[x : p \vee x.m]((X \underset{x,y:b}{\triangleright} Y) \times F_m))\\
&\quad\equiv\quad \pi_X(\sigma[x : p \vee true]((X \underset{x,y:b}{\ltimes} Y) \times T_m) \cup \sigma[x : p \vee false]((X \underset{x,y:b}{\triangleright} Y) \times F_m))\\
&\quad\equiv\quad \pi_X(((X \underset{x,y:b}{\ltimes} Y) \times T_m) \cup \sigma[x : p]((X \underset{x,y:b}{\triangleright} Y) \times F_m))\\
&\quad\equiv\quad \pi_X((X \underset{x,y:b}{\ltimes} Y) \times T_m) \cup \sigma[x : p](\pi_X((X \underset{x,y:b}{\triangleright} Y) \times F_m))\\
&\quad\equiv\quad (X \underset{x,y:b}{\ltimes} Y) \cup \sigma[x : p](X \underset{x,y:b}{\triangleright} Y)
\end{aligned}$$

The definition of the markjoin is expanded, the selection is pushed through, the selection predicate is simplified, and finally the projection is pushed through.

(Induction step)

(Disjunction)

$$\begin{aligned}
&\sigma[x : p \vee P_{dis}(\exists y \in Y \bullet b)](X) \\
&\quad \equiv \quad \sigma[x : p](X) \cup \sigma[x : P_{dis}(\exists y \in Y \bullet b)](X) \\
&\quad \equiv \quad \sigma[x : p](X) \cup \pi_X(\sigma[x : P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y)) \textbf{ (induction)}
\end{aligned}$$

Disjunction is transformed into union, and then induction is used. On the other hand we have:

$$\begin{aligned}
&\pi_X(\sigma[x : p \vee P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y)) \\
&\quad \equiv \quad \pi_X(\sigma[x : p](X \underset{x,y:b;m}{-} Y) \cup \sigma[x : P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y)) \\
&\quad \equiv \quad \pi_X(\sigma[x : p](X \underset{x,y:b;m}{-} Y)) \cup \pi_X(\sigma[x : P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y)) \\
&\quad \equiv \quad \sigma[x : p](\pi_X(X \underset{x,y:b;m}{-} Y)) \cup \pi_X(\sigma[x : P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y)) \\
&\quad \equiv \quad \sigma[x : p](X) \cup \pi_X(\sigma[x : P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y))
\end{aligned}$$

Disjunction is transformed into union, and next the left union operand is simplified by pushing through the projection.

(Conjunction)

$$\begin{aligned}
&\sigma[x : p \wedge P_{dis}(\exists y \in Y \bullet b)](X) \\
&\quad \equiv \quad \sigma[x : p](\sigma[x : P_{dis}(\exists y \in Y \bullet b)](X)) \\
&\quad \equiv \quad \sigma[x : p](\pi_X(\sigma[x : P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y))) \textbf{ (induction)} \\
&\quad \equiv \quad \pi_X(\sigma[x : p](\sigma[x : P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y))) \\
&\quad \equiv \quad \pi_X(\sigma[x : p \wedge P_{dis}(x.m)](X \underset{x,y:b;m}{-} Y))
\end{aligned}$$

□

We give two example applications of Rule 4.6:

Rewriting example 4.5 Introduction of partial joins

Let b be an atomic predicate, then:

- $\sigma[x : p(x) \wedge \nexists y \in Y \bullet b(x,y)](X) \equiv \pi_X(\sigma[x : p(x) \wedge true](X \underset{x,y:b(x,y)}{\triangleright} Y))$
- $\sigma[x : q(x) \wedge (p(x) \vee \nexists y \in Y \bullet b(x,y))](X) \equiv$
  $\pi_X(\sigma[x : q(x) \wedge (p(x) \vee \neg x.m))](X \underset{x,y:b(x,y);m}{-} Y))$

A second step in the translation is to check whether the scope of some quantifier contains one or more basic quantifier expressions that do not occur within the scope of yet another quantifier. If so, we apply the following transformation rules:

Rule 4.7 Introduction of partial joins (range nest) Let b be a predicate that does not contain quantification, and let $FV(b) = \{x, y\}$, then:

1. $\exists x \in X \bullet P_{con}(\exists y \in Y \bullet b) \equiv \exists x \in (X \underset{x,y:b}{\ltimes} Y) \bullet P_{con}(true)$
2. $\exists x \in X \bullet P_{con}(\nexists y \in Y \bullet b) \equiv \exists x \in (X \underset{x,y:b}{\triangleright} Y) \bullet P_{con}(true)$
3. $\exists x \in X \bullet P_{dis}(\exists y \in Y \bullet b) \equiv \exists x \in (X \underset{x,y:b;m}{-} Y) \bullet P_{dis}(x.m)$

The third step is to remove the remaining quantifiers. This is done by means of the standard rules listed below:

Rule 4.8 Basic rules

1. $\sigma[x : p \wedge q](X) \equiv \sigma[x : p](\sigma[x : q](X))$ **(split)**
2. $\sigma[x : p \vee q](X) \equiv \sigma[x : p](X) \cup \sigma[x : q](X)$ **(split)**
3. $\sigma[x : \exists y \in Y \bullet p(x, y)](X) \equiv \pi_X(\sigma[v : p(v_X, v_Y)](X \times Y))$ **(unnest)**
4. $\sigma[x : \nexists y \in Y \bullet p(x, y)](X) \equiv (\sigma[v : \neg p(v_X, v_Y)](X \times Y)) \div Y$ **(unnest)**

Top-level selection predicates, which are in MNF, are split until the quantifiers present occur at the top level, and then the quantification is removed by rewriting into a product, followed by division in case the quantifier is negated. The application of the basic rules may result in intermediate expressions that allow for the introduction of partial joins, so Rule 4.6 and 4.7 always are applied prior to the basic rules.

Starting from a selection with the predicate in MNF, the rules presented here are sufficient to obtain a fully unnested algebraic expression, in which base tables occur at the top level only. To further improve the result, we need rules for pushing through selections to joins and join operands; these rules, and the rules for pushing through projections are listed in Appendix A.

Rules in which quantification is removed from selection predicates are called unnest rules; the rules for pushing through predicates to quantifier ranges (Rule 4.7) are called range nest rules. Below, we describe the steps our translation algorithm consists of:

1. Whenever possible, push through selections to joins and join operands. However, join predicates are allowed to be basic predicates only, i.e. predicates in which no quantification occurs.

2. Whenever possible, introduce partial join operators by means of Rule 4.6 (unnest) and Rule 4.7 (range nest).

3. If the above is not possible, and selection predicates still contain quantification, then apply the basic rules for splitting predicates, until quantification occurs at the top level, and then use the basic rules for unnesting.

In the translation, we do not pay much attention to projections. Of course, for each operation, it suffices to preserve only the attributes that are needed in the computation, or in subsequent operations.

4.5.2 Example translations

In this section, we illustrate the results obtained by our proposed rewrite algorithm by showing the translation of the following expression:

$$\sigma[x : \mathcal{Q}y \in Y \bullet p(x,y) \ \boldsymbol{c} \ \mathcal{Q}z \in Z \bullet q(x,z) \ \boldsymbol{c} \ r(y,z)](X)$$

Quantifiers $\mathcal{Q}$ may be $\exists$ or $\nexists$; connectives $\boldsymbol{c}$ are either $\vee$ or $\wedge$ (both occurrences of $\mathcal{Q}$ and $\boldsymbol{c}$ may be different). We distinguish between chain, tree(I), tree(II), and cyclic queries:

Chain $\sigma[x : \mathcal{Q}y \in Y \bullet p(x,y) \ \boldsymbol{c} \ \mathcal{Q}z \in Z \bullet r(y,z)](X)$

Tree(I) $\sigma[x : \mathcal{Q}y \in Y \bullet p(x,y) \ \boldsymbol{c} \ \mathcal{Q}z \in Z \bullet q(x,z)](X)$

Tree(II) $\sigma[x : \mathcal{Q}y \in Y \bullet \mathcal{Q}z \in Z \bullet q(x,z) \ \boldsymbol{c} \ r(y,z)](X)$

Cyclic $\sigma[x : \mathcal{Q}y \in Y \bullet p(x,y) \ \boldsymbol{c} \ \mathcal{Q}z \in Z \bullet q(x,z) \ \boldsymbol{c} \ r(y,z)](X)$

Is is assumed that predicates p, q, and r are atomic predicates. In Tables 4.2 and 4.3, the translation of the expressions above is given, varying the type of the quantifiers and the connective present. In each of the expressions above except for the last only one connective is present; the precise format of the second connective of cyclic queries is not considered. The number of variants thus achieved is 32.

Note that tree(I) queries are not in MNF, so the first step in translation is to obtain MNF by means of descoping. Tree(II) queries are in MNF, however, in some cases exchange or distribution of quantification makes it possible to improve the result. The set of rules that consists of Rule 4.6, 4.7, and 4.8, in combination with the rules for descoping (Rule 4.5), for the exchange of quantification (Rule 4.4), and for pushing through selections and projections (Rule A.7 and A.6, given in Appendix A) is referred to as the **standard rule set** $\mathcal{R}$. Rewriting by means of rule set $\mathcal{R}$ is called **standard rewriting**.

Table 4.2 lists the results for disjunctive queries, and Table 4.3 for conjunctive ones. In the rewritings and tables, we have omitted the projections necessary due to the introduction of markjoin operators. In the tables, we have also omitted selection and join variables, as we assume they are clear from the context.

The full rewritings are listed in Appendix B. Note that the results in the first and third part of the tables are the complement of the corresponding results listed in the second and fourth part of the tables, respectively, and vice versa.

For chain queries, which are the easiest to handle, the first step is to apply Rule 4.7, i.e. to push through the quantification $\exists z \in Z \bullet r(y,z)$ to table Y; this step was called 'range nest'. This results into a nested semi- or antijoin in case the quantification is part of a conjunction, and into a nested markjoin in case the quantification is part of a disjunction. Further rewriting is easy. In all cases, the result is linear.

Tree(I) queries are rewritten by first applying the descoping rules, transforming the query into a query with multiple subqueries.[2] Whenever the quantification with range Y is

[2]Recall that in applying the descoping rules we do not take into account the possibility of empty quantifier ranges.

$\exists\,\exists$	chain	1	$X \underset{p\vee m}{\ltimes} (Y \underset{r;m}{-} Z)$
	tree(I)	2	$\sigma[m \vee m']((X \underset{q;m}{-} Z) \underset{p;m'}{-} Y)$
	tree(II)	3	$X \underset{q\vee m}{\ltimes} (Z \underset{r;m}{-} Y)$
	cyclic	4	$\pi_X(\sigma[p \vee m]((X \times Y) \underset{q\ \boldsymbol{c}\ r;m}{-} Z))$
$\not\exists\,\exists$	chain	5	$X \underset{p\vee m}{\rhd} (Y \underset{r;m}{-} Z)$
	tree(I)	6	$(X \underset{q}{\rhd} Z) \underset{p}{\rhd} Y$
	tree(II)	7	$X \underset{q\vee m}{\rhd} (Z \underset{r;m}{-} Y)$
	cyclic	8	$((X \underset{\neg p}{\bowtie} Y) \underset{q\ \boldsymbol{c}\ r}{\rhd} Z) \div Y$
$\exists\ \not\exists$	chain	9	$X \underset{p\vee\neg m}{\ltimes} (Y \underset{r;m}{-} Z)$
	tree(I)	10	$\sigma[\neg m \vee m']((X \underset{q;m}{-} Z) \underset{p;m'}{-} Y)$
	tree(II)	11	$\mathbf{if}\ (Y \underset{y,z:r(y,z)}{\rhd} Z) \neq \emptyset\ \mathbf{then}\ (X \underset{x,z:q(x,z)}{\rhd} Z)\ \mathbf{else}\ \emptyset$
	cyclic	12	$\pi_X(\sigma[p \vee \neg m]((X \times Y) \underset{q\ \boldsymbol{c}\ r;m}{-} Z))$
$\not\exists\ \not\exists$	chain	13	$X \underset{p\vee\neg m}{\rhd} (Y \underset{r;m}{-} Z)$
	tree(I)	14	$(X \underset{q}{\ltimes} Z) \underset{p}{\rhd} Y$
	tree(II)	15	$\mathbf{if}\ (Y \underset{y,z:r(y,z)}{\rhd} Z) \neq \emptyset\ \mathbf{then}\ (X \underset{x,z:q(x,z)}{\ltimes} Z)\ \mathbf{else}\ X$
	cyclic	16	$((X \underset{\neg p}{\bowtie} Y) \underset{q\ \boldsymbol{c}\ r}{\ltimes} Z) \div Y$

Table 4.2: Illustration of the translation algorithm—disjunction

$\exists\,\exists$	chain	17	$X \underset{p}{\ltimes} (Y \underset{r}{\ltimes} Z)$
	tree(I)	18	$(X \underset{q}{\ltimes} Z) \underset{p}{\ltimes} Y$
	tree(II)	19	$X \underset{q}{\ltimes} (Z \underset{r}{\ltimes} Y)$
	cyclic	20	$\pi_X((X \underset{p}{\bowtie} Y) \underset{q\ \boldsymbol{c}\ r}{\ltimes} Z)$
$\not\exists\,\exists$	chain	21	$X \underset{p}{\rhd} (Y \underset{r}{\ltimes} Z)$
	tree(I)	22	$\sigma[\neg m \vee \neg m'](X \underset{q;m}{-} Z) \underset{p;m'}{-} Y)$
	tree(II)	23	$X \underset{q}{\rhd} (Z \underset{r}{\ltimes} Y)$
	cyclic	24	$(\sigma[\neg p \vee \neg m]((X \times Y) \underset{q\ \boldsymbol{c}\ r;m}{-} Z) \div Y$
$\exists\ \not\exists$	chain	25	$X \underset{p}{\ltimes} (Y \underset{r}{\rhd} Z)$
	tree(I)	26	$(X \underset{q}{\rhd} Z) \underset{p}{\ltimes} Y$
	tree(II)	27	$\pi_X((X \times Y) \underset{q\wedge r}{\rhd} Z)$
	cyclic	28	$\pi_X((X \underset{p}{\bowtie} Y) \underset{q\ \boldsymbol{c}\ r}{\rhd} Z)$
$\not\exists\ \not\exists$	chain	29	$X \underset{p}{\rhd} (Y \underset{r}{\rhd} Z)$
	tree(I)	30	$\sigma[m \vee \neg m'](X \underset{q;m}{-} Z) \underset{p;m'}{-} Y)$
	tree(II)	31	$((X \times Y) \underset{q\wedge r}{\ltimes} Z) \div Y$
	cyclic	32	$(\sigma[\neg p \vee \neg m]((X \times Y) \underset{q\ \boldsymbol{c}\ r;m}{-} Z) \div Y$

Table 4.3: Illustration of the translation algorithm—conjunction

negated (cases 6, 14, 22, and 30), the connective changes due to descoping, i.e. a conjunct becomes a disjunct, and vice versa. Therefore, if the quantification over Y is negated, then the result is good if the connective is $\vee$ (cases 6 and 14), and not to good if the connective is $\wedge$ (cases 22 and 30). In the latter case, the result contains two markjoin operators, which should be avoided, if possible—probably the use of set union is preferable. Tree(I) queries of which the quantification is not negated and the connective is $\wedge$ (cases 18 and 26) do not pose a problem; in case the connective is $\vee$ (cases 2 and 10), again the result contains two markjoin operators.

Tree(II) queries are rewritten by exchanging the quantifier, if possible (cases 3, 7, 19, and 23), transforming the query into a chain query, and then the result is satisfactory in general. In case quantifier exchange is not possible, the quantification over Z is distributed, if possible (cases 11 and 15). The result is simple. If distribution is not possible (cases 27 and 31), the result contains a Cartesian product, and has to be optimized further.

In the translation of cyclic queries, the only rules used are the basic rules for unnesting, removing quantifiers by rewriting into a product, possibly with a division. Certainly in case of cyclic queries, but possibly for other queries as well, the results achieved by means of our set of rewrite rules $\mathcal{R}$ can be improved. In the next section, we discuss some optional solutions for the translation of calculus into algebra.

4.6 Optimizations

In the previous section, we discussed some example translations. Our translation is such that:

- MNF is the starting point for translation; for disjunctive tree(I) queries, which are not in MNF, we first apply descoping.
- Quantifiers are exchanged if possible, for tree(II) queries, which enables to apply descoping.
- Quantification is distributed, but only for disjunctive tree(II) queries, in case exchange is not possible.
- The semi- and antijoin are used whenever possible.
- The markjoin is used to handle disjunction whenever possible, and set union is used otherwise.
- Division is used to handle universal quantification.

In this section, we evaluate the results achieved by our translation, and discuss additional rewrite techniques and optional solutions that might give better results.

4.6.1 Markjoin versus set union versus bypass processing

Translation of tree(I) queries often results in an expression that contains two markjoin operators (cases 2, 10, 22, and 30). For example, expression 2 is translated into:

$$\sigma[x : x.m \vee x.m']((X \underset{x,z:q(x,z);m}{-} Z) \underset{x,y:p(x,y);m'}{-} Y)$$

The translation is linear, in the sense that each base table that occurs in the calculus expression occurs in the algebraic expression at most once. The price paid for the use of the markjoin is the non-reduction of left markjoin operands and the additional selection. Using set union to handle the disjunction, the result for expression 2 is:

$$(X \underset{x,z:q(x,z)}{\ltimes} Z) \cup (X \underset{x,y:p(x,y)}{\ltimes} Y)$$

The translation by means of set union is not linear, and also duplicates have to be removed from the result. Yet another equivalent algebraic expression can be achieved by scoping the disjunct $x.m$ that is obtained due to the introduction of the first markjoin operator:

Rewriting example 4.6 (2) Tree(I)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet (p(x,y) \vee \exists z \in Z \bullet q(x,z))](X)\\
&\quad\equiv \quad \sigma[x : \exists z \in Z \bullet q(x,z) \vee \exists y \in Y \bullet p(x,y)](X) \textbf{ (descope)}\\
&\quad\equiv \quad \sigma[x : x.m \vee \exists y \in Y \bullet p(x,y)](X \underset{x,z:q(x,z);m}{-} Z) \textbf{ (unnest)}\\
&\quad\equiv \quad \sigma[x : \exists y \in Y \bullet x.m \vee p(x,y)](X \underset{x,z:q(x,z);m}{-} Z) \textbf{ (scope)}\\
&\quad\equiv \quad (X \underset{x,z:q(x,z);m}{-} Z) \underset{x,y:x.m \vee p(x,y)}{\ltimes} Y \textbf{ (unnest)}
\end{aligned}$$

Of course, the question is which of the above algebraic expressions is the best with respect to performance. The answer is not clear. For example, the higher the selectivity of predicates p and q, the less disadvantageous the operand-preserving property of the markjoin operator, and also, the larger the number of duplicates that is present in the union operands. It does seem wise to prevent the second markjoin operator by means of scoping as shown above. The semijoin can be implemented such that predicate p is tested only for the tuples marked false in the preceding markjoin operation.

We next discuss the result of the translation of expression 4. Rewriting using our set of rules $\mathcal{R}$ proceeds as follows:

Rewriting example 4.7 (4) Cyclic

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \ \boldsymbol{c}\ r(y,z)](X)\\
&\quad\equiv \quad \pi_X(\sigma[v : p(v_X, v_Y) \vee \exists z \in Z \bullet q(v_X,z) \ \boldsymbol{c}\ r(v_Y,z)](X \times Y)) \textbf{ (unnest)}\\
&\quad\equiv \quad \pi_X(\sigma[v : p(v_X, v_Y) \vee v.m]((X \times Y) \underset{v,z:q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z);m}{-} Z)) \textbf{ (unnest)}
\end{aligned}$$

In this case, the rules presented so far do not give especially good results. The result contains a Cartesian product that is the left operand of a markjoin operator that *preserves* its left operand. Let us abbreviate predicate $q(x,z)\ \boldsymbol{c}\ r(y,z)$ to $t(x,y,z)$, then standard rewriting using the union to solve the disjunction and pushing through selections and projections proceeds as follows:

Rewriting example 4.8 (4) Cyclic

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet t(x,y,z)](X)\\
&\quad\equiv\quad \pi_X(\sigma[v : p(v_X,v_Y) \vee \exists z \in Z \bullet t(v_X,v_Y,z)](X \times Y)) \textbf{ (unnest)}\\
&\quad\equiv\quad \pi_X(\sigma[x : p(x,y)](X \times Y) \cup \sigma[v : \exists z \in Z \bullet t(v_X,v_Y,z)](X \times Y)) \textbf{ (split)}\\
&\quad\equiv\quad \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \cup ((X \times Y) \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z)) \textbf{ (unnest)}\\
&\quad\equiv\quad (X \underset{x,y:p(x,y)}{\ltimes} Y) \cup \pi_X((X \times Y) \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z) \textbf{ (push)}
\end{aligned}$$

The result contains a simple semijoin, and a product that is the left operand of another semijoin. The result achieved by standard rewriting using $\mathcal{R}$ can be rewritten using one of the rules for the simplification of markjoin expressions (Rule A.9) in the following way:

Rewriting example 4.9

$$\begin{aligned}
&\pi_X(\sigma[v : p(v_X,v_Y) \vee v.m]((X \times Y) \underset{v,z:t(v_X,v_Y,z);m}{-} Z))\\
&\quad\equiv\quad \pi_X(\pi_{XY}((\sigma[v : p(v_X,v_Y) \vee v.m]((X \times Y) \underset{v,z:t(v_X,v_Y,z);m}{-} Z)))\\
&\quad\equiv\quad \pi_X(((X \times Y) \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z) \cup \sigma[v : p(v_X,v_Y)]((X \times Y) \underset{v,z:t(v_X,v_Y,z)}{\rhd} Z))\\
&\quad\equiv\quad \pi_X(((X \times Y) \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z) \cup ((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:t(v_X,v_Y,z)}{\rhd} Z))
\end{aligned}$$

Compared to the solution achieved by the use of the union operator, the advantage of the latter result is that there are no duplicates present in the union operands. Expression 4 seems hard to translate efficiently—the markjoin does not seem very well suited for the translation of this expression.

In [KMPS94], another solution to handle disjunctive queries is proposed: a two-stream selection operator σ^2. In [KMPS95], the idea of two-stream operator is generalized to two-stream semi- and regular join, denoted by $\ltimes^2$ and $\bowtie^2$, respectively. Let $+$ denote union without duplicate removal (merge). We have the following equivalences:

Rule 4.9 Bypass processing

1. $\sigma[x : p(x) \vee q(x)](X) \equiv$
   $T + \sigma[x : q(x)](F)\ \textbf{with}\ (T,F) = \sigma^2[x : p(x)](X)$
2. $\sigma[x : (\exists y \in Y \bullet p(x,y)) \vee q(x)](X) \equiv$
   $T + \sigma[x : q(x)](F)\ \textbf{with}\ (T,F) = X \underset{x,y:p(x,y)}{\ltimes^2} Y$
3. $\sigma[x : (\nexists y \in Y \bullet p(x,y)) \vee q(x)](X) \equiv$
   $F + \sigma[x : q(x)](T)\ \textbf{with}\ (T,F) = X \underset{x,y:p(x,y)}{\ltimes^2} Y$

4. $\sigma[x : \exists y \in Y \bullet p(x,y) \vee q(x,y)](X) \equiv$
   $\pi_X(T + \sigma[x : q(x,y)](F))\ \mathbf{with}\ (T,F) = X \underset{x,y:p(x,y)}{\ltimes^2} Y$

The above equivalence rules are based on the simple logical equivalence:

$$a \vee b \equiv a \vee (\neg a \wedge b)$$

We have, for a selection with arbitrary predicates p and q:

Rewriting example 4.10 Two-stream selection

$$\begin{aligned}
&\sigma[x : p \vee q](X)\\
&\quad\equiv\quad \sigma[x : p \vee (\neg p \wedge q)](X)\\
&\quad\equiv\quad \sigma[x : p](X) + \sigma[x : \neg p \wedge q](X)\\
&\quad\equiv\quad \sigma[x : p](X) + \sigma[x : q](\sigma[x : \neg p](X))\\
&\quad\equiv\quad T + \sigma[x : q](F)\ \mathbf{with}\ (T,F) = \sigma^2[x : p(x)](X)
\end{aligned}$$

Instead of set union, the merge operator $+$, i.e. set union *without duplicate removal*, can be used to solve the disjunction, because the two sets $\sigma[x : p](X)$ and $\sigma[x : \neg p \wedge q](X)$ are disjoint. The sets are complementary as well, therefore, a two-stream selection can be implemented as a simple modification of selection, splitting the output into a positive and a negative stream. The negative stream is processed further, the positive stream can be included into the result right away. We also derive the equivalence for the join:

Rewriting example 4.11 Two-stream join

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee q(x,y)](X)\\
&\quad\equiv\quad \pi_X(\sigma[v : p(v_X, v_Y) \vee q(v_X, v_Y)](X \times Y))\\
&\quad\equiv\quad \pi_X(T + \sigma[v : q(v_X, v_Y)](F))\ \mathbf{with}\ (T,F) = \sigma^2[v : p(v_X, v_Y)](X \times Y)\\
&\quad\equiv\quad \pi_X(T + \sigma[v : q(v_X, v_Y)](F))\ \mathbf{with}\ (T,F) = X \underset{x,y:p(x,y)}{\ltimes^2} Y
\end{aligned}$$

We note that two-stream semijoin can be looked upon as the dynamic version of the markjoin operator—recall that we used the equivalence $a \vee b \equiv a \vee (\neg a \wedge b)$ in the proof concerning introduction of the markjoin. The markjoin marks the positive stream with $true$, the negative with $false$. Assuming a conditional or, for example w.r.t. the selection in $\sigma[x : x.m \vee q(x)](X -_{x,y:q(x,y);m} Y)$, it is only necessary to evaluate q for the tuples marked $false$; the tuples marked $true$ can be included into the result without further processing. The ideas underlying both operators are the same: the partitioning of the input set, the further processing of only one partition, and the avoidance of duplicate removal.

The predicate of two-stream selection (and also two-stream semi- and regular join) is required to be predicate without quantification; two-stream operator expressions cannot be rewritten further at will.

We adhere to the following heuristic rule for the transformation of queries that contain disjunction:

Heuristic rule 4.1 For disjunctive selections we use two-stream operators whenever possible, and set union otherwise.

Returning to our example expressions, their translations are:

(2) $T + F \underset{x,z:q(x,z)}{\ltimes} Z \textbf{ with } (T,F) = X \underset{x,y:p(x,y)}{\ltimes} Y$

(4) $\pi_X(T + (F \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z)) \textbf{ with } (T,F) = X \underset{x,y:p(x,y)}{\bowtie^2} Y$

We show the latter rewriting:

Rewriting example 4.12

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet t(x,y,z)](X) \\
&\quad \equiv \quad \pi_X(T + \sigma[v : \exists z \in Z \bullet t(v_X, v_Y, z)](F)) \textbf{ with } (T,F) = X \underset{x,y:p(x,y)}{\bowtie^2} Y \\
&\quad \equiv \quad \pi_X(T + (F \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z)) \textbf{ with } (T,F) = X \underset{x,y:p(x,y)}{\bowtie^2} Y
\end{aligned}$$

Note that the projection cannot be pushed through, because then the two merge operands might contain duplicates: $\pi_A(X + Y) \not\equiv \pi_A(X) + \pi_A(Y)$.

4.6.2 Division versus set difference

The results of queries 8, 16, 24, 31, and 32 contain the expensive division operator. Negated existential (universal) quantification can be handled by means of set difference as well as division:

$$\sigma[x : \nexists y \in Y \bullet p](X) \equiv X - \sigma[x : \exists y \in Y \bullet p](X)$$

In the translation, we have chosen to use division to handle universal quantification. However, in some cases the use of set difference gives better results. Consider for example the expression (case 24 of our list):

$$\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet t(x,y,z)](X)$$

that was rewritten into:

$$(\sigma[v : \neg p(v_X, v_Y) \vee \neg v.m]((X \times Y) \underset{v,z:t(v_X,v_Y,z);m}{-} Z) \div Y$$

A much better result is achieved if we use set difference:

Rewriting example 4.13

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet t(x,y,z)](X) \\
&\quad \equiv \quad X - \sigma[x : \exists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet t(x,y,z)](X) \\
&\quad \equiv \quad X - \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z)
\end{aligned}$$

Because the scope of the quantification with range Y is a conjunction, set difference gives a better result than division. In the latter case, the conjunction becomes a disjunction, whereas set difference leaves the predicate as it is.

On the other hand, a standard use of set difference can give results much worse than those achieved with division. For example, consider case 31 of our list:

$$\sigma[x : \nexists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \wedge r(y,z)](X)$$

which, using set difference for both negations, is translated into:

$$X - \pi_X((X \times Y) - ((X \times Y) \underset{v,z:q(v_X,z)\wedge r(v_Y,z)}{\ltimes} Z))$$

instead of:

$$((X \times Y) \underset{v,z:q(v_X,z)\wedge r(v_Y,z)}{\ltimes} Z) \div Y$$

Whenever the operand of a selection of which the predicate is a negated quantification is an expensive expression (in this case the product $X \times Y$), it does not seem wise to use set difference, because set difference creates common subexpressions. Moreover, in further rewriting either the common subexpression (a Cartesian product) is factorised (!), or it is undone by moving around predicates q and r in further algebraic optimization.

Our conjecture is that set difference should be used only if the selection operand is a base table, or some algebraic expression roughly corresponding to a base table, i.e. a semi- or antijoin. However, probably not even with base table operands set difference should be used in a standard way. Handling the first negation by set difference and the second by division, the result for expression 31 is:

$$X - \pi_X(((X \times Y) \underset{v,z:\neg t(v_X,v_Y,z)}{\ltimes} Z) \div Z)$$

The standard solution that handles negation by division undoes the second. In the latter case, we have obtained an additional set difference, but predicate t is negated. In case t is a disjunction, this has a positive effect, but the effect is negative in case t is a conjunction.

In short, it is difficult to determine exactly when to use set difference instead of division. We adhere to the simple heuristic rule:

Heuristic rule 4.2 Set difference is used to handle negated existential quantification whenever the scope of the quantifier concerned is a conjunction. Division is used otherwise, unless the quantifier scope itself is a non-negated existential quantification.

Thus for queries 24 and 32 of our list a different result is achieved:

(24) $X - \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z)$

(32) $X - \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:t(v_X,v_Y,z)}{\triangleright} Z)$

Note that the results are precisely the complement of the results obtained for expressions 20 and 28, respectively.

In the previous section, we discussed bypass processing. W.r.t. queries 8 and 16, we may use difference to handle the negated quantification, and then apply bypass processing for the disjunction. For query 8, rewriting proceeds as follows:

Rewriting example 4.14 (8) Cyclic

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet t(x,y,z)](X) \\
&\quad\equiv\quad X - \sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet t(x,y,z)](X) \text{ (\textbf{difference})} \\
&\quad\equiv\quad X - \pi_X(T + (F \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z)) \textbf{ with } (T,F) = X \underset{x,y:p(x,y)}{\bowtie^2} Y \text{ (\textbf{case 4})}
\end{aligned}$$

The standard result is:

$$((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:t(v_X,v_Y,z)}{\triangleright} Z) \div Y$$

It is difficult to compare the above results. We have one two-stream join, a semijoin and difference against one regular join, an antijoin, and division.

4.6.3 Constant terms

Assume that we have included a conditional expression into our algebra, then selection predicates that are constants may be handled by means of the following rules:

Rule 4.10 Constant selection predicates Let x not occur free in c, then:

1. $\sigma[x : p \wedge c](X) \equiv \textbf{if } c \textbf{ then } \sigma[x : p](X) \textbf{ else } \emptyset$
2. $\sigma[x : p \vee c](X) \equiv \textbf{if } c \textbf{ then } X \textbf{ else } \sigma[x : p](X)$

However, in doing so, we leave the set-oriented framework. For example, if the rules above are applied to the expression:

$$\sigma[x : \exists y \in Y \bullet y.a = 1](X)$$

it is translated into:

$$\textbf{if } \exists y \in Y \bullet y.a = 1 \textbf{ then } X \textbf{ else } \emptyset$$

If we want to stay within the algebraic framework, we need the additional rule:

Rule 4.11 $\exists x \in X \bullet p \equiv \sigma[x : p](X) \neq \emptyset$

The equivalent expression which uses the conditional probably can be evaluated more efficiently, by breaking off the iteration over Y as soon as some tuple satisfies the predicate.

Heuristic rule 4.3 Constant selection predicates are handled by means of the conditional.

4.6.4 Quantifier exchange

The exchange of quantifiers is a profitable strategy for tree(II) queries. After exchange of quantifiers, descoping can take place, followed by range nest, and then the result is linear. For example, without quantifier exchange, query 3 is translated as follows:

Rewriting example 4.15 (3) Tree(II)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
&\quad\equiv\quad \pi_X(\sigma[v : \exists z \in Z \bullet q(v_X,z) \vee r(v_Y,z)](X \times Y)) \textbf{ (unnest)} \\
&\quad\equiv\quad \pi_X((X \times Y) \underset{v,y:q(v_X,z)\vee r(v_Y,z)}{\ltimes} Z) \textbf{ (unnest)}
\end{aligned}$$

To exchange quantifiers is advantageous whenever it introduces a constant term in the scope of the inner quantification. A constant term in a selection predicate or a quantifier scope is a term in which the corresponding variable does not occur free. For example $p(x)$, with $FV(p) = \{x\}$ is constant in the quantifier scope in $\exists y \in Y \bullet p(x)$, but not in $\exists x \in X \bullet p(x)$.

Heuristic rule 4.4 Quantifiers are exchanged if possible, whenever the result contains a constant term.

4.6.5 Distribution of quantification

In our example rewritings, we distributed existential quantification for queries 11 and 15. For these tree(II) queries, the inner quantification is negated, so that the exchange of outer and inner quantifier is not possible. Distribution of quantification is a non-linear technique, but for the queries concerned it is a profitable strategy. For example, should we rewrite expression 11 *without* distribution, then we have:

Rewriting example 4.16 (11) Tree(II)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
&\quad\equiv\quad \pi_X(\sigma[x : \nexists z \in Z \bullet q(v_X,z) \vee r(v_Y,z)](X \times Y)) \\
&\quad\equiv\quad \pi_X((X \times Y) \underset{v,z:q(v_X,z)\vee r(v_Y,z)}{\triangleright} Z)
\end{aligned}$$

The result is an antijoin with a disjunctive join predicate, the left operand of which is a Cartesian product. The resulting expression can be rewritten further in a purely algebraic way, by means of Rule A.12, for splitting join predicates, and Rule 4.15 and 4.16, for changing the join order:

Rewriting example 4.17

$$\begin{aligned}
&\pi_X((X \times Y) \underset{v,z:q(v_X,z)\vee r(v_Y,z)}{\triangleright} Z) \\
&\quad\equiv\quad \pi_X(((X \times Y) \underset{v,z:q(v_X,z)}{\triangleright} Z) \underset{v,z:r(v_Y,z)}{\triangleright} Z) \textbf{ (split)} \\
&\quad\equiv\quad \pi_X(((X \underset{x,y:q(x,z)}{\triangleright} Z) \times Y) \underset{v,z:r(v_Y,z)}{\triangleright} Z) \textbf{ (exchange)} \\
&\quad\equiv\quad \pi_X((X \underset{x,y:q(x,z)}{\triangleright} Z) \times (Y \underset{y,z:r(y,z)}{\triangleright} Z)) \textbf{ (associate)} \\
&\quad\equiv\quad (X \underset{x,y:q(x,z)}{\triangleright} Z) \underset{x,y:true}{\ltimes} (Y \underset{y,z:r(y,z)}{\triangleright} Z) \textbf{ (push)} \\
&\quad\equiv\quad \textbf{if } (Y \underset{y,z:r(y,z)}{\triangleright} Z) \neq \emptyset \textbf{ then } (X \underset{x,z:q(x,z)}{\triangleright} Z) \textbf{ else } \emptyset \textbf{ (constant } \ltimes \textbf{ predicate)}
\end{aligned}$$

For expressions 11 and 15, distribution is profitable, because it reduces the number of free variables in the quantification with range Z by one, so that the use of partial join operators becomes possible, after further range nesting and descoping. Not distributing the quantifier results into a simpler translation, but a more extensive algebraic optimization process, if we want to achieve the same result. The question is whether distribution of quantification may be a profitable strategy in other cases as well. With respect to our example rewritings, for conjunctive queries (expressions 17–32), quantifiers can be distributed for cyclic queries only in case the second connective is a disjunctive one, but we have not taken into consideration the precise format of this second connective until now. With respect to disjunctive queries, distribution is possible in all cases. We investigate the effect of distribution for queries 1–8; the rewritings are listed in Appendix B, Section B.2.

With respect to chain queries that have the format:

$$\sigma[x : \mathcal{Q}y \in Y \bullet p(x,y) \vee \mathcal{Q}z \in Z \bullet r(y,z)](X)$$

distribution of quantification introduces a constant disjunct, i.e. a disjunct in which the outer loop variable x does not occur free. If we treat constants by means of a conditional expression, then the results for queries 1 and 5 (and analogously for 7 and 13) are:

$$(1)\quad \textbf{if } Y \underset{y,z:r(y,z)}{\ltimes} Z \neq \emptyset \textbf{ then } X \textbf{ else } X \underset{x,y:p(x,y)}{\ltimes} Y$$

$$(5)\quad \textbf{if } Y \underset{y,z:r(y,z)}{\ltimes} Z \neq \emptyset \textbf{ then } \emptyset \textbf{ else } X \underset{x,y:p(x,y)}{\triangleright} Y$$

We conclude that distribution of quantification is advantageous whenever distribution has the effect of introducing constant terms.

With respect to tree(I) queries, distribution does not make sense, because variable y does not occur free in the second disjunct—descoping is an obligatory first step in this case to obtain MNF, which we have chosen as the starting point for translation.

With respect to tree(II) queries that are of the format:

$$\sigma[x : \mathcal{Q}y \in Y \bullet \mathcal{Q}z \in Z \bullet q(x,z) \vee r(y,z)](X)$$

in case 11 and 15 distribution already is the first step in rewriting. For expressions 3 and 7, after distribution descoping becomes possible, which introduces a constant term again, and then the same discussion as above applies. The results are:

(3) $\textbf{if } Z \underset{y,z:r(y,z)}{\ltimes} Y \neq \emptyset \textbf{ then } X \textbf{ else } X \underset{x,z:p(x,z)}{\ltimes} Z$

(7) $\textbf{if } Z \underset{y,z:r(y,z)}{\ltimes} Y \neq \emptyset \textbf{ then } \emptyset \textbf{ else } X \underset{x,z:p(x,z)}{\triangleright} Z$

With respect to cyclic queries (expressions 4 and 8) both disjuncts contain the same set of free variables. From the example rewritings, it does not clearly show that distribution is advantageous. Recall that we decided to apply bypass processing to expression 4, which resulted in:

$$\pi_X(T + (F \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z)) \textbf{ with } (T, F) = X \underset{x,y:p(x,y)}{\bowtie^2} Y$$

For expression 4, the result, after distribution of the quantification, and also performing bypass processing, the result is:

$$T + \pi_F((F \times Y) \underset{v,z:t(v_X,v_Y,z)}{\ltimes} Z) \textbf{ with } (T, F) = X \underset{x,y:p(x,y)}{\ltimes^2} Y$$

Instead of a two-stream join, we have obtained a two-stream semijoin and an additional Cartesian product.

For expression 8, distribution changes the disjunction into a conjunction, and then set difference can be used to handle the negated quantification. However, it is difficult to compare the standard result:

$$((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:t(v_X,v_Y,z)}{\triangleright} Z) \div Y$$

with the result achieved after distribution:

$$X - \pi_X(((X \underset{x,y:p(x,y)}{\triangleright} Y) \times Y) \underset{v,y:t(v_X,v_Y,z)}{\ltimes} Z)$$

Instead of division we have set difference, instead of an antijoin a semijoin, and instead of the join $X \bowtie_{x,y:\neg p(x,y)} Y$, we have $(X \triangleright_{x,y:p(x,y)} Y) \times Y$. For one thing, the cardinality of the join expression probably is larger than that of the antijoin/product expression; in Section B.3 we explain why (see Rewriting example B.49).

We propose the following rule w.r.t. distribution:

Heuristic rule 4.5 Quantification is distributed whenever it leads to (1) one or more disjuncts in which the number of free variables is less than in the original quantifier scope or (2) the modification of a disjunctive predicate into a conjunctive one (i.e., when distributing $\nexists$).

To conclude the present discussion, we note that, whenever set union is used to handle disjunction, distribution of existential quantification has no effect on the outcome of the translation process:

Rewriting example 4.18 Union without distribution

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p_1 \vee p_2](X) \\
&\quad \equiv \quad \pi_X(\sigma[p_1 \vee p_2](X \times Y)) \\
&\quad \equiv \quad \pi_X(\sigma[p_1](X \times Y) \cup \sigma[p_2](X \times Y)) \\
&\quad \equiv \quad \pi_X(\sigma[p_1](X \times Y)) \cup \pi_X(\sigma[p_2](X \times Y))
\end{aligned}$$

Rewriting example 4.19 Union with distribution

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p_1 \vee p_2](X) \\
&\quad \equiv \quad \sigma[x : \exists y \in Y \bullet p_1 \vee \sigma[x : \exists y \in Y \bullet p_2](X) \\
&\quad \equiv \quad \sigma[x : \exists y \in Y \bullet p_1](X) \cup \sigma[x : \exists y \in Y \bullet p_2](X) \\
&\quad \equiv \quad \pi_X(\sigma[p_1](X \times Y)) \cup \pi_X(\sigma[p_2](X \times Y))
\end{aligned}$$

The result is the same for both rewritings.

4.6.6 Range nesting

Range nesting involves pushing through expressions to quantifier ranges, which enables the introduction of partial joins (Rule 4.7). Until now, we have applied range nesting only to predicates that are basic quantifications. By means of our set of rules $\mathcal{R}$, the expression:

$$\sigma[x : \exists y \in Y \bullet a_1(x, y) \wedge a_2(y)](X)$$

in which a_1 and a_2 are atomic, is translated into:

$$X \underset{x,y:a_1(x,y) \wedge a_2(y)}{\ltimes} Y$$

However, it is also possible to push through predicate a_2 to the quantifier range Y:

Rewriting example 4.20

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet a_1(x, y) \wedge a_2(y)](X) \\
&\quad \equiv \quad \sigma[x : \exists y \in \sigma[y : a_2(y)](Y) \bullet a_1(x, y)](X) \\
&\quad \equiv \quad X \underset{x,y:a_1(x,y)}{\ltimes} \sigma[y : a_2(y)](Y)
\end{aligned}$$

We have the following algebraic equivalence rule:

Rule 4.12 Distribution to right join operand Let θ be either $\bowtie$, $\ltimes$, $\rhd$, or $-$, then:

$$X \underset{x,y:p_1(x,y) \wedge p_2(y)}{\theta} Y \equiv X \underset{x,y:p_1(x,y)}{\theta} \sigma[y : p_2(y)](Y)$$

This rule can be derived in a pure algebraic way as well. Range nesting, both of basic quantification (as in Rule 4.7) and of atomic predicates as shown above, can be regarded as the analogue of descoping. Descoping corresponds to the pushing through of predicates to left, and range nesting to right join operands. The rule for range nesting is:

Rule 4.13 Range nesting Let $FV(p) = \{x\}$, then:

$$\exists x \in X \bullet P_{con}(p(x)) \equiv \exists x \in \sigma[x : p(x)](X) \bullet P_{con}(true)$$

In case predicate p contains quantification, the nested selection must be rewritten further.

Heuristic rule 4.6 Range nesting is applied whenever possible.

4.6.7 Common subexpressions

In the transformation of a nested expression to an algebraic one, often common subexpressions are introduced. For example, the rule that translates disjunction into set union, and also the rule that handles negation by means of set difference introduces a common subexpression. Another example is given by the rule for distribution of quantifiers.

Common subexpressions can be named using the **with**-clause, or can be left as they are. Naming of a common subexpression makes further rewriting of the different occurrences impossible. Not naming common subexpressions means that expressions can be further rewritten. This further rewriting may modify the different occurrences in different ways, leading to a separate evaluation. For example, consider the expression:

$$\sigma[p(x) \vee r(y)](X \underset{x,y:t(x,y)}{\bowtie} Y)$$

in which predicates p and r refer only to attributes of X and Y, respectively, and the following rewritings:

Rewriting example 4.21 Naming common subexpressions

$$\begin{aligned}
&\sigma[p(x) \vee r(y)](X \underset{x,y:t(x,y)}{\bowtie} Y) \\
&\quad \equiv \quad \sigma[p(x) \vee r(y)](Z) \textbf{ with } Z = X \underset{x,y:t(x,y)}{\bowtie} Y \\
&\quad \equiv \quad \sigma[p(x)](Z) \cup \sigma[r(y)](Z) \textbf{ with } Z = X \underset{x,y:t(x,y)}{\bowtie} Y
\end{aligned}$$

Rewriting example 4.22 Rewriting common subexpressions

$$\begin{aligned}
&\sigma[p(x) \vee r(y)](X \underset{x,y:t(x,y)}{\bowtie} Y) \\
&\quad \equiv \quad \sigma[p(x)](X \underset{x,y:t(x,y)}{\bowtie} Y) \cup \sigma[r(y)](X \underset{x,y:t(x,y)}{\bowtie} Y) \\
&\quad \equiv \quad \sigma[p(x)](X) \underset{x,y:t(x,y)}{\bowtie} Y \cup X \underset{x,y:t(x,y)}{\bowtie} \sigma[x : r(y)](Y)
\end{aligned}$$

Which of the two result expressions is the cheapest depends on predicate selectivities. The more restrictive the monadic predicates are, the more likely it is that the latter result is the cheapest.

So far, we have presented a standard transformation algorithm, and discussed alternative transformation techniques and extensions. More than often, it is not possible to reach a definitive conclusion about the effect of the various options. In Section 4.8, we will discuss some more example rewritings, which are listed in Appendix B, Section B.3. In some cases, equivalence of results can be easily shown by means of algebraic rewriting, and therefore, in the next section, we present some algebraic equivalence rules that concern the join operators that we use.

4.7 Some algebraic equivalence rules

So far, we have mainly presented rewrite rules for expressions consisting of both calculus-like and pure algebraic constructs. In this section, we list some algebraic equivalence rules.

4.7.1 Join order

To achieve the goal of a linear translation, various non-standard join operators have been used. A disadvantage of these new join operators is that in general they are neither commutative nor associative, which may cause a problem in determining (cost-based) join ordering. In this section we present some of the rules for the rewriting of join sequences that involve the regular join, the semi-, the anti-, and/or the markjoin. The rules presented here resemble those listed in [RoGa90, GaRo90], reporting work on the ordering of join/outerjoin operator sequences.

For the reordering of join sequences, the properties of commutativity and associativity are important. We introduce one other: that of *exchange*, a combination of the properties of associativity and commutativity. Exchange can be used to rearrange the join order of tree queries. Let θ denote an arbitrary join operator, then we ask whether and when the following properties hold:

Commutativity	$X\ \theta\ Y \equiv Y\ \theta\ X$
Associativity	$(X\ \theta\ Y)\ \theta\ Z \equiv X\ \theta\ (Y\ \theta\ Z)$
Exchange	$(X\ \theta\ Y)\ \theta\ Z \equiv (X\ \theta\ Z)\ \theta\ Y$

The discussion focuses on the operators $\bowtie$, $\ltimes$, $\rhd$, and $-$. Of the various join operators, only the regular join is commutative. The basic rules, often used in proofs, are the following:

Rule 4.14 Basic join rules

1. $X \underset{p}{\bowtie} Y \equiv Y \underset{p}{\bowtie} X$ (commutativity)
2. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\bowtie} Z \equiv X \underset{p(x,y)}{\bowtie} (Y \underset{r(y,z)}{\bowtie} Z)$ (associativity)
3. $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)}{\bowtie} Z \equiv (X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)}{\bowtie} Y$ (exchange)

In the rules presented below, predicate t is arbitrary; in the most general case it is a three-variable predicate $t(x, y, z)$.

Rule 4.15 Associativity

1. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z) \wedge t}{\bowtie} Z \equiv X \underset{p(x,y) \wedge t}{\bowtie} (Y \underset{r(y,z)}{\bowtie} Z)$
2. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z) \wedge t}{\ltimes} Z \equiv \pi_{XY}(X \underset{p(x,y) \wedge t}{\bowtie} (Y \underset{r(y,z)}{\bowtie} Z))$
3. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\ltimes} Z \equiv X \underset{p(x,y)}{\bowtie} (Y \underset{r(y,z)}{\ltimes} Z)$
4. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\triangleright} Z \equiv X \underset{p(x,y)}{\bowtie} (Y \underset{r(y,z)}{\triangleright} Z)$

Rule 4.16 Exchange

1. $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z) \wedge t}{\bowtie} Z \equiv (X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y) \wedge t}{\bowtie} Y$
2. $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z) \wedge t}{\ltimes} Z \equiv \pi_{XY}((X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y) \wedge t}{\bowtie} Y)$
3. Let θ^1 and θ^2 be either $\bowtie$, $\ltimes$, or $\triangleright$, then:
   $(X \underset{p(x,y)}{\theta^1} Y) \underset{q(x,z)}{\theta^2} Z \equiv (X \underset{q(x,z)}{\theta^2} Z) \underset{p(x,y)}{\theta^1} Y$
   Which means that exchange can be applied to tree(I) queries regardless of the type of join operators present.

4.7.2 Join distribution

Relationships between joins and set operators are important. We have a number of join operators (including the Cartesian product), and the set operators $\cup$ and $-$ (we do not consider $\cap$). The property we discuss here is distribution. Let θ be an arbitrary join operator, and let Θ be either union or set difference, then we ask whether the following properties hold:

Left distribution $X\ \theta\ (Y\ \Theta\ Z) \equiv (X\ \theta\ Y)\ \Theta\ (X\ \theta\ Z)$

Right distribution $(X\ \Theta\ Y)\ \theta\ Z \equiv (X\ \theta\ Z)\ \Theta\ (Y\ \theta\ Z)$

In Table 4.4, we list our findings.

Rule 4.17 Joins and set operators

Join operators distribute left or right over union and set difference as indicated in Table 4.4, a "1" meaning that distribution holds, a "0" meaning that it does not.

The proofs are simple, and we do not give them, except for the proof that concerns product and set difference (see Appendix A). As an aside, in proving the equivalences sometimes the algebraic formalism is more convenient, but there are also cases in which the calculus-like formalism leads to the result faster. Assume that join predicate p involves equality of set elements, then a counterexample for those cases in which distribution does not hold is the instantiation $X = \{1, 2, 3\}$, $Y = \{2, 3\}$, $Z = \{1, 2\}$.

operators	left	right
$\times/\cup$	1	1
$\bowtie/\cup$	1	1
$\ltimes/\cup$	1	1
$\rhd/\cup$	0	1
$-/\cup$	0	1
$\times/-$	1	1
$\bowtie/-$	1	1
$\ltimes/-$	1	1
$\rhd/-$	0	1
$-/-$	0	1

Table 4.4: Join distribution

4.7.3 Division

The query that lists the suppliers supplying all parts:

$$\pi_1(\sigma[x : \forall y \in P \bullet \exists z \in SP \bullet x[1] = z[1] \wedge y[1] = z[2]](SP))$$

which corresponds to expression 31 of our list, is translated into the algebraic expression:

$$\pi_1(\pi_{1,2,3}(((SP \underset{x,y:x[1]=y[1]}{\bowtie} SP) \underset{v,z:v[3]=z[2]}{\bowtie} P)) \div [2/1]P)$$

The above expression was shown to be equivalent to the expression $SP \div [2/1]P$ [Naka90]; we did not achieve a result better than the result which is obtained by means of the reduction algorithm of Codd. Further optimization is required. We must either improve the translation, or perform further algebraic rewriting. Our conjecture is that algebraic rewriting of expressions that contain division is much harder than finding better translation rules.

4.8 Evaluation

In Section B.3, we have listed some more example rewritings. The input queries are the cyclic queries of our list (expressions 4, 8, 12, 16, 20, 24, 28, and 32). The second connective present is now taken into consideration too. We compare the results achieved using the reduction algorithm of Codd, the results achieved by means of standard rewriting with $\mathcal{R}$, and consider some of the results achieved after distribution of quantification. In some cases, we perform some algebraic rewriting as well.

Generally, the results achieved with $\mathcal{R}$ are better than those achieved by means of the algorithm of Codd. The number of division operators is less, and the number of partial join operators higher. In addition, distribution of quantification may be helpful. We note that:

- The Miniscope Normal Form is advantageous, because it enables the use of partial join operators.
- The exchange and distribution of quantification may enable descoping. Distribution of quantification seems to be advantageous if it leads to a reduction of free variables, or if a disjunction is changed into a conjunction. But distribution in some cases has a negative effect.
- In case relatively simple algebraic operators like joins are involved, algebraic rewriting can do much to achieve an optimized result. However, if division is involved, the result achieved by means of the algorithm of Codd cannot always be rewritten by means of purely algebraic rewriting, simply because we do not have at our disposal the proper algebraic equivalences rules.

Our research approach can be characterized as 'research by example'. By tediously working out examples, for one thing, it is shown that it is not easy to find good heuristics to guide the transformation. We have shown that, under certain conditions, specific techniques like distribution of quantification and descoping are advantageous, but there always remain notable exceptions. The transformation of nested queries is a complex process, and the rewrite steps are not mutually independent. Decisions taken in one phase of the process influence the possible course of action in subsequent rewritings. Usually, the (final) effect of certain decisions cannot be foreseen. To solve the problem of how to achieve an efficient transformation, techniques from the field of artificial intelligence such as machine learning might be appropriate.

For the transformation of expressions we have at our disposal a basic set of rules $\mathcal{R}$, and an algorithm that prescribes the order of rule application. The transformation is deterministic, i.e. only one solution is generated, and the result is always the same. The transformation can be modified in two different ways: by rearrangement of the order of rule application, and by modification of the rule set by modifying or adding transformation rules.

Given an expression, the result obtained with standard rewriting sometimes can be improved; experiments may be carried out to decide when and how better results can be achieved. Alternative solutions can be generated by modification of the transformation algorithm. Assume that some transformation rule is added, for example the rule that prescribes to distribute quantifiers, whenever possible. We have two possibilities: either we can prove that the modification is always beneficial, or we cannot provide such a proof. In the latter case, we must compare the result achieved by standard rewriting with the result achieved by means of the modified algorithm on the basis of specific example rewritings. To compare results, we must have a cost model. Because we are involved in logical rewriting, the cost model is based on qualitative cost parameters. Possible cost parameters are the number of operators, the relative cost of the various operators, the size and cardinality of operands, etc. To design such a logical cost model is not an easy task—as we have seen, sometimes it is hard to reach decisive conclusions with respect to the relative cost of expressions.

Given a cost model, for some example expressions, the result achieved by the modified algorithm may be better, for others it may be worse, so the exact conditions under

which to apply the additional rule must be determined. The transformation algorithm, i.e., the rewrite rule added needs refinement. A characterization of expressions is needed to decide when or not to use the modification. For example, while investigating the effect of distribution of quantification, in the first instance we decided to distribute quantification whenever possible (see Section 4.6.5). We evaluated the results achieved for some example expressions, and then decided to distribute quantification only in case that it leads to a reduction of free variables or if a conjunction is changed into a conjunction.

At first glance, our method can be qualified as symbolic learning by induction. In [Hopg93], this is described as follows. Rule induction involves the generation of general rules from specific examples. The accuracy of a specific rule (application of this rule leads to the most efficient algebraic expression) is never certain—it can be invalidated by a counterexample. The aim is to build rules that are as general as possible (in fact, the rule set $\mathcal{R}$ that we started with is rather general) and to modify them when they are found to be wrong (in our case, the search for counterexamples is an active generate-and-test process). The rules should match the positive examples but not the negative ones.

4.9 Summary

In this chapter, we made an attempt to achieve an efficient translation of relational calculus into relational algebra. To enable an efficient translation, i.e., a translation that results in efficient algebraic expressions, we combined optimization with translation, and extended relational algebra with some non-standard join operators. An efficient translation was defined as a linear translation in which Cartesian products are avoided, selections and projections are pushed through as far as possible, and partial joins (which preserve only one of the join operands) are given preference over full joins (which preserve both operands). The translation algorithm was described in the form of a set of equivalence (or rewrite) rules in the language that is a mixture of calculus and algebra. The main task in translation is the removal of quantification from selection predicates—the emphasis is on the transformation of nested queries, i.e., queries that involve nested quantifier expressions.

Our approach is systematic, and deals with disjunction as well as universal quantification. We start with expressions of which selection predicates are in MNF; quantification is removed from selection predicates by means of the introduction of partial join operators whenever possible. We noticed that our standard translation algorithm can be improved, and we discussed some additional rewrite techniques and alternative solutions that may be used to achieve better results. Some of these techniques are generally beneficial, other are beneficial under certain conditions only. However, the result of translation is still unnecessarily inefficient in some cases.

If anything, the conclusion that has to be drawn from the discussion in this chapter is that, even for the simple relational calculus that we defined, translation into the algebra is a complicated issue. Translation of an SQL query language into an algebra usually is considered as the task of the parser [GrMc93]. In our opinion, such a transformation cannot be left to a parser, which we see as a relatively simple syntax-manipulating device. To provide a good transformation, i.e., a transformation that provides a good starting point for

further optimization, one has to think about the logical operators that should be offered, and about the proper mapping of the one language into the other. Transformation of SQL into the algebra provides a starting point for logical and cost-based optimization. Logical optimization mainly involves pushing through selections and projections. Cost-based optimization is done by generating equivalent physical algebra expressions, and choosing the best among them; the number of expressions that actually can be generated is limited. A naive mapping of SQL into the algebra may result into a logical algebra expression such that even the optimized physical algebra expression will never have, or even come close to the performance of an expression that is the result of an 'intelligent' transformation.

References

[Bry89] Bry, F., "Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited," *Proceedings ACM SIGMOD*, Portland, Oregon, June 1989, pp. 193–204.

[CeGo85] Ceri, S. and G. Gottlob, "Translating SQL into Relational Algebra: Optimization, Semantics, and equivalence of SQL Queries," *IEEE Transactions on Software Engineering*, Vol. SE–11, No. 4, April 1985.

[Codd72] Codd, E.F., "Relational Completeness of Data Base Sublanguages," in: *Data Base Systems*, ed. R. Rustin, Prentice Hall, 1972.

[Daya87A] Dayal, U. "Processing Queries with Quantifiers: A Horticultural Approach," *Proceedings ACM PODS*, 1983.

[GaRo90] Galindo-Legaria, C. and A. Rosenthal, "How to Extend a Conventional Optimizer to Handle One- and Two-Sided Outerjoin," *Proceedings IEEE Conference on Data Engineering*, Tempe, Arizona, February 1992, pp. 402–409.

[Grae93] Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2), June 1993, pp. 73–170.

[GrMc93] Graefe, G. and W.J. McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search," *Proceedings of the IEEE Conference on Data Engineering*, 1993, pp. 209–218.

[Hopg93] Hopgood, A.A., **Knowledge-Based Systems for Engineers and Scientists**, CRC Press, Boca Raton, Florida, 1993.

[JaKo84] Jarke, M. and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, 16(2), June 1984, pp. 111–152.

[Kamb85] Kambayashi, Y, "Cyclic Query Processing," in: *Query Processing in Database Systems*, eds. W. Kim, D.S. Reiner, and D.S. Batory, Springer Verlag, 1985, pp. 62–78.

[KMPS94] Kemper, A., G. Moerkotte, K. Peithner, and M. Steinbrunn, "Optimizing Disjunctive Queries with Expensive Predicates," *Proceedings ACM SIGMOD*, Minneapolis, Minnesota, 1994, pp. 336–347.

[KMPS95] Kemper, A., G. Moerkotte, K. Peithner, and M. Steinbrunn, "Bypassing Joins in Disjunctive Queries," *Proceedings VLDB*, Zürich, Switzerland, 1995.

[Klug82] Klug, A., "Equivalence of Relational Algebra and Relational Calculus Queries Having Aggregate Functions," *Journal of the ACM*, 29(3), July 1982, pp. 699–717.

[Naka90] Nakano, R., "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS*, 15(4), December 1990, pp. 518–557.

[PBGG89] Paredaens, J., P. de Bra, M. Gyssens, and D. van Gucht, **The Structure of the Relational Database Model**, EATCS Monographs on Theoretical Computer Science, eds. W. Brauer, G. Rozenberg, and A. Salomaa, Springer-Verlag, 1989.

[RoGa90] Rosenthal, A. and C. Galindo-Legaria, "Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins," *Proceedings ACM SIGMOD*, Atlantic City, 1990.

[Ullm89] Ullman, J.D., **Database and Knowledge-base Systems**, Computer Science Press, Rockville, USA, 1989.

[Zhon89] Zhongwan, L., **Mathematical Logic for Computer Science**, World Scientific Publishing, 1989.

Chapter 5

Optimization of Nested Queries in a Complex Object Model

This chapter contains a reprint from:

Steenhagen, H. J., P. M. G. Apers, and H. M. Blanken, “Optimization of Nested Queries in a Complex Object Model,” *Proceedings EDBT*, Cambridge, March 1994, pp. 337–350.

Optimization of Nested Queries in a Complex Object Model

Hennie J. Steenhagen Peter M. G. Apers Henk M. Blanken

Department of Computer Science, University of Twente
PO Box 217, 7500 AE Enschede, the Netherlands
{hennie,apers,blanken}@cs.utwente.nl

Abstract

Transformation of nested SQL queries into join queries is advantageous because a nested SQL query can be looked upon as a nested-loop join, which is just one of the several join implementations that may be available in a relational DBMS. In join queries, dangling (unmatched) operand tuples are lost, which causes a problem in transforming nested queries having the aggregate function COUNT between query blocks–a problem that has become well-known as the COUNT bug. In the relational context, the outerjoin has been employed to solve the COUNT bug. In complex object models supporting an SQL-like query language, transformation of nested queries into join queries is an important optimization issue as well. The COUNT bug turns out to be a special case of a general problem being revealed in a complex object model. To solve the more general problem, we introduce the *nestjoin* operator, which is a generalization of the outerjoin for complex objects.

5.1 Introduction

Currently, at the University of Twente, work is being done on the high-level object-oriented data model TM. TM is a database specification language incorporating standard object-oriented features such as classes and types, object identity, complex objects, and multiple inheritance of data, methods, and constraints. In TM, methods and constraints are specified in a high-level, declarative language of expressions. An important language construct is the SELECT-FROM-WHERE (SFW) construct. The SFW-construct of TM is comparable to the SFW-query block from HDBL, the query language of the experimental DBMS AIM [PiAn86]. HDBL is an orthogonal extension of SQL for extended NF^2 data structures. Optimization of TM SFW-expressions therefore has much in common with optimization of the SFW-expressions of SQL and HDBL.

Optimization of SQL queries has received quite some attention the last decade. An important problem in this area is the optimization of nested SQL queries [Kim82, GaWo87, Daya87B, Mura89, Mura92]. SQL offers possibilities to formulate nested queries: SFW-query blocks containing other SFW-blocks in the WHERE clause. In [Kim82], it was

pointed out that it is advantageous to replace nested SQL queries by flat, or join queries. Flat SQL queries are SFW-blocks not containing subqueries in the WHERE clause. Replacing nested SQL queries by join queries is advantageous because a nested SQL query can be looked upon as a nested-loop join, which is just one of the several join implementations possible. After rewriting a nested query into a join query, the optimizer has better possibilities to choose the most appropriate join implementation.

In nested queries, inner operand tuples are grouped by the values of the outer operand tuples. Whenever aggregate functions occur between query blocks, the transformed, i.e. join query also requires grouping (expressed by means of the GROUP BY clause). In nested queries, dangling outer operand tuples, i.e. outer operand tuples that are not matched by any of the inner operand tuples, deliver a subquery result equal to the empty set. Transformation of a nested query into a join query causes the loss of dangling tuples. In the relational context, this may cause a problem when the aggregate function COUNT occurs between query blocks. As a solution to this problem (that has become well known as the COUNT bug), it has been proposed to use the outerjoin instead of the regular join [GaWo87].

In complex object models supporting an SQL-like query language, transformation of nested queries into join queries is just as important as in the relational context. However, in complex object models grouping of the inner operand is required not only if aggregate functions occur between query blocks, but in many other cases as well. The reason for this is that attributes may be set valued. Moreover, each time grouping is necessary, we have to deal with some kind of COUNT bug caused by the loss of dangling tuples, i.e. the COUNT bug is just a special case of a more general problem being revealed in a complex object model. An important result of this paper is that from the form of the predicate between query blocks it can easily be derived when grouping is not necessary. Nested queries that do not require grouping can be transformed into join queries; for the efficient and correct processing of nested queries that do require grouping a new join operator is introduced–the *nestjoin* operator.

Instead of producing the concatenation of every pair of matching tuples as in the regular join operation, in the nestjoin operation each left operand tuple is extended with the *set* of matching right operand tuples. This way two birds are killed with one stone: grouping is performed, and also dangling tuples are preserved. Implementation of the nestjoin operator is a simple modification of any common join implementation method, however, like the outerjoin operator, the nestjoin has limited rewrite possibilities compared to the regular join operator.

In general, in the logical optimization of a declarative query language, two approaches can be distinguished: (1) rewriting expressions in the query language itself and (2) translation into and rewriting in some intermediate language, for example an algebraic language. Also a combination of the two approaches is possible. For the logical optimization of TM, we have defined the language ADL, an algebra for complex objects which is an extension of the NF^2 algebra of [ScSc86]. This work will be used in the ESPRIT III project IMPRESS (Integrated, Multi-Paradigm, Reliable, and Extensible Storage System). The IMPRESS project started in 1992, and one of the subtasks is to translate (a subset of) the language TM into an algebra for complex objects resembling ADL. In this paper, our ideas

with regard to query transformation will be presented using the language TM; we will not introduce the algebra for reasons of simplicity.

The structure of this paper is as follows. In Section 5.2, we briefly review the work that has been done with regard to nested SQL queries. In Section 5.3, we describe the language TM and the types of nested SFW-expressions that are of interest for the purpose of this paper. Nesting in the WHERE clause and in the SELECT clause are discussed in Sections 5.4 and 5.5, respectively, and we will see that in the transformation of nested queries into join queries in many cases grouping is needed, each case leading to some kind of COUNT bug if relational transformation techniques are used. Then, in Section 5.6, we introduce the nestjoin operator. The nestjoin is an operator that allows efficient processing of nested queries that cannot be transformed into join queries without grouping. Bugs are avoided by preserving dangling tuples. In Section 5.7 we show, for two-block queries, which types of nested queries can be transformed into join queries without problem. An example of query processing for an arbitrary linear nested query (having only one subquery per WHERE clause) is then given in Section 5.8. The paper is concluded by a section discussing future work.

5.2 Nested SQL queries

In this section we briefly review the work that has been done on optimization of nested SQL queries. We do not give a complete overview; we merely indicate the ideas behind optimization of nested SQL queries with a view on the additional problems that come up with optimization of nested queries in a data model supporting complex objects.

Nested SQL queries are SFW-query blocks containing other (possibly nested) SFW-query blocks in the WHERE clause. For example, assume we have relation schemas $R(A, B, C)$ and $S(C, D)$, and consider the following SQL query:

SELECT *
FROM R
WHERE $R.B$ IN SELECT $S.D$
FROM S
WHERE $R.C = S.C$

Disregarding duplicates, the nested query given above is easily transformed into the flat, or join query:

SELECT $R.A$, $R.B$, $R.C$
FROM R, S
WHERE $R.B = S.D$ AND $R.C = S.C$

which, in relational algebra, is simply a join between tables R and S followed by a projection on R, i.e. a semijoin. The advantage of transforming nested queries to join queries is clear: a nested-loop join is just one of the several possible implementations of the join operator, and after transformation to a join query the optimizer can choose the most suitable

join execution method. The method chosen may be a nested-loop join, but not necessarily– alternative join implementations are the sort-merge join, the hash join etc.

In [Kim82], five types of nesting have been distinguished and an algorithm has been given to transform nested queries into join queries for each of these different types of nesting. In case aggregate functions occur between query blocks (one of the types of nesting) SQL's GROUP BY clause is employed to compute the aggregates needed. However, in [Kies84] it has been shown that Kim's algorithm is not correct if the aggregate function COUNT occurs between query blocks. This flaw has become known as the COUNT bug. Consider the query:

SELECT *
FROM R
WHERE $R.B$ = SELECT COUNT (*)
FROM S
WHERE $R.C$ = $S.C$

Following Kim's algorithm, we get the following queries:

(1) $T(C, CNT)$ = SELECT $S.C$, COUNT (*)
FROM S
GROUP BY $S.C$

SELECT $R.A, R.B, R.C$
FROM R, T
WHERE $R.B = T.CNT$ AND $R.C$ = $T.C$

Alternatively, if the relation R does not contain duplicates, the nested query may be transformed into:

(2) SELECT $R.A, R.B, R.C$
FROM R, S
WHERE $R.C$ = $S.C$
GROUP BY $R.A, R.B, R.C$
HAVING $R.B$ = COUNT ($S.C$)

In the former, grouping of the inner operand and computation of the aggregate precedes the join operation; in the latter the join is executed first.

The queries resulting from the transformations do not give the correct result. In the original, nested query, dangling R-tuples for which $R.B = 0$, are included in the result; these tuples are lost in the join queries.

To solve the COUNT bug, it has been proposed in [GaWo87] to modify (2) by using outerjoins instead of joins in case the COUNT function occurs between query blocks. The right outerjoin operator preserves dangling tuples of the left join operand: unmatched left operand tuples are extended with NULL values in the right operand attribute positions.

As another solution, it has been proposed in [Mura92] to modify (1), because in some cases (1) is more efficient than (2). It is proposed to have two types of join predicates in the second query of (1): a regular join predicate and an additional, so-called antijoin predicate,

to be applied to the dangling tuples. In the example given above the antijoin predicate would be: $R.B = 0$. In Kim's second query the join is replaced by an outerjoin operation with join predicate $R.C = T.C$; to the tuples that match the predicate $R.B = T.CNT$ is applied, and to the unmatched tuples in R the antijoin predicate $R.B = 0$ is applied.

5.3 Nested TM queries

5.3.1 General description of TM

In this section we describe the features of TM that are important for the purpose of this paper–support for complex objects and the SELECT-FROM-WHERE construct. We refer the reader to [Bal et al. 93, BaBZ93, BaVr91] for a more comprehensive description of TM.

TM is a high-level, object-oriented database specification language. It is formally founded in the language FM, a typed lambda calculus allowing for subtyping and multiple inheritance. Characteristic features of TM are the distinction between types, classes, and sorts, support for object identity and complex objects, and multiple inheritance of attributes, methods, and constraints. In TM, attribute types may be arbitrarily complex: the type constructors supported are the tuple, variant, set, and list type constructor; type constructors may be arbitrarily nested. Besides basic types, class names may be used in type specifications. Sets do not contain duplicates.

In constraint and method specifications we may use the SELECT-FROM-WHERE construct, having the following general format:

SELECT <result expression>
FROM <operand expression> <variable>
WHERE <predicate>

The meaning of the SFW-expression is as follows. The operand expression is evaluated; a variable is iterated over the resulting set; for each value of the variable it is determined whether the predicate holds, and if so, the result expression is evaluated and this value is included in the resulting set.

5.3.2 Types of nesting in TM

One important difference between SQL on the one hand, and TM and HDBL on the other is that TM and HDBL are *orthogonal* languages. The operand and result expression of the SFW-query block of TM may be arbitrary expressions, also containing other (nested) SFW-expressions, provided they are correctly typed. The predicate may also be built up from arbitrary expressions (including quantifiers FORALL and EXISTS), as long as it delivers a Boolean result.

We give some examples of SFW-expressions. Assume we have specified classes 'Employee' and 'Department':

```
CLASS Employee WITH EXTENSION EMP
ATTRIBUTES
        name : STRING,
        address : Address,
        sal : INT,
        children : ℙ⟨name : STRING, age : INT⟩
END Employee

CLASS Department WITH EXTENSION DEPT
ATTRIBUTES
        name : STRING,
        address : Address,
        emps : ℙEmployee
END Department

SORT Address
TYPE ⟨street : STRING, nr : STRING, city : STRING⟩
END Address
```

The symbol $\mathbb{P}$ denotes the set type constructor, brackets $\langle\rangle$ denote the tuple type constructor. In TM, class extensions are explicitly named. The class 'Employee' has four attributes, of which the attribute 'address' has a complex type specified as a sort. Sorts are used to describe commonly used types such as 'Address', 'Date', 'Time' etc.

$Q1$: Select the departments that have at least one employee living in the same street the department is located.

```
SELECT d
FROM DEPT d
WHERE ⟨s = d.address.street, c = d.address.city⟩
        IN SELECT ⟨s = e.address.street, c = e.address.city⟩
           FROM d.emps e
```

$Q2$: Select for all departments the names of the departments and the employees living in the same city the department is located.

```
SELECT ⟨dname = d.name, emps = SELECT e
                               FROM EMP e
                               WHERE e.address.city = d.address.city⟩
FROM DEPT d
```

We make a distinction in the types of nested queries. In a SFW-expression, other SFW-expressions may occur in the SELECT clause (query $Q2$), in the FROM clause, and in the WHERE clause (query $Q1$). In this paper, the expressions of interest are nested SFW-expressions having subqueries in which free variables (correlated subqueries) occur; subqueries without free variables simply are constants. We do not consider SFW-expressions with subqueries in the FROM clause, because these can be rewritten easily. Furthermore,

operands of subqueries may be either set-valued attributes, as in query Q1, or distinct tables, as in query Q2. Only if subquery operands are distinct tables, transformation to join queries is desirable. There is no use to flatten nested queries in which subquery operands are set-valued attributes, because set-valued attributes are stored with the objects themselves (as materialized joins), at least conceptually.

In short, the nested queries of interest are SFW-expressions having subqueries in the SELECT- and/or WHERE clause containing free variables and having distinct tables as operands. Initially, we will restrict ourselves to two-block queries. In Section 5.8 we will briefly discuss queries with multiple nesting levels.

5.4 Nesting in the WHERE clause

Assume we have a two-block query with one-level deep nesting. The general format of such a query is:

SELECT $F(x)$
FROM $X\ x$
WHERE $P(x, z)$
WITH z = SELECT $G(x, y)$
FROM $Y\ y$
WHERE $Q(x, y)$

The WITH clause is a TM construct enabling local definitions, used here to facilitate the description of the syntactical form of the predicate P. In this paper, we do not consider multiple subqueries, $P(x, z)$ contains only one occurrence of z.

We want to transform the nested query into a join query of the following format (remember that, in SQL, grouping is necessary only if aggregate functions occur between query blocks):

SELECT $F(x)$
FROM $X\ x, Y\ y$
WHERE $P'(x, v) \wedge Q(x, y)$
WITH $v = G(x, y)$

For notational convenience, also the expression $G(x, y)$ has been named by means of a WITH clause.

The goal of the transformation process is to transform the predicate $P(x, z)$, whose second argument z is set valued, into a predicate $P'(x, v)$, where values v are the members of z. The types of P and P' clearly differ: from the second argument of P a set constructor is removed, resulting in predicate P'.

5.4.1 Example predicates

Assume that the predicate P only involves attribute a of the outer operand X and z, the subquery result. Because the attribute a may be set valued, this (already restricted) pred-

icate between query blocks may take many different forms. We may have for instance:[1]

- $x.a$ *OP* z with *OP* $\in \{\in, \subset, \subseteq, =, \supseteq, \supset, \ni\}$,
- expressions involving quantifiers, for example $\exists s \in z\ (s = x.a)$,
- $x.a$ *OP* $H(z)$ with H an aggregate function and *OP* an arithmetical comparison operator,
- expressions involving set operators, for example $x.a \cap z = \emptyset$, or
- negations of expressions listed above.

Predicates can be divided into two groups: predicates that require grouping of the inner operand tuples and the predicates that do not. In Section 5.7, we give a formal characterization of predicates that do and do not require grouping; below, the need for grouping is discussed more informally.

Grouping is not necessary if the question whether outer operand tuples belong to the result or not (whether, for some outer operand tuple, the predicate evaluates to $true$ or $false$) can be answered on the basis of the individual members of the subquery result, i.e. by *scanning* the subquery result. For example, consider the expression $x.a \in z$. The moment we encounter a tuple y in the inner operand Y such that the condition $Q(x, y)$ holds and $x.a$ equals the value of v, we know that tuple x belongs to the result. If no such v is found in the end, the predicate evaluates to $false$.

Grouping is necessary if the subquery result has to be available *as a whole* to decide whether the predicate holds. In this case, all tuples belonging to the subquery result must be kept, because the predicate can only be evaluated having all values in the subquery result at hand. An obvious example of a predicate requiring grouping is the expression $x.a =$ COUNT(z): not until the entire subquery result is at our disposal it is possible to compute (or output, if accumulated) the cardinality of the subquery result. Another predicate requiring grouping is for example the expression $x.a \subseteq z$.

Whenever a predicate needs grouping, we have to deal with some sort of COUNT bug if the nested query is transformed according to the algorithm of [Kim82]. For example, the nested query:

SELECT x
FROM X x
WHERE $x.a \subseteq z$ WITH z = SELECT $y.a$
FROM Y y
WHERE $x.b = y.b$

is, following the ideas of [Kim82], transformed into the following TM queries:

T = SELECT $\langle\ b = y.b,\ as$ = SELECT $y'.a$
FROM Y y'
WHERE $y'.b = y.b\ \rangle$
FROM Y y

[1] In the rest of the paper we will use the common set-theoretical notation for comparison operators and boolean connectives occurring in TM-predicates because of its conciseness.

SELECT x
FROM $X\ x, T\ t$
WHERE $x.b = t.b \wedge x.a \subseteq t.as$

The first query groups the set of $y.a$ values by the values of the attribute b (cf. the operator nest (ν) from the NF^2 algebra [ScSc86]). The transformed query also suffers from a bug (which we might call the SUBSETEQ bug in this case): X-tuples for which $x.a = \emptyset$ that are not matched by any t-tuple on the condition $x.b = t.b$ are lost.

In summary, in TM grouping of the inner operand is required not only if aggregate functions occur between query blocks, but in many other cases too. If Kim's solution is chosen, the transformed query will suffer from a bug each time grouping is needed. We will not use the outerjoin operator to solve these bugs. Instead, in Section 5.6, we will introduce the nestjoin operator, which is a much cleaner solution in a model supporting complex objects.

5.5 Nesting in the SELECT clause

In this section, we will show that, with one notable exception, nesting in the SELECT clause always requires grouping of the inner operand. SFW-expressions having subqueries in the SELECT clause are not new. In HDBL, it is also allowed to have SFW-query blocks in the SELECT clause. SFW-expressions nested in the SELECT clause commonly describe nested results, as in query Q2 from Section 5.3.2, where employees are grouped by departments. Consider the general format of a two-block query with nesting in the SELECT clause:

SELECT $F(x, z)$
  WITH z = SELECT $G(x, y)$
    FROM $Y\ y$
    WHERE $Q(x, y)$
FROM $X\ x$
WHERE $P(x)$

If this query is to be transformed into a join query, the inner operand values have to be grouped by the outer operand values. Grouping may take place preceding or following the join. In both cases again dangling tuples are lost.

With regard to nesting in the SELECT clause, there is one special case in which grouping can be avoided. In TM, a SFW-expression may be nested directly in the SELECT clause, meaning the result is a set of sets. This set of sets may be 'collapsed' by applying the operator UNNEST, which is defined as UNNEST$(S) = \bigcup\{s \mid s \in S\}$. Consider the following query:

UNNEST (SELECT (SELECT $\langle a = x.a, b = y.b \rangle$
    FROM $Y\ y$
    WHERE $x.b = y.a$)
  FROM $X\ x$

This nested query is equivalent to the join query:

SELECT $\langle a = x.a, b = y.b \rangle$
FROM $X\ x, Y\ y$
WHERE $x.b = y.a$

5.6 The nestjoin operator

In the previous sections we have shown that in a complex object model, in many cases grouping seems to be an essential step in the transformation of nested queries to join queries. Queries requiring grouping may always be handled by means of nested-loop processing, which gives correct results but may be very inefficient. If we, though, choose to transform nested queries into join queries, we have to take special measures when queries need grouping because dangling tuples are lost. In the relational model the outerjoin is used to take care of dangling tuples: for subqueries that deliver empty sets the NULL is used to represent the empty set. In a complex object model, however, we do not have to represent the empty set: *the empty set is part of the model*.

Definition

The nestjoin operator, denoted by the symbol Δ, is simply a modification of the join operator. Instead of producing the concatenation of every pair of matching tuples, for each left operand tuple a set is created to hold the (possibly modified) right operand tuples that match. The nestjoin of two tables X and Y on predicate Q with function G (the function applied to the right-hand tuples satisfying the join predicate) is defined as:

$$X \underset{x,y:Q,G;a}{\Delta} Y \stackrel{d}{=} \{x \mathbin{++} \langle a = z \rangle \mid x \in X \wedge z = \{G(x,y) \mid y \in Y \wedge Q(x,y)\}\}$$

In this expression, $x \mathbin{++} \langle a = z \rangle$ denotes the concatenation of the tuple x and the unary tuple $\langle a = z \rangle$, in which a is an arbitrary label not occurring on the top level of X. An example of the nestjoin operation is found in Figure 5.1, where flat relations X and Y are equijoined on the second attribute (the join function is the identity function). Note that for dangling tuples $x \in X$, the tuple $x \mathbin{++} \langle a = \emptyset \rangle$ is present in the result.

The nestjoin operation is a neatly defined operation. Grouping, which is performed during the join, is made explicit by means of a set-valued attribute. Because dangling tuples are preserved, bugs like the COUNT bug are prevented.

Algebraic properties

Assuming that the nestjoin function is identity, the nestjoin can be expressed using the outerjoin, denoted by the symbol $\odot$, and the nest operator ν^*:

$$X \underset{x,y:Q,id;a}{\Delta} Y \equiv \nu^*_{X;a}(X \underset{x,y:Q}{\odot} Y)$$

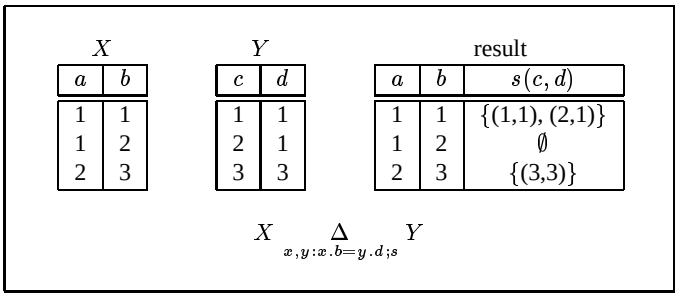


Figure 5.1: Nestjoin example

In this algebraic expression, the operator ν^* is a slightly modified version of the standard nest operator performing nesting in the usual way, and then mapping nested sets consisting of a NULL-tuple to the empty set [Scho86]. By using the nestjoin instead of the outerjoin followed by the nest operator, we do not have to resort to NULLs to avoid the loss of dangling tuples.

A disadvantage of the nestjoin operator is that it, like the outerjoin, has less pleasant algebraic properties. For example, the nestjoin operation is neither commutative nor associative. As another example, the nestjoin does not always associate with the regular join: $X \ \Delta \ (Y \bowtie Z)$ is not equivalent to $(X \ \Delta \ Y) \bowtie Z$, the two expressions already being typed differently. Below we list some examples of algebraic equivalences concerning the nestjoin operator.

Let $X \ \Delta_p \ Y$ denote a nestjoin operation on predicate p in which the nestjoin function equals the identity function (for simplicity omitting variable names and the nestjoin label). Let $r(a, b)$ denote a predicate referencing attributes in tables A and B (and no other), then we have:

- $\pi_X(X \underset{r(x,y)}{\Delta} Y) \equiv X$
- $(X \underset{r(x,y)}{\bowtie} Y) \underset{r(x,z)}{\Delta} Z \equiv (X \underset{r(x,z)}{\Delta} Z) \underset{r(x,y)}{\bowtie} Y)$
- $(X \underset{r(x,y)}{\bowtie} Y) \underset{r(y,z)}{\Delta} Z \equiv X \underset{r(x,y)}{\bowtie} (Y \underset{r(y,z)}{\Delta} Z)$

Implementation

To implement the nestjoin, common join implementation methods like the sort-merge join, or the hash join can be used. However, some restrictions hold. First, in nestjoin implementations, an output tuple can be produced not until the entire set of matching right operand tuples has been found. Second, because in the nestjoin operation the output has to be grouped according to the values of the left operand tuples, the choice for outer and inner loop operand is restricted. For example, in a (simple) hash join implementation, if the join attribute does not form a key attribute of the right join operand, only the right join operand may be the build table. (For the regular join, usually the smaller operand is chosen as the build table.)

Use

The nestjoin operation can be employed to process queries with nesting in the SELECT- as well in the WHERE clause. Queries having subqueries in the SELECT clause often describe nested results, so processing by means of the nestjoin operation will be an appropriate method of processing. For queries with nesting in the WHERE clause, however, sometimes there are other, more efficient possibilities. In the following section we show in which cases grouping certainly is not necessary, so that, instead of the nestjoin operator, we may employ some flat join operator to obtain the results needed.

5.7 The need for grouping

In this section, we present a class of predicate expressions for which it is known that grouping is *not* necessary. Again consider the general format of a two-block query with nesting in the WHERE clause given in Section 5.4, then we have:

Theorem 5.1 Grouping is *not* necessary if the predicate expression $P(x, z)$ can be rewritten into a calculus expression of the form (1) $\exists v \in z\ (P'(x, v))$ or (2) $\nexists v \in z\ (P'(x, v))$. In this expression, $P'(x, v)$ may be arbitrary.

Proof of Theorem 5.1 is omitted due to lack of space. It is an open question whether grouping is always necessary in case predicate P cannot be rewritten into one of the two forms given above.

Generally, a nested query may be processed by applying nestjoin operators, possibly followed by (nested) function applications (e.g. projections) and selections. However, sometimes nestjoin operators may be replaced by flat join operators. For a two-block query, in case the predicate between the two blocks can be rewritten into a calculus expression of the first format, involving a non-negated existential quantifier, a semijoin operation will provide the correct result. If it is possible to rewrite the predicate into a calculus expression involving a negated existential quantifier, then the flat join operation needed is the antijoin operation. The join predicate is $P'(x, G(x, y)) \wedge Q(x, y)$. (Remember that the semi-and antijoin operations are defined as follows. Let X and Y be tables (sets of tuples having possibly complex attributes), and let P be a predicate, then the semijoin operation $X \ltimes_{x,y:P} Y$ is defined as $\{x \mid x \in X \wedge \exists y \in Y\ (P(x, y))\}$ and the antijoin operation $X \rhd_{x,y:P} Y$ as $\{x \mid x \in X \wedge \nexists y \in Y\ (P(x, y))\}$.)

Examples of predicates that may or may not be rewritten into the desired format are listed in Table 5.1. Predicates above the separation line are predicates that may occur in SQL (being a subset of TM); predicates below the separation line are specific TM predicates involving set-valued attributes.

5.8 Query processing example

In this section, we illustrate our ideas by means of an example concerning an acyclic query with nesting in the WHERE clause in which correlation predicates are all neighbour pred-

$P(x,z)$	$P'(x,v)$
$z = \emptyset$	$\nexists v \in z\ (true)$
$z \neq \emptyset$	$\exists v \in z\ (true)$
$count(z) = 0$	$\nexists v \in z\ (true)$
$x.a = count(z)$	
$x.a \in z$	$\exists v \in z\ (v = x.a)$
$x.a \notin z$	$\nexists v \in z\ (v = x.a)$
$x.a \subset z$	
$x.a \supset z$	
$x.a \subseteq z$	
$x.a \supseteq z$	$\nexists v \in z\ (v \notin x.a)$
$x.a \not\supseteq z$	$\exists v \in z\ (v \notin x.a)$
$x.a = z$	
$x.a \cap z = \emptyset$	$\nexists v \in z\ (v \in x.a)$
$x.a \cap z \neq \emptyset$	$\exists v \in z\ (v \in x.a)$
$x.a \ni z$	
$\forall w \in x.a\ (w \subseteq z)$	
$\forall w \in x.a\ (w \supseteq z)$	$\nexists v \in z\ (\exists w \in x.a : (v \notin w))$

Table 5.1: Rewriting TM predicates

icates (having free variables declared in the immediately surrounding block).

In a preprocessing phase, predicates between query blocks are rewritten into calculus expressions if possible. The purpose of this rewrite phase is to determine whether nestjoin operations may be replaced by flat join operations (semi- or antijoin), as indicated in the previous section.

Nested queries are processed by performing a number of join operations and executing a number of function applications (for example projections) and selections on the join results. If predicates between query blocks require grouping, a nestjoin operator is applied; if predicates do not need grouping a flat join operation is executed. Replacement of a nestjoin operator by a semijoin or antijoin operator is advantageous because the semi- and antijoin can be implemented more efficiently than the nest- (or regular) join operator. Note that, like the semijoin, the antijoin operation may be implemented as a modification of the regular join. Consider the following query:

SELECT x
FROM $X\ x$
WHERE $x.a \subseteq$ SELECT $y.a$ (P_1)
FROM $Y\ y$
WHERE $x.b = y.b \wedge$
$y.c \subseteq$ SELECT $z.c$ (P_2)
FROM $Z\ z$
WHERE $y.d = z.d$

Predicates P_1 and P_2 between query blocks do require grouping (see Table 5.1), so we may have the following execution strategy:

1. A nestjoin with operands Y and Z on join predicate $y.d = z.d$. Each element of Z satisfying the join predicate is projected on the c attribute. The result of this step is the set:

   $$T_1 = \{y \mathbin{++} \langle zs = \{z.c \mid z \in Z \wedge y.d = z.d\}\rangle \mid y \in Y\}$$

   Note that zs is an arbitrary label.

2. The result of step 1 is restricted such that $y.c \subseteq y.zs$:

   $$T_2 = \{y \mid y \in T_1 \wedge y.c \subseteq y.zs\}$$

3. The result of step 2 is nest-joined with X on join predicate $x.b = y.b$ and projected on attribute a. (A projection on attributes a and b may proceed the nestjoin operation.) We now have:

   $$T_3 = \{x \mathbin{++} \langle ys = \{y.a \mid y \in T_2 \wedge x.b = y.b\} \mid x \in X\}$$

   Again, label ys is arbitrary.

4. Finally, the result of step 3 is restricted such that $x.a \subseteq x.ys$ and projected on the attributes of X (attributes a and b):

   $$T_4 = \{\langle a = x.a, b = x.b\rangle \mid x \in T_3 \wedge x.a \subseteq x.ys\}$$

Now assume that the operators $\subseteq$ in predicates P_1 and P_2 are changed in $\in$ and $\notin$, respectively, then the nestjoin operation in (1) may be replaced by an antijoin operation, and the nestjoin in (3) may be replaced by a semijoin operation. The additional join predicates are $y.c = z.c$ and $x.a = y.a$, respectively.

5.9 Conclusions and future work

As in relational systems supporting SQL, in complex object models supporting an SQL-like query language optimization of nested queries is an important issue. A naive way to handle nested queries is by nested-loop processing. However, it is better to transform nested queries into flat, or join queries, because join queries can be implemented in many different ways. In a complex object model, it is not always possible to flatten nested queries. In this paper, we have described a class of nested SFW-expressions that can be flattened without problem. For those nested queries that cannot be transformed into join queries we have introduced the nestjoin operator, allowing correct and efficient processing of general nested queries.

Future work concerns a number of issues. We need a formal algorithm to translate general SFW-query blocks of TM into the algebra, also taking into account nesting in the SELECT clause, multiple subqueries, and multiple nesting levels (including cyclic queries). Logical optimization (rewriting algebraic expressions) may follow the translation process. Therefore, the algebraic properties of the nestjoin operator have to be further investigated, by analogy with for example the work of [RoGa90] concerning the outerjoin operator.

Acknowledgments

We thank our colleagues Rolf de By and Paul Grefen for comments on earlier drafts of this paper.

References

[Bal et al. 93] Bal, R. et al., The TM Manual version 2.0, University of Twente, Enschede, 1993.

[BaBZ93] Balsters, H., R.A. de By, and R. Zicari, "Typed Sets as a Basis for Object-Oriented Database Schemas," *Proceedings ECOOP*, Kaiserslautern, 1993.

[BaVr91] Balsters, H. and C.C. de Vreeze, "A Semantics of Object-Oriented Sets", *Proceedings 3rd International Workshop on Database Programming Languages"* , Nafplion, Greece, 1991.

[Daya87B] Dayal, U., "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers," *Proceedings VLDB*, Brighton, 1987.

[GaWo87] Ganski, R.A., Wong, A.K.T., "Optimization of Nested SQL Queries Revisited," *Proceedings SIGMOD*, 1987.

[Kies84] Kiesling, W., "SQL-like and QUEL-like Correlation Queries with Aggregates Revisited," Memorandum UCB/ERL 84/75, Berkeley, 1984.

[Kim82] Kim, W., "On Optimizing an SQL-like Nested Query," *ACM TODS*, 7(3), September 1982, pp. 443–469.

[Mura89] Muralikrishna, M., "Optimization and Dataflow Algorithms for Nested Tree Queries," *Proceedings VLDB*, Amsterdam, 1989.

[Mura92] Muralikrishna, M., "Improved Unnesting Algorithms for Join Aggregate SQL Queries," *Proceedings VLDB*, Vancouver, 1992.

[PiAn86] Pistor, P., and Andersen, F., "Designing a Generalized NF^2 Model with an SQL-Type Language Interface," *Proceedings VLDB*, Kyoto, August 1986.

[RoGa90] Rosenthal, A. and C. Galindo-Legaria, "Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins," *Proceedings ACM SIGMOD*, Atlantic City, 1990.

[ScSc86] Schek, H.-J., and M.H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems* , 11(2), 1986, pp. 137–147.

[Scho86] Scholl, M.H., "Theoretical Foundations of Algebraic Optimization Utilizing Unnormalized Relations," *Proceedings First International Conference on Database Theory*, LNCS 243, Springer-Verlag, 1986.

Chapter 6

From Nested-Loop to Join Queries in OODB

This chapter contains a reprint from:

Steenhagen, H. J., P. M. G. Apers, H. M. Blanken, and R. A. de By, “From Nested-Loop to Join Queries in OODB,” *Proceedings VLDB,* Santiago de Chile, September 1994.

From Nested-Loop to Join Queries in OODB

Hennie J. Steenhagen Peter M. G. Apers Henk M. Blanken
Rolf A. de By

Department of Computer Science, University of Twente
PO Box 217, 7500 AE Enschede, the Netherlands
{hennie,apers,blanken,deby}@cs.utwente.nl

Abstract

Most declarative SQL-like query languages for object-oriented database systems (OSQL) are orthogonal languages allowing for arbitrary nesting of expressions in the **select-**, **from-**, and **where-**clause. Expressions in the **from-**clause may be base tables as well as set-valued attributes. In this paper, we propose a general strategy for the optimization of nested OSQL queries. As in the relational model, the translation/optimization goal is to move from tuple- to set-oriented query processing. Therefore, OSQL is translated into the algebraic language ADL, and by means of algebraic rewriting nested queries are transformed into join queries as far as possible. Three optimization options are described, and a strategy to assign priorities to options is proposed.

6.1 Introduction

To support technical applications like CAD/CAM, GIS etc, relational technology has its shortcomings. In these areas, the popularity of object-oriented technology is growing. First, from the field of programming languages, *persistent* programming languages like GemStone [Bretl et al. 89] came, later followed by object-oriented *database systems* such as O_2 [BaDK92] and HP OpenODB [Lyng91]. The historical background is still visible in the sense that too little attention has been paid to *ad hoc* query facilities and database design tools. A number of proposals for declarative query languages for extended NF^2

Proceedings of the 20th VLDB Conference Santiago, Chile, 1994

and object-oriented data models has appeared, e.g. [PiAn86, BaCD92, Catt93]. We will use the term OSQL for the various proposals of SQL-like languages for OODB. In this paper, we concentrate on efficient support for ad hoc queries formulated in a declarative query language.

An OSQL query facility is inherently more complex than one for SQL, because nesting is allowed in all clauses, i.e. in the **select**-, **from**-, and **where**-clause. Expressions in the **from**-clause, the query block operands, may be base tables as well as set-valued attributes. Also, the predicates that are used in the **where**-clause are more complex, because comparisons between set-valued attributes, or set-valued attributes and base table expressions are allowed. An example is the test whether two parts have an overlap in their sets of subparts.

Due to the complexity of OSQL, the dominant strategy to handle nesting is to execute it by means of nested-loop processing, leaving no room for optimization. We distinguish between two types of nesting: nesting required to iterate over base tables and nesting required to iterate over set-valued attributes. Ideally, nesting over base tables should be translated to some kind of join so that a choice can be made between various efficient join implementations. Of course, this problem already occurred in SQL [Kim82]. In general, nested SQL queries can be translated to a join in the relational algebra to get a better execution than the nested-loop execution. However, not all SQL queries can be translated to a join due to loss of dangling tuples in the join, a phenomenon known as the COUNT-bug. In [StAB94] we have shown that the COUNT-bug is only a special case of a more general problem occurring in nested OSQL queries.

The main focus of our research is to translate OSQL queries to an extended relational algebra for complex objects, called ADL, to allow for an efficient execution. In this paper, we deal with the problem of trying to translate nested OSQL queries to join queries in ADL, taking advantage of efficient implementations of join operators. We present a general approach for handling nesting in the **where**-clause, the **from**-, as well as in the **select**-clause; the discussion concentrates on nesting in the **where**-clause, though.

In the (extended) NF^2, as well as in the object-oriented literature, little work has been reported concerning efficient translation of SQL-like query languages into algebraic languages. For the NF^2 model, a translation of a calculus into an NF^2 algebra has been presented in [RoKS88], however, little attention has been paid to efficiency. Work has been done on the implementation of the extended NF^2 query language HDBL of the AIM project [SLPW89, SüLi90]; to our knowledge HDBL has not been translated into an algebra. In [ClMo93], a proposal for the optimization of nested O_2SQL queries has been made. O_2SQL is translated into an extension of the GOM algebra [KeMo93], and examples of optimization of nested algebra queries are given.

The organization of this paper is as follows. In Section 6.2, nesting in the various clauses of the select statement is introduced by means of examples, together with an example schema of an OODB. In Section 6.3, an algebra for complex objects called ADL is shown with a general translation of OSQL queries to ADL queries. Section 6.4 addresses the problem of optimizing ADL queries. Three alternatives are discussed. Two of them, rewriting into relational join queries and the introduction of new operators are considered in Section 6.5 and Section 6.6. The paper ends in discussing future work.

6.2 Example

In this section, we give a simple example schema of the supplier-part database in OSQL, together with some example queries.

Class Supplier **with extension** SUPPLIER
attributes
sname : **string**,
parts_supplied : { Part }
end Supplier

Class Part **with extension** PART
attributes
pname : **string**,
price : **int**,
colour : **string**
end Part

Class Delivery **with extension** DELIVERY
attributes
supplier : Supplier,
supply : { ⟨part : Part, quantity : **int**⟩ },
date : **date**
end Delivery

The schema only shows the structural properties of the entities stored in the database; method and constraint definitions have been left out. Brackets ⟨ ⟩ and { } denote the tuple and set type constructor, respectively. Analogous to relational convention, we call the class extensions base tables.

Below we give example queries for nesting in the various clauses. With regard to the **where**-clause, one example is given for nesting over a base table, and another for nesting over a set-valued attribute.

Example 6.1 Nesting in the **select**-clause is used to produce set-valued attributes in a complex object.

Select the names of the suppliers together with the names of the red parts supplied:

select ⟨sname = s.sname,
pnames = **select** p.pname
from p **in** s.parts_supplied
where p.colour = "red"⟩
from s **in** SUPPLIER

Example 6.2 Nesting in the **from**-clause denotes query composition, that for example may occur as the result of expanding views or named intermediate tables.

Select all deliveries that concern supplier s_1 with date January 1, 1994:

```
select d
from d in (select e
           from e in DELIVERY
           where e.supplier.sname = "s1")
where d.date = 940101
```

Nesting in the **from**-clause is a type of nesting that does not pose problems with respect to translation/optimization; it can be removed easily.

Example 6.3 Nesting in the **where**-clause is used for restrictions.

1. Select the names of the suppliers supplying all parts supplied by supplier s_1:

```
select s.sname
from s in SUPPLIER
where s.parts_supplied ⊇ select t.parts_supplied
                         from t in SUPPLIER
                         where t.sname = "s1"
```

2. Select all deliveries that include red parts.

```
select d
from d in DELIVERY
where exists x in (select s
                   from s in d.supply
                   where s.part.colour = "red")
```

In the first query, the operand of the inner **sfw**-block is the base table SUPPLIER: in the second the operand is the set-valued attribute supply. In the first, we have a set comparison between blocks, in the second a quantifier expression.

One important difference between SQL and OSQL is that OSQL is an *orthogonal* language. The expressions in the **from**- and **select**-clause of OSQL may be arbitrary, also containing other **select-from-where** (**sfw**) expressions (subqueries), provided they are correctly typed. Predicates may also be built up from arbitrary expressions including quantifiers **forall** and **exists** and set comparison operators. The focus of this paper is a general strategy for dealing with nested OSQL queries with nesting in the **select**- or **where**-clause. (Nesting in the **from**-clause is handled easily.) The discussion in subsequent sections will be centered around nested queries with nesting in the **where**-clause, however, techniques presented apply to nested queries with nesting in the **select**-clause as well.

Following the relational line of work, the goal in translation and optimization of OSQL is to move from tuple- to set-oriented query processing. Our approach, as in [ClMo93], is to translate nested OSQL queries into nested algebraic expressions, and then to try to rewrite nested algebraic expressions into join expressions. In the following section, we briefly present the algebraic language ADL.

6.3 The complex object algebra ADL

The language ADL is a typed algebra for complex objects in the style of the NF^2 algebra of [ScSc86], allowing for nesting of expressions. Among the constructors supported are

the tuple ($\langle\,\rangle$) and set ($\{\,\}$) type constructor; the basic type **oid** is used to represent object identity.

Roughly, the algebraic operators of the language ADL are the standard set (comparison) operators and multiple union (flatten), extended Cartesian product (in which operand tuples are concatenated) and division, the map operator α, selection σ, projection π, the renaming operator ρ, and the restructuring operators nest (ν) and unnest (μ). The map operator, well-known from functional languages and appearing under many different names in the literature, is used to apply a function to every element of a set. The function applied may be arbitrarily complex, so that the effect of a map operation may vary from a simple projection to the production of complex results. Furthermore, a number of join operators is supported: the regular join $\bowtie$, the semijoin $\ltimes$, and the antijoin $\rhd$. The semijoin (a regular join followed by the projection on the left-hand join operand attributes) is a join operator that is useful in processing so-called tree queries [Kamb85]. The antijoin is defined as a semijoin followed by a set difference of the left-hand join operand and the semijoin result. The antijoin operator is less well-known than the semijoin operator; it can be employed to efficiently process tree queries involving universal quantification. In selections and joins arbitrarily complex predicates can be used, including predicates containing quantifiers. Of course aggregate functions are part of the language too. Below, we give the semantics of some of the ADL operators used in this paper. For presentation purposes, a simplified notation is used; it is assumed no attribute naming conflicts occur. The operator $\circ$ is used to denote tuple concatenation. The schema function Sch, when applied to a table expression, delivers the top level attribute names.

1. (Flatten) $\bigcup(e) = \{x \mid x \in X \wedge X \in e\}$
2. (Tuple subscription) $e[a_1, \ldots, a_n] = \langle a_1 = e.a_1, \ldots, a_n = e.a_n \rangle$
3. (Tuple "update") Let $Sch(e) = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, then:
   e **except** $(a_1 = e_1, \ldots, a_n = e_n, c_1 = e'_1, \ldots, c_n = e'_k) =$
   $\langle a_1 = e_1, \ldots, a_n = e_n, b_1 = e.b_1, \ldots, b_m = e.b_m, c_1 = e'_1, \ldots, c_n = e'_k \rangle$
   The except operator may update existing tuple fields ($a_i = e_i$), leaving the remaining as they are ($b_i = e.b_i$), and may also extend the tuple with some new fields ($c_i = e'_i$).
4. (Map, or function application) $\alpha[x : f(x)](e) = \{f(x) \mid x \in e\}$
5. (Selection) $\sigma[x : p(x)](e) = \{x \in e \mid p(x)\}$
6. (Projection) $\pi_{a_1,\ldots,a_n}(e) = \{x[a_1, \ldots, a_n] \mid x \in e\}$
7. (Unnest) Let $Sch(e) = \{a, b_1, \ldots, b_m\}$, then:
   $\mu_a(e) = \{x' \circ x[b_1, \ldots, b_m] \mid x \in e \wedge x' \in x.a\}$
8. (Nest)
   Let $Sch(e) = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, let $A = a_1, \ldots, a_n$, and let $B = b_1, \ldots, b_m$, then:
   $\nu_{A;a}(e) = \{x[B] \circ \langle a = X \rangle \mid x \in e \ \wedge X = \{x'[A] \mid x' \in e \wedge x'[B] = x[B]\}\}$
9. (Cartesian product) $e_1 \times e_2 = \{x_1 \circ x_2 \mid x_1 \in e_1 \wedge x_2 \in e_2\}$
10. (Regular join) $e_1 \underset{x_1,x_2:p(x_1,x_2)}{\bowtie} e_2 = \{x_1 \circ x_2 \mid x_1 \in e_1 \wedge x_2 \in e_2 \wedge p(x_1, x_2)\}$
11. (Semijoin) $e_1 \underset{x_1,x_2:p(x_1,x_2)}{\ltimes} e_2 = \{x_1 \mid x_1 \in e_1 \wedge \exists x_2 \in e_2 \bullet p(x_1, x_2)\}$

12. (Antijoin) $e_1 \underset{x_1,x_2:p(x_1,x_2)}{\triangleright} e_2 = \{x_1 \mid x_1 \in e_1 \wedge \nexists x_2 \in e_2 \bullet p(x_1,x_2)\}$

ADL operators that allow for nesting, the so-called *iterators*, are the map, select, and join operators, and also quantifiers. Iterators are operators having functions (lambda expressions $\lambda x.e$, denoted as $x : e$) as parameters; within the function body other operators may occur. Note that the parameter of join operators (the join predicate) is a function having two arguments; the function (denoted as $x_1, x_2 : p$) is written as a subscript to the join operator symbol. In the evaluation of joins, variables x_1 and x_2 are iterated over operands e_1 and e_2 respectively, and tuples are included in the result depending on the value of $p(x_1, x_2)$.

Mapping OSQL to the algebra involves a mapping of types and a mapping of expressions. The mapping of *types* is carried out in the phase of logical database design. Here, we assume that each class extension is mapped to a table of (possibly complex) objects; a field of type **oid** is added to represent object identity, and class references are implemented by pointers, also of type **oid**. We also assume that class hierarchies are mapped to ADL types in some way or the other (the algebra does not support inheritance).

With respect to translation and optimization of *expressions*, we take the following approach. Translation of OSQL queries into the algebra is done in a simple, almost one-to-one way. The algebra supports nesting of expressions, representing tuple-oriented, or nested-loop query processing, as well as a number of purely algebraic set-oriented operators. In the translation phase, nested OSQL queries are translated into nested algebraic expressions. Following translation, in the phase of logical optimization, nested expressions are rewritten into set operations.

The OSQL construct that does not have an immediate algebraic equivalent is the **sfw**-query block. In the translation phase, an **sfw**-query block is mapped to an algebraic expression consisting of a selection followed by a map:

$$\mathbf{select}\ e_1\ \mathbf{from}\ x\ \mathbf{in}\ e_2\ \mathbf{where}\ e_3 \equiv \alpha[x : e_1](\sigma[x : e_3](e_2))$$

where σ computes the selection e_3 and α the "projection" e_1. In the select operation, variable x is iterated over operand e_2 and the operand is restricted according to the values of the **where**-clause predicate e_3; in the map operation α, variable x is iterated over the resulting operand subset, and for each of the tuples in this set the **select**-clause expression e_1 is evaluated.

To conclude this section, we formulate the goal in optimization of nested ADL queries. Operands of operators nested within parameter expressions of iterators may be either set-valued attributes or base table expressions. In this paper, the *goal* is to transform nested expressions, in which iterators having base tables as operands occur nested within parameter expressions of other iterators, into join expressions in which base tables occur *only at top level*. Of course the goal of unnesting applies to correlated subqueries[1] only; uncorrelated subqueries simply are constants, and treated as such. Assuming set-valued attributes are stored clustered, the unnesting of expressions with nested iterators having set-valued attributes as operands is not desirable. For example, $\sigma[x : \exists y \in Y \bullet p](X)$ with Y a base

[1] Correlated subqueries are iterator expressions that use variables from iterators in which they are nested.

table, is transformed into the semijoin operation $X \ltimes_{x,y:p} Y$, but $\sigma[x : \exists y \in x.c \bullet p](X)$ with c a set-valued attribute stored with the X-tuples is left as it is. In short, the goal in translation/optimization is to remove base tables from the parameter expressions of iterators, moving from tuple- to set-oriented query processing.

6.4 Optimization of nested algebra queries

In this section we present a general approach to optimize nested ADL queries. Three optimization options are given, together with example queries. The section ends with a strategy to give a priority to the options.

The example queries given below concern the supplier-part database of which the OSQL schema was given in Section 6.2. In ADL, the types of SUPPLIER and PART are as follows:

SUPPLIER : $\{\langle sid : \mathbf{oid},$
$sname : \mathbf{string},$
$parts : \{\langle pid : \mathbf{oid}\rangle\}\rangle\}$
PART : $\{\langle pid : \mathbf{oid},$
$pname : \mathbf{string},$
$price : \mathbf{int},$
$colour : \mathbf{string}\rangle\}$

We distinguish three ways of optimizing nested ADL queries: (1) the unnesting of attributes by using the unnest operator, (2) the unnesting of nested expressions by transforming them into relational join queries, and (3) using new operators that (analogous to for example the relational join) are defined especially to enhance performance. Below, we discuss the options in more detail.

Unnesting of attributes

If nesting is caused by iteration over a set-valued attribute it is possible to unnest this attribute. Depending on whether the result is nested or not, the nest operator has to be applied. The unnesting of attributes has some disadvantages. First, nest and unnest are each others inverse only for PNF relations (nested relations of which the atomic attributes recursively form a key) that have no empty set-valued attributes [RoKS88]. Second, first unnesting and later nesting again will be expensive due to duplication of attribute values and overhead caused by restructuring. Therefore, we only use this option if the final nesting is not required, and empty set-valued attributes cause no problem. Consider the query:

Example 6.4 Select the identifiers of suppliers supplying non-existing parts (violating referential integrity).

$$\pi_{sid}(\sigma[s : \exists z \in s.parts \bullet \nexists p \in \text{PART} \bullet z = p[pid]](\text{SUPPLIER}))$$

The set-valued attribute $parts$ is not needed in the result, so the above query may be rewritten into the antijoin query:

$$\pi_{sid}(\mu_{parts}(\text{SUPPLIER}) \underset{s,p:s.pid=p.pid}{\triangleright} \text{PART})$$

Note that, because z is existentially quantified, the loss of tuples with empty set-valued attribute $parts$ causes no problem (existential quantification over the empty set delivers $false$).

Transformation into join queries

In some cases two or more consecutive levels of nesting can be replaced by a join, antijoin, or semijoin operator, reducing the number of levels of nesting. In the ideal case all nesting has disappeared. Query 6.5 below shows such an example.

Example 6.5 Select the suppliers supplying red parts.

$$\sigma[s : \exists z \in s.parts \bullet \exists p \in \text{PART} \bullet z = p[pid] \wedge p.colour = \text{"}red\text{"}](\text{SUPPLIER})$$

This query can be rewritten into the semijoin query:

$$\text{SUPPLIER} \underset{s,p:p[pid] \in s.parts}{\ltimes} \sigma[p : p.colour = \text{"}red\text{"}](\text{PART})$$

In Section 6.5 this option is discussed in detail.

Using special operators

The relational join is not really necessary for the expressive power of the relational algebra; it was introduced to allow for various efficient implementations. The same can of course be done in an algebra for complex objects. Quite often we encounter that an efficient execution of a query is prohibited if we stick to generally-accepted operators. Therefore, we expect that introducing new operators is really necessary to obtain an efficient implementation. In Section 6.6, some new operators are discussed. The following query cannot be rewritten into a relational join query:

Example 6.6 Select suppliers names together with the parts supplied.

$$\alpha[s : \langle sname = s.sname, parts\_suppl = \sigma[p : p[pid] \in s.parts](\text{PART})\rangle](\text{SUPPLIER})$$

However, using the so-called nestjoin operator Δ (see Section 6.6), the nested query can be rewritten into an efficient set operation:

$$\pi_{sname,parts\_suppl}(\text{SUPPLIER} \underset{s,p:p[pid] \in s.parts;parts\_suppl}{\Delta} \text{PART})$$

Note that each of the options above can be applied to the top level expression as well as to subexpressions thereof. Given these options for optimization of nested ADL queries, the rewrite strategy is as follows:

1. Try to rewrite to the various relational join operators (join, antijoin, or semijoin).

2. If the above is not possible, try to flatten set-valued attributes; if the nesting phase can be skipped, this may be a strategy worthwhile considering.

3. If the above is not possible, try to rewrite to one of the newly defined operators, because they were introduced to get a better performance compared to nested-loop processing.

4. If none of the above works, leave the query as it is, which means that it is executed by means of nested loops.

The options "rewriting into relational join queries" and "using special operators" are discussed in more detail in the following two sections.

6.5 Rewriting into flat relational algebra

In the previous section, we have discussed some options for processing nested ADL queries. In this section, we investigate one of them: the rewriting of nested expressions into join expressions without unnesting set-valued attributes. The discussion concentrates on rewriting nested select (σ) expressions, the algebraic equivalent of nested OSQL queries with nesting in the **where**-clause. However, rewriting nested expressions into join expressions is a strategy that can be applied to nesting in the map operator as well; the section is concluded by briefly discussing an example of this type of nesting.

6.5.1 General query format

In this section, we discuss transformation of nested OSQL queries with nesting in the **where**-clause *in the presence of set-valued attributes*. Nesting in the **where**-clause is an important (and only) type of nesting allowed in the flat relational model; in complex object models it is considered equally important. The goal here, as in [Kim82], is to rewrite nested queries into *join queries* without unnesting set-valued attributes, so that instead of performing a naive nested-loop execution, the optimizer may choose from a number of different join processing strategies.

The general format of a two-block OSQL query with nesting in the **where**-clause is the following:

$$
\begin{array}{l}
\textbf{select}\ F(x) \\
\textbf{from}\ x\ \textbf{in}\ X \\
\textbf{where}\ P(x, Y') \\
\qquad \textbf{with}\ Y' = \textbf{select}\ G(x, y) \\
\qquad\qquad\qquad\ \ \textbf{from}\ y\ \textbf{in}\ Y \\
\qquad\qquad\qquad\ \ \textbf{where}\ Q(x, y)
\end{array}
$$

(The **with**-construct, enabling local definitions, is used here for reasons of convenience.) In Section 6.3, we have given the equivalence rule for translating the **sfw**-block into the algebra:

$$\mathbf{select}\ e_1\ \mathbf{from}\ x\ \mathbf{in}\ e_2\ \mathbf{where}\ e_3 \equiv \alpha[x : e_1](\sigma[x : e_3](e_2))$$

so the algebraic equivalent of an **sfw**-expression is a selection σ followed by a general function application α to compute the "projection". The map operator α is needed to compute arbitrarily structured results, as opposed to standard projections in the flat relational model.

The general format of the two-block **sfw**-expression with nesting in the **where**-clause as shown above is:

$$\alpha[x : F(x)](\sigma[x : P(x, Y')](X))\ \mathbf{with}\ Y' = \alpha[y : G(x, y)](\sigma[y : Q(x, y)](Y))$$

For simplicity, the functions F and G are assumed to be identity, so we have the format:

$$\sigma[x : P(x, Y')](X)\ \mathbf{with}\ Y' = \sigma[y : Q(x, y)](Y)$$

The query above is a nested query involving nested iteration over a base table: the outer selection predicate contains a subquery, which is a selection on base table Y. We want to transform this nested query into a join query, i.e. a query having no subqueries with base table operands.

6.5.2 Set comparison operations

To illustrate our ideas, at first we concentrate on two-block nested expressions with set comparison operations between query blocks. We investigate two unnesting techniques: unnesting by rewriting into quantifier expressions, and unnesting by grouping, a technique well-known from the relational model [Kim82].

Below, we show that, in some cases, queries having set comparison operators between query blocks can be transformed into join queries by rewriting the set comparison operator into a quantifier expression. However, in other cases, rewriting into quantifiers has a negative effect on performance; other solutions have to be sought. The results are generalized to a rewrite heuristic for transforming two-block select queries with arbitrary quantifier expressions between blocks.

To the queries that cannot be unnested by rewriting into quantifier expressions, we apply the methods of [Kim82, GaWo87] for unnesting relational queries with aggregate functions between blocks. We show that the methods of [Kim82, GaWo87] are general techniques for transforming nested queries into join queries, however, to be of good use in complex models, they have to be adapted. We will do so by defining a new algebraic operator, the *nestjoin operator*, in Section 6.6.

We now continue by describing the query format of interest in this section. Assume that predicate P involves some set comparison operation relating attribute c of the outer operand X and the set Y', the subquery result. More formally, the query format is:

$$\sigma[x : x.c\ \theta\ Y'](X)\ \mathbf{with}\ Y' = \sigma[y : Q(x, y)](Y)$$

set comparison operation	quantifier expression
$x.c \in Y'$	$\exists y \in Y' \bullet y = x.c$
$x.c \subset Y'$	$x.c \subseteq Y' \wedge x.c \not\supseteq Y'$
	$(\forall z \in x.c \bullet \exists y \in Y' \bullet z = y) \wedge (\not\forall y \in Y' \bullet y \in x.c)$
$x.c \subseteq Y'$	$\forall z \in x.c \bullet \exists y \in Y' \bullet z = y$
$x.c = Y'$	$x.c \subseteq Y' \wedge x.c \supseteq Y'$
	$(\forall z \in x.c \bullet \exists y \in Y' \bullet z = y) \wedge (\forall y \in Y' \bullet y \in x.c)$
$x.c \supseteq Y'$	$\forall y \in Y' \bullet y \in x.c$
$x.c \supset Y'$	$x.c \supseteq Y' \wedge x.c \not\subseteq Y'$
	$(\forall y \in Y' \bullet y \in x.c) \wedge (\not\forall z \in x.c \bullet \exists y \in Y \bullet z = y)$
$x.c \ni Y'$	$\exists z \in x.c \bullet z = Y'$

Table 6.1: Rewriting set comparison operations

with $\theta \in \{\in, \subset, \subseteq, =, \supseteq, \supset, \ni\}$. Note that the type of attribute c varies with the comparison operator used; the type may be atomic ($\in$), or an arbitrary set type. If the operator used is $\ni$, c has a set-of-set type.

Unnesting by rewriting into quantifier expressions

In this section we show that translating set comparison operators into quantifier expressions offers possibilities to unnest nested queries. In [CeGo85], presenting a translation from SQL to the relational algebra, set comparison operators are dealt with by rewriting them into quantifier expressions in a preprocessing phase. Nested relational queries with quantifiers are easily translated into relational algebra operations. Existential quantification is mapped to a projection on a join (or product); universal quantification is handled by means of the division operator [Codd72].

For example, from the relational model we know that a set membership predicate can be translated into an existential subquery that is easily translated into a (semi)join operation. Let $q \equiv Q(x, y)$, then:

Rewriting example 6.1 Set membership

$$
\begin{aligned}
&\sigma[x : x.c \in \sigma[y : q](Y)](X) \\
&\quad \equiv \quad \sigma[x : \exists y \in \sigma[y : q](Y) \bullet y = x.c](X) \\
&\quad \equiv \quad \sigma[x : \exists y \in Y \bullet y = x.c \wedge q](X) \\
&\quad \equiv \quad X \underset{x, y : y = x.c \wedge q}{\ltimes} Y
\end{aligned}
$$

First the operator $\in$ is rewritten into an existential quantification. Next, the select operation is removed from the operand (the range expression) of the existential quantifier, providing the possibility to translate the existential subquery into a semijoin operation in the last rewrite step. In this last step the actual unnesting is performed; the preceding rewrite steps are necessary to transform the input expression into the format suitable for unnesting. The

equivalence rules for unnesting quantifier expressions nested within select operators are the following.

Rule 6.1 Unnesting quantifier expressions Let X and Y be table expressions, and let x not be free in Y, then:

1. $\sigma[x : \exists y \in Y \bullet p](X) \equiv X \underset{x,y:p}{\ltimes} Y$
2. $\sigma[x : \nexists y \in Y \bullet p](X) \equiv X \underset{x,y:p}{\triangleright} Y$

A nested query with existential quantification is translated into a semijoin operation; negated existential (i.e. universal) quantification is dealt with by means of the antijoin operator.

The same method, rewriting set comparison operators by means of quantifiers, can be applied in complex object models as well. Again, let $q \equiv Q(x, y)$, and consider the following example dealing with the set inclusion operator:

Rewriting example 6.2 Set inclusion

$$
\begin{aligned}
&\sigma[x : \sigma[y : q](Y) \subseteq x.c](X) \\
&\quad\equiv \quad \sigma[x : \forall y \in \sigma[y : q](Y) \bullet y \in x.c](Y)](X) \\
&\quad\equiv \quad \sigma[x : \nexists y \in \sigma[y : q](Y) \bullet y \notin x.c](X) \\
&\quad\equiv \quad \sigma[x : \nexists y \in Y \bullet q \wedge y \notin x.c](X) \\
&\quad\equiv \quad X \underset{x,y:q \wedge y \notin x.c}{\triangleright} Y
\end{aligned}
$$

In the example above, the same procedure as in Rewriting Example 6.1 is followed (rewriting into quantification, transformation of the range expression, and unnesting). In addition, the universal quantifier is transformed into a negated existential quantifier by pushing through negation to enable transformation into the antijoin operation.

All set comparison operators can be rewritten into quantifier expressions, as shown in Table 6.1. In the table, in all cases except for the last we have expanded operators until quantification(s) over Y' take(s) place. We see that expanding operators $\in$ and $\supseteq$ leads to a (negated) existential quantifier expression that is suited for unnesting by applying Rule 6.1; expansion of the other operators leads to a multiple subquery expression, that cannot be unnested that way. Note that negation of the set comparison operation does not influence the possibilities for unnesting. Negating the operator negates the quantifier expression; antijoins are used instead of semijoins and vice versa.

So far, we restricted our discussion to two-block nested queries with predicates of the form $x.c\ \theta\ Y'$, with θ a set comparison operator. In Table 6.2, we show some more examples of predicates that can be rewritten into (negated) existential quantification, the form suitable for transformation in relational join expressions. To determine exactly which types of predicates can be rewritten is a topic of future research.

$P(x, Y')$	quantifier expression
$Y' = \emptyset$	$\nexists y \in Y' \bullet true$
$count(Y') = 0$	$\nexists y \in Y' \bullet true$
$x.c \cap Y' = \emptyset$	$\nexists y \in Y' \bullet y \in x.c$
$\forall z \in x.c \bullet z \supseteq Y'$	$\nexists y \in Y' \bullet \exists z \in x.c \bullet y \notin z$

Table 6.2: Rewriting predicates

Rewrite heuristic

From the discussion above, an important rewrite heuristic for nested expressions with predicates consisting of arbitrary quantifier expressions can be derived. Difficulties with unnesting arise whenever subqueries with base tables as operands are nested within iterators with set-valued attributes as operands, and the order of nesting cannot be changed. Consider the last expression of Table 6.2. We have:

Rewriting example 6.3 Exchanging quantifiers

$$
\begin{aligned}
&\forall z \in x.c \bullet z \supseteq Y' \\
&\quad \equiv \quad \forall z \in x.c \bullet \forall y \in Y' \bullet y \in z \\
&\quad \equiv \quad \forall y \in Y' \bullet \forall z \in x.c \bullet y \in z \\
&\quad \equiv \quad \nexists y \in Y' \bullet \exists z \in x.c \bullet y \notin z
\end{aligned}
$$

By expanding the comparison operator and exchanging universal quantifiers, the predicate is put in a form suitable for unnesting according to Rule 6.1. The general rewrite heuristic is formulated as follows. Let P be a quantifier expression in Prenex Normal Form:

$$P \equiv \mathcal{Q}x_1 \in e_1 \bullet \mathcal{Q}x_2 \in e_2 \cdots \mathcal{Q}x_n \in e_n \bullet p$$

in which the range expressions e_i are either base tables or set-valued attributes. To enable unnesting of (sub)expressions, the goal is to move quantification over base tables to the left of the quantifier expression. This goal may be achieved by exchanging universal or existential quantifiers.

Unnesting by grouping

Another way to deal with set comparison operators is to use grouping. In the relational context, grouping is used to transform nested queries with aggregate functions between query blocks [Kim82, GaWo87]. As we will see, the method of [GaWo87] in fact represents a general way of treating nested queries that can be applied in complex object models as well.[2] However, in some cases the results achieved are not correct due to the loss of dangling tuples in the relational join operation.

[2]The method of [Kim82] can be applied only when the correlation predicate (or join predicate) is equality.

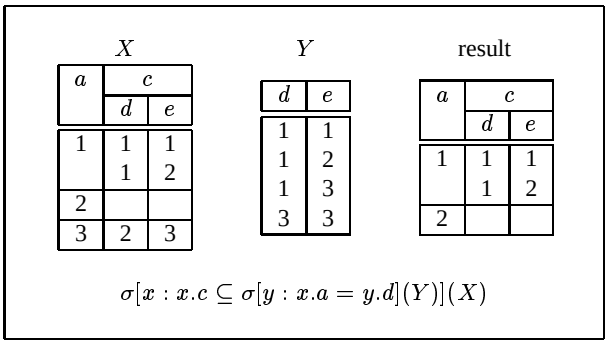


Figure 6.1: Nesting involving set-valued attribute

In [Kim82, GaWo87], methods for unnesting queries with aggregate functions between query blocks of the form $\sigma[x : P(x, agg(\sigma[y : Q(x, y)](Y)))](X)$ are presented. Below, we apply the method of [GaWo87] to nested queries with set comparators between blocks.

Consider the following nested query, an example of which is given in Figure 6.1.

$$\sigma[x : x.c \subseteq \sigma[y : x.a = y.d](Y)](X)$$

Applying the transformation technique of [GaWo87] to the expression above, we have, in our own formalism:

$$\pi_X(\sigma[x : x.c \subseteq x.ys](\nu_{Y;ys}(X \underset{x,y:x.a=y.d}{\bowtie} Y)))$$

The nested query is transformed into a flat join query consisting of (1) a join to evaluate the inner query block predicate, (2) a nest operation for grouping, (3) a selection for evaluating P, the predicate between blocks, and (4) a final projection. Example tables X and Y and the intermediate results of the join, nest, and project/select operation are shown in Figure 6.2.

We note that, as with relational queries involving the COUNT function, in the join query some kind of bug occurs, due to the loss of dangling tuples in the join; in analogy with the phrase "COUNT bug," we call this bug the "Complex Object bug." In the example, the tuple $\langle a = 2, c = \emptyset \rangle$ in X is not matched by any of the tuples $y \in Y$, so the subquery result is empty. In the join, this tuple is lost; in the nested query, the expression $\emptyset \subseteq \emptyset$ evaluates to $true$, so the tuple has to be included.

Now consider a variant of the query above, in which $\subseteq$ is changed into $\supseteq$:

$$\sigma[x : x.c \supseteq \sigma[y : x.a = y.d](Y)](X)$$

Here as well, applying the same unnesting technique yields a Complex Object bug. All tuples $x \in X$ for which it holds that the subquery Y' is equal to the empty set should be included into the result, but are lost in the join.

In Table 6.3, we have listed the set comparison operations under consideration, together with the value of the predicate for subqueries delivering empty sets (a question mark

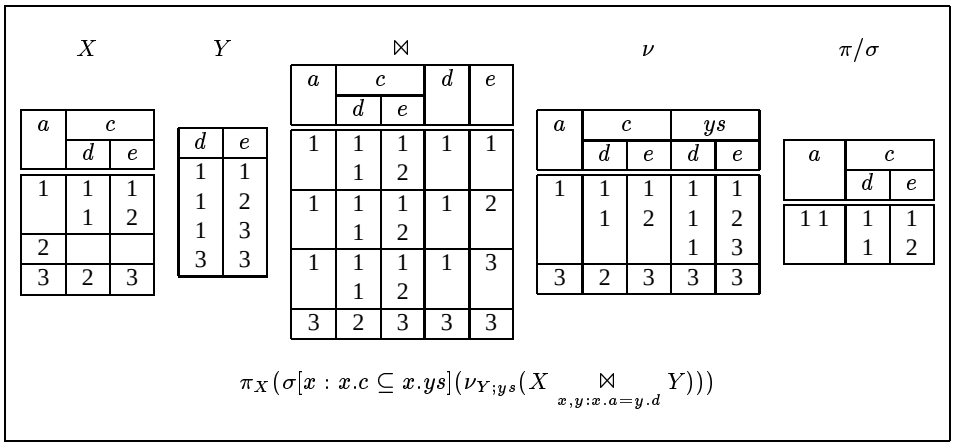


Figure 6.2: The complex object bug

meaning run-time dependence). Negated predicates are treated in the same way. We have the following result.

- The relational transformation technique of [GaWo87] for unnesting nested queries having aggregate functions between query blocks, using grouping, may be applied to nested complex object queries involving set comparison operators between query blocks as well. However, in some cases the loss of dangling outer operand tuples in the join causes incorrect results.

- The value of the expression $P(x, Y')$, with the empty set substituted for Y', determines whether or not dangling tuples should be included into the result. Whenever $P(x, \emptyset)$ can be reduced statically to $true/false$, all/none of the dangling tuples $x \in X$ must be included into the result; whenever this value is undetermined at compile time, it is run-time dependent whether or not dangling tuples $x \in X$ should be included (cf. the predicate $x.c = count(Y')$). In other words, the unnesting technique used here is guaranteed to deliver correct results only if $P(x, \emptyset)$ can be statically reduced to $false$.

If not for the occurrence of bugs, the techniques of [Kim82, GaWo87] can be applied to nested queries with arbitrary predicates between blocks. One way to solve the COUNT bug in the relational model is to employ the outerjoin operator [GaWo87]. In using the outerjoin, NULL values are used to represent the empty set. This method may be applied in a slightly adapted way in complex object models as well. Another way to solve the COUNT bug is to use a binary aggregation operator [ÖÖMa87, Naka90]. In Section 6.6, we discuss a new operator for unnesting nested queries that is based on the idea underlying binary aggregation, but separates predicate evaluation from join and grouping.

$P(x, Y')$	$P(x, \emptyset)$
$x.c \in Y'$	$false$
$x.c \subset Y'$	$false$
$x.c \subseteq Y'$	?
$x.c = Y'$	?
$x.c \supseteq Y'$	$true$
$x.c \supset Y'$	?
$x.c \ni Y'$	?

Table 6.3: Set comparison operators and bugs

6.5.3 Nesting in the map operator

To conclude, we give another example of the strategy of rewriting nested expressions into relational join expressions, but now concerning nesting in the map operator (i.e. in the **select**-clause). The following equivalence rule can be used to transform a nested map operation into a join query:

Rule 6.2 Nesting in the map operator

$$\bigcup(\alpha[x : \alpha[y : x \circ y](\sigma[y : p])(Y)](X) \equiv X \underset{x,y:p}{\bowtie} Y$$

The nested map operation on the left hand side creates a set of sets that is flattened immediately afterwards; the same result is achieved by the right hand join expression.

Briefly summarizing this section, we have seen that (1) rewriting predicates into quantifier expressions may enable the transformation of nested expressions involving set-valued attributes into relational join expressions, (2) unnesting by grouping is a transformation technique that is generally applicable, if not for the occurrence of bugs. In the next section, we show how to avoid the occurrence of bugs by using the nestjoin operator; the general transformation strategy then is to transform nested queries into nestjoin expressions, but to use relational join operators whenever possible.

6.6 New algebraic operators

In this section, we give three examples of new algebraic operators that are well-suited for efficient implementation of nested OSQL queries. Generally speaking, it is worthwhile to define new logical algebra operators whenever there can be found new access algorithms (or physical algebra operators [Grae93]) that are an improvement over nested-loop query processing. For example, the join can be implemented as an index nested-loop join, a sort-merge join, a hash join, etc. In this section, we give some examples of operators that might be of use for improving performance in OO query processing. The first operator to be discussed is the nestjoin operator, defined in [StAB94] for the processing of nested queries requiring grouping. The second operation to be discussed is the PNHL algorithm

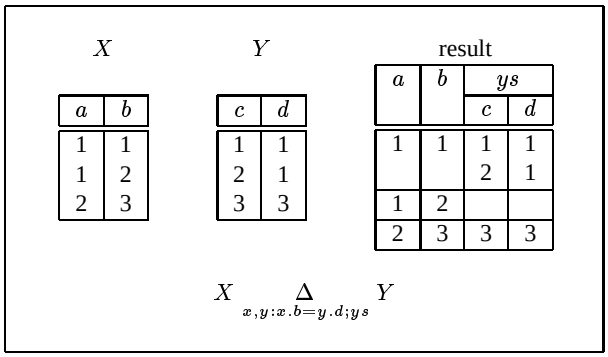


Figure 6.3: Nestjoin example

of [DeLa92], useful for materializing set-valued attributes, and the third is the materialize operator of [BlMG93].

6.6.1 The nestjoin operator—grouping during join

In the previous section we have shown that unnesting by using grouping is a transformation strategy generally applicable, if not for the occurrence of bugs due to the loss of dangling tuples in the join. In [StAB94], we have defined an operator that combines grouping and join without losing dangling left operand tuples: the *nestjoin* operator. The nestjoin operator is to be used for the unnesting of nested queries that cannot be rewritten into flat relational join operations.

The nestjoin operator, denoted by the symbol Δ, is a simple modification of the join operator. Instead of producing the concatenation of every pair of matching tuples, each left operand tuple is concatenated with the *set* of matching right operand tuples. To implement the nestjoin, common join implementation methods like the sort-merge join, or the hash join can be adapted. The definition of the nestjoin is as follows.

Definition 6.1 The nestjoin operator (simple) Let label a not occur in the schema of e_1, then:

$$e_1 \underset{x_1,x_2:p(x_1,x_2);a}{\Delta} e_2 = \{x_1 \circ \langle a = X \rangle \mid x_1 \in e_1 \wedge X = \{x_2 \mid x_2 \in e_2 \wedge p(x_1, x_2)\}\} \quad \square$$

Variables x_1 and x_2 are iterated over operands e_1 and e_2, respectively. Each left operand tuple $x_1 \in e_1$ is concatenated with the unary tuple $\langle a = X \rangle$, in which the set X contains those right hand operand tuples $x_2 \in e_2$ for which the predicate $p(x_1, x_2)$ holds. An example of the nestjoin operation is given in Figure 6.3, where relations X and Y are equijoined on the second attribute.

The nestjoin operator as defined above can be used for the transformation of two-block select expressions with arbitrary predicates between blocks. The simplified version of the two-block select query:

$$\sigma[x : P(x, Y')](X) \textbf{ with } Y' = \sigma[y : Q(x, y)](Y)$$

is transformed into the nestjoin expression:

$$\pi_X(\sigma[z : P'](X \underset{x,y:Q(x,y);ys}{\Delta} Y))$$

In the nestjoin operation, for each tuple $x \in X$ the set of tuples $y \in Y$ is restricted according to predicate Q. In the selection, the nestjoin result is restricted according to predicate P'. Predicate P has to be adapted by substituting $z[X]$ (nestjoin tuple z projected on its X attributes) and $z.ys$ (the subquery result as attribute of nestjoin tuples z) for x and Y', respectively, i.e. $P' \equiv P(x, Y')[z[X]/x, z.ys/Y']$. A projection on the attribute values of X completes the computation.

The nestjoin operation can be used to process queries with nesting in the **select**- or **where**-clause. Queries having subqueries in the **select**-clause often denote nested results, so processing by means of the nest join operation will be appropriate. The general format of a query with nesting in the **select**-clause is:

$$\begin{array}{l}\textbf{select } F(x, Y') \\ \quad \textbf{with } Y' = \textbf{select } G(x, y) \\ \qquad\qquad \textbf{from } y \textbf{ in } Y \\ \qquad\qquad \textbf{where } Q(x, y) \\ \textbf{from } x \textbf{ in } X \\ \textbf{where } P(x)\end{array}$$

Assume function G and predicate P are identity, then in the algebra we have:

$$\alpha[x : F(x, Y')](X) \textbf{ with } Y' = \sigma[y : Q(x, y)](Y)$$

which is equivalent to:

$$\alpha[z : F'](X \underset{x,y:Q(x,y);ys}{\Delta} Y)$$

in which function F is adapted by performing the necessary substitutions:

$$F' \equiv F(x, Y')[z[X]/x, z.ys/Y']$$

Above, we have given a simplified definition of the nestjoin operator. For the transformation of general nested queries with deeper nesting levels, the nestjoin needs as an extra parameter a function to be applied to the right hand operand tuples [StAB94].

6.6.2 Materializing set-valued attributes

Below, we describe two proposals for the materialisation of (set-valued) attributes, in complex object models an operation presumed to occur frequently. In the first proposal of [DeLa92], an algorithm was given without defining a corresponding logical algebra operator; in [BlMG93], both a logical and a corresponding physical algebra operation are described.

The PNHL algorithm

Below, we describe the algorithm of [DeLa92] for efficiently processing a nested expression in which a set-valued attribute is joined with a base table. This algorithm can be considered as a new physical algebra operation. Though the correspondence between logical and physical algebra operators usually is not one-to-one [Grae93], the question is whether it is useful to define new logical operators for algorithms such as that of [DeLa92]. The following query expresses a nested natural join ($\star$) operation:

$$\alpha[x : x \ \mathbf{except}\ (parts = x.parts \underset{z,y:z.pid=y.pid}{\star} \text{PART})](\text{SUPPLIER})$$

In [DeLa92], a hash-based algorithm called Partitioned Nested-Hashed-Loops (PNHL) algorithm for computing this type of join operation is described and performance measures are reported. The algorithm builds a hash table for those segments of operand PART that fit into main memory and then probes operand SUPPLIER against each segment of the hash table, thus building partial results. Partial results are merged in the second phase of the algorithm. Compared to the unnest-join-nest processing method, the algorithm achieves better performance. Comparing the PNHL algorithm with traditional hash join, we see that in the PNHL algorithm, only the flat table can be the build table (the inner operand PART in the example), whereas in relational hash join usually the smaller operand is chosen as build table.

The materialize operator

In object-oriented database systems the concepts of object identity and path expressions play an important role. Object identifiers can be implemented either as physical or as logical pointers. Implementing object-identifiers as physical pointers opens the way to new join implementation methods (pointer-based joins, [ShCa90]).

Also, object identifiers can be usefully employed to implement path expressions, i.e. the user-defined relationships or links between object classes. In [BlMG93], path expressions are represented by the operator materialize. Materialize is defined as a new logical algebra operator, with the purpose to explicitly indicate the use of inter-object references, i.e. to indicate where path expressions are used and where therefore algebraic transformations can be applied. The operator is implemented by an access algorithm called assembly, a generalization of the concept of a pointer-based join.

6.7 Conclusion and future work

As in relational systems supporting SQL, in OO data models supporting an SQL-like query language (OSQL), optimization of nested queries is an important issue. A naive way to handle nested queries is by nested-loop processing (tuple-oriented query processing), however, it is better to transform nested queries into join queries, because join queries can be implemented in many different ways (set-oriented query processing).

In this paper, we have presented a general approach to optimization of nested OSQL queries. In OSQL, nesting may occur in the **where**-, **from**-, and **select**-clause. An additional complication in complex object models is the support for iteration over set-valued attributes. The goal is to transform nested OSQL queries having correlated subqueries with base table expressions as operands into join queries in which base tables occur only at top level. First, we try to rewrite nested expressions into relational join operations. Second, we consider whether the unnesting of set-valued attributes is a possible (for theoretical reasons) and a worthwhile (for reasons of performance) optimization option. Third, if the previous steps do not give the result wanted, we use new operators especially defined to improve performance. Finally, if none of the previous steps work, we resort to nested-loop processing.

We have shown that transformation of nested OSQL queries dealing with set-valued attributes into relational join queries is not always possible. In many cases, the unnesting of nested OSQL queries requires some form of grouping in the unnested, or join query. Relational transformation techniques for nested queries requiring grouping (nested queries with aggregate functions between blocks) do not always give correct results; to improve matters we have defined a new operator called the nestjoin operator.

Future work concerns a number of issues. First, we need a precise characterization of nested queries requiring grouping or not. Second, for those queries that do require grouping, new implementation techniques have to be investigated. Third, new features characteristic of OO data models, like object identity and path expressions, provide new opportunities to improve performance. At the logical as well as the physical algebra level new operators may be defined and implemented. Finally, the ultimate goal of course is a general (syntax-driven) translation/optimization algorithm for arbitrary nested OSQL queries, including queries with multiple subqueries and multiple nesting levels.

References

[BaCD92] Bancilhon, F., S. Cluet, and C. Delobel, *A Query Language for O_2*, in: *Building an Object-Oriented Database System—The Story of O_2*, eds. F. Bancilhon, C. Delobel, and P. Kanellakis, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[BaDK92] Bancilhon, F., C. Delobel, and P. Kanellakis (eds.), *Building an Object-Oriented Database System—The Story of O_2*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[BlMG93] Blakeley, J.A., W.J. McKenna, and G. Graefe, "Experiences Building the Open OODB Optimizer," *Proceedings ACM SIGMOD*, 1993.

[Bretl et al. 89] Bretl, R. et al., "The GemStone Database Management System," in: *Object-Oriented Concepts, Databases, and Applications*, eds. W. Kim and F.H. Lochovsky, Addison-Wesley, Reading, MA, 1989.

[Catt93] R.G.G. Cattell, ed., *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[CeGo85] Ceri, S. and G. Gottlob, "Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries," *IEEE Transactions on Software Engineering*, 11(4), April 1985.

[ClMo93] Cluet, S. and G. Moerkotte, "Nested Queries in Object Bases," *Proceedings Fourth International Workshop on Database Programming languages*, New York, Sept. 1993.

[Codd72] Codd, E.F., "Relational Completeness of Data Base Sublanguages," In: *Data Base Systems*, ed. R. Rustin, Prentice Hall, 1972.

[DeLa92] Desphande, V. and P.-Å. Larson, "The Design and Implementation of a Parallel Join Algorithm for Nested Relations on Shared-Memory Multiprocessors," *Proceedings IEEE Conference on Data Engineering*, pp. 68-77, Tempe, Arizona, February 1992.

[GaWo87] Ganski, R.A. and A.K.T. Wong, "Optimization of Nested SQL Queries Revisited," *Proceedings ACM SIGMOD*, 1987.

[Grae93] Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2), pp. 73-170, June 1993.

[Kamb85] Kambayashi, Y., "Cyclic Query Processing," in: *Query Processing in Database Systems*, eds. W. Kim, D.S. Reiner, and D.S. Batory, Springer Verlag, pp. 62-78, 1985.

[KeMo93] Kemper, A. and G. Moerkotte, "Query Optimization in Object Bases: Exploiting Relational Techniques," in: *Query Processing for Advanced Database Systems*, eds. J.-C. Freytag, D. Maier, and G. Vossen, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Kim82] Kim, W., "On Optimizing an SQL-like Nested Query," *ACM TODS*, 7(3), pp. 443-469, September 1982.

[Lyng91] Lyngbaek, P., "From Relational Databases to Objects and Beyond," in: *Advances in Data Management*, eds. P. Sadanandan, T.M. Vijayaraman, McGraw-Hill Publishing Company Ltd, 1991.

[Naka90] Nakano, R. "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS*, 15(4), pp. 518-557, December 1990.

[ÖÖMa87] Özsoyoğlu, G., Z.M. Özsoyoğlu, and V. Matos, "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions," *ACM TODS*, 12(4), pp. 566-592, December 1987.

[PiAn86] Pistor, P. and F. Andersen, "Designing a Generalized NF^2 Model with an SQL-Type Language Interface," *Proceedings VLDB*, Kyoto, August 1986.

[RoKS88] Roth, M.A., H.F. Korth, and A. Silberschatz, "Extended Algebra and Calculus for Nested Relational Databases," *ACM TODS*, 13(4), pp. 389-417, December 1988.

[SLPW89] Saake, G., V. Linnemann, P. Pistor, and L. Wegner, "Sorting, Grouping and Duplicate Elimination in the Advanced Information Management Prototype," *Proceedings VLDB*, Amsterdam, 1989.

[ScSc86] Schek, H.-J. and M.H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems*, 11(2), pp. 137-147, 1986.

[ShCa90] Shekita, E.J. and M.J. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proceedings ACM SIGMOD*, pp. 300-311, Atlantic City, May 1990.

[StAB94] Steenhagen, H.J., P.G.M. Apers, and H.M. Blanken, "Optimization of Nested Queries in a Complex Object Model," *Proceedings EDBT*, Cambridge, March 1994.

[SüLi90] Südkamp, N. and V. Linnemann, "Elimination of Views and Redundant Variables in an SQL-like Database Language for extended NF^2 Structures," *Proceedings VLDB*, Brisbane, 1990.

Chapter 7

Translating OSQL Queries into Efficient Set Expressions

This chapter contains an article accepted for the EDBT conference in 1996 in Avignon, France.

Translating OSQL Queries into Efficient Set Expressions

Hennie J. Steenhagen Rolf A. de By Henk M. Blanken

Department of Computer Science, University of Twente
PO Box 217, 7500 AE Enschede, the Netherlands
{hennie,deby,blanken}@cs.utwente.nl

Abstract

Efficient query processing is one of the key promises of database technology. With the evolution of supported data models—from relational via nested relational to object-oriented—the need for such efficiency has not diminished, and the general problem has increased in complexity.

In this paper, we present a complete, heuristics-based and extensible algorithm for the translation of object-oriented query expressions in a variant of OSQL to an algebra extended with specialized join operators, designed for the task. We claim that the resulting algebraic expressions are cost-efficient.

Our approach builds on well-known optimization strategies for the relational model, but extends them to include relations and more arbitrary sets as values. We pay special attention to the most costly forms of OSQL queries, namely those with full subqueries in the SELECT- or WHERE-clause. The paper builds on earlier results [StAB94, SABB94].

7.1 Introduction

Currently, the ODMG group is working on the standards for object-oriented database management systems [Catt93]; the ODMG proposal includes a description of an object query language named OQL, which is an SQL-like language. How to implement such a language, i.e. the subject of query processing in object-oriented database systems, is an important research topic. This paper studies the efficiency of query processing in such systems.

We consider the first phase of query processing, i.e. the translation of an object-oriented SQL-like language (OSQL) into an algebra supporting complex objects. We believe that, as for the relational model, set-orientation is an appropriate query processing paradigm for object-oriented models also. Set operators allow to apply techniques such as sorting or hashing to improve performance. The goal is to achieve a translation that results in algebraic expressions that have good performance.

Important features of OSQL are object identity, inheritance, the presence of complex objects, and the possibility to define methods. In our opinion, OSQL can be considered

as an extension of SQL-like languages for extended nested relational models [PiAn86]. Common features are the presence of complex objects and the orthogonality of language design. In the implementation of OSQL, precisely these features are of major importance. The work presented here is meant to serve as the basis for the implementation of OSQL. Specific object-oriented features can be handled as an addition or an extension. For example, the presence of object identity allows to speed up join algorithms [ShCa90]. We remark that, in our framework [BaBZ91], methods are written using OSQL instead of some general purpose programming language. Hence, method calls in a query can be textually substituted by their OSQL definition, allowing for additional optimization.

The nested relational model has been studied in depth. To our knowledge however, a translation from OSQL into nested relational algebra, comparable to the algorithm of [CeGo85] for the translation of relational SQL queries, has never been published. A translation of nested calculus into nested algebra was made in [RoKS88], but that work is of a more theoretical nature, paying no attention to implementation aspects. The algebra used is a minimal extension of relational algebra, not suitable to be used for implementation purposes.

We build on previous work. In [StAB94], we showed that in complex object models it is impossible to transform arbitrary nested queries into flat join queries. As a solution, we introduced the nestjoin operator. In [SABB94], we described a general approach to handle nested queries; here, we are more concrete and present a translation algorithm. Related to our work is that of [ClMo93], in which the optimization of nested O_2SQL queries is discussed.

In Section 7.2, we briefly describe the language used. In Section 7.3, we discuss the problem of translating OSQL in more detail. The presentation of this paper is one of stepwise refinement. First, we show in Section 7.4 that a nested OSQL expression can be translated into an expression akin to the project-select-product form of the relational model. The result thus obtained is very inefficient, and must be optimized. We try to obtain better results by using better translation rules. Next, in Section 7.5, the main steps of the algorithm are described, and the basic set of rewrite rules and an initial rewrite strategy is given. It becomes clear that, in order to obtain an efficient result, (multi-variable) parameter expressions have to be split. As it turns out, the way expressions are split strongly determines the outcome of the transformation, given a fixed rewrite strategy. Heuristics are needed to guide the process of splitting expressions; these are presented in Section 7.6. A new, again fixed, rewrite strategy is presented. However, we are forced to conclude that, due to the possible complexity of nested expressions in OSQL, a fixed rewrite strategy does not suffice. The unnesting procedure (and therefore the precise way of parameter splitting) has to be guided by a (heuristic) cost model that weighs the costs of the various join operators, taking into account the level of nesting, as well as the operand types. The search space is determined by the various join links that are present between iterator operands. In Section 7.7, some extensions are discussed, and in Section 7.8, we compare our work with that of others. Section 7.9 gives conclusions and discusses future work.

7.2 Preliminaries

We work within one language. The type system of our language is that of the nested relational model, extended in the sense that, besides relation-valued attributes, arbitrary set-valued attributes are allowed as well. The language used is a combination of SQL-like constructs which allow for nesting and pure algebraic operators such as set operators and join. Below, we give the definitions of the main operators used in this paper.

Collect	$\Gamma[x : f(x) \mid p(x)](e)$	=	$\{f(x) \mid x \in e \wedge p(x)\}$
Semijoin	$e_1 \underset{x_1,x_2:p(x_1,x_2)}{\ltimes} e_2$	=	$\{x_1 \mid x_1 \in e_1 \wedge \exists x_2 \in e_2 \bullet p(x_1,x_2)\}$
Antijoin	$e_1 \underset{x_1,x_2:p(x_1,x_2)}{\triangleright} e_2$	=	$\{x_1 \mid x_1 \in e_1 \wedge \nexists x_2 \in e_2 \bullet p(x_1,x_2)\}$
Nestjoin	$e_1 \underset{x_1,x_2:f(x_1,x_2)\mid p(x_1,x_2);a}{\Delta} e_2$	=	$\{x_1 ++ \langle a = X\rangle \mid x_1 \in e_1 \wedge X = \{f(x_1,x_2) \mid x_2 \in e_2 \wedge p(x_1,x_2)\}\}$

The collect operator Γ is just a syntactic variant of the SELECT-FROM-WHERE construct of SQL:

$$\text{SELECT } f(x) \text{ FROM } x \text{ IN } e \text{ WHERE } p(x) \equiv \Gamma[x : f(x) \mid p(x)](e)$$

We have two special forms of Γ-expression:

1. $\Gamma[x : f \mid true](X) \equiv \alpha[x : f](X)$ which is also known as the map operator, and
2. $\Gamma[x : x \mid p](X) \equiv \sigma[x : p](X)$ which is also known as the selection operator.

The nestjoin operator was introduced in [StAB94]. The operator is a combination of join and grouping and was introduced to avoid problems with dangling tuples. Parameters of the nestjoin operator are a predicate p, a function f, and a label a. Each tuple in the left-hand join operand is concatenated with the unary tuple $\langle a = X\rangle$; the set X consists of the right-hand operand tuples that satisfy p, modified according to function f.

In addition to the operators listed above, we have the standard set operators union, difference, and intersection, the tuple constructor $\langle a_1 = e_1, \ldots, a_n = e_n\rangle$, tuple projection $x[L]$, which is denoted as x_L, projection π, join $\bowtie$, nest ν and unnest μ, etc. Predicates of the language may be arbitrary Boolean expressions, involving set comparison operators and quantifier expressions $\exists x \in X \bullet p$ and $\forall x \in X \bullet p$.

In this paper, capitals X, Y, Z are used to denote table expressions, i.e. base tables or set expressions with base table operands only. The expression $FV(e)$ stands for the set of free variables that occur in some expression e.

Operators collect, select, map, and quantifiers $\exists$ and $\forall$ are called *iterators*. In translation, the goal is to remove nested iterator occurrences as much as possible, by rewriting into joins. We give some example expressions:

Example 7.1 Nested expressions

1. $\sigma[x : \exists y \in Y \bullet x.a = y.a](X)$
2. $\Gamma[x : x.a \mid x.c \subseteq \alpha[y : y.a](Y)](X)$

3. $\alpha[x : \langle a = x.a, c = \Gamma[y : y.c \mid x.a = y.a](Y)\rangle](X)$
4. $\alpha[x : \Gamma[y : x.a + y.a \mid y.b = 1](Y)](X)$

The top level collect operator (or, comparably a map or select operator) contains nested iterators $\exists$ and Γ, either in the predicate or in the function. Nested expressions as shown above express a pure nested-loop execution strategy. By rewriting into join expressions, other implementation options come within reach:

Example 7.2 Join expressions

1. $X \underset{x,y:x.a=y.a}{\ltimes} Y$
2. $\Gamma[v : v.a \mid x.c \subseteq v.ys](X \underset{ys}{\Delta} \alpha[y : y.a](Y))$
3. $\alpha[v : \langle a = v.a, c = v.ys\rangle](X \underset{x,y:y.c|x.a=y.a;ys}{\Delta} Y)$
4. $\alpha[v : v.ys](X \underset{x,y:x.a+y.a|true;ys}{\Delta} \sigma[y : y.b = 1](Y))$

For the unnesting of an existential quantifier we may use the semijoin operator. The other examples concern expressions that cannot be transformed into flat relational join queries; we use the nestjoin operator.

7.3 Approach

Following relational tradition, we assume a three-level architecture of user language, logical algebra, and physical algebra. Query processing then consists of the following steps: translation into the logical algebra, logical query optimization, the generation of an access plan (translation into the physical algebra), and query execution [Grae93]. As said, we consider the first phase of query processing: the translation of OSQL into the logical algebra.

The goal in translating OSQL into the algebra is two-fold, namely (1) to obtain set-orientation, and (2) to achieve an efficient translation. Our first goal is motivated by the work done in the relational context. SQL languages offer the possibility to formulate nested queries, i.e. queries that contain subqueries (nested query blocks). In the relational model, the subquery operand is a base table or another subquery. In the translation to relational algebra, subqueries are removed by transforming the nested query into a join query. Transformation into join queries is advantageous because the join can be implemented such that its performance is better than pure nested-loop execution expressed by a nested query. Also for nested relational and object-oriented systems, the set-oriented paradigm seems appropriate. Though navigation has been considered as the prevailing method to access object-oriented databases in the past, recently more attention has been paid to set-oriented access methods, like pointer-based joins [ShCa90].

Our second goal originates from our belief that the actual translation into the logical algebra strongly influences performance. In our opinion, query optimization should not be restricted to the phases of algebraic rewriting and plan compilation; optimization should

play a role in all phases of query processing. Simple, standard algorithms for translating the user language (either SQL or calculus-like) into the algebra may result in very inefficient expressions. The inefficiency introduced in the translation phase is assumed to be reduced in the phase of logical optimization. This is quite a hard task, because in a pure algebraic context, in which information about the original query structure is scattered throughout the expression, it becomes difficult to find the proper optimization rules and to control the sequence of rule application [Naka90]. To support this claim, we invite the reader to transform the expression:

$$((\sigma[x : p(x)](X) \times Y) \cup (X \underset{x,y:q(x,y)}{\bowtie} Y)) \div Y$$

into:

$$X - \sigma[x : \neg p(x)](X) \underset{x,y:\neg q(x,y)}{\ltimes} Y$$

using algebraic rewrite rules only. Both expressions are translations of the expression:

$$\sigma[x : \forall y \in Y \bullet p(x) \vee q(x,y)](X)$$

The first uses division to handle the universal quantifier, the second set difference. In our opinion, it is better to try to achieve a 'good' translation right away than to try to rewrite inefficient algebraic expressions afterwards.

The main cause of the problems that have to be solved is the presence of set-valued attributes. In a language like OSQL, arbitrary nesting of query blocks may take place, in the SELECT- as well as in the WHERE-clause. The operands of nested query blocks may be base tables or set-valued attributes (or other subqueries); the two forms of iteration may alternate in arbitrary ways. The goal in translation is to achieve set-orientation, i.e. to remove nested iteration as much as possible. Nested iteration can be removed in two different ways:

Unnesting of expressions Unnesting rules may be applied either to the top level expression, moving nested base table occurrences to the top level, or to nested expressions, introducing nested set operations with base table operands and/or set-valued operands.

Unnesting of attributes Set-valued attributes can be unnested using the operator μ, and nested later on, if necessary, using ν.

We give an example. Consider the expression:

$$\sigma[x : \exists y \in Y \bullet y \in x.c](X)$$

or, equivalently:

$$\sigma[x : \exists y \in Y \bullet \exists v \in x.c \bullet y.d = v.d](X)$$

Attribute c of X is set-valued; we assume that both base table Y and attribute c are unary tables with only one attribute d. Below, we discuss some possibilities for the translation of the above expression.

The first option is to leave the expression as it is, adhering to a simple nested-loop execution strategy. Second, for the translation of existential quantification we may use the semijoin operator, resulting in:

$$X \underset{x,y:y \in x.c}{\ltimes} Y$$

Base table occurrence Y is moved to the top level. However, the result contains a complex join predicate, i.e., a predicate that concerns non-atomic attribute comparisons, and present join implementation techniques are not capable of handling such predicates. Third, we may introduce a nested semijoin operation:

$$\sigma[x : \exists y \in (Y \underset{y,v:y.d=v.d}{\ltimes} x.c) \bullet true](X)$$

Instead of performing a join between base tables X and Y, a local join is executed for each tuple $x \in X$. The fourth and last option is to flatten attribute c by means of the unnest operator μ. The result then is:

$$\nu_{d;c}(\mu_c(X) \underset{x,y:x.d=y.d}{\ltimes} Y)$$

Depending on the expression concerned, one or more options may be appropriate. Clearly, a cost model is needed to compare the performance of the various options, if applicable. In this paper, we do not consider the option of attribute unnesting; it can be treated independently and easily incorporated into our transformation algorithm.

7.4 Translation into nestjoin expression

In this section, we show that any base table occurrence can be moved to the top level by translating nested expressions into nested product expressions. A nested product expression can be looked upon as the equivalent of the relational project-select-product expression. A nested product expression, just like its relational equivalent, is highly inefficient; it can be considered as the Most Costly Normal Form [KeMo93]. Instead of trying to optimize the algebraic expression, we try to find better translation rules.

In further reading, it may be of help to realize that we simply draw the analogy between the translation of relational languages and that of OSQL. Below, we shortly describe the procedure followed in [Codd72] in translating relational calculus into relational algebra. The reduction algorithm of Codd takes as input an alpha expression in which the predicate is in Prenex Normal Form (PNF). First, the Cartesian product of base tables involved in the query is formed; next the Cartesian product is restricted according to the predicate that is the matrix of the PNF expression. Finally, projections and division operators are added to take care of quantifiers $\exists$ and $\forall$, respectively. In the phase of logical optimization, the Cartesian products are removed as much as possible by pushing through selections, modifying the join order, etc.

In our language, we have at our disposal the complex object equivalent of the relational Cartesian product, namely the *nested Cartesian product*, that consists of a nestjoin operator with join predicate $true$ and nestjoin function identity:

$$X \underset{x,y:y|true;a}{\Delta} Y \equiv X \times \{\langle a = Y \rangle\}$$

A product expression $X\ \Delta_{x,y:y|true;a}\ Y$ is abbreviated as $X\ \Delta_a\ Y$. Also, if the join predicate is $true$ or the join function is identity, they may be omitted: $X\ \Delta_{x,y:y|p;a}\ Y$ is

abbreviated as $X\ \Delta_{x,y:p;a}\ Y$, and $X\ \Delta_{x,y:f|true;a}\ Y$ as $X\ \Delta_{x,y:f;a}\ Y$.[1] In the footsteps of relational tradition, a nested OSQL query can be transformed into an expression that consists of a sequence of operations that work on the nested Cartesian product. The optimization objective then is to push through operations (not just projections and selections).

7.4.1 Example

Consider the following two-block query:

SELECT x
FROM x IN X
WHERE $x.c \subseteq$ SELECT $y.a$
FROM y IN Y
WHERE $y.b = x.d$

In [StAB94], we have shown that this query cannot be translated into a flat join query; grouping is needed. In our syntax, the corresponding expression is:

$$\Gamma[x : x \mid x.c \subseteq \Gamma[y : y.a \mid y.b = x.d](Y)](X)$$

Transformation of the above nested expression into a collect operation on the nested Cartesian product is easy and results in:

$$\Gamma[v : v_X \mid v_X.c \subseteq \Gamma[y : y.a \mid y.b = v_X.d](v.ys)](X \underset{ys}{\Delta} Y)$$

In SQL syntax this is:

SELECT v_X
FROM v IN $X \underset{ys}{\Delta} Y$
WHERE $v_X.c \subseteq$ SELECT $y.a$
FROM y IN $v.ys$
WHERE $y.b = v_X.d$

To obtain a nested product expression, the nested product of X and the (uncorrelated) subquery operand Y is formed. In the original collect expression, now having as operand the nested product instead of table X, the expression Y is replaced by the newly formed set-valued attribute ys. In addition, the occurrences of outer loop variable x must be adapted to account for the fact that the outer loop no longer iterates over X, but over the nested product that has an additional attribute. The expression v_X delivers the original tuple value of x. Note that $v_X.a$, with a an arbitrary label, is equivalent to $v.a$.

Of course, left as it is, the above transformation is no real improvement; the nested product expression must be further optimized. For example, operations applied to the newly formed set-valued attribute may be applied during the nestjoin operation itself:

$$\Gamma[v : v_X \mid v.c \subseteq v.ys](X \underset{x,y:y.a|y.b=x.d;ys}{\Delta} Y)$$

In SQL syntax we have:

[1] Because a predicate is a function too, this convention may be confusing; we assume that the type of the nestjoin parameter is clear from the context.

SELECT v_X
FROM v IN $X \underset{x,y:y.a|y.b=x.d;ys}{\Delta} Y$
WHERE $v.c \subseteq v.ys$

The subquery result is computed in advance, and stored as the set-valued attribute named ys. Below, we visualize the transformation:

Rewriting example 7.1 Unnesting

$$\Gamma[x : x \mid x.c \subseteq \overbrace{\Gamma[y : y.a \mid y.b = x.d](Y)}^{\text{subquery}}](X) \equiv \Gamma[v : v_X \mid v.c \subseteq \underbrace{v.ys}_{\text{result}}](X \underset{x,y:y.a|y.b=x.d;ys}{\Delta} Y)$$

The above optimized nestjoin expression can be implemented more efficiently than the original nested expression that contains the subquery and expresses a nested-loop execution. Like the regular join, the nestjoin operation can be implemented using indices, sorting, hashing, or whatever implementation technique that is available and profitable.[2]

7.4.2 General rule

We have the following general equivalence rule:

Rule 7.1 Transformation into nested Cartesian product

$$\Gamma[x : E(x, Y)](X) \equiv \Gamma[v : E(v_X, v.ys)](X \underset{ys}{\Delta} Y)$$

The left-hand side of the equivalence above is a collect expression, in which for each $x \in X$ some expression E is evaluated. E corresponds to some parameter expression $f \mid p$; because it is irrelevant whether Y occurs in f or in p, we may abstract from the form of the parameter expression. In E, we find occurrences of loop variable x and some uncorrelated subquery Y; this is denoted as $E(x, Y)$. After having formed the nested product of X and Y, we have to perform some substitutions in the original collect expression. Loop variable x is replaced by a fresh variable v, the occurrences of x in E are replaced by the tuple projection v_X, and the occurrences of Y in E are replaced by the expression $v.ys$, with ys a label that does not occur in X. All this is denoted as $E(v_X, v.ys)$. Below, we give some example applications of the rule.

Rewriting example 7.2 Transformation into nested product

- $\alpha[x : x.c - Y](X) \equiv \alpha[v : v.c - v.ys](X \underset{ys}{\Delta} Y)$
- $\alpha[x : x.c - \sigma[y : y.a = 1](Y)](X) \equiv \alpha[v : v.c - v.ys](X \underset{ys}{\Delta} \sigma[y : y.a = 1](Y))$
- $\sigma[x : \exists y \in Y \bullet x.a = y.a](X) \equiv \Gamma[v : v_X \mid \exists y \in v.ys \bullet v.a = y.a](X \underset{ys}{\Delta} Y)$

[2] Some restrictions hold, though, because the nestjoin operator is not commutative, because its output is structured.

7.4.3 Optimization of the general rule

As remarked, a nested product expression does have not especially good performance. We have to find better translation rules; below, we present one such rule: the rule for unnesting a collect nested in another collect.

Rule 7.2 Unnesting collect within collect

$$\Gamma[x : E(x, \Gamma[y : f \mid p](Y))](X) \equiv \Gamma[v : E(v_X, v.ys)](X \underset{x,y:f \mid p;ys}{\Delta} Y)$$

Again, for each $x \in X$, some expression E is evaluated. E contains occurrences of x and a nested collect expression $\Gamma[y : f \mid p](Y)$. It is indifferent whether the subquery occurs in the predicate part of E or in the function, meaning that the rule can be used for removing subqueries from the WHERE- as well as from the SELECT-clause. The same procedure as described above is followed, forming the nestjoin of X and Y, and performing the necessary substitutions. The expression that is the parameter of the nested collect, i.e. $f \mid p$, now becomes the nestjoin parameter expression; instead of a nested Cartesian product, we have a true nestjoin operation. An example application of this rule was given in Example 7.1. In Section 7.5.2, we discuss the types of join predicates and functions allowed.

Transformations may not introduce free variables. If the above rule is applied on the top level, then it must hold that the nestjoin predicate and function contain no free variables other then the nestjoin variables themselves; if the rule is applied at nested levels, variables from higher levels may occur free in function and predicate.

Of course, the problem of achieving a translation, and an efficient one too, is not solved by giving the above rules. Many issues remain to be discussed. For example, we have to decide what unnesting strategy should be followed, we have to consider expressions containing quantifiers etc.

7.4.4 Why is a nestjoin better than a nested loop?

Or, stated differently: how to implement the nestjoin such that the performance of a nestjoin expression is better than that of the corresponding nested-loop expression, that possibly is implemented with loop optimization methods such as caching, the use of indices, etc. The answer to the above question consists of two parts:

- Each individual nestjoin operator such that its performance is better than the performance achieved by means of nested-loop execution.
- To have nestjoin sequences instead of nested expressions offers additional optimization opportunities.

First, a nestjoin operator can be implemented efficiently. If an (atomic) nestjoin predicate is present, and the nestjoin function is identity, the nestjoin can be implemented as a modification of implementation methods for the regular join. Any join algorithm that has an output ordered according to the left hand operand tuple values is suitable to be used for the implementation of the nestjoin. The only difference with relational join lies in in

the structure of the output—in the nestjoin, each left-hand tuple is concatenated with the *set* of matching right-hand operand tuples. If a function is present as well, then still any suitable join algorithm can be used to retrieve the tuples needed, and the function values can be computed in a separate intermediate step between the execution of the join and the creation of the result tuples, just like additional selections and projection can be computed after the join, by means of a single physical algebra operator.

Also, if the nestjoin predicate is missing (equivalent to $true$), but the function is present (not equivalent to identity), the nestjoin can be implemented efficiently. The use of indices and caching of (partial) results enables to avoid duplicate removal and duplicate computations. Consider for example the expression:

$$X \underset{x,y:x.a+y.b|true;ys}{\Delta} Y$$

If an index exists on attribute b of table Y, an index-scan suffices to retrieve the attribute values of b.

Second, the use of (nest)joins instead of nested expressions offers additional optimization opportunities, because of the presence of intermediate results, which can be optimized independently. Consider the nested expression:

$$\alpha[x : \alpha[y : \alpha[z : x.a + y.b + z.c](Z)](Y)](X)$$

In a pure nested-loop execution, first variables x, y, z are bound and then the innermost map function is evaluated. The use of indices and caching of (partial) results enables to avoid duplicate removal and duplicate computations. But to do this, i.e., to implement nested queries with the use of indices, caching, or even other access methods, causes a heavy administrative overload, which is just the problem that is solved in the translation into an algebraic join expression. Nested expressions allow for the use of multi-variable expressions (predicates or functions), and nestjoins, which are binary operators, do not. (Nest)joins allow for the construction of *partial results* that are *closed*, i.e. that do not contain free variables (at least, when occurring at the top level). Additional access methods can be used to speed up the evaluation of these intermediate results. Translation into an algebraic join expression can be looked upon as a divide-and-conquer approach to the task of efficient evaluation of a nested expression.

7.5 Outline of the algorithm

In this section, we outline the basic steps of our translation algorithm. The discussion is restricted in that we consider nested iterators only; the operands of these may be either set-valued attributes or tables. We do not pay attention to nested occurrences of set (comparison) operators.

The input to the translation is a collect expression $\Gamma[x : f \mid p](e)$. Recall that the collect is the syntactical equivalent of the SQL SELECT-FROM-WHERE construct. The expressions f, p, and e may be arbitrary, containing other collects and/or quantifier expressions. The goal in translation is to remove nested collects and quantifiers by rewriting into join operators. We want to achieve an efficient translation, i.e. we try to avoid Cartesian

product as much as possible, we try to push through predicates and functions, and give preference to cheap operators, given a choice.

The basic rewrite algorithm consists of the following steps:

1. **Standardization**. Standardization involves composition and predicate transformation.

2. **Translation**. In this step, subqueries are removed from parameter expressions by rewriting nested expressions into set expressions.

These steps are described in more detail below.

7.5.1 Standardization

The input to the translation is an arbitrary collect expression $\Gamma[x : f \mid p](e)$. In composition, collect operands and quantifier range expressions that are iterator expressions are transformed into table or attribute expressions. Composition means the combination two iterators into one; in decomposition one iteration is transformed into two separate iterations that each perform part of the work. As in the relational context, composition is needed because the user/system-prescribed order of operations (evaluation of predicates and functions) is not necessarily the most efficient. In addition, composition may offer additional optimization opportunities.

We deal with the three iterators Γ, $\exists$, and $\forall$, so we have the following rules:

Rule 7.3 Composition

1. $\Gamma[x : f(x) \mid p(x)](\Gamma[x : g(x) \mid q(x)](X)) \equiv \Gamma[x : f(g(x)) \mid p(g(x)) \wedge q(x)](X)$
   Note that the right-hand side contains the common subexpression $g(x)$.
2. $\exists x \in \Gamma[x : f(x) \mid p(x)](X) \bullet q(x) \equiv \exists x \in X \bullet p(x) \wedge q(f(x))$
3. $\forall x \in \Gamma[x : f(x) \mid p(x)](X) \bullet q(x) \equiv \forall x \in X \bullet \neg p(x) \vee q(f(x))$

The output of the phase of composition is a possibly nested collect expression in which the operand of each iterator is either a base table, a set-valued attribute, or a set expression (union, product), but not an iterator expression.

In the relational context, predicates usually are rewritten into Prenex Normal Form (PNF). After transformation into PNF, the matrix of the PNF expression can be optimized [JaKo84]. However, in [Bry89], it is proposed to use another normal form for predicates, namely the Miniscope Normal Form (MNF), in which quantifier scopes do not contain subexpressions that do not depend on the quantifier variable itself. Allegedly, MNF allows for a better translation, i.e. a translation with better results. As we will see, rewriting into MNF is one example of the generally beneficial rewrite strategy to remove local constants from iterator parameter expressions; therefore, we rewrite into MNF.

7.5.2 Translation

In this section, we describe the actual translation. We present a basic set of rewrite rules for the translation of (nested) expressions. The goal in translation is (1) to remove nested iteration, and (2) to push through operations (predicates and functions). It is important to note that we simply follow relational tradition, in which nested expressions are transformed into product expressions, and selections and projections are pushed through. In a complex object model, besides the standard Cartesian product, we need the nested Cartesian product, and instead of pushing through projections, we are concerned with pushing through arbitrary functions. Consequently, the transformation rules are more complex than in the relational context.

Basic set of equivalence rules

Our approach to translation may be characterized as heuristic [Naka90]. The translation algorithm is presented in the form of a set of rewrite rules; translation proceeds by means of pattern matching. For each syntactic construct we have an ordered set of rewrite rules that consists of a basic rewrite rule, and some (zero or more) rules that are applied in priority to the basic rule. Depending on the syntactic form of the query under consideration, there may exist transformations that give better results (according to some cost model) than the standard transformation. For example, in the relational model, universal quantification usually is handled by means of division. However, in some cases the antijoin operator may be used, resulting in an expression that has better performance. Whenever a better translation rule, or more efficient logical operators (e.g. [DeLa92, BlMG93]) are found, the set of rewrite rules can be easily extended. In general, the rules presented may be applied to the top-level expression, as well as to nested expressions. In some cases, multiple rules can be applied to multiple (sub)expressions, so we have to provide a rewrite strategy.

In this section, we give rules for unnesting quantifiers nested within collects, and for unnesting collects nested within collects. In theory, the only condition placed on transformations is that they do not introduce free variables. However, we require that (1) join predicates consist of atomic terms only, (2) that both join variables occur free in the join predicate, and (3) that nestjoin functions do not contain base table occurrences. Atomic terms are comparisons between attribute values and/or constants of atomic type. In principle, any predicate can be used as join predicate. For example, it is perfectly legal to write $X \bowtie_{x,y:x.a \in y.c} Y$, or even $X \bowtie_{x,y:x.a \in \Gamma[z:z|p(x,y,z)](Z)} Y$. However, present join implementation techniques are not capable of handling complex join predicates; joins with complex predicates probably will be handled by nested-loop execution after all.

Whenever quantifiers occur in predicates between blocks, we apply the rewrite rules presented below, translating nested expressions with quantification into relational join (product, join, semi-, or antijoin) operations. In case it is not possible to apply the first alternative, we use the second, and so on.

Rule 7.4 Unnesting existential quantification

1. $\Gamma[x : f \mid \exists y \in Y \bullet p(x,y)](X) \equiv \Gamma[x : f \mid true](X \underset{x,y:p(x,y)}{\ltimes} Y)$
2. $\Gamma[x : f(x) \mid \exists y \in \sigma[y : p(x,y)](Y) \bullet q(x,y))](X) \equiv$
   $\Gamma[v : f(v_X) \mid q(v_X, v_Y))](X \underset{x,y:p(x,y)}{\bowtie} Y)$
3. $\Gamma[x : f(x) \mid \exists y \in Y \bullet p(x,y)](X) \equiv \Gamma[v : f(v_X) \mid p(v_X, v_Y)](X \times Y)$

Rule 7.5 Unnesting universal quantification

1. $\Gamma[x : f \mid \forall y \in Y \bullet p(x,y)](X) \equiv \Gamma[x : f \mid true](X \underset{x,y:\neg p(x,y)}{\triangleright} Y)$
2. $\Gamma[x : f \mid \forall y \in Y \bullet p(x,y)](X) \equiv \Gamma[x : f \mid true]((\Gamma[v : v \mid p(v_X, v_Y)](X \times Y)) \div Y)$

The rules given below were discussed in the previous section. In case it is impossible to apply the first, we try the second.

Rule 7.6 Unnesting collect

1. $\Gamma[x : E(x, \Gamma[y : f \mid p](Y))](X) \equiv \Gamma[v : E(v_X, v.ys)](X \underset{x,y:f|p;ys}{\Delta} Y)$
2. $\Gamma[x : E(x,Y)](X) \equiv \Gamma[v : E(v_X, v.ys)](X \underset{ys}{\Delta} Y)$

During translation, operations are pushed through whenever possible (and allowed, considering our requirements with respect to join predicates and functions). The rules for pushing through selection predicates to standard relational join (product) or join operands are simple and well-known or derived easily; we do not give them here (see for example [ElNa89] for rules concerning products). Below, we give rules for pushing through operations to nestjoin operands and to the nestjoin itself.

Rule 7.7 Pushing through operations to nestjoin (operands)

1. **To left operand**
   $\Gamma[v : E(v_X)](X \underset{x,y:g|q;ys}{\Delta} Y) \equiv \Gamma[v : E(x)](X) \underset{x,y:g|q;ys}{\Delta} Y$
   In this rule, E is an expression referencing only attributes of the left-hand join operand X. In case the collect does not modify values $x \in X$, but restricts or extends them, it may be evaluated before the join. For example, E may correspond to a simple selection, as in $v_X \mid v_X.a = 1$, or involve a collect, as in v_X **except** $(c = \Gamma[z : z \mid v_X.a = z.a](Z))$.
2. **To right operand**
   $\Gamma[v : E(e(v.ys))](X \underset{x,y:g|q;ys}{\Delta} Y) \equiv \Gamma[v : E(v.ys)](X \underset{x,y:g|q;ys}{\Delta} e(Y))$
   In this rule, e is an expression with subexpression $v.ys$, that contains no free variables other than v in $v.ys$. Expression e may be computed before the join, provided that expression e does not modify values $y \in Y$, but restricts or extends them. For example, e may involve a selection $\Gamma[y : y \mid p(y)](v.ys)$, or a join $v.ys \bowtie Z$, but not a semijoin $Z \ltimes v.ys$.

3. **To the join itself**

$$\Gamma[v : E(\Gamma[y : f(v_X, y) \mid p(v_X, y)](v.ys))](X \underset{x,y:g(x,y)|q(x,y);ys}{\Delta} Y) \equiv$$
$$\Gamma[v : E(v.ys)](X \underset{x,y:f(x,g(x,y))|p(x,g(x,y))\wedge q(x,y);ys}{\Delta} Y)$$

A function applied to the new nestjoin attribute may be applied during the nestjoin operation itself. The rule as given above expresses pushing through in its most general form. In our rewritings, it is assumed that the final nestjoin predicate consists of atomic terms only, and that the nestjoin function does not contain table expressions.

Rewrite strategy

In the first instance, the top level iterator is a collect expression that possibly contains nested iterators with base table or set-valued operands. The rules given above are generally applicable, so they may be applied at will: at nested levels, to expressions with base table as well as set-valued attributes etc. The only 'natural' restriction is that it is not allowed to introduce free variables; recall that we have made some other restrictions too.

In the translation, whenever possible, we try to apply the rules for pushing through predicates and functions. In unnesting, we use a top-down strategy, recursively joining outermost iterator operand with next inner, if possible. We first consider pairs of table expressions, then pairs of table expressions and set-valued attributes, and finally pairs of set-valued attributes. In other words, first we try to join the top level operand, which is a table, with each of the nested base table occurrences in the query. Next, we take the next inner iterator operand that is a base table, and follow the same procedure. We repeat this three times, first trying to join set-valued attributes and tables, next joining tables and set-valued attributes, and finally pairs of set-valued attributes. After each introduction of a local join, we return to the top level. In case there is a choice, with multiple subqueries, the order of unnesting is arbitrary. We avoid Cartesian products: these are introduced only if everything else has failed. The advantage of a top-down unnesting strategy is that the level of nesting in the algebraic expression is kept to a minimum.

7.5.3 Splitting expressions

We now investigate the effect of our initial rewrite strategy by considering the translation of the following expression:

$$\alpha[x : x \ \mathbf{except}\ (c = \sigma[y : p(x, y) \wedge \exists z \in Z \bullet q(x, z) \wedge r(y, z)](Y))](X)$$

We assume that predicates p, q, and r are atomic, and that X, Y, and Z are base tables. The result of the query is the set of tuples $x \in X$, extended with a new set-valued attribute c that contains all tuples $y \in Y$ that satisfy the predicate, which involves a nested quantification. For simplicity, we omit the **except** construct, so that the result is a set of sets:

$$\alpha[x : \sigma[y : p(x, y) \wedge \exists z \in Z \bullet q(x, z) \wedge r(y, z)](Y)](X)$$

Given our set of rewrite rules presented above, for unnesting of quantifiers and collects together with the rules for pushing through operations to join (operands), often the only option is to introduce (nested) Cartesian products, also because predicates are in MNF and

therefore the rules for unnesting quantification are not always applicable. W.r.t. the above expression, the selection predicate is not atomic, so we cannot use the rule for unnesting a collect (or equivalently, a selection or a map) and the quantifier cannot be removed because it does not occur at the top level. Before we can apply the unnesting rules, the predicate has to be rewritten. Splitting of expressions enables us to introduce joins instead of Cartesian products. The rules used for splitting predicates are the following:

Rule 7.8 Splitting predicates

1. $\Gamma[x : f \mid p \wedge q](X) \equiv\equiv \Gamma[x : f \mid q](\sigma[x : p](X))$
2. $\Gamma[x : f \mid p \vee q](X) \equiv \Gamma[x : f \mid p](X) \cup \Gamma[x : f \mid q](X)$
3. $\exists x \in X \bullet p \wedge q \equiv \exists x \in \sigma[x : p](X) \bullet q$
4. $\exists x \in X \bullet p \vee q \equiv \exists x \in X \bullet p \vee \exists x \in X \bullet q$

Expressions that denote function results can be split of as well, using the rules:

Rule 7.9 Splitting functions

1. (a) Let x not occur free in f outside of $g(x)$, then:
   $\Gamma[x : f(g(x)) \mid p(x)](X) \equiv \alpha[x : f(x)](\Gamma[x : g(x) \mid p(x)](X))$
   (b) Let x not occur free in f outside of instances of $g_i(x)$, then:
   $\Gamma[x : f(g_1(x), \ldots, g_n(x)) \mid p(x)](X) \equiv$
   $\alpha[x : f(x.a_1, \ldots, x.a_n)](\Gamma[x : \langle a_1 = g_1(x), \ldots a_n = g_n(x)\rangle \mid p(x)](X))$
2. Let X be a table, let labels a_i not occur in the schema of X, and let x not occur free in p outside of instances of $g_i(x)$, then:
   $\Gamma[x : f(x) \mid p(g_1(x), \ldots, g_n(x))](X) \equiv$
   $\Gamma[v : f(v_X) \mid p(v.a_1, \ldots, v.a_n)](\Gamma[x : x \textbf{ except } (a_1 = g_1(x), \ldots a_n = g_n(x))](X))$
3. (a) Let x not occur free in p outside of $g(x)$, then:
   $\exists x \in X \bullet p(g(x)) \equiv \exists x \in \alpha[x : g(x)](X) \bullet p(x)$
   (b) Let x not occur free in p outside of instances of $g_i(x)$, then:
   $\exists x \in X \bullet p(g_1(x), \ldots, g_n(x)) \equiv$
   $\exists x \in \Gamma[x : \langle a_1 = g_1(x), \ldots a_n = g_n(x)\rangle \mid true](X) \bullet p(x.a_1, \ldots, x.a_n)$

Note that for splitting a collect predicate we can only extend operand tuples because the original values are needed for the evaluation of the function. The rules that deal with multiple occurrences of expressions in which the iterator variable occurs free correspond to the use of the **define**-clause as proposed in [ClMo93].

Below, we show that the way of splitting predicates (and functions) strongly influences the outcome of the rewrite process, because it determines which unnesting rules can be applied at some point in rewriting. We proceed with our example expression. First, we split the selection predicate, and evaluate (atomic) predicate p before the quantification. The quantifier scope is left as it is, and the result is an expression that contains a nested semijoin operation with one set-valued and one base table operand. The semijoin predicate is not closed, i.e. variable v occurs free in it.

Rewriting example 7.3

$$
\begin{aligned}
&\alpha[x : \sigma[y : p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y)](X) \\
&\quad\equiv\quad \alpha[x : \sigma[y : \exists z \in Z \bullet q(x,z) \wedge r(y,z)](\sigma[y : p(x,y)](Y))](X)\ (\textbf{split}) \\
&\quad\equiv\quad \alpha[v : \sigma[y : \exists z \in Z \bullet q(v_X,z) \wedge r(y,z)](v.ys)](X \underset{x,y:p(x,y);ys}{\Delta} Y)\ (\textbf{unnest}) \\
&\quad\equiv\quad \alpha[v : v.ys \underset{y,z:q(v_X,z)\wedge r(y,z)}{\ltimes} Z](X \underset{x,y:p(x,y);ys}{\Delta} Y)\ (\textbf{unnest})
\end{aligned}
$$

It is not allowed to join $X\ \Delta\ Y$ with Z at top level, because then the result would be a Cartesian product, so we perform a local join between attribute ys and Z.

Instead of leaving the quantifier scope as it is, it can be split as well, in two different ways. First, after having split the selection predicate, we may push through conjunct q. Following the top-down unnest strategy, we then can join X with Y at the top level, and next join the result with Z, also at the top level:

Rewriting example 7.4

$$
\begin{aligned}
&\alpha[x : \sigma[y : p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y)](X) \\
&\quad\equiv\quad \alpha[x : \sigma[y : \exists z \in Z \bullet q(x,z) \wedge r(y,z)](\sigma[y : p(x,y)](Y))](X)\ (\textbf{split}) \\
&\quad\equiv\quad \alpha[x : \sigma[y : \exists z \in \sigma[z : q(x,z)](Z) \bullet r(y,z)](\sigma[y : p(x,y)](Y))](X)\ (\textbf{split}) \\
&\quad\equiv\quad \alpha[v : \sigma[y : \exists z \in \sigma[z : q(v_X,z)](Z) \bullet r(y,z)](v.ys)](X \underset{x,y:p(x,y);ys}{\Delta} Y)\ (\textbf{unnest}) \\
&\quad\equiv\quad \alpha[w : \sigma[y : \exists z \in w.zs \bullet r(y,z)](w.ys)]((X \underset{x,y:p(x,y);ys}{\Delta} Y) \underset{v,z:q(v_X,z);zs}{\Delta} Z) \\
&\qquad\ (\textbf{unnest}) \\
&\quad\equiv\quad \alpha[w : w.ys \underset{y,z:r(y,z)}{\ltimes} w.zs]((X \underset{x,y:p(x,y);ys}{\Delta} Y) \underset{v,z:q(v_X,z);zs}{\Delta} Z)\ (\textbf{unnest})
\end{aligned}
$$

The last step involves the introduction of a local semijoin between set-valued attributes.

Alternatively, we can push through conjunct r. Again, X is joined with Y at the top level. Next, we cannot join the result with Z at the top level because variable y occurs free in the predicate of the selection with operand Z—we are forced to introduce a local join between attribute ys and Z.

Rewriting example 7.5

$$
\begin{aligned}
&\alpha[x : \sigma[y : p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y)](X) \\
&\quad\equiv\quad \alpha[x : \sigma[y : \exists z \in Z \bullet q(x,z) \wedge r(y,z)](\sigma[y : p(x,y)](Y))](X)\ (\textbf{split}) \\
&\quad\equiv\quad \alpha[x : \sigma[y : \exists z \in \sigma[z : r(y,z)](Z) \bullet q(x,z)](\sigma[y : p(x,y)](Y))](X)\ (\textbf{split}) \\
&\quad\equiv\quad \alpha[v : \sigma[y : \exists z \in \sigma[z : r(y,z)](Z) \bullet q(v_X,z)](v.ys)](X \underset{x,y:p(x,y);ys}{\Delta} Y)\ (\textbf{unnest}) \\
&\quad\equiv\quad \alpha[v : \Gamma[w : w_{ys} \mid q(v_X,w_Z)](v.ys \underset{y,z:r(y,z)}{\bowtie} Z)](X \underset{x,y:p(x,y);ys}{\Delta} Y)\ (\textbf{unnest})
\end{aligned}
$$

We remark that by means of relatively simple additional transformation rules the first and the third result can be easily rewritten into the second, which is a fully unnested expression, i.e. an expression that does not contain nested base table occurrences.

In the previous rewritings, predicate p is evaluated before the quantification. Instead, we can split the selection predicate such that the quantification is evaluated first. We do not introduce a nested Cartesian product $X \ \Delta \ Y$ at the top level, because we want to avoid Cartesian products. Instead, a nested regular join between Y and Z can be formed, after pushing through conjunct r, which avoids to introduce a local join expression with free variable. The result achieved then contains one regular join and one nestjoin, and no nested join operators.

Rewriting example 7.6

$$
\begin{aligned}
&\alpha[x : \sigma[y : p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y)](X) \\
&\quad \equiv \quad \alpha[x : \sigma[y : p(x,y)](\sigma[y : \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y))](X) \ (\textbf{split}) \\
&\quad \equiv \quad \alpha[x : \sigma[y : p(x,y)](\sigma[y : \exists z \in \sigma[z : r(y,z)](Z) \bullet q(x,z)](Y))](X) \ (\textbf{split}) \\
&\quad \equiv \quad \alpha[x : \sigma[y : p(x,y)](\Gamma[w : w_Y \mid q(x,w_Z)](Y \underset{y,z:r(y,z)}{\bowtie} Z))](X) \ (\textbf{unnest}) \\
&\quad \equiv \quad \alpha[v : \sigma[y : p(v_X,y)](v.cs)](X \underset{x,w:w_Y \mid q(x,w_Z);cs}{\Delta} (Y \underset{y,z:r(y,z)}{\bowtie} Z)) \ (\textbf{unnest}) \\
&\quad \equiv \quad \alpha[v : v.cs](X \underset{x,w:w_Y \mid p(x,w_Y) \wedge q(x,w_Z);cs}{\Delta} (Y \underset{y,z:r(y,z)}{\bowtie} Z)) \ (\textbf{push})
\end{aligned}
$$

The latter result is essentially different from the former. Instead of two nestjoins and one nested semijoin, we have obtained an expression with one nest- and one regular join. Without doubt, the two can be rewritten into each other, but we do not immediately see how this can be done easily—by means of one or two algebraic equivalence rules, like the following:

Rule 7.10 Join/nestjoin order

1. $(X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:z \mid r(v_Y,z);zs}{\Delta} Z \equiv X \underset{x,v:p(x,v_Y)}{\bowtie} (Y \underset{y,z:z \mid r(y,z);zs}{\Delta} Z)$ (**associativity**)
2. Let θ be either $\bowtie$ or Δ, then:
   $(X \underset{x,y:p(x,y)}{\theta} Y) \underset{v,z:q(v_X,z)}{\theta} Z \equiv (X \underset{x,z:q(x,z)}{\theta} Z) \underset{v,y:p(v_X,y)}{\theta} Y$ (**exchange**)

The conclusion is that the outcome of the translation is strongly determined by (1) the set of rewrite rules that we have at our disposal, (2) the order of rule application. Given our fixed rewrite strategy, the way predicates are split determines which unnest rules can be applied; the syntactic form of the unnest rules, and also the restrictions posed on their application require that predicates and functions are split.

The above observation is important. One could think that nothing is lost, because in further algebraic optimization one expression can be easily rewritten into another, but this is not so. Even in the relational model, the transformation of algebraic expressions that contain arbitrary operators like set operators and division is a difficult task. For languages that support complex objects, algebraic rewriting is even harder. The equivalence rules are hard to find, and to find the proper order of rule application is even harder, because the algebraic operators are more complex, and because we have to deal with nesting. For the translation from nested expression into more set-oriented ones, we have to find proper heuristics to guide the process. To find heuristics is not easy either. One heuristic that is

expected to be good is to evaluate cheaper predicates first. An atomic predicate presumably is less costly than a predicate that contains quantification, so w.r.t. our example discussed above, predicate p should be evaluated before the quantification. However, in this case, to evaluate the quantification before predicate p leads to an expression that cannot be considered as a bad result at first glance.

7.6 Heuristic rewriting

We need heuristics to guide the transformation process. In this section, we present some of the heuristic rules that can be used to achieve a better translation. Many others can be invented. Next, we present a new rewrite strategy, that prescribes how to use the heuristic rules, and then we discuss the result, which still can be improved in many cases.

7.6.1 Heuristics

Given a choice, expressions are split such that the cheapest (most restrictive predicate, less costly function) expression part is evaluated first. Predicates and functions, either nested or at the top level, can be classified on a pure syntactic basis according to the following criteria:

1. The number of variables that occurs free (constant, monadic, dyadic, multi-variable).
2. The presence of other iterators, i.e. the nesting level.
3. The type of the iterator operand (table or (set-valued) attribute subquery).
4. The number of subqueries present in the parameter expression (single or multiple subqueries).

We present a nested expression that will serve as the leading example in this section:

$$\sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet p_1(x) \wedge p_2(y) \wedge p_3(x,y) \wedge p_4(z) \wedge p_5(x,z) \wedge p_6(y,z)](X)$$

We alert the reader that we have chosen the above example just for illustration purposes—it looks deceptively relational, but the same nesting pattern can be achieved with collects instead of quantifiers. For the time being, assume that X, Y, and Z are base tables and that all predicates p_i are atomic.

We note that the selection predicate is in PNF, and that the matrix of the PNF expressions contains conjuncts that do not depend on one or both quantifier variables. The predicate therefore is rewritten into MNF, as proposed in [Bry89]:

$$\begin{array}{l}\sigma[x : p_1(x) \wedge \\ \quad \exists y \in Y \bullet p_2(y) \wedge p_3(x,y) \wedge \\ \quad\quad \exists z \in Z \bullet p_4(z) \wedge p_5(x,z) \wedge p_6(y,z)](X)\end{array}$$

A comparable transformation that concerns a map operator is the following:

$$\alpha[x : \alpha[y : x.b + y.b](Y)](X) \equiv \alpha[x : \alpha[y : x + y.b](Y)](\alpha[x : x.b](X))$$

The subexpression $x.b$ does not depend on the inner map variable y, so it can be evaluated outside of the scope of y. The above transformation corresponds to the idea of pushing through a projection. Even if attribute names occur at different nesting levels, projections can be pushed through:

$$\alpha[x : \langle a = x.a, c = \alpha[y : x.b + y.b](Y)\rangle](X) \equiv \\ \alpha[v : \langle a = v.a, c = \alpha[y : v.b + y.b](Y)\rangle](\alpha[x : \langle a = x.a, b = x.b\rangle](X))$$

We see that in a complex object model, as in the relational model, it is only necessary to preserve those attributes that are needed in subsequent computations. Subexpressions of parameter expressions that are constant w.r.t. the corresponding iterator variable, i.e. subexpressions in which the iterator variable does not occur free are called *local constants*. We have a first heuristic transformation rule:

Heuristic rule 7.1 Local constants are removed from the iterator parameter expression as much as possible.

To remove independent subformulas from parameter expressions, for predicates we use the descoping rules (possibly others are needed too, see[Bry89]):

Rule 7.11 Descoping Let $x \notin FV(p)$, then:

1. $\exists x \in X \bullet p \wedge q(x, y) \equiv p \wedge \exists x \in X \bullet q(x, y)$
2. $\exists x \in X \bullet p \vee q(x, y) \equiv p \vee \exists x \in X \bullet q(x, y)$[3]

To obtain independent subformulas, the technique of quantifier exchange may be of help. For functions, we use the rules for splitting of functions as given in Rule 7.9. Note that w.r.t. quantifier scopes, independent subformulas can be removed completely. W.r.t. functions, this is not possible—in the above map transformation, the inner map still contains the local constant x.

Second, another type of constant expression is one in which no variables from higher levels occur free; this type of expression is called an *global constant*. Global constants are evaluated independently, which becomes possible by naming them by means of a local definition facility:

$$\sigma[x : \exists y \in \sigma[y : p(y)](Y) \bullet q(x, y)](X) \equiv \\ \sigma[x : \exists y \in Y' \bullet q(x, y)](X) \textbf{ with } Y' = \sigma[y : p(y)](Y)$$

Heuristic rule 7.2 Global constants are named with a local definition facility.

[3] Because variables are range-restricted, we have to take into account the possibility of empty ranges. The correct transformation is:

$$\textbf{if } X = \emptyset \textbf{ then } false \textbf{ else } p \vee \exists x \in X \bullet q(x, y)$$

For reasons of simplicity, we assume quantifier ranges are never empty.

In [ClMo93], a transformation called dependency-based optimization is described, which is comparable to the transformations described above. In dependency-based optimization, subformulas are named with a local definition facility, and then pushed out of query blocks whenever possible. However, it is not precisely clear what type of subformulas are considered. Naming is mainly applied to path expressions, and it is not clear, for example, how independent predicates are dealt with. In our approach, we remove independent subformulas of maximum size, and for predicates we prefer transformation into MNF above naming.

We return to our leading example. We notice two monadic conjuncts at the top level, i.e. $p_1(x)$ and the quantifier expression itself, and also two at a nested level, namely $p_2(y)$, and $p_4(z)$. Monadic expressions can be pushed through to the corresponding iterator operand by means of the rules for splitting predicates, thereby eventually creating global constants that can be evaluated independently:

$$\sigma[x : \exists y \in \sigma[y : p_2(y)](Y) \bullet p_3(x,y) \wedge \\ \exists z \in \sigma[z : p_4(z)](Z) \bullet p_5(x,z) \wedge p_6(y,z)](\sigma[x : p_1(x)](X))$$

Heuristic rule 7.3 Monadic expressions are evaluated first, if possible.

An example of Heuristic rule 7.3 that involves the map operator is the following:

$$\alpha[x : \alpha[y : x.b + y.b](Y)](X) \equiv \alpha[x : \alpha[y : x.b + y](\alpha[y : y.b](Y))](X)$$

Again, the above example deals with pushing through a projection. Notice however, that also very complex functions can be pushed through.

We made the assumption that predicate p_1 is atomic, so it seems a good strategy to restrict operand X before evaluation of the quantifier expression. However, the quantifier expression is monadic as well. Given a choice, it seems advantageous to evaluate expressions that do not contain quantification before ones that do, i.e. to evaluate expressions in order of their respective nesting level. Whenever the nesting level is the same, we may choose for arbitrary evaluation order, or invent some other heuristic rule.

Heuristic rule 7.4 Expressions are evaluated in order of nesting level.

So far so good. We have discussed constant and monadic expressions, and now we have to decide what to do with dyadic and multi-variable parameter expressions. Joins are binary operators, so dyadic expressions are candidates for join predicates and functions. In our example expression, we notice dyadic conjuncts p_3, p_5, and p_6 that mutually link the base tables that occur in the query. It is tried to split multi-variable predicates and functions as needed, i.e., as prescribed by the unnesting strategy. Whenever we investigate the possibility of joining two tables A and B, we search for maximal closed expressions that refer to attributes of A and B.

Heuristic rule 7.5 Predicates and functions are split as prescribed by the unnesting strategy.

In a complex object model, splitting of multi-variable expressions into dyadic expressions is not always possible. In the relational model, predicates consist of atomic attribute comparisons only, and multi-variable selection predicates and also quantifier scopes can be split into dyadic and/or monadic formulas, by means of decomposition for conjunction, and the union or distribution of quantification for disjunction. In the algebra, literals can be moved around with great freedom, at least for select-project-join expressions. In a complex object model, we may have functions as well as predicates, and splitting functions is not always possible. Consider the three-variable expressions:

1. $\sigma[z : x.a + z.a = y.a](Z)$
2. $\sigma[z : x.a \cup z.a = y.a](Z)$

Expression 1 can be rewritten to allow for splitting off a local dyadic constant of maximum size: $\sigma[z : x.a - y.a = z.a](Z)$. We cannot apply such a rule to expression 2.

Nested collects can be removed independent from the nesting level, but quantification can be removed only by joining adjacent operands, i.e., in which the one operand occurs nested immediately within the other (or within a conjunction of predicates[4]). Because we prefer flat joins above nestjoins, before starting the top-down unnesting procedure, we first try to introduce local flat joins between base table expressions as much as possible. It is required that the local join expressions are closed, to be able to move them to the top level. Given our example query from the previous section:

$$\alpha[x : \sigma[y : p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y)](X)$$

we note an existential quantifier nested within a selection. Therefore, we split the multi-variable quantifier scope as needed, and first introduce a local join between Y and Z:

Rewriting example 7.7

$$\begin{aligned}
&\alpha[x : \sigma[y : p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y)](X) \\
&\quad \equiv \quad \alpha[x : \sigma[y : p(x,y) \wedge \exists z \in \sigma[z : r(y,z)](Z) \bullet q(x,z)](Y)](X) \ (\textbf{split}) \\
&\quad \equiv \quad \alpha[x : \sigma[v : p(x,v_Y) \wedge q(x,v_Z)](Y \underset{y,z:r(y,z)}{\bowtie} Z)](X) \ (\textbf{unnest})
\end{aligned}$$

Heuristic rule 7.6 Before starting the top-down unnesting procedure, we introduce closed flat join expressions at local levels, if possible.

7.6.2 Rewrite strategy

We now summarize the foregoing in the following rewrite strategy:

- Whenever possible:

[4] For the introduction of a join, for example, we have the following extended rule:

$$\begin{aligned}
&\Gamma[x : f(x) \mid t(x) \wedge \exists y \in \sigma[y : p(x,y)](Y) \bullet q(x,y))](X) \equiv \\
&\quad \Gamma[v : f(v_X) \mid t(v_X) \wedge q(v_X, v_Y)](X \underset{x,y:p(x,y)}{\bowtie} Y)
\end{aligned}$$

  - Name global constants.
  - Remove local constants.
  - Push through monadic subexpressions, in order of nesting level.
  - Push through operations to joins and join operands.

- Whenever a parameter expression contains nested iterators:

  1. Introduce closed flat join operations at nested levels, whenever possible.
  2. Unnest, top down. First join tables, then tables with set-valued attributes, and finally set-valued attributes. After each local join, try global unnest again. To enable the introduction of real joins instead of Cartesian products, split parameter expressions as needed.

The above rewrite strategy is a starting point, i.e., in many cases it may be better to modify the strategy in one way or another.

7.6.3 Optimal join order

Traditionally, determination of the join order is done cost-based. We stress that in a complex object model with set-valued attributes that supports an algebra that contains a rich variety of (join) operators such a cost-based optimization may be hard to do. First, the proper algebraic equivalence rules have to be found. Second, the process of algebraic rewriting is very difficult to control. By means of heuristic translation rules we try to achieve an algebraic expression that has performance that is not too bad right from the start.

The search space for an optimal join order is restricted. First, links between tables may be missing—we do not want to introduce Cartesian products. Second, iterator operands may be tables, as well as set-valued attributes, which cannot be moved to the top level. Whenever an iteration with an attribute operand is cast between iterators with table operands, the result may contain nested join operators. Also, the links between tables may be of a different nature, i.e., predicate and/or function, and the creation of predicate links between operands may be preferable to the establishment of function links. We want to know what constitutes an optimal join order, from the viewpoint of logical optimization, i.e. not taking into consideration physical database characteristics.

At the moment, we do not have the definitive answer to the question of optimal join order. (Sub)expressions can be classified according to the criteria of (1) the specific operators that occur in it (2) the fact whether operators occur nested, or at the top level, and (3) the kind of operator operands, i.e. attribute or base table (expression). It seems reasonable to assume that:

- Set operators are better than iterator expressions.
- Partial joins (semi- and antijoin) are better than regular joins, that in turn are better than Cartesian products. A flat join is better than a nestjoin.[5] Also, predicate links

[5] This is questionable: a nestjoin does not suffer from data replication, but is not commutative like the regular join.

are better than function links, and atomic links are better than complex ones.

- Top-level operations are better than nested ones.
- Table operations are better than operations on set-valued attributes.

But is a nested semijoin better or worse than a top-level join? The criteria listed above are not orthogonal criteria, and have to be refined.

We note that it is not always easy to judge expressions on relative performance without the use of a more or less detailed cost model. For example, returning to our latter leading example, assume that operand Y is not a base table, but the set-valued attribute c of table X. We simplify our example:

$$\sigma[x : \exists y \in x.c \bullet p_3(x,y) \wedge \exists z \in Z \bullet p_5(x,z) \wedge p_6(y,z)](X)$$

The expression contains predicates that mutually link all three iterator operands. Existential quantifiers may be exchanged to move the attribute iterator inside, but this is not a generally valid strategy. We may choose to join X with Z at the top level, and then to execute a nested quantification:

$$\sigma[v : \exists y \in x.c \bullet p_3(v_X, y) \wedge p_6(y, v_Z)](X \underset{x,z:p_5(x,z)}{\bowtie} Z)$$

Also, it is possible to join table Z with attribute c at a local level:

$$\sigma[x : \exists v \in (x.c \underset{y,z:p_6(y,z)}{\bowtie} Z) \bullet p_3(x, v_Y) \wedge p_5(x, v_Z)](X)$$

The former is likely to be better than the second, because in the second tuples of Z are replicated for each matching tuple in attribute c, for each (set) value c. However, the performance of both expressions depends on join methods used, join selectivities, the respective cardinalities of join operands, etc. For illustration purposes, in Appendix C, we present a computation of costs for both expressions, assuming that both joins are executed by means of a nested-loop strategy.

7.7 Extensions and optimizations

Extensions to the framework given in this paper may concern the following issues:

1. Nested set operators with table and/or attribute operands.
2. Nested set comparators with table and/or attribute operands

We have required that join predicates consist of atomic attribute comparisons only. However, in the past few years work has been done on the efficient implementation of set comparison operators in join operators. For example, indexes can be used, either on set elements themselves, or on sets as a whole [HePf94]. Another technique proposed is the use of signature files [IsKO93]. To allow for set comparison operators in join predicates is a simple extension of our general framework. Also, as pointed out in [SABB94], the

rewriting of set comparison operators into quantifier expressions and/or exchanging the order of quantifiers may enable the use of the quantifier unnest rules.

Also, work has been done for example on the efficient implementation of a local join between a set-valued attribute and a base table expression [DeLa92]. As far as we know, no work has been done yet on the efficient implementation of set operators. For example, the expression: $\alpha[x : x.c - Y](X)$ can be evaluated by means of nested-loop execution, using a standard implementation method for the set difference operator, the set difference can be rewritten into negated existential quantification, or some new implementation method can be devised.

The unnesting procedure described above and the results achieved can be extended and optimized further in many ways. For example, we have not considered how to deal with quantification in collect functions, i.e. in the SELECT-clause. Orthogonality of the language allows to write for example:

$$\alpha[x : \langle a = x.a, b = \exists y \in Y \bullet x.b = y.b\rangle](X)$$

7.8 Related work

Relational SQL The work presented here follows that done on the translation and optimization of relational SQL. Basically, there are two ways of optimizing SQL: (1) the rewriting of SQL expressions themselves [Kim82], and (2) translation of SQL into relational algebra, followed by algebraic rewriting. Relational algebra is a pure set-oriented language, which does not allow for nested expressions. The transformation of nested SQL queries into SQL join queries is completely in analogy with the transformation of SQL into relational algebra, either directly [CeGo85], or by using relational calculus as an intermediate language [Codd72]. The underlying idea of all work mentioned above, and ours, is that nested-loop expressions should be transformed into set expressions that do not contain nested operators. However, the important differences are with the work presented here is that (1) we have to deal with nesting in the SELECT-clause, which is not allowed in relational SQL, and (2) the presence of set-valued attributes, which is non-relational as well. SQL-like languages proposed for complex object models usually are orthogonal languages; the problem of choosing an algebra for, and of translation into the algebra of such a language is much more complicated.

Cluet & Moerkotte The work presented in [ClMo93] and [ClMo94] has much in common with ours. In [ClMo94] a binary grouping operator is defined that differs slightly from the nestjoin in the sense that the join function is applied to the *set* of matching right-hand operand tuples, not to the elements themselves. This has as a consequence that nested expressions with so-called projection dependency, that involve a free variable occurring in a SELECT-clause, cannot be unnested. Iteration over set-valued attributes that are not extents is not considered, but aggregate functions are.

Generally speaking, the algebraic operators used and the building blocks of optimization, the (algebraic) equivalence rules are similar in both approaches. However, in this paper, we consider queries with deeper nesting levels, incorporating nested iterators with set-valued attribute operands.

Own work In [StAB94], we showed that in complex object models, the regular, i.e. flat relational join operator does not suffice for the unnesting of nested queries. To solve this problem, we introduced the nestjoin. In [SABB94], we presented a general strategy for unnesting. We proposed to use relational (join) operators whenever possible, and to use the nestjoin otherwise. In this paper, the general strategy outlined in [SABB94] is made mode concrete. Starting form a basic set of transformation rules, we have discussed what can be done to achieve efficient algebraic expressions. We use heuristic rules to determine an initial join order, and to push through predicates and functions.

7.9 Conclusion

In this paper, we have presented a heuristics-based, extensible algorithm for the translation of nested OSQL queries into efficient join expressions. Queries that involve nested quantifier expressions are translated using relational algebra operators. For the translation of nested queries that cannot be translated into flat join queries, the nestjoin operator is used, which is a combination of join and grouping. During translation, predicates and functions are pushed through as far as possible. We have presented a general framework that can can easily be extended. Additional rules can be found, involving either better transformations or additional algebraic operators.

The main problem in the translation of nested OSQL queries is to find a good unnesting strategy. We have proposed a top-down unnesting strategy that minimizes the nesting level in nestjoin expressions. Our goal is to rewrite nested expressions into algebraic expressions such that expensive operators (e.g. Cartesian products), nested base table occurrences, and nested joins are avoided as much as possible. How to achieve this goal in the best possible way is topic of further research. A (heuristic) cost model is needed to guide the transformation of nested queries; such a cost model is much more complex than the heuristic model used in logical optimization in the relational context, that merely prescribes to push through selections and projections.

References

[BaBZ91] Balsters, H., R.A. de By, and R. Zicari, "Typed Sets as a Basis for Object-Oriented Database Schemas," *Proceedings ECOOP*, Kaiserslautern, 1993.

[BlMG93] Blakeley, J.A., W.J. McKenna, and G. Graefe, "Experiences Building the Open OODB Optimizer," *Proceedings ACM SIGMOD*, 1993, pp. 287–296.

[Bry89] Bry, F., "Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited," *Proceedings ACM SIGMOD*, Portland, Oregon, June 1989, pp. 193–204.

[Catt93] R.G.G. Cattell, ed., *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[CeGo85] Ceri, S. and G. Gottlob, "Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries," *IEEE Transactions on Software Engineering*, 11(4), April 1985, pp. 324–345.

[ClMo93] Cluet, S. and G. Moerkotte, "Nested Queries in Object Bases," *Proceedings Fourth International Workshop on Database Programming languages*, New York, Sept. 1993.

[ClMo94] Cluet, S. and G. Moerkotte, "Classification and Optimization of Nested Queries in Object Bases," manuscript, 1994.

[Codd72] Codd, E.F., "Relational Completeness of Data Base Sublanguages," in: *Data Base Systems*, ed. R. Rustin, Prentice Hall, 1972, pp. 65–98.

[DeLa92] Desphande, V. and P.-Å. Larson, "The Design and Implementation of a Parallel Join Algorithm for Nested Relations on Shared-Memory Multiprocessors," *Proceedings IEEE Conference on Data Engineering*, Tempe, Arizona, February 1992, pp. 68–77.

[ElNa89] Elmasri, R. and S.B. Navathe, **Fundamentals of Database Systems**, Benjamin/Cummings Publishing Company Inc., 1989.

[Grae93] Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2), June 1993, pp. 73–170.

[HePf94] Hellerstein, J.M. and A. Pfeffer, "The RD-Tree: An Index Structure for Sets," Technical Report #1252, University at Wisconsin at Madison, October 1994.

[IsKO93] Ishikawa, Y., H. Kitagawa, and N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs," *Proceedings ACM SIGMOD*, 1993, pp. 247–256.

[JaKo84] Jarke, M. and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, 16(2), June 1984, pp. 111–152.

[KeMo93] Kemper, A. and G. Moerkotte, "Query Optimization in Object Bases: Exploiting Relational Techniques," in: *Query Processing for Advanced Database Systems*, eds. J.-C. Freytag, D. Maier, and G. Vossen, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Kim82] Kim, W., "On Optimizing an SQL-like Nested Query," *ACM TODS*, 7(3), September 1982, pp. 443–469.

[Naka90] Nakano, R., "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS*, 15(4), December 1990, pp. 518–557.

[PiAn86] Pistor, P. and F. Andersen, "Designing a Generalized NF^2 Model with an SQL-Type Language Interface," *Proceedings VLDB*, Kyoto, August 1986, pp. 278–285.

[RoKS88] Roth, M.A., H.F. Korth, and A. Silberschatz, "Extended Algebra and Calculus for Nested Relational Databases," *ACM TODS*, 13(4), December 1988, pp. 389–417.

[ScSc86] Schek, H.-J. and M.H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems*, 11(2), 1986, pp. 137–147.

[ShCa90] Shekita, E.J. and M.J. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proceedings ACM SIGMOD*, Atlantic City, May 1990, pp. 300–311.

[StAB94] Steenhagen, H.J., P.M.G. Apers, and H.M. Blanken, "Optimization of Nested Queries in a Complex Object Model," *Proceedings EDBT*, Cambridge, March 1994, pp. 337–350.

[SABB94] Steenhagen, H.J., P.M.G. Apers, H.M. Blanken, and R.A. de By, "From Nested-Loop to Join Queries in OODB," *Proceedings VLDB*, Santiago de Chile, September 1994.

Chapter 8

Transformation techniques

In this chapter, we first present an overview of transformation techniques that are used in the transformation of nested expressions in Section 8.1. We identify four general transformation principles that can be understood as the lessons learned from from the previous chapters: they address some overall principles of efficiency gain that can be formally quantified. In Section 8.2, we briefly summarize the transformation strategy presented in the previous chapter, and in Section 8.3, we discuss some work done on the efficient implementation of complex parameter expressions.

8.1 Transformation techniques

In this section, we present four general transformation rules, and we also show that the general rules can be refined for specific types of selection and quantifier expressions. We give some example applications, and discover that common algebraic operators correspond to, i.e., implement, specific expressions that are obtained by application of the general rules. The general techniques are used for the transformation of calculus-like expressions; we describe also some other transformation techniques that are mainly used in an algebraic context.

8.1.1 Transformation principles

The guiding principle in the transformation of nested queries is **localization**. Localization is defined as the attempt to perform the necessary operations, *and no others*, to only the data needed, *and no other*. Localization means the avoidance of redundant data and superfluous computations. The basic transformation techniques are:

Extension Materialisation of partial results.

Generalized projection The selection and precomputation of only those data that are needed in subsequent computations.

Decomposition An optimization of projection.

Naming The use of a local definition facility.

The above list is a hierarchical list—from top to bottom, each technique is a special case of the previous one.

8.1.2 Basic transformation rules

As we will see in Section 8.1.4, **extension** is the principle that underlies the transformation of nested queries into join queries. Extension is applied to remove monadic subexpressions from nested loops, in case the original operand values are still needed in subsequent computations (for example with selections). The operand tuple values are concatenated with attribute values that materialize the precomputed, partial results. The rule for extension is:

Transformation 8.1 Extension Let X be a table, and let label a not occur in the schema of X, then:

$$\Gamma[x : e(x, g(x))](X) \equiv \Gamma[v : e(v_X, x.a)](\alpha[x : x \ \mathbf{except}\ (a = g(x))](X))$$

The parameter expression e (which stands for some expression $f \mid p$) of the left-hand side collect operator contains a subexpression $g(x)$, which may be arbitrary. Variable x may occur free in e outside of g. Expression g may occur on arbitrary deep nesting levels, meaning that g is evaluated within arbitrary many loops. In the transformation of the left- into the right-hand side g is brought to the top level. In the right-hand side expression, g is evaluated within only one loop: only for each $x \in X$. The results are stored as a new attribute of X. Recall that it is not allowed to introduce free variables, so when applied at top level, g may have no free variables other than x. Note that the above definition of extension can be modified such that it allows for the precomputation of multiple subexpressions, resulting in multiple extensions—we have not done so for reasons of simplicity.

An example application of extension is the following (we remark once more that σ and α are special cases of Γ):

$$\begin{array}{l}\sigma[x : x.c \subseteq \sigma[y : y.c \subseteq \sigma[z : x.a = z.a](Z)](Y)](X) \equiv \\ \quad \Gamma[v : v_X \mid x.c \subseteq \sigma[y : y.c \subseteq v.zs](Y)](\\ \quad\quad \alpha[x : x \ \mathbf{except}\ (zs = \sigma[z : x.a = z.a](Z))](X))\end{array}$$

In the left-hand side expression, the subquery $g(x) = \sigma[z : x.a = z.a](Z)$ is evaluated for each value $x \in X$ and each value $y \in Y$. In the right-hand side expression, the subquery is evaluated only for each value $x \in X$.

Generalized projection involves the precomputation of *all* subexpressions in which a certain iterator variable occurs free. Only these results are preserved, no other data is passed through. The rule for projection is:

Transformation 8.2 Generalized projection Let x not occur free in e outside of instances of the form $g_i(x)$, then:

$$\Gamma[x : e(g_1(x), \ldots, g_n(x))](X) \equiv \\ \Gamma[x : e(x.a_1, \ldots, x.a_n)](\alpha[x : \langle a_1 = g_1(x), \ldots a_n = g_n(x)\rangle](X))$$

Projection is a special case of extension, in the sense that it does not preserve the original operand tuples. An example of generalized projection is the following:

$$\alpha[x : x.c - \alpha[y : x.b + y.b](Y)](X) \equiv \\ \alpha[x : x.a_1 - \alpha[y : x.a_2 + y.b](Y)](\alpha[x : \langle a_1 = x.c, a_2 = x.b\rangle](X))$$

Decomposition can be applied whenever it is possible to identify a *single* subexpression in which some iterator variable occurs free. Decomposition is a special case of projection: the explicit tuple construction present in projection can be avoided.

Transformation 8.3 Decomposition Let x not occur free in e outside of g, then:

$$\Gamma[x : e(g(x))](X) \equiv \Gamma[x : e(x)](\alpha[x : g(x)](X))$$

An example of decomposition is:

$$\alpha[x : \alpha[y : x.a + y.a](Y)](X) \equiv \alpha[x : \alpha[y : x + y.a](Y)](\alpha[x : x.a](X))$$

The **local definition** rule concerns subexpressions that contain no free variables at all (global constants):

Transformation 8.4 Local definition Let $FV(c) = \emptyset$, then:

$$\Gamma[x : e(c)](X) \equiv \Gamma[x : e(C)](X) \textbf{ with } C = eval(c)$$

Rules for existential quantification are analogous. The preferred order of rule application is from bottom to top. The above transformations are applied to remove local and global constants $g(x)$ from parameter expressions, when it is profitable to evaluate them separately, i.e. whenever a transformation leads to a reduction of:

- the number of times $g(x)$ is evaluated, or of
- data volume.

A reduction of data volume can be achieved by means of explicit restrictions or projections, or implicitly, at the physical level, for example when scanning an index instead of a relation. A projection reduces tuple size, but may as well reduce cardinality.

8.1.3 Refinements for predicates

For some types of expressions, the above rules can be further refined. Whenever the scope of some quantifier or a selection predicate is a conjunction, we can apply decomposition instead of extension:

Rule 8.1 Conjunction: decomposition instead of extension

1. $\sigma[x : p(x) \wedge q(x)](X) \equiv \sigma[x : q(x)](\sigma[x : p(x)](X))$
2. $\exists x \in X \bullet p(x) \wedge q(x) \equiv \exists x \in \sigma[x : p(x)](X) \bullet q(x)$

Consider the above rule for selection. Even though x occurs free in q, it is allowed to perform a transformation that is analogous to decomposition, instead of applying extension, which would be done as follows:

$$\sigma[x : p(x) \wedge q(x)](X) \equiv \Gamma[v : v_X \mid b \wedge q(v_X)](\alpha[x : x \ \mathbf{except}\ (b = p(x))](X))$$

Now consider the expression:

$$\sigma[x : \exists y \in Y \bullet p(x) \wedge q(x, y)](X)$$

Because x occurs free in q, we are required to use extension in case we want to remove the local constant $p(x)$ from the quantifier scope:

$$\Gamma[v : v_X \mid \exists y \in Y \bullet v.b \wedge q(v_X, y)](\alpha[x : x \ \mathbf{except}\ (b = p(x))](X))$$

However, the above transformation is unnecessary complex, because $p(x)$ can be removed from the quantifier scope directly, by means of the well-known rules for descoping (see Section 4.4.4):

Rule 8.2 Descoping instead of extension Let x not be free in p, then:

1. $\exists x \in X \bullet p \wedge q(x) \equiv p \wedge \exists x \in X \bullet q(x)$
2. $\exists x \in X \bullet p \vee q(x) \equiv p \vee \exists x \in X \bullet q(x)$

In this case, we do not need even a second iterator. Note that descoping, or, in general, transformation into MNF, implements the principle of localization for quantifier expressions.

8.1.4 Applications

In this section we show that application of the transformation rules as described in the previous section may result into expressions that closely correspond to well-known algebraic operators. The operators mark-, nest-, and regular join implement specific extension expressions. The extension consists of a Boolean value, a set, and plain attribute values, respectively.

Rewriting example 8.1 Markjoin

$$\begin{aligned}
&\sigma[x : E(x, \exists y \in Y \bullet q(x,y))](X) \\
&\quad\equiv\quad \Gamma[v : E(v_X, v.m)](\alpha[x : x \ \mathbf{except}\ (m = \exists y \in Y \bullet q(x,y))](X)) \\
&\quad\equiv\quad \Gamma[v : E(v_X, v.m)](X \underset{x,y:q(x,y);m}{-} Y))
\end{aligned}$$

The latter transformation step represents one possible definition of the markjoin operator:

$$X \underset{x,y:q(x,y);m}{-} Y \equiv \alpha[x : x \ \mathbf{except}\ (m = \exists y \in Y \bullet q(x,y))](X)$$

(It can be shown that the above definition is equivalent to the one given in Section 4.5.) We discover that a markjoin can be used to remove nested existential quantification regardless of the context it occurs in.

Rewriting example 8.2 Nestjoin

$$\begin{aligned}
&\Gamma[x : E(x, \Gamma[y : e(x,y)](Y))](X) \\
&\quad\equiv\quad \Gamma[v : E(v_X, v.ys)](\alpha[x : x \ \mathbf{except}\ (ys = \Gamma[y : e(x,y)](Y))](X)) \\
&\quad\equiv\quad \Gamma[v : E(v_X, v.ys)](X \underset{x,y:e(x,y);ys}{\Delta} Y)
\end{aligned}$$

Note that the nestjoin can be defined as follows:

$$X \underset{x,y:e(x,y);ys}{\Delta} Y \equiv \alpha[x : x \ \mathbf{except}\ (ys = \Gamma[y : e(x,y)](Y))](X)$$

Rewriting example 8.3 Join

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge q(x,y)](X) \\
&\quad\equiv\quad \sigma[x : \exists y \in \sigma[y : p(x,y)](Y) \bullet q(x,y)](X) \\
&\quad\equiv\quad \Gamma[v : v_X \mid \exists y \in v.ys \bullet q(v_X, y)](X \underset{x,y:\mathtt{id}|p(x,y);ys}{\Delta} Y) \\
&\quad\equiv\quad \Gamma[v : v_X \mid q(v_X, v_Y)](\mu_{ys}(X \underset{x,y:\mathtt{id}|p(x,y);ys}{\Delta} Y)) \\
&\quad\equiv\quad \Gamma[v : v_X \mid q(v_X, v_Y)](X \underset{x,y:p(x,y)}{\bowtie} Y)
\end{aligned}$$

An existential quantifier that ranges over a set-valued attribute can be removed by the introduction of an unnest operator, provided the attribute is not needed afterwards:

$$\sigma[x : \exists y \in x.c \bullet p](X) \equiv \sigma[x : p](\mu_c(X))$$

We used the following definition of the join operator:

$$X \underset{x,y:q(x,y)}{\bowtie} Y \equiv \mu_{ys}(X \underset{x,y:\mathtt{id}|q(x,y);ys}{\Delta} Y)$$

The above rewriting may seem rather artificial, but, nonetheless, it is obvious that the regular join is a special case of the extension principle. Note that the semi- and antijoin also

can be looked upon as special cases of extension: the extension is empty. We have for example:

$$X \underset{x,y:p(x,y)}{\ltimes} Y \equiv \sigma[x : \exists y \in x.ys \bullet true](X \underset{x,y:\mathtt{id}|p(x,y);ys}{\Delta} Y)$$

Relational projection is a special case of generalized projection as described above:

Rewriting example 8.4 Relational projection

$$\begin{aligned}
&\Gamma[x : x.a \mid \exists y \in Y \bullet x.b = y.b](X) \\
&\quad \equiv \quad \Gamma[x : x.a \mid \exists y \in Y \bullet x.b = y.b](\alpha[x : \langle a = x.a, b = x.b\rangle](X)) \\
&\quad \equiv \quad \Gamma[x : x.a \mid \exists y \in Y \bullet x.b = y.b](\pi_{a,b}(X))
\end{aligned}$$

An example of decomposition is the following:

$$\alpha[x : \alpha[y : x.a + y.b](Y)](X) \equiv \alpha[x : \alpha[y : x + y.b](Y)](\alpha[x : x.a](X))$$

Instead of evaluating the expression $x.a$ within the inner loop, it can be precomputed. Because variable x does not occur elsewhere, tuple construction can be avoided. We can define a simplified projection operator π^s as follows:

$$\pi^s_a(X) = \alpha[x : x.a](X)$$

One efficient implementation method for π^s is index-scan, which avoids access to duplicate values and to redundant data. The same can be done with subexpression $y.b$, and a full transformation results in:

$$\alpha[v : v.ys](\pi^s_a(X) \underset{x,y:x+y|true;ys}{\Delta} \pi^s_b(Y))$$

The concatenation implicit in the nestjoin is superfluous here; perhaps we need a new physical operator (and maybe a new logical operator too) that avoids the concatenation.

Naming can be applied to constants and common subexpressions. In the following example, naming is applied to two almost-common subexpressions: the complementary selections $\sigma[x : p(x)](X)$ and $\sigma[x : \neg p(x)](X)$. As we have seen in Chapter 4, in [KMPS94], the two-stream selection operator (σ^2) was introduced for the efficient implementation of such a pair of selections:

Rewriting example 8.5 Two-stream selection

$$\begin{aligned}
&\sigma[x : p(x) \vee q(x)](X) \\
&\quad \equiv \quad \sigma[x : p(x) \vee (\neg p(x) \wedge q(x))](X) \\
&\quad \equiv \quad \sigma[x : p(x)](X) + \sigma[x : \neg p(x) \wedge q(x)](X) \\
&\quad \equiv \quad \sigma[x : p(x)](X) + \sigma[x : q(x)](\sigma[x : \neg p(x)](X)) \\
&\quad \equiv \quad T + \sigma[x : q(x)](F) \ \mathbf{with}\ T = \sigma[x : p(x)](X), F = \sigma[x : \neg p(x)](X) \\
&\quad \equiv \quad T + \sigma[x : q(x)](F) \ \mathbf{with}\ (T, F) = \sigma^2[x : p(x)](X)
\end{aligned}$$

Recall that the operator $+$ stands for set union without duplicate removal.

Our conclusion is that we have found a method for deciding which algebraic operators to include in our algebra: for specific combinations of language constructs that occur often as the result of our general transformation rules (that are generally beneficial if applied as specified), we can define an operator, and search for efficient implementations. The underlying transformation principle is that of definition; this and some other transformation techniques we describe in the next section.

8.1.5 Other transformation techniques

First, we have the technique of definition:

Transformation 8.5 Definition is the transformation of an operator sequence into one logical algebra operator.

The operator sequence concerned is supposed to occur frequently, and the performance of its corresponding algebraic operator is assumed to be at least equal to, but preferably better than the original sequence. In the previous section, we discussed definition as a transformation of nested expressions into algebraic ones. Definition can also be applied in a pure algebraic context, e.g., when defining the join as a selection on the Cartesian product.

The principle of definition has its inverse: that of expansion. For example, set comparison operators can be rewritten into quantifier expressions. Also, a set operator like difference can be rewritten into an iterator expression.

Transformation 8.6 Expansion is the transformation of an operator into an equivalent operator sequence that does not contain the operator itself.

Expansion is useful whenever a specific operator is not implemented (cf. relational algebra with its limited set of operators). Expansion may also be useful whenever it allows for further rewriting that positively influences performance. For example, expansion of set comparison operators may enable unnesting (e.g., see Section 6.5.2).

One other technique is that of reordering the operator tree. Reordering is mainly applied in algebraic transformations, e.g. when pushing through projections or selections, but it can also be used in a calculus context, for example when exchanging quantifiers.

Transformation 8.7 Reordering concerns the transformation of operator trees while preserving the specific collection of operators present.

In Section 4.6.5, we have discussed distribution of quantification:

$$\exists x \in X \bullet p \vee q \equiv (\exists x \in X \bullet p) \vee (\exists x \in X \bullet q)$$

This rule is of a different nature than the rules discussed until now; it is applied for its assumed beneficial effect upon further rewriting, but it does not necessarily improve efficiency itself.

Transformation 8.8 Heuristic transformation rules are rules that improve the outcome of some standard rewriting process.

For example, in Section 4.6.2, we decided to treat a negated existential quantification with a conjunctive scope by means of set difference instead of division.

8.2 Transformation strategy

In the previous section, we described the principal transformation techniques, as well as some refinements to the general rules for quantifier expressions and selections. Furthermore, we have shown that (more or less) common algebraic operators implement specific types of expressions that are obtained by application of the general rules.

The transformation strategy, which for a large part determines the efficiency of the results, prescribes how, where, and when transformation techniques are applied. In the previous chapter, we described a top-down rewrite strategy. Flat joins are preferred over nestjoins, and Cartesian products are introduced only after everything else has failed. Throughout, global constants are named. In addition, expressions that contain the same set of free variables are evaluated in order of nesting level.

Whenever joins are about to be introduced:

- we push through monadic subexpressions to the join operands, and
- we gather dyadic subexpressions that are considered suitable as join parameter expressions, as many as possible.

We decided that in join parameter expressions both join variables must occur free, that predicates must consist of atomic terms only, and that nestjoin functions may not contain base table occurrences. These restrictions can be relaxed whenever profitable: in the next section we discuss some work that concerns the efficient implementation of joins with complex join predicates containing set comparison operators. In the previous chapter, we assumed that:

- Set operators are better than iterator expressions.
- Partial joins (semi- and antijoin) are better than regular joins, which in turn are better than Cartesian products. A flat join is better than a nestjoin. Also, predicate links are better than function links, and atomic links are better than complex ones.
- Top-level operations are better than nested ones.
- Table operations are better than operations on set-valued attributes.

and we observed that the above heuristics need refinement. Our conjecture is that to ensure a proper translation from calculus into algebra, also physical properties should be taken into consideration. We give an example. Consider the expression:

$$\alpha[x : \sigma[y : \exists z \in x.c \bullet y.a = z](Y)](X)$$

One possible transformation is given by:

$$\alpha[x : Y \underset{y,z:y.a=z}{\ltimes} x.c](X)$$

A nested semijoin is executed for each $x \in X$. Another possible transformation results in a top-level nestjoin operator:

$$\alpha[x : x.ys](X \underset{x,y:y|y.a \in x.c;ys}{\Delta} Y)$$

(Recall that the concatenation and the projection can be avoided.) The choice is between a nested semijoin with an atomic predicate and a top-level nestjoin with a complex predicate, i.e., a predicate involving a set comparison operator. Logical properties such as table cardinality and tuple width cannot help us to decide which of the two algebraic expressions is to be preferred. However, assume that an index exists on attribute a of table Y, then the obvious way of processing the query is to scan table X, scan its set-valued attribute c, and retrieve the corresponding Y-values by means of the index. Neither one of the algebraic expressions forms an immediate representation of this query processing strategy, if we assume that in partial join implementations the left-hand operand is taken as the outer loop operand. The following expression comes closer:

$$\alpha[x : \pi_Y(Y \underset{y,z:y.a=z}{\bowtie} x.c)](X)$$

Because the regular join is commutative, attribute values $x.c$ can be taken as outer loop operand in an index nested-loop join implementation.

8.3 Complex parameter expressions

In this section, we pay attention to some issues concerning the implementation of the logical algebra ADL. As stated, the logical algebra is supposed to bridge the gap between the user language and the system-specific physical algebra, which is assumed to provide clever access methods to speed up the evaluation of logical algebra operators (or parts or sequences thereof). The relationship between the logical and the physical algebra is strong. The physical algebra implements the logical algebra in such a way that the performance at least improves the performance of naive query execution. However, there does not exist a standard physical (nor logical) algebra for object-oriented or even NF^2 database management systems; research into logical and physical algebras for next-generation database management systems is in development. Of course, we assume that the physical algebra offers at least the functionality of that of relational database management systems. Below, we discuss some of the access methods that allow for the efficient implementation of ADL operators that are more complex than their relational counterparts.

8.3.1 Predicates

Operators that allow for nesting of (arbitrary) expressions are the collect, select, and map and the universal and existential quantifier, but also the various join operators. Join operators are introduced during the transformation of OSQL expressions. In principle, in join predicates also nesting of expressions may occur, and join predicates may contain arbitrary operators, among which quantifier expressions and set comparison operators. Consider the following expression, in which attributes c and d are possibly complex:

$$X \underset{x,y:x.c=y.d}{\bowtie} Y$$

We have two questions:

1. Is it possible having to avoid to compare each left-hand operand tuple with every right-hand operand tuple, as is done in naive nested-loop query processing? In the relational model, the join is implemented using indices, sorting or hashing, thus establishing an improvement over nested-loop processing in the sense that only parts of operands need to be compared. The question is whether the same effect can be achieved with complex predicates.

2. Is it possible to evaluate the join predicate itself efficiently? In the relational model, join and selection predicates are simple, comparing atomic attribute values only. In complex object models, a predicate that tests for example the equality of attribute values, may involve extensive computations if the attribute values are complex. Attention must be paid to the efficient evaluation of predicates.

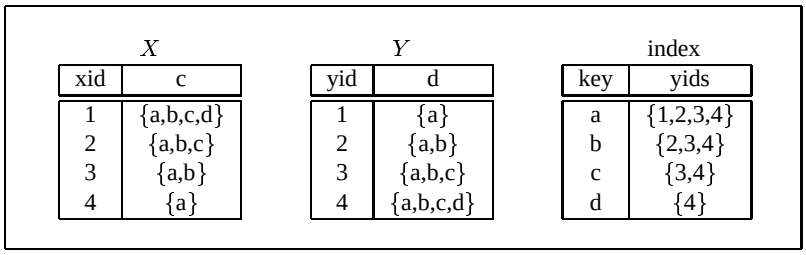
File organization techniques like indexing, sorting, and hashing, also called access methods, can be considered as preprocessing techniques aimed at reducing the search space (full table scan) in query evaluation. The search space is reduced by *partitioning* the input set(s), so that it is necessary to consider or compare only a subset of the data that is actually present. Relational query execution algorithms for selection and join are based on, and can efficiently handle, attribute comparisons involving atomic domains only. (In fact, the transformation of a nested query into a join query can be looked upon as the transformation of a complex selection predicate into an atomic join predicate.) Consider the operation:

$$X \underset{x,y:x.c\subseteq y.d}{\bowtie} Y$$

which is equivalent to the expression:

$$X \underset{x,y:\forall v\in x.c\bullet\exists w\in y.d\bullet v=w}{\bowtie} Y$$

In naive query evaluation, for each of the left-hand operand tuples, each of the right-hand operand tuples is retrieved, and the set comparison operation is evaluated. Naive evaluation of the subset predicate means that for each value in $x.c$ it is checked whether it occurs in $y.d$. The question is how to improve on naive query evaluation; below we describe some options.



X

xid	c
1	{a,b,c,d}
2	{a,b,c}
3	{a,b}
4	{a}

Y

yid	d
1	{a}
2	{a,b}
3	{a,b,c}
4	{a,b,c,d}

index

key	yids
a	{1,2,3,4}
b	{2,3,4}
c	{3,4}
d	{4}

Figure 8.1: Indexing set elements

Filters (Naive) predicate evaluation can be avoided altogether if there exists some natural or system-imposed property from which it can be deduced that the predicate certainly does not hold.

For example, the cardinality of sets may provide information concerning the outcome of set comparison operators. For instance, whenever the cardinality of some set s_1 is strictly smaller than that of s_2, it can never be the case that s_2 is a subset of s_1:

$$\#s_1 < \#s_2 \Rightarrow s_2 \not\subseteq s_1$$

Assuming the cardinality of sets is maintained by the system, comparison of set cardinalities may preclude the evaluation of the subset predicate. In our example, it is not necessary to actually test the subset relationship for those tuples $x \in X, y \in Y$ for which $\#x.c > \#y.d$. In the above approach, still every left-hand tuple is compared with every right-hand tuple, either comparing cardinalities only, or fully evaluating the subset predicate as well. Sorting table Y on the cardinality of the set-valued attributes (in descending order) can reduce the search space in that the inner loop can be broken of as soon as $\#x.c > \#y.d$.

The cardinality is a natural property of sets; other set abstractions that can be used are signature files [IsKO93] or hash values [KhFr88].

Element indexing If attributes c or d are sets of atomic type, an index on the elements of the attributes can be used, associating the identifiers (oids, key values, addresses, etc.) of operands tuples with elements of the set-valued attribute. In Figure 8.1, tables X and Y are illustrated, and a table showing the index on the elements of attribute d of Y.

Given some $x \in X$, the set of $y \in Y$ such that $x.c \subseteq y.d$ is retrieved as follows. For each value v in $x.c$, the corresponding index entry is retrieved; the intersection of the sets of $yids$ delivers the identifiers of tuples $y \in Y$ that are in the result. To retrieve the set of $y \in Y$ such that $x.c \supseteq y.d$, for some $x \in X$, for each value v in $x.c$, the corresponding index entry is retrieved; next the union of the sets of $yids$ is taken. For each of the identifiers in the resulting set, it is checked whether the tuple

actually satisfies the predicate. Of course, set equality can be computed by taking the intersection of the yid sets retrieved in sub- and superset testing.

Note that in our specific example (Figure 8.1), no advantage is gained by the index for the evaluation of the superset predicate; an index on attribute c of X would be more appropriate.

Set indexing Indexing and sorting are techniques based on a natural ordering of attribute domains, so they are easy to use with integer- and string-valued attributes. However, in complex object models, attributes may be set-valued, and designing index structures to support predicates involving set comparison operators is not straightforward. In [HePf94], an index structure called the RD-tree for the set-of-integer domain is proposed. The RD-tree is based on set inclusion: each index entry I provides access to (leaf or other index) nodes that provides access to sets that are subsets of I. The search key values used in the index are sets, instead of atomic set elements as in simple index structure described above. The RD-tree is suitable for the evaluation of superset predicates; for the evaluation of subset predicates the inverted RD-tree is used. In the inverted RD-tree, each index entry I provides access to nodes containing supersets of I.

Set-valued attributes have something in common with spatial data structures. A set of integers for example can be looked upon as a two-dimensional spatial object consisting of (unconnected) points situated parallel to the x-axis. Such a set can be "described" by means of a bounding set (in analogy with the well-known bounding box) that consists of the minimum and maximum values present in the data set. Therefore, it may be the case that techniques developed in the field of spatial query processing can be used for processing set comparison operators.

Concluding the above discussion, we believe that, although the work on access methods for the evaluation of set comparison operators is still in its infancy, in the future it will be possible to evaluate set comparison operators efficiently. We expect that new evaluation strategies will be discovered that allow for better performance than that of nested-loop execution (expressed by the corresponding equivalent quantifier expression). Therefore, set comparison operators are included in our language, and used as much as possible. For example, whenever a quantifier expression can be suitably rewritten in a set comparator, we do so.

8.3.2 Functions

Relational access methods can be used to improve the performance of algebraic operators such as selection and join, operators of which the parameter expression is a predicate. In OSQL, arbitrary expressions may occur not only in the **iff**-, but also in the **collect**-clause. For example:

$$\alpha[x : x \ \mathbf{except}\ x.c = \alpha[y : x.a + y.b](Y)](X)$$

In the above expression, the parameter expression of the outer map function is an arbitrary function, not a predicate expression. The question is whether standard, or other, new access methods can be used to improve performance.

In predicate evaluation, access methods are used to reduce the search space, i.e. to avoid unnecessary comparisons. In the evaluation of parameter expressions that are arbitrary functions, indices and other techniques like caching of results can be used to avoid duplicate computations. Again, consider the above expression. The expression is equivalent to the nestjoin expression:

$$X \underset{x,y:x.a+y.b|true;c}{\Delta} Y$$

Following a naive execution strategy, the expression $x.a + y.b$ is evaluated for each pair of tuples in X and Y. Assuming there is an index on attribute b of table Y, then, for each $x \in X$, it suffices to compute the value of the parameter expression for each key value occurring in the index, which is especially advantageous if the actual domain of attribute a is small. Another technique that can be used is memorizing, i.e. caching of the subquery result for the distinct values of attribute a of X. Before computation of the subquery for some $x \in X$, it is checked whether the result is already available.

Of course both techniques mentioned above can be used for predicate evaluation as well. Caching of results accounts for multiple occurrences of a attribute values, concerning the left join operand; the use of an index accounts for duplicate occurrences of b attribute values concerning the right join operand.

References

[HePf94] Hellerstein, J.M. and A. Pfeffer, "The RD-Tree: An Index Structure for Sets," Technical Report #1252, University at Wisconsin at Madison, October 1994.

[IsKO93] Ishikawa, Y., H. Kitagawa, and N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs," *Proceedings ACM SIGMOD*, 1993, pp. 247–256.

[KMPS94] Kemper, A., G. Moerkotte, K. Peithner, and M. Steinbrunn, "Optimizing Disjunctive Queries with Expensive Predicates," *Proceedings ACM SIGMOD*, Minneapolis, Minnesota, 1994, pp. 336–347.

[KhFr88] Khoshafian, S. and D. Frank, "Implementation Techniques for Object-Oriented Databases," *Proceedings 2nd International Workshop on Object-Oriented Database systems*, LNCS 334, Springer-Verlag, September 1988, pp. 60–79.

Chapter 9

Summary and conclusions

In this thesis, we have studied the translation of a prototype query language for advanced data models in the style of SQL into an algebra that supports complex objects; emphasis lies on the translation of nested queries.

9.1 Summary

In Chapter 2, an overview of query optimization issues in relational, nested relational, and object-oriented database systems is presented. In Chapter 3, we define OSQL, a prototype SQL language for data models that support complex objects, and its extension ADL. OSQL allows for nesting of expressions; ADL is a nested relational algebra extended with some non-standard join operators.

We start from the assumptions that (1) also in advanced database systems, the set-oriented query processing paradigm is appropriate, and (2) optimization aspects should play a role in the translation process too.

To support the second claim, we review the translation of relational calculus into relational algebra in Chapter 4. Instead of taking the Prenex Normal Form as a starting point for translation, we use the Miniscope Normal Form, in which quantifier scopes do not contain independent terms, i.e., atoms in which the quantified variable does not occur free. We describe a basic transformation algorithm, which gives better results than the algorithms found in the literature, and discuss some optional transformations. We investigate the effect of the optional transformations by translating example queries, and observe that, depending on the specific expression at hand, the effect of specific transformation rules may be positive or negative. We present heuristic rules to further improve the translation.

Next, in Chapter 5, we discuss some of the difficulties encountered in the translation of nested queries in complex object models. In OSQL, which is an orthogonal language, expressions may be nested arbitrarily. Not only in the WHERE-, but also in the SELECT-clause of the SELECT-FROM-WHERE query block other query blocks may occur, and the operands of these may be tables as well as set-valued attributes. OSQL queries can-

not always be translated into flat relational queries; often grouping is needed. To perform grouping before or after a join may give incorrect results due to the loss of dangling tuples. To solve this problem, we define an operator called the nestjoin, which combines join with grouping.

In Chapter 6, we present a framework for the translation of nested OSQL queries, and in Chapter 7, concrete transformation rules and an initial rewrite strategy are provided. We show that nested expressions can be translated into nested product expressions. These are, however, very inefficient because a nested product expression corresponds to the relational select-project-join (or, rather, select-project-product) expression, which is the most costly normal form. Like select-project-join expressions, nested product expressions need optimization, and we try to obtain better results by providing a better translation.

Nested iteration is removed as much as possible by rewriting nested queries into join queries, avoiding Cartesian products. Flat relational join operators are used whenever possible. Due to the presence of set-valued attributes, a complete unnesting in which no nested operators with set-valued operands (iterators or set operators) occur is not always possible. It is also not always desirable. Often the transformation result contains nested joins or nested set (comparison) operators, and also nested iterators with set-valued attribute operands.

In Chapter 8, we present an overview of transformation techniques used in this thesis. Transformation techniques can be divided into (1) generally beneficial techniques used for the transformation of nested expressions, and (2) techniques that play a role in algebraic rewriting mainly. The former class consists of rules that can be applied to ensure localization, which is the principle of doing only that what is strictly needed, and accessing only the data strictly required.

9.2 Conclusions

In the past, the phases of translation and logical optimization have been looked upon as separate phases in query processing, and little attention has been paid to optimization of other than select-project-join queries with conjunctive predicates. In Chapter 4, we reviewed the translation of relational calculus into relational algebra. We showed that the performance of relational query processing can be much improved by (1) extending relational algebra with non-standard operators like the semi- and antijoin and two-stream selection, and by (2) a careful design of the translation algorithm. Standard translation algorithms do not pay attention to performance issues: the result often is unnecessarily inefficient. In the phase of logical optimization, this inefficiency is supposed to be reduced. However, the damage cannot always be repaired because purely algebraic rewriting of expressions that involve arbitrary, complex, operators is a difficult process. Consequently, research into relational query optimization has often been restricted to optimization of project-select-join queries. Only recently, attention has been paid to optimization of queries that contain other operators like disjunction. Our conclusion is that the strict separation between the phases of translation and optimization should be readdressed; much can be gained by combining translation with optimization.

We provide a framework for the translation of nested OSQL queries. We define an extended relational algebra, which includes the nestjoin operator that is well-suited for the translation of nested queries in complex object models. We provide equivalence rules, and propose a rewrite strategy. The work presented here follows relational tradition by adhering to the set-oriented query processing paradigm. We believe that our work can serve as the basis for the implementation of query calculi for advanced (object-oriented) data models. However, many open problems remain.

In our opinion, the difficulty in the translation and optimization of (O)SQL queries lies in the number of equivalent logical expressions, and the lack of a suitable cost model. In the relational context, considering select-project-join queries only, the goal in optimization is to push through selections and projections, to transform products into joins, and to determine an optimal join order. Transformations are based on the localization principle, which is equally valid in a complex model. Optimization of select-project-join queries merely concerns a rearrangement of the operator tree. However complex this reordering process is from a cost-based point of view, from a syntactic point of view it is simple, because it is based on simple logical equivalences.

In a complex object algebra that supports a rich variety of operators, the number of equivalent logical algebra expressions is much larger than in the relational model. The rewriting of logical expressions is complicated by the fact that nesting of operators may occur, so the equivalence rules are more complex. Moreover, simple heuristics as used in the relational model do not always suffice for choosing between expression forms. For example, often the choice is between a top-level nestjoin operator, or a nested relational join operator. To decide between either one of them, logical properties like table cardinality and tuple width do not suffice. The goal is not only to perform the most restrictive operations first, but also to choose that specific logical algebra expression that can be implemented best, in other words, that makes best use of the opportunities offered by the physical algebra. Our conjecture is that physical properties should play a role in the translation of OSQL into the algebra.

9.3 Future work

We established a framework for the translation of (nested) OSQL queries into an algebra. Equivalence rules were defined, and a rewrite strategy was proposed. As indicated in the previous section, future work may concern (1) the implementation of expressions that contain nested set and join operators, (2) a (qualitative) cost model, and (3) a refinement of the proposed rewrite strategy. Given a proper set of equivalence rules, the rewrite strategy determines the outcome of the rewrite process. Rewriting must be guided by a cost model; a simple logical cost model as is used in the relational context (push through projections and selections) does not suffice. In this thesis, we have taken a top-down approach to the implementation of OSQL. Starting with nested expressions, we have tried to obtain set-orientation as much as possible. We believe that a bottom-up approach, starting with access methods and investigating the opportunities they offer for efficient implementation will give new insights into the problem of how to translate OSQL into the algebra.

We have not considered in our discussion specific object-oriented features such as class hierarchies, methods in imperative style, and object identity. Such features allow for additional optimization opportunities, and should be taken into consideration as well.

Appendix A

Rules and Proofs

A.1 Additional rules

In this section, we present some general rewrite rules that are used in various rewrite stages. We note that the set of rules listed is not complete. Additional rules for the transformation of relational algebra expressions can be found in [ElNa89] and [Kuijk91].
The logical rules given below are well-known:

Rule A.1 Some logical equivalence rules

1. $\nexists x \in X \bullet p \equiv \forall x \in X \bullet \neg p$
   $\nforall x \in X \bullet p \equiv \exists x \in X \bullet \neg p$
2. $\neg(p \wedge q) \equiv \neg p \vee \neg q$
   $\neg(p \vee q) \equiv \neg p \wedge \neg q$
   $\neg(\neg p) \equiv p$
3. $p \wedge (\neg p \vee q) \equiv p$
   $p \vee (p \wedge q) \equiv p$
4. $p \wedge (\neg p \vee q) \equiv p \wedge q$
   $p \vee (\neg p \wedge q) \equiv p \vee q$
5. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge q)$
   $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

Rules for set operators that we use are the following:

Rule A.2 Set operators

1. $X - (Y \cup Z) \equiv (X - Y) - Z$
2. $(X \cup Y) - Z \equiv (X - Z) \cup (Y - Z)$

The following rules deal with join operators with constant join predicates and empty right operands.

Rule A.3 Constants

1. $X \underset{p}{\ltimes} \emptyset \equiv \emptyset$
2. $X \underset{p}{\rhd} \emptyset \equiv X$
3. $X \underset{true}{\ltimes} Y \equiv \textbf{if } Y = \emptyset \textbf{ then } \emptyset \textbf{ else } X \equiv \textbf{if } Y \neq \emptyset \textbf{ then } X \textbf{ else } \emptyset$
4. $X \underset{false}{\ltimes} Y \equiv \emptyset$
5. $X \underset{true}{\rhd} Y \equiv \textbf{if } Y = \emptyset \textbf{ then } X \textbf{ else } \emptyset \equiv \textbf{if } Y \neq \emptyset \textbf{ then } \emptyset \textbf{ else } X$
6. $X \underset{false}{\rhd} Y \equiv X$
7. $X \underset{true;m}{-} Y \equiv X \times T_m$
8. $X \underset{false;m}{-} Y \equiv X \times F_m$

Rule A.4 Projection

1. **Commuting projection with selection**
   $\pi_A(\sigma[x : p](X)) \equiv \sigma[p](\pi_A(X))$, if $Attr(p) \subseteq A$
2. **Introduction of projection**
   $\pi_A(X) \equiv \pi_A(\pi_{A'}(X))$, if $A \subseteq A' \subseteq Sch(X)$

Rule A.5 Selection

$\sigma[x : \neg p](X) \equiv X - \sigma[x : p](X)$

Selection distributes over any set operator. Projection distributes over set union only, not over difference and intersection.

Rule A.6 Pushing through projections

1. Let $Attr(p_X)$ denote the set of attribute names of X referenced in predicate p, then:
   (a) $\pi_A(X \underset{p}{\ltimes} Y) \equiv \pi_A(X) \underset{p}{\ltimes} Y$, if $A \subseteq Attr(p_X)$
   (b) $\pi_A(X \underset{p}{\rhd} Y) \equiv \pi_A(X) \underset{p}{\rhd} Y$, if $A \subseteq Attr(p_X)$
2. $\pi_{AB}(X \times Y) \equiv \pi_A(X) \times \pi_B(Y)$, if $A \subseteq Sch(X), B \subseteq Sch(Y)$
3. $\pi_{AB}(X \underset{p}{\bowtie} Y) \equiv \pi_A(X) \underset{p}{\bowtie} \pi_B(Y)$, if $A \subseteq Sch(X), B \subseteq Sch(Y), Attr(p) \subseteq (A \cup B)$

Rule A.7 Pushing through selections

1. **Regular join**
   (a) $\sigma[v : p(v_X)](X \underset{x,y:q}{\bowtie} Y) \equiv \sigma[x : p(x)](X) \underset{x,y:q}{\bowtie} Y$
   (b) $\sigma[v : p(v_Y)](X \underset{x,y:q}{\bowtie} Y) \equiv X \underset{x,y:q}{\bowtie} \sigma[y : p(y)](Y)$
   (c) $\sigma[v : p(v_X, v_Y)](X \underset{x,y:q}{\bowtie} Y) \equiv X \underset{x,y:p(v_X,v_Y) \wedge q}{\bowtie} Y$

2. **Product** $X \times Y \equiv X \underset{x,y:true}{\bowtie} Y$
3. **Semijoin** $\sigma[x : p(x)](X \underset{x,y:q}{\ltimes} Y) \equiv \sigma[x : p(x)](X) \underset{x,y:q}{\ltimes} Y$
4. **Antijoin** $\sigma[x : p(x)](X \underset{x,y:p}{\rhd} Y) \equiv \sigma[p(x)](X) \underset{x,y:p}{\rhd} Y$

   Proof: $\sigma[x : p(x)](X \underset{x,y:q}{\rhd} Y)$

   $$\begin{aligned} &\equiv \sigma[x : p(x)](X - X \underset{x,y:q}{\ltimes} Y) \\ &\equiv \sigma[x : p(x)](X) - \sigma[x : p(x)](X \underset{x,y:q}{\ltimes} Y) \\ &\equiv \sigma[x : p(x)](X) - \sigma[x : p(x)](X) \underset{x,y:q}{\ltimes} Y \\ &\equiv \sigma[x : p(x)](X) \underset{x,y:q}{\rhd} Y \end{aligned}$$

Note that for partial joins right distribution does not apply, and that selection does not left distribute over the markjoin operator.

Rule A.8 Semi/antijoin

1. $X - (X \ltimes Y) \equiv X \rhd Y$
2. $X - (X \rhd Y) \equiv X \ltimes Y$
3. $X' - (X \ltimes Y) \equiv X' \rhd Y$, whenever $X' \subseteq X$

   Proof: Let X' denote some selection $\sigma[p](X)$, then:

   $$\begin{aligned} &X' - (X \ltimes Y) \\ &\equiv \sigma[p](X) - (X \ltimes Y) \\ &\equiv \sigma[p](X) - \sigma[p \vee \neg p](X \ltimes Y) \\ &\equiv \sigma[p](X) - (\sigma[p](X \ltimes Y) \cup \sigma[\neg p](X \ltimes Y)) \\ &\equiv (\sigma[p](X) - \sigma[\neg p](X \ltimes Y)) - \sigma[p](X \ltimes Y) \\ &\equiv \sigma[p](X) - \sigma[p](X \ltimes Y) \\ &\equiv \sigma[p](X) - \sigma[p](X) \ltimes Y \\ &\equiv \sigma[p](X) \rhd Y \\ &\equiv X' \rhd Y \end{aligned}$$

4. $X' - (X \rhd Y) \equiv X' \ltimes Y$, whenever $X' \subseteq X$

Rule A.9 Simplification of markjoin expressions (1)

1. $\pi_X(X \underset{p;m}{-} Y) \equiv X$
2. (a) $\pi_X(\sigma[x : x.m](X \underset{p;m}{-} Y)) \equiv X \underset{p}{\ltimes} Y$

(b) $\pi_X(\sigma[x : \neg x.m](X \underset{p;m}{-} Y)) \equiv X \underset{p}{\triangleright} Y$

3. $\pi_X(\sigma[x : p(x) \vee x.m](X \underset{x,y:q;m}{-} Y)) \equiv (X \underset{x,y:q}{\ltimes} Y) \cup \sigma[x : p(x)](X \underset{x,y:q}{\triangleright} Y)$
   if $m \notin Attr(p)$

Proof: $\pi_X(\sigma[x : p(x) \vee x.m](X \underset{x,y:q;m}{-} Y))$

$$\begin{aligned}
&\equiv \pi_X(\sigma[x : p(x) \vee x.m](((X \underset{x,y:q}{\ltimes} Y) \times T_m) \cup ((X \underset{x,y:q}{\triangleright} Y) \times F_m))) \\
&\equiv \pi_X(\sigma[x : p(x) \vee x.m]((X \underset{x,y:q}{\ltimes} Y) \times T_m) \cup \\
&\qquad \sigma[x : p(x) \vee x.m]((X \underset{x,y:q}{\triangleright} Y) \times F_m)) \\
&\equiv \pi_X(\sigma[x : p(x) \vee true]((X \underset{x,y:q}{\ltimes} Y) \times T_m) \cup \\
&\qquad \sigma[x : p(x) \vee false]((X \underset{x,y:q}{\triangleright} Y) \times F_m)) \\
&\equiv \pi_X(((X \underset{x,y:q}{\ltimes} Y) \times T_m) \cup \sigma[x : p(x)]((X \underset{x,y:q}{\triangleright} Y) \times F_m)) \\
&\equiv \pi_X((X \underset{x,y:q}{\ltimes} Y) \times T_m) \cup \pi_X(\sigma[x : p(x)]((X \underset{x,y:q}{\triangleright} Y) \times F_m)) \\
&\equiv (X \underset{x,y:q}{\ltimes} Y) \cup \sigma[x : p(x)](\pi_X((X \underset{x,y:q}{\triangleright} Y) \times F_m)) \\
&\equiv (X \underset{x,y:q}{\ltimes} Y) \cup \sigma[x : p(x)](X \underset{x,y:q}{\triangleright} Y)
\end{aligned}$$

Rule A.10 Simplification of markjoin expressions (2)

1. (a) $X \underset{x,y:p(x,y)}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z) \equiv X \underset{x,y:p(x,y)}{\ltimes} Y$, if $m \notin Attr(p)$

   (b) $X \underset{x,y:p(x,y)}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z) \equiv X \underset{x,y:p(x,y)}{\triangleright} Y$, if $m \notin Attr(p)$

2. (a) $X \underset{x,y:y.m}{\ltimes} (Y \underset{y,z:p;m}{-} Z) \equiv X \underset{x,y:true}{\ltimes} (Y \underset{y,z:p}{\ltimes} Z)$

   (b) $X \underset{x,y:y.m}{\triangleright} (Y \underset{y,z:p;m}{-} Z) \equiv X \underset{x,y:true}{\triangleright} (Y \underset{y,z:p}{\ltimes} Z)$

   (c) $X \underset{x,y:\neg y.m}{\ltimes} (Y \underset{y,z:p;m}{-} Z) \equiv X \underset{x,y:true}{\ltimes} (Y \underset{y,z:p}{\triangleright} Z)$

   (d) $X \underset{x,y:\neg y.m}{\triangleright} (Y \underset{y,z:p;m}{-} Z) \equiv X \underset{x,y:true}{\triangleright} (Y \underset{y,z:p}{\triangleright} Z)$

Proof: $X \underset{x,y:\neg y.m}{\triangleright} (Y \underset{y,z:p;m}{-} Z)$

$$\begin{aligned}
&\equiv \sigma[x : \nexists y \in (Y \underset{y,z:p;m}{-} Z) \bullet \neg y.m](X) \\
&\equiv \sigma[x : \nexists y \in \sigma[y : \neg y.m](Y \underset{y,z:p;m}{-} Z) \bullet true](X) \\
&\equiv \sigma[x : \nexists y \in (Y \underset{y,z:p}{\triangleright} Z) \bullet true](X) \\
&\equiv X \underset{x,y:true}{\triangleright} (Y \underset{y,z:p}{\triangleright} Z)
\end{aligned}$$

Rule A.11 Simplification of markjoin expressions (3)

1. $X \underset{x,y:p(x,y)\vee y.m}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z) \equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } (X \underset{x,y:p(x,y)}{\ltimes} Y)$

   Proof: $X \underset{x,y:p(x,y)\vee y.m}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z)$
   $\equiv X \underset{x,y:p(x,y)}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z) \cup X \underset{x,y:y.m}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z)$
   $\equiv (X \underset{x,y:p(x,y)}{\ltimes} Y) \cup X \underset{x,y:true}{\ltimes} (Y \underset{y,z:r(y,z)}{\ltimes} Z)$
   $\equiv (X \underset{x,y:p(x,y)}{\ltimes} Y) \cup \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } \emptyset$
   $\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } (X \underset{x,y:p(x,y)}{\ltimes} Y)$

2. $X \underset{x,y:p(x,y)\vee y.m}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z) \equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } \emptyset \textbf{ else } (X \underset{x,y:p(x,y)}{\triangleright} Y)$

   Proof: $X \underset{x,y:p(x,y)\vee y.m}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z)$
   $\equiv (X \underset{x,y:p(x,y)}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z)) \underset{x,y:y.m}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z)$
   $\equiv (X \underset{x,y:p(x,y)}{\triangleright} Y) \underset{x,y:true}{\triangleright} (Y \underset{y,z:r(y,z)}{\ltimes} Z)$
   $\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } \emptyset \textbf{ else } (X \underset{x,y:p(x,y)}{\triangleright} Y)$

Rule A.12 Splitting join predicates

1. $X \underset{p\vee q}{\ltimes} Y \equiv (X \underset{p}{\ltimes} Y) \cup (X \underset{q}{\ltimes} Y)$
2. $X \underset{p\wedge q}{\ltimes} Y \equiv \pi_X((X \underset{p}{\bowtie} Y) \cap (X \underset{q}{\bowtie} Y))$
   $X \underset{p\wedge q}{\ltimes} Y \equiv \pi_X(\sigma[p](X \underset{q}{\bowtie} Y))$
3. $X \underset{p\vee q}{\triangleright} Y \equiv (X \underset{p}{\triangleright} Y) \underset{q}{\triangleright} Y$

   Proof: $X \underset{x,y:p\vee q}{\triangleright} Y$
   $\equiv X - (X \underset{x,y:p\vee q}{\ltimes} Y)$
   $\equiv X - (X \underset{x,y:p}{\ltimes} Y \cup X \underset{x,y:q}{\ltimes} Y)$
   $\equiv (X - X \underset{x,y:p}{\ltimes} Y) - X \underset{x,y:q}{\ltimes} Y$
   $\equiv (X \underset{x,y:p}{\triangleright} Y) - X \underset{x,y:q}{\ltimes} Y$
   $\equiv (X \underset{x,y:p}{\triangleright} Y) \underset{x,y:q}{\triangleright} Y$

A.2 Proofs

Proof Rule 4.3 Scoping Let $x \notin FV(p)$, then:

1. $\exists x \in X \bullet p \wedge q \equiv p \wedge \exists x \in X \bullet q$

   Proof: $\exists x \in X \bullet p \wedge q$
   $$\begin{aligned} &\equiv \exists x \bullet x \in X \wedge p \wedge q \\ &\equiv p \wedge \exists x \bullet x \in X \wedge q \\ &\equiv p \wedge \exists x \in X \bullet q \end{aligned}$$

2. $p \vee \exists x \in X \bullet q \equiv ((p \wedge X = \emptyset) \vee (\exists x \in X \bullet p \vee q)$

   Proof: $p \vee \exists x \in X \bullet q$
   $$\begin{aligned} &\equiv (p \wedge true) \vee \exists x \in X \bullet q \\ &\equiv (p \wedge (X = \emptyset \vee X \neq \emptyset)) \vee \exists x \in X \bullet q \\ &\equiv (p \wedge X = \emptyset) \vee (p \wedge X \neq \emptyset) \vee \exists x \in X \bullet q \\ &\equiv ((p \wedge X = \emptyset) \vee (p \wedge \exists x \in X \bullet true) \vee \exists x \in X \bullet q \\ &\equiv ((p \wedge X = \emptyset) \vee (\exists x \in X \bullet p) \vee \exists x \in X \bullet q \\ &\equiv ((p \wedge X = \emptyset) \vee (\exists x \in X \bullet p \vee q) \end{aligned}$$

Proof Rule 4.15 Join associativity

1. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z) \wedge t}{\bowtie} Z \equiv X \underset{p(x,y) \wedge t}{\bowtie} (Y \underset{r(y,z)}{\bowtie} Z)$

   Proof: $(X \underset{p(x,y)}{\bowtie} Y) \underset{t \wedge r(y,z)}{\bowtie} Z$
   $$\begin{aligned} &\equiv \sigma[t]((X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\bowtie} Z) \\ &\equiv \sigma[t](X \underset{p(x,y)}{\bowtie} (Y \underset{r(y,z)}{\bowtie} Z)) \\ &\equiv X \underset{p(x,y) \wedge t}{\bowtie} (Y \underset{r(y,z)}{\bowtie} Z) \end{aligned}$$

2. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z) \wedge t}{\ltimes} Z \equiv \pi_{XY}(X \underset{p(x,y) \wedge t}{\bowtie} (Y \underset{r(y,z)}{\bowtie} Z))$
3. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\ltimes} Z \equiv X \underset{p(x,y)}{\bowtie} (Y \underset{r(y,z)}{\ltimes} Z)$
4. $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\rhd} Z \equiv X \underset{p(x,y)}{\bowtie} (Y \underset{r(y,z)}{\rhd} Z)$

   Proof: $(X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\rhd} Z$
   $$\begin{aligned} &\equiv (X \underset{p(x,y)}{\bowtie} Y) - (X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\ltimes} Z \\ &\equiv (X \underset{p(x,y)}{\bowtie} Y) - X \underset{p(x,y)}{\bowtie} (Y \underset{r(y,z)}{\ltimes} Z) \\ &\equiv X \underset{p(x,y)}{\bowtie} (Y - (Y \underset{r(y,z)}{\ltimes} Z)) \\ &\equiv X \underset{p(x,y)}{\bowtie} (Y \underset{r(y,z)}{\rhd} Z) \end{aligned}$$

Proof Rule 4.16 Join exchange

1. $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)\wedge t}{\bowtie} Z \equiv (X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)\wedge t}{\bowtie} Y$

   Proof: $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)\wedge t}{\bowtie} Z$
   $\equiv \sigma[t]((X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)}{\bowtie} Z)$
   $\equiv \sigma[t]((Y \underset{p(x,y)}{\bowtie} X) \underset{q(x,z)}{\bowtie} Z)$
   $\equiv \sigma[t](Y \underset{p(x,y)}{\bowtie} (X \underset{q(x,z)}{\bowtie} Z))$
   $\equiv \sigma[t]((X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)}{\bowtie} Y)$
   $\equiv (X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)\wedge t}{\bowtie} Y$

2. $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)\wedge t}{\ltimes} Z \equiv \pi_{XY}((X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)\wedge t}{\bowtie} Y)$

3. (a) $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)}{\bowtie} Z \equiv (X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)}{\bowtie} Y$ See above.

   (b) $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)}{\ltimes} Z \equiv (X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\bowtie} Y$ See above.

   (c) $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)}{\rhd} Z \equiv (X \underset{q(x,z)}{\rhd} Z) \underset{p(x,y)}{\bowtie} Y$

   Proof: $(X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)}{\rhd} Z$
   $\equiv (Y \underset{p(x,y)}{\bowtie} X) \underset{q(x,z)}{\rhd} Z$
   $\equiv Y \underset{p(x,y)}{\bowtie} (X \underset{q(x,z)}{\rhd} Z)$
   $\equiv (X \underset{q(x,z)}{\rhd} Z) \underset{p(x,y)}{\bowtie} Y$

   (d) $(X \underset{p(x,y)}{\ltimes} Y) \underset{q(x,z)}{\bowtie} Z \equiv (X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)}{\ltimes} Y$ By symmetry.

   (e) $(X \underset{p(x,y)}{\ltimes} Y) \underset{q(x,z)}{\ltimes} Z \equiv (X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\ltimes} Y$

   Proof: $(X \underset{p(x,y)}{\ltimes} Y) \underset{q(x,z)}{\ltimes} Z$
   $\equiv \pi_X((X \underset{p(x,y)}{\ltimes} Y) \underset{q(x,z)}{\bowtie} Z)$
   $\equiv \pi_X((X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\bowtie} Y)$
   $\equiv \pi_X(X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\bowtie} \pi_Y(Y))$
   $\equiv (X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\ltimes} Y$

(f) $(X \underset{p(x,y)}{\ltimes} Y) \underset{q(x,z)}{\rhd} Z \equiv (X \underset{q(x,z)}{\rhd} Z) \underset{p(x,y)}{\ltimes} Y$

Proof: $(X \underset{p(x,y)}{\rhd} Y) \underset{q(x,z)}{\ltimes} Z$

$$\begin{aligned} &\equiv \pi_X((X \underset{p(x,y)}{\rhd} Y) \underset{q(x,z)}{\bowtie} Z) \\ &\equiv \pi_X((X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)}{\rhd} Y) \\ &\equiv \pi_X(X \underset{q(x,z)}{\bowtie} Z) \underset{p(x,y)}{\rhd} Y \\ &\equiv (X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\rhd} Y \end{aligned}$$

(g) $(X \underset{p(x,y)}{\rhd} Y) \underset{q(x,z)}{\bowtie} Z \equiv (X \underset{q(x,z)}{\rhd} Z) \underset{p(x,y)}{\rhd} Y$ By symmetry.

(h) $(X \underset{p(x,y)}{\rhd} Y) \underset{q(x,z)}{\ltimes} Z \equiv (X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\rhd} Y$ By symmetry.

(i) $(X \underset{p(x,y)}{\rhd} Y) \underset{q(x,z)}{\rhd} Z \equiv (X \underset{q(x,z)}{\rhd} Z) \underset{p(x,y)}{\rhd} Y$

Proof: $(X \underset{p(x,y)}{\rhd} Y) \underset{q(x,z)}{\rhd} Z$

$$\begin{aligned} &\equiv (X \underset{p(x,y)}{\rhd} Y) - (X \underset{p(x,y)}{\rhd} Y) \underset{q(x,z)}{\ltimes} Z \\ &\equiv (X \underset{p(x,y)}{\rhd} Y) - (X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\rhd} Y \\ &\equiv (X - X \underset{q(x,z)}{\ltimes} Z) \underset{p(x,y)}{\rhd} Y \\ &\equiv (X \underset{q(x,z)}{\rhd} Z) \underset{p(x,y)}{\rhd} Y \end{aligned}$$

Proof Rule 4.17 Distribution of join over set operators: $(X - Y) \times Z = (X \times Z) - (Y \times Z)$

We define:

- $X^+ = \sigma[x : x \in Y](X)$
- $X^- = \sigma[x : x \notin Y](X)$
- $Y^+ = \sigma[x : x \in X](Y)$
- $Y^- = \sigma[x : x \notin X](Y)$

We have:

- $X = X^+ \cup X^-$, $Y = Y^+ \cup Y^-$
- $X^+ = Y^+$
- $X^- \cap Y^- = \emptyset$

Also, whenever $A \cap B = \emptyset$, then $(A \times C) - (B \times C) = A \times C$.

Left-hand side

$$\begin{aligned} &(X - Y) \times Z \\ &\equiv \sigma[x : x \notin Y](X) \times Z \\ &\equiv X^- \times Z \end{aligned}$$

Right-hand side

$$
\begin{aligned}
& (X \times Z) - (Y \times Z) \\
\equiv\ & ((X^+ \cup X^-) \times Z) - ((Y^+ \cup Y^-) \times Z) \\
\equiv\ & ((X^+ \times Z) \cup (X^- \times Z)) - ((Y^+ \times Z) \cup (Y^- \times Z)) \\
\equiv\ & (X^+ \times Z) - ((Y^+ \times Z) \cup (Y^- \times Z)) \cup \\
& \quad (X^- \times Z) - ((Y^+ \times Z) \cup (Y^- \times Z)) \\
\equiv\ & ((X^+ \times Z) - (X^+ \times Z)) - (Y^- \times Z) \cup \\
& \quad ((X^- \times Z) - (X^+ \times Z)) - (Y^- \times Z) \\
\equiv\ & \emptyset - (Y^- \times Z) \cup \\
& \quad (X^- \times Z) - (Y^- \times Z) \\
\equiv\ & (X^- \times Z) - (Y^- \times Z) \\
\equiv\ & X^- \times Z
\end{aligned}
$$

References

[ElNa89] Elmasri, R. and S.B. Navathe, **Fundamentals of Database Systems**, Benjamin/Cummings Publishing Company Inc., 1989.

[Kuijk91] van Kuijk, H.J.A., "Semantic Query Optimization in Distributed Database Systems—a Knowledge-Based Approach," Ph.D. Thesis, University of Twente, 1991.

Appendix B

Rewritings

B.1 Standard rewritings

Rewriting example B.1 (1) Chain

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet r(y,z)](X) \\
&\quad \equiv \quad \sigma[x : \exists y \in (Y \underset{y,z:r(y,z);m}{-} Z) \bullet p(x,y) \vee y.m](X) \ (\textbf{range nest}) \\
&\quad \equiv \quad X \underset{x,y:p(x,y)\vee y.m}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.2 (2) Tree(I)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z)](X) \\
&\quad \equiv \quad \sigma[x : \exists z \in Z \bullet q(x,z) \vee \exists y \in Y \bullet p(x,y)](X) \ (\textbf{descope}) \\
&\quad \equiv \quad \sigma[x : x.m \vee \exists y \in Y \bullet p(x,y)](X \underset{x,z:q(x,z);m}{-} Z) \ (\textbf{unnest}) \\
&\quad \equiv \quad \sigma[x : x.m \vee x.m'](((X \underset{x,z:q(x,z);m}{-} Z) \underset{x,y:p(x,y);m'}{-} Y) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.3 (3) Tree(II)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
&\quad \equiv \quad \sigma[x : \exists z \in Z \bullet \exists y \in Y \bullet q(x,z) \vee r(y,z)](X) \ (\textbf{exchange}) \\
&\quad \equiv \quad \sigma[x : \exists z \in Z \bullet q(x,z) \vee \exists y \in Y \bullet r(y,z)](X) \ (\textbf{descope}) \\
&\quad \equiv \quad \sigma[x : \exists z \in (Z \underset{y,z:r(y,z);m}{-} Y) \bullet q(x,z) \vee z.m](X) \ (\textbf{range nest}) \\
&\quad \equiv \quad X \underset{x,z:q(x,z)\vee z.m}{\ltimes} (Z \underset{y,z:r(y,z);m}{-} Y) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.4 (4) Cyclic

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \ \boldsymbol{c} \ r(y,z)](X) \\
&\quad \equiv \quad \pi_X(\sigma[v : p(v_X, v_Y) \vee \exists z \in Z \bullet q(v_X,z) \ \boldsymbol{c} \ r(v_Y,z)](X \times Y)) \ (\textbf{unnest}) \\
&\quad \equiv \quad \pi_X(\sigma[v : p(v_X, v_Y) \vee v.m]((X \times Y) \underset{v,z:q(v_X,z) \ \boldsymbol{c} \ r(v_Y,z);m}{-} Z)) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.5 (5) Chain

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists y \in (Y \underset{y,z:r(y,z);m}{-} Z) \bullet p(x,y) \vee y.m](X) \text{ (\textbf{range nest})}\\
&\quad\equiv\quad X \underset{x,y:p(x,y)\vee y.m}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z) \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.6 (6) Tree(I)

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \nexists y \in Y \bullet p(x,y)](X) \text{ (\textbf{descope})}\\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet p(x,y)](X \underset{x,z:q(x,z)}{\triangleright} Z) \text{ (\textbf{unnest})}\\
&\quad\equiv\quad (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\triangleright} Y \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.7 (7) Tree(II)

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet \exists y \in Y \bullet q(x,z) \vee r(y,z)](X) \text{ (\textbf{exchange})}\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \vee \exists y \in Y \bullet r(y,z)](X) \text{ (\textbf{descope})}\\
&\quad\equiv\quad \sigma[x : \nexists z \in (Z \underset{z,y:r(y,z);m}{-} Y) \bullet q(x,z) \vee z.m](X) \text{ (\textbf{range nest})}\\
&\quad\equiv\quad X \underset{x,z:q(x,z)\vee z.m}{\triangleright} (Z \underset{z,y:r(y,z);m}{-} Y) \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.8 (8) Cyclic

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \ \boldsymbol{c}\ r(y,z)](X)\\
&\quad\equiv\quad (\sigma[v : \neg p(v_X, v_Y) \wedge \nexists z \in Z \bullet q(v_X,z) \ \boldsymbol{c}\ r(v_Y,z)](X \times Y)) \div Y \text{ (\textbf{division})}\\
&\quad\equiv\quad (\sigma[v : \neg p(v_X, v_Y)]((X \times Y) \underset{v,z:q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z)}{\triangleright} Z)) \div Y \text{ (\textbf{unnest})}\\
&\quad\equiv\quad ((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z)}{\triangleright} Z) \div Y \text{ (\textbf{push})}
\end{aligned}$$

Rewriting example B.9 (9) Chain

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists y \in (Y \underset{y,z:r(y,z);m}{-} Z) \bullet p(x,y) \vee \neg y.m](X) \text{ (\textbf{range nest})}\\
&\quad\equiv\quad X \underset{x,y:p(x,y)\vee\neg y.m}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z) \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.10 (10) Tree(I)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \vee \exists y \in Y \bullet p(x,y)](X) \text{ (\textbf{descope})}\\
&\quad\equiv\quad \sigma[x : \neg x.m \vee \exists y \in Y \bullet p(x,y)](X \underset{x,z:q(x,z);m}{-} Z) \text{ (\textbf{unnest})}\\
&\quad\equiv\quad \sigma[x : \neg x.m \vee x.m'](X \underset{x,z:q(x,z);m}{-} Z) \underset{x,y:p(x,y);m'}{-} Y) \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.11 (11) Tree(II)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv \sigma[x : \exists y \in Y \bullet (\nexists z \in Z \bullet q(x,z) \wedge \nexists z \in Z \bullet r(y,z))](X) \text{ (\textbf{distribute})}\\
&\quad\equiv \sigma[x : \exists y \in (Y \underset{y,z:r(y,z)}{\triangleright} Z) \bullet \nexists z \in Z \bullet q(x,z)](X) \text{ (\textbf{range nest})}\\
&\quad\equiv \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \exists y \in (Y \underset{y,z:r(y,z)}{\triangleright} Z) \bullet true](X) \text{ (\textbf{descope})}\\
&\quad\equiv \sigma[x : \exists y \in (Y \underset{y,z:r(y,z)}{\triangleright} Z) \bullet true](X \underset{x,z:q(x,z)}{\triangleright} Z)) \text{ (\textbf{unnest})}\\
&\quad\equiv (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:true}{\ltimes} (Y \underset{y,z:r(y,z)}{\triangleright} Z) \text{ (\textbf{unnest})}\\
&\quad\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\triangleright} Z) \neq \emptyset \textbf{ then } (X \underset{x,z:q(x,z)}{\triangleright} Z) \textbf{ else } \emptyset \text{ (\textbf{constant} $\ltimes$ \textbf{predicate})}
\end{aligned}$$

Rewriting example B.12 (12) Cyclic

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z) \boldsymbol{c}\, r(y,z)](X)\\
&\quad\equiv \pi_X(\sigma[v : p(v_X, v_Y) \vee \nexists z \in Z \bullet q(v_X,z) \boldsymbol{c}\, r(v_Y,z)](X \times Y)) \text{ (\textbf{unnest})}\\
&\quad\equiv \pi_X(\sigma[v : p(v_X, v_Y) \vee \neg v.m](X \underset{v,z:q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z);m}{-} Z)) \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.13 (13) Chain

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet r(y,z)](X)\\
&\quad\equiv \sigma[x : \nexists y \in (Y \underset{y,z:r(y,z);m}{-} Z) \bullet p(x,y) \vee \neg y.m](X) \text{ (\textbf{range nest})}\\
&\quad\equiv X \underset{x,y:p(x,y)\vee\neg y.m}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z) \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.14 (14) Tree(I)

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z)](X)\\
&\quad\equiv \sigma[x : \exists z \in Z \bullet q(x,z) \wedge \nexists y \in Y \bullet p(x,y)](X) \text{ (\textbf{descope})}\\
&\quad\equiv (X \underset{x,z:q(x,z)}{\ltimes} Z) \underset{x,y:p(x,y)}{\triangleright} Y \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.15 (15) Tree(II)
The result for this query is the complement of the result achieved for expression 11:

$$\begin{aligned}
&X - \textbf{if } (Y \underset{y,z:r(y,z)}{\triangleright} Z) \neq \emptyset \textbf{ then } (X \underset{x,z:q(x,z)}{\triangleright} Z) \textbf{ else } \emptyset\\
&\quad\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\triangleright} Z) \neq \emptyset \textbf{ then } (X \underset{x,z:q(x,z)}{\ltimes} Z) \textbf{ else } X
\end{aligned}$$

We do not show the full rewriting for this expression, because we would have to take into account the possibility of empty ranges to ensure that we indeed achieve the complementary result. This would result into a rewriting sequence that is too complex for the purpose of this section.

Rewriting example B.16 (16) Cyclic

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z) \boldsymbol{c}\, r(y,z)](X)\\
&\quad\equiv (\sigma[v : \neg p(v_X, v_Y) \wedge \exists z \in Z \bullet q(v_X,z) \boldsymbol{c}\, r(v_Y,z)](X \times Y)) \div Y \text{ (\textbf{unnest})}\\
&\quad\equiv ((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z)}{\ltimes} Z) \div Y \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.17 (17) Chain

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists y \in (Y \underset{y,z:r(y,z)}{\ltimes} Z) \bullet p(x,y)](X) \ (\textbf{range nest})\\
&\quad\equiv\quad X \underset{x,y:p(x,y)}{\ltimes} (Y \underset{y,z:r(y,z)}{\ltimes} Z) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.18 (18) Tree(I)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists z \in Z \bullet q(x,z) \wedge \exists y \in Y \bullet p(x,y)](X) \ (\textbf{descope})\\
&\quad\equiv\quad \sigma[x : \exists y \in Y \bullet p(x,y)](X \underset{x,z:q(x,z)}{\ltimes} Z) \ (\textbf{unnest})\\
&\quad\equiv\quad (X \underset{x,z:q(x,z)}{\ltimes} Z) \underset{x,y:p(x,y)}{\ltimes} Y \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.19 (19) Tree(II)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \wedge r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists z \in Z \bullet \exists y \in Y \bullet q(x,z) \wedge r(y,z)](X) \ (\textbf{exchange})\\
&\quad\equiv\quad \sigma[x : \exists z \in Z \bullet q(x,z) \wedge \exists y \in Y \bullet r(y,z)](X) \ (\textbf{descope})\\
&\quad\equiv\quad \sigma[x : \exists z \in (Z \underset{y,z:r(y,z)}{\ltimes} Y) \bullet q(x,z)](X) \ (\textbf{range nest})\\
&\quad\equiv\quad X \underset{x,z:q(x,z)}{\ltimes} (Z \underset{y,z:r(y,z)}{\ltimes} Y) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.20 (20) Cyclic

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z) \ \boldsymbol{c} \ r(y,z)](X)\\
&\quad\equiv\quad \pi_X(\sigma[v : p(v_X, v_Y) \wedge \exists z \in Z \bullet q(v_X,z) \ \boldsymbol{c} \ r(v_Y,z)](X \times Y)) \ (\textbf{unnest})\\
&\quad\equiv\quad \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z) \ \boldsymbol{c} \ r(v_Y,z)}{\ltimes} Z) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.21 (21) Chain

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists y \in (Y \underset{y,z:r(y,z)}{\ltimes} Z) \bullet p(x,y)](X) \ (\textbf{range nest})\\
&\quad\equiv\quad X \underset{x,y:p(x,y)}{\rhd} (Y \underset{y,z:r(y,z)}{\ltimes} Z) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.22 (22) Tree(I)

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \vee \nexists y \in Y \bullet p(x,y)](X) \ (\textbf{descope})\\
&\quad\equiv\quad \sigma[x : \neg x.m \vee \nexists y \in Y \bullet p(x,y)](X \underset{x,z:q(x,z);m}{-} Z)) \ (\textbf{unnest})\\
&\quad\equiv\quad \sigma[x : \neg x.m \vee \neg x.m'](X \underset{x,z:q(x,z);m}{-} Z) \underset{x,y:p(x,y);m'}{-} Y) \ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.23 (23) Tree(II)

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet q(x,z) \wedge r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet \exists y \in Y \bullet q(x,z) \wedge r(y,z)](X)\ (\textbf{exchange})\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \exists y \in Y \bullet r(y,z)](X)\ (\textbf{descope})\\
&\quad\equiv\quad \sigma[x : \nexists z \in (Z \underset{z,y:r(y,z)}{\ltimes} Y) \bullet q(x,z)](X)\ (\textbf{range nest})\\
&\quad\equiv\quad X \underset{x,z:q(x,z)}{\triangleright} (Z \underset{z,y:r(y,z)}{\ltimes} Y)\ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.24 (24) Cyclic

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z)\ \boldsymbol{c}\ r(y,z)](X)\\
&\quad\equiv\quad (\sigma[v : \neg p(v_X, v_Y) \vee \nexists z \in Z \bullet q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z)](X \times Y) \div Y)\\
&\quad\equiv\quad (\sigma[v : \neg p(v_X, v_Y) \vee \neg v.m]((X \times Y) \underset{v,z:q(v_X,z)\ \overline{\boldsymbol{c}}\ r(v_Y,z);m}{} Z) \div Y\ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.25 (25) Chain

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists y \in (Y \underset{y,z:r(y,z)}{\triangleright} Z) \bullet p(x,y)](X)\ (\textbf{range nest})\\
&\quad\equiv\quad X \underset{x,y:p(x,y)}{\ltimes} (Y \underset{y,z:r(y,z)}{\triangleright} Z)\ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.26 (26) Tree(I)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \exists y \in Y \bullet p(x,y)](X)\ (\textbf{descope})\\
&\quad\equiv\quad (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\ltimes} Y\ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.27 (27) Tree(II)

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \wedge r(y,z)](X)\\
&\quad\equiv\quad \pi_X(\sigma[v : \nexists z \in Z \bullet q(v_X,z) \wedge r(v_Y,z)](X \times Y))\ (\textbf{unnest})\\
&\quad\equiv\quad \pi_X((X \times Y) \underset{v,z:q(v_X,z)\wedge r(v_Y,z)}{\triangleright} Z)\ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.28 (28) Cyclic

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z)\ \boldsymbol{c}\ r(y,z)](X)\\
&\quad\equiv\quad \pi_X(\sigma[v : p(v_X, v_Y) \wedge \nexists z \in Z \bullet q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z)](X \times Y))\ (\textbf{unnest})\\
&\quad\equiv\quad \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z)}{\triangleright} Z)\ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.29 (29) Chain

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists y \in (Y \underset{y,z:r(y,z)}{\triangleright} Z) \bullet p(x,y)](X)\ (\textbf{range nest})\\
&\quad\equiv\quad X \underset{x,y:p(x,y)}{\triangleright} (Y \underset{y,z:r(y,z)}{\triangleright} Z)\ (\textbf{unnest})
\end{aligned}$$

Rewriting example B.30 (30) Tree(I)

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z)](X)\\
&\quad\equiv \sigma[x : \exists z \in Z \bullet q(x,z) \vee \nexists y \in Y \bullet p(x,y)](X) \text{ (\textbf{descope})}\\
&\quad\equiv \sigma[x : x.m \vee \nexists y \in Y \bullet p(x,y)](X \underset{x,z:q(x,z);m}{-} Z) \text{ (\textbf{unnest})}\\
&\quad\equiv \sigma[x : x.m \vee \neg x.m'](((X \underset{x,z:q(x,z);m}{-} Z) \underset{x,y:p(x,y);m'}{-} Y) \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.31 (31) Tree(II)

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \wedge r(y,z)](X)\\
&\quad\equiv (\sigma[v : \exists z \in Z \bullet q(v_X,z) \wedge r(v_Y,z)](X \times Y)) \div Y \text{ (\textbf{unnest})}\\
&\quad\equiv ((X \times Y) \underset{v,z:q(v_X,z)\wedge r(v_Y,z)}{\ltimes} Z) \div Y \text{ (\textbf{unnest})}
\end{aligned}$$

Rewriting example B.32 (32) Cyclic

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z) \boldsymbol{c}\, r(y,z)](X)\\
&\quad\equiv (\sigma[v : \neg p(v_X,v_Y) \vee \exists z \in Z \bullet q(v_X,z) \boldsymbol{c}\, r(v_Y,z)](X \times Y)) \div Y \text{ (\textbf{unnest})}\\
&\quad\equiv (\sigma[v : \neg p(v_X,v_Y) \vee v.m]((X \times Y) \underset{v,z:q(v_X,z)}{\overline{\boldsymbol{c}}}\, \underset{r(v_Y,z);m}{} Z) \div Y \text{ (\textbf{unnest})}
\end{aligned}$$

B.2 Distribution of quantification

If we use the conditional for constant terms, then we have:

Rewriting example B.33 (1) Chain

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet r(y,z)](X)\\
&\quad\equiv \sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists y \in Y \bullet \exists z \in Z \bullet r(y,z)](X) \text{ (\textbf{distribute})}\\
&\quad\equiv \sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists y \in (Y \underset{y,z:r(y,z)}{\ltimes} Z) \bullet true](X) \text{ (\textbf{range nest})}\\
&\quad\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } \sigma[x : \exists y \in Y \bullet p(x,y)(X) \text{ (\textbf{constant})}\\
&\quad\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } X \underset{x,y:p(x,y)}{\ltimes} Y \text{ (\textbf{unnest})}
\end{aligned}$$

We can rewrite the standard result into the latter:

$$\begin{aligned}
&X \underset{x,y:p(x,y)\vee y.m}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z)\\
&\quad\equiv (X \underset{x,y:p(x,y)}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z)) \cup (X \underset{x,y:y.m}{\ltimes} (Y \underset{y,z:r(y,z);m}{-} Z)) \text{ (\textbf{split})}\\
&\quad\equiv (X \underset{x,y:p(x,y)}{\ltimes} Y) \cup (X \underset{x,y:true}{\ltimes} (Y \underset{y,z:r(y,z)}{\ltimes} Z)) \text{ (\textbf{simplify})}\\
&\quad\equiv (X \underset{x,y:p(x,y)}{\ltimes} Y) \cup \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } \emptyset \text{ (\textbf{constant} $\ltimes$ \textbf{predicate})}\\
&\quad\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \cup (X \underset{x,y:p(x,y)}{\ltimes} Y) \textbf{ else } \emptyset \cup (X \underset{x,y:p(x,y)}{\ltimes} Y)\\
&\qquad \text{(\textbf{distribute})}\\
&\quad\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } X \underset{x,y:p(x,y)}{\ltimes} Y \text{ (\textbf{simplify})}
\end{aligned}$$

In the above rewriting, we have used rules for splitting join predicates and for the simplification of expressions that contain markjoin operators; these are listed in Appendix A.

Rewriting example B.34 (3) Tree(II)

$$
\begin{aligned}
&\sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
&\quad\equiv\quad \sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \vee \exists z \in Z \bullet r(y,z)](X) \ (\textbf{distribute}) \\
&\quad\equiv\quad \sigma[x : \exists z \in Z \bullet q(x,z) \vee \exists y \in Y \bullet \exists z \in Z \bullet r(y,z)](X) \ (\textbf{descope}) \\
&\quad\equiv\quad \sigma[x : \exists z \in Z \bullet q(x,z) \vee \exists y \in (Y \underset{y,z:r(y,z)}{\ltimes} Z) \bullet true](X) \ (\textbf{range nest}) \\
&\quad\equiv\quad \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } \sigma[x : \exists z \in Z \bullet q(x,z)](X) \ (\textbf{constant}) \\
&\quad\equiv\quad \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } X \underset{x,z:q(x,z)}{\ltimes} Z \ (\textbf{unnest})
\end{aligned}
$$

Rewriting example B.35 (4) Cyclic

$$
\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \ \boldsymbol{c}\ r(y,z)](X) \\
&\quad\equiv\quad \sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \ \boldsymbol{c}\ r(y,z)](X) \ (\textbf{distribute}) \\
&\quad\equiv\quad T + \sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \ \boldsymbol{c}\ r(y,z)](F) \\
&\qquad\quad \textbf{with } (T,F) = X \underset{x,y:p(x,y)}{\ltimes^2} Y \ (\textbf{bypass}) \\
&\quad\equiv\quad T + \pi_F((F \times Y) \underset{v,z:q(v_X,z)\ \boldsymbol{c}\ r(v_Y,z)}{\ltimes} Z) \\
&\qquad\quad \textbf{with } (T,F) = X \underset{x,y:p(x,y)}{\ltimes^2} Y \ (\textbf{unnest})
\end{aligned}
$$

Rewriting example B.36 (5) Chain

$$
\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet r(y,z)](X) \\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists y \in Y \bullet \exists z \in Z \bullet r(y,z)](X) \ (\textbf{distribute}) \\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists y \in (Y \underset{y,z:r(y,z)}{\ltimes} Z) \bullet true](X) \ (\textbf{range nest}) \\
&\quad\equiv\quad \sigma[x : \nexists y \in (Y \underset{y,z:r(y,z)}{\ltimes} Z) \bullet true](X \underset{x,y:p(x,y)}{\triangleright} Y) \ (\textbf{unnest}) \\
&\quad\equiv\quad (X \underset{x,y:p(x,y)}{\triangleright} Y) \underset{x,y:true}{\triangleright} (Y \underset{y,z:r(y,z)}{\ltimes} Z) \ (\textbf{unnest}) \\
&\quad\equiv\quad \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } \emptyset \textbf{ else } X \underset{x,y:p(x,y)}{\triangleright} Y \ (\textbf{constant} \triangleright \textbf{predicate})
\end{aligned}
$$

Note that the standard result can be rewritten into the latter easily:

$$
\begin{aligned}
&X \underset{x,y:p(x,y) \vee y.m}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z) \\
&\quad\equiv\quad (X \underset{x,y:p(x,y)}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z)) \underset{x,y:y.m}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z) \ (\textbf{split}) \\
&\quad\equiv\quad (X \underset{x,y:p(x,y)}{\triangleright} Y) \underset{x,y:y.m}{\triangleright} (Y \underset{y,z:r(y,z);m}{-} Z) \ (\textbf{simplify}) \\
&\quad\equiv\quad (X \underset{x,y:p(x,y)}{\triangleright} Y) \underset{x,y:true}{\triangleright} (Y \underset{y,z:r(y,z)}{\ltimes} Z) \ (\textbf{simplify})
\end{aligned}
$$

We use the rules for simplification of markjoin expressions and for splitting of join predicates.

Rewriting example B.37 (7) Tree(II)

$$
\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet q(x,z) \vee \exists z \in Z \bullet r(y,z)](X) \text{ (**distribute**)} \\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \nexists y \in Y \bullet \exists z \in Z \bullet r(y,z)](X) \text{ (**descope**)} \\
&\quad\equiv\quad \sigma[x : \nexists y \in (Y \underset{y,z:r(y,z)}{\ltimes} Z) \bullet true](X \underset{x,z:q(x,z)}{\triangleright} Z) \text{ (**unnest/nest**)} \\
&\quad\equiv\quad (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:true}{\triangleright} (Y \underset{y,z:r(y,z)}{\ltimes} Z) \text{ (**unnest**)}
\end{aligned}
$$

Rewriting example B.38 (8) Cyclic

$$
\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \; \boldsymbol{c} \; r(y,z)](X) \\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists y \in Y \bullet \exists z \in Z \bullet q(x,z) \; \boldsymbol{c} \; r(y,z)](X) \\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet q(x,z) \; \boldsymbol{c} \; r(y,z)](X \underset{x,y:p(x,y)}{\triangleright} Y) \\
&\quad\equiv\quad X - \sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet q(x,z) \; \boldsymbol{c} \; r(y,z)](X \underset{x,y:p(x,y)}{\triangleright} Y) \\
&\quad\equiv\quad X - \pi_X(((X \underset{x,y:p(x,y)}{\triangleright} Y) \times Y) \underset{v,y:q(v_X,z) \; \boldsymbol{c} \; r(v_Y,z)}{\ltimes} Z)
\end{aligned}
$$

B.3 Cyclic queries

We rewrite some cyclic queries of our list. A query is called disjunctive or conjunctive whenever the inner connective is $\vee$ or $\wedge$, respectively.

Query 4, disjunction

Rewriting example B.39 (4), disjunctive, without distribution

$$
\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
&\quad\equiv\quad \pi_X(T + \sigma[v : \exists z \in Z \bullet q(v_X,z) \vee r(v_Y,z)](F)) \\
&\qquad \mathbf{with}\ (T,F) = X \underset{x,y:p(x,y)}{\bowtie^2} Y \text{ (**bypass**)} \\
&\quad\equiv\quad \pi_X(T + (F \underset{v,z:q(v_X,z) \vee r(v_Y,z)}{\ltimes} Z)) \\
&\qquad \mathbf{with}\ (T,F) = X \underset{x,y:p(x,y)}{\bowtie^2} Y \text{ (**unnest**)}
\end{aligned}
$$

Rewriting example B.40 (4), disjunctive, with distribution

$$
\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \vee \exists z \in Z \bullet r(y,z)](X)\\
&\qquad (\textbf{distribute inner})\\
&\quad\equiv\quad \sigma[x : \exists z \in Z \bullet q(x,z) \vee \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet r(y,z)](X)\ (\textbf{descope})\\
&\quad\equiv\quad T + \sigma[x : \exists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet r(y,z)](F)\\
&\qquad \textbf{with}\ (T,F) = X \underset{x,z:q(x,z)}{\ltimes^2} Z\ (\textbf{bypass})\\
&\quad\equiv\quad T + \textbf{if}\ Y \underset{y,z:r(y,z)}{\ltimes} Z \neq \emptyset\ \textbf{then}\ F\ \textbf{else}\ F \underset{x,y:p(x,y)}{\ltimes} Y\\
&\qquad \textbf{with}\ (T,F) = X \underset{x,z:q(x,z)}{\ltimes^2} Z\ (\textbf{case 1})\\
&\quad\equiv\quad \textbf{if}\ Y \underset{y,z:r(y,z)}{\ltimes} Z \neq \emptyset\ \textbf{then}\ T + F\ \textbf{else}\ T + F \underset{x,y:p(x,y)}{\ltimes} Y\\
&\qquad \textbf{with}\ (T,F) = X \underset{x,z:q(x,z)}{\ltimes^2} Z\ (\textbf{distribute}\ +)\\
&\quad\equiv\quad \textbf{if}\ Y \underset{y,z:r(y,z)}{\ltimes} Z \neq \emptyset\ \textbf{then}\ X\ \textbf{else}\ T + F \underset{x,y:p(x,y)}{\ltimes} Y\\
&\qquad \textbf{with}\ (T,F) = X \underset{x,z:q(x,z)}{\ltimes^2} Z\ (\textbf{simplify})
\end{aligned}
$$

Whenever the set $Y \underset{y,z:r(y,z)}{\ltimes} Z$ is not empty, distribution of the inner quantification results in major cost savings—only one semijoin has to be performed.

Query 8, disjunction

Rewriting example B.41 (8), disjunctive, Codd

$$
\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet p(x,y) \vee q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \forall y \in Y \bullet \nexists z \in Z \bullet p(x,y) \vee q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \forall y \in Y \bullet \forall z \in Z \bullet \neg p(x,y) \wedge \neg q(x,z) \wedge \neg r(y,z)](X)\ (\textbf{PNF})\\
&\quad\equiv\quad (\sigma[v : \forall z \in Z \bullet \neg p(v_X,v_Y) \wedge \neg q(v_X,z) \wedge \neg r(v_Y,z)](X \times Y)) \div Y\\
&\quad\equiv\quad ((\sigma[v : \neg p(v_X,v_Y) \wedge \neg q(v_X,v_Z) \wedge \neg r(v_Y,v_Z)]((X \times Y) \times Z) \div Z) \div Y\\
&\quad\equiv\quad ((((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:\neg q(v_X,z) \wedge \neg r(v_Y,z)}{\bowtie} Z) \div Z) \div Y
\end{aligned}
$$

Note that, due to the use of division, the use of partial join operators is impossible. The result achieved by means of standard rewriting with $\mathcal{R}$ contains one less division:

$$
((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z) \vee r(v_Y,z)}{\rhd} Z) \div Y
$$

Instead of a join followed by division, we now have a antijoin of which the predicate is a disjunction that can be split. The standard result can be rewritten further algebraically as follows:

$$
\begin{aligned}
&((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)\vee r(v_Y,z)}{\triangleright} Z) \div Y \\
&\quad\equiv\quad (((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)}{\triangleright} Z) \underset{v,z:r(v_Y,z)}{\triangleright} Z) \div Y \ (\textbf{split}) \\
&\quad\equiv\quad (((X \underset{v,z:q(v_X,z)}{\triangleright} Z) \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:r(v_Y,z)}{\triangleright} Z) \div Y \ (\textbf{exchange}) \\
&\quad\equiv\quad (((X \underset{v,z:q(v_X,z)}{\triangleright} Z) \underset{x,y:\neg p(x,y)}{\bowtie} (Y \underset{v,z:r(v_Y,z)}{\triangleright} Z)) \div Y \ (\textbf{associate})
\end{aligned}
$$

The advantage of the latter expression is that the operands of the regular join present are reduced by the antijoin operations. The price paid is an additional access to table Z. Below, we show that, with distribution of the inner quantification, division can be avoided altogether:

Rewriting example B.42 (8) disjunctive, distribution

$$
\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet p(x,y) \vee (\exists z \in Z \bullet q(x,z) \vee \exists z \in Z \bullet r(y,z))](X) \\
&\qquad (\textbf{distribute inner}) \\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet r(y,z)](X) \ (\textbf{descope}) \\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet r(y,z)](X \underset{x,z:q(x,z)}{\triangleright} Z) \ (\textbf{unnest}) \\
&\quad\equiv\quad \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } \emptyset \textbf{ else } (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\triangleright} Y \ (\textbf{case 5})
\end{aligned}
$$

Distribution of the innermost quantification leads to what seems the best result. No division operators are present, and all joins are partial. Due to the distribution, descoping can take place, followed by unnesting of the quantification over Z, and then the selection is rewritten as in Rewriting example B.36, with distribution of the quantification with range Y.
Recall that query 4 is the complement of 8, so for 4 we have:

$$X - \textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } \emptyset \textbf{ else } (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\triangleright} Y$$

which is equivalent to:

$$\textbf{if } (Y \underset{y,z:r(y,z)}{\ltimes} Z) \neq \emptyset \textbf{ then } X \textbf{ else } X - (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\triangleright} Y$$

We have found a useful equivalence rule:

$$X - (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\triangleright} Y \equiv T + F \underset{x,y:p(x,y)}{\ltimes} Y \textbf{ with } (T,F) = X \underset{x,z:q(x,z)}{\ltimes^2} Z$$

We show the rewriting of the left-hand side into the right-hand side:

Rewriting example B.43

$$
\begin{aligned}
& X - (X \underset{x,z:q(x,z)}{\rhd} Z) \underset{x,y:p(x,y)}{\rhd} Y \\
\equiv\ & ((X \underset{x,z:q(x,z)}{\ltimes} Z) \cup (X \underset{x,z:q(x,z)}{\rhd} Z)) - (X \underset{x,z:q(x,z)}{\rhd} Z) \underset{x,y:p(x,y)}{\rhd} Y \\
\equiv\ & ((X \underset{x,z:q(x,z)}{\ltimes} Z - (X \underset{x,z:q(x,z)}{\rhd} Z) \underset{x,y:p(x,y)}{\rhd} Y) \cup \\
& (X \underset{x,z:q(x,z)}{\rhd} Z - (X \underset{x,z:q(x,z)}{\rhd} Z) \underset{x,y:p(x,y)}{\rhd} Y) \\
\equiv\ & (X \underset{x,z:q(x,z)}{\ltimes} Z) \cup (X \underset{x,z:q(x,z)}{\rhd} Z) \underset{x,y:p(x,y)}{\ltimes} Y \\
\equiv\ & T + F \underset{x,y:p(x,y)}{\ltimes} Y \ \mathbf{with}\ (T,F) = X \underset{x,z:q(x,z)}{\ltimes^2} Z
\end{aligned}
$$

Query 8, conjunction

Rewriting example B.44 (8), conjunctive, Codd

$$
\begin{aligned}
& \sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \wedge r(y,z)](X) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet p(x,y) \vee (q(x,z) \wedge r(y,z))](X)\ (\textbf{scope}) \\
\equiv\ & \sigma[x : \forall y \in Y \bullet\ \nexists z \in Z \bullet p(x,y) \vee (q(x,z) \wedge r(y,z))](X)\ (\textbf{push}\ \neg) \\
\equiv\ & \sigma[x : \forall y \in Y \bullet \forall z \in Z \bullet \neg p(x,y) \wedge (\neg q(x,z) \vee \neg r(y,z))](X)\ (\textbf{push}\ \neg, \textbf{PNF}) \\
\equiv\ & (\sigma[v : \forall z \in Z \bullet \neg p(v_X, v_Y) \wedge (\neg q(v_X, z) \vee \neg r(y,z))](X \times Y)) \div Y\ (\textbf{unnest}) \\
\equiv\ & ((\sigma[v : \neg p(v_X, v_Y) \wedge (\neg q(v_X, v_Z) \vee \neg r(v_Y, v_Z))](X \times Y) \times Z) \div Z) \div Y \\
& (\textbf{unnest}) \\
\equiv\ & ((((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:\neg q(v_X,z) \vee \neg r(v_Y,z)}{\bowtie} Z) \div Z) \div Y\ (\textbf{push})
\end{aligned}
$$

The result achieved by means of standard rewriting using $\mathcal{R}$ is:

$$((X \underset{\neg p}{\bowtie} Y) \underset{q \wedge r}{\rhd} Z) \div Y$$

Again, we have one less division operator. However, the antijoin predicate, which is a conjunction, cannot be split, as far as we know, so further algebraic rewriting seems impossible. With distribution of the outer quantification we have:

Rewriting example B.45 (8), conjunctive, distribution

$$
\begin{aligned}
& \sigma[x : \nexists y \in Y \bullet p(x,y) \vee \exists z \in Z \bullet q(x,z) \wedge r(y,z)](X) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge\ \nexists y \in Y \bullet \exists z \in Z \bullet q(x,z) \wedge r(y,z)](X)\ (\textbf{distribute}) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet q(x,z) \wedge r(y,z)](X \underset{x,y:p(x,y)}{\rhd} Y)\ (\textbf{unnest}) \\
\equiv\ & (X \underset{x,y:p(x,y)}{\rhd} Y) \underset{x,z:q(x,z)}{\rhd} (Z \underset{z,y:r(y,z)}{\ltimes} Y)\ (\textbf{case 23})
\end{aligned}
$$

Again, the best result is achieved after distribution of quantification. Distribution is advantageous even though the number of free variables in the disjuncts is not reduced compared to the number present in the original predicate. But, due to the distribution, which concerns a negated quantifier, the disjunction is transformed into a conjunction.

Query 16, disjunction

Rewriting example B.46 (16), disjunctive, Codd

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv \sigma[x : \nexists y \in Y \bullet \nexists z \in Z \bullet \neg p(x,y) \wedge (q(x,z) \vee r(y,z))](X) \text{ (\textbf{scope})}\\
&\quad\equiv (\sigma[v : \exists z \in Z \bullet \neg p(v_X, v_Y) \wedge (q(v_X, z) \vee r(v_Y, z))](X \times Y)) \div Y \text{ (\textbf{division})}\\
&\quad\equiv (\sigma[v : \neg p(v_X, v_Y) \wedge (q(v_X, v_Z) \vee r(v_Y, v_Z))]((X \times Y) \times Z)) \div Y \text{ (\textbf{unnest})}
\end{aligned}$$

The result contains one division, and the selection predicate involves a disjunction. We try distribution:

Rewriting example B.47 (16), disjunctive, distribution

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv \sigma[x : \nexists y \in Y \bullet p(x,y) \vee (\nexists z \in Z \bullet q(x,z) \wedge \nexists z \in Z \bullet r(y,z))](X)\\
&\qquad (\textbf{distribute inner})
\end{aligned}$$

After distribution of the inner quantification, which both reduces the number of free variables and changes the dis- into a conjunction, the selection predicate is not in MNF. Transformation into MNF requires to distribute $\vee$ over $\wedge$, and then to distribute the outer quantification, resulting in a complex expression. Distributing the outer quantification, we have:

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\qquad (\textbf{distribute outer})\\
&\quad\equiv \sigma[x : \nexists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X \underset{x,y:p(x,y)}{\triangleright} Y) \text{ (\textbf{unnest})}\\
&\quad\equiv \textbf{if } (Y \underset{y,z:r(y,z)}{\triangleright} Z) \neq \emptyset \textbf{ then } \emptyset \textbf{ else } ((X \underset{x,y:p(x,y)}{\triangleright} Y) \underset{x,z:q(x,z)}{\ltimes} Z) \text{ (\textbf{case 15})}
\end{aligned}$$

whereas the result achieved by standard rewriting is:

$$((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z) \vee r(v_Y,z)}{\ltimes} Z) \div Y$$

Query 16, conjunction

Rewriting example B.48 (16), conjunctive, Codd

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z) \wedge r(y,z)](X)\\
&\quad\equiv \sigma[x : \nexists y \in Y \bullet \nexists z \in Z \bullet \neg p(x,y) \wedge q(x,z) \wedge r(y,z)](X)\\
&\quad\equiv (\sigma[v : \exists z \in Z \bullet \neg p(v_X, v_Y) \wedge q(v_X, z) \wedge r(v_Y, z)](X \times Y)) \div Y\\
&\quad\equiv (\sigma[v : \neg p(v_X, v_Y) \wedge q(v_X, v_Z) \wedge r(v_Y, v_Z)]((X \times Y) \times Z)) \div Y
\end{aligned}$$

which is equivalent to the result achieved by standard rewriting:

$$((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z) \wedge r(v_Y,z)}{\ltimes} Z) \div Y$$

Rewriting example B.49 (16), conjunctive, distribution

$$
\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \vee \nexists z \in Z \bullet q(x,z) \wedge r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \wedge r(y,z)](X) \ (\textbf{distribute})\\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet \nexists z \in Z \bullet q(x,z) \wedge r(y,z)](X \underset{x,y:p(x,y)}{\triangleright} Y) \ (\textbf{unnest})\\
&\quad\equiv\quad ((((X \underset{x,y:p(x,y)}{\triangleright} Y) \times Y) \underset{v,z:q(v_X,z)\wedge r(v_Y,z)}{\ltimes} Z) \div Y \ (\textbf{case 31})
\end{aligned}
$$

With distribution, the result contains an additional antijoin operation. Instead of the subexpression $(X \bowtie_{x,y:\neg p(x,y)} Y)$, we have $((X \triangleright_{x,y:p(x,y)} Y) \times Y)$. We can say something about the cardinality of the latter expressions. Let R_1, R_2, respectively, denote the result sets of the above expressions, then it holds that $R_2 \subseteq R_1$. We explain why. Let x be some tuple of X such that $\nexists y \in Y \bullet p$, then $\{x\} \times Y$ belongs to R_2, and also to R_1, because $\forall y \in Y \bullet \neg p$. On the other hand, let x be some tuple of X such that $\exists y \in Y \bullet p$, then this tuple does not contribute to R_2. But the tuple does contribute to R_1, unless it holds that $\forall y \in Y \bullet p$, which is unlikely. Therefore, the cardinality of the join expression will be equal to or, most probably, larger than that of the antijoin expression. So, in this case distribution is profitable as well.

Query 24, disjunction

Rewriting example B.50 (24) disjunctive, Codd

$$
\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \nexists y \in Y \bullet \exists z \in Z \bullet p(x,y) \wedge (q(x,z) \vee r(y,z))](X)\\
&\quad\equiv\quad \sigma[x : \forall y \in Y \bullet \nexists z \in Z \bullet p(x,y) \wedge (q(x,z) \vee r(y,z))](X)\\
&\quad\equiv\quad \sigma[x : \forall y \in Y \bullet \forall z \in Z \bullet \neg p(x,y) \vee (\neg q(x,z) \wedge \neg r(y,z))](X)\\
&\quad\equiv\quad (\sigma[v : \forall z \in Z \bullet \neg p(v_X,v_Y) \vee (\neg q(v_X,z) \wedge \neg r(v_Y,z))](X \times Y)) \div Y\\
&\quad\equiv\quad ((\sigma[v : \neg p(v_X,v_Y) \vee (\neg q(v_X,v_Z) \wedge \neg r(v_Y,v_Z))]((X \times Y) \times Z) \div Z) \div Y\\
&\quad\equiv\quad (((X \underset{x,y:\neg p(x,y)}{\bowtie} Y) \times Z) \cup ((X \underset{x,z:\neg q(x,z)}{\bowtie} Z) \underset{v,y:\neg r(y,v_Z)}{\bowtie} Y)) \div Z) \div Y
\end{aligned}
$$

Rewriting example B.51 (24), disjunctive, $\mathcal{R}$

$$
\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad X - \sigma[x : \exists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \ (\textbf{difference})\\
&\quad\equiv\quad X - \pi_X(\sigma[v : p(v_X,v_Y) \wedge \exists z \in Z \bullet q(v_X,z) \vee r(v_Y,z)](X \times Y)) \ (\textbf{unnest})\\
&\quad\equiv\quad X - \pi_X(\sigma[x : \exists z \in Z \bullet q(v_X,z) \vee r(v_Y,z)](X \underset{x,y:p(x,y)}{\bowtie} Y)) \ (\textbf{push})\\
&\quad\equiv\quad X - \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)\vee r(v_Y,z)}{\ltimes} Z) \ (\textbf{unnest})\\
&\quad\equiv\quad X - \pi_X(((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)}{\ltimes} Z) \cup ((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:r(v_Y,z)}{\ltimes} Z))\\
&\qquad(\textbf{split})
\end{aligned}
$$

Compared to the result achieved by means of the reduction algorithm of Codd, instead of two divisions, we have one difference operator. In addition, the result achieved by means of standard rewrit-

ing contains a common subexpression, which is the left operand of two partial joins. Further algebraic rewriting may proceed as follows:

$$
\begin{aligned}
& X - \pi_X(((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)}{\ltimes} Z) \cup ((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:r(v_Y,z)}{\ltimes} Z)) \\
\equiv\ & X - \pi_X(((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)}{\ltimes} Z) \cup (X \underset{x,y:p(x,y)}{\bowtie} (Y \underset{v,z:r(v_Y,z)}{\ltimes} Z))) \\
& (\textbf{associate}) \\
\equiv\ & X - (((X \underset{x,y:p(x,y)}{\ltimes} Y) \underset{v,z:q(v_X,z)}{\ltimes} Z) \cup (X \underset{x,y:p(x,y)}{\ltimes} (Y \underset{v,z:r(v_Y,z)}{\ltimes} Z))) \ (\textbf{push})
\end{aligned}
$$

The common subexpression has disappeared; instead of one regular and two partial joins, we have four partial joins.

Rewriting example B.52 (24), disjunctive, distribution

$$
\begin{aligned}
& \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge (\exists z \in Z \bullet q(x,z) \vee \exists z \in Z \bullet r(y,z))](X) \ (\textbf{distribute})
\end{aligned}
$$

The selection predicate is not in MNF. Below, we show the effect of rewriting the predicate into MNF, using the union to handle disjunction:

$$
\begin{aligned}
& \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \exists z \in Z \bullet q(x,z) \vee r(y,z)](X) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge (\exists z \in Z \bullet q(x,z) \vee \exists z \in Z \bullet r(y,z))](X) \ (\textbf{distribute}) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet (p(x,y) \wedge \exists z \in Z \bullet q(x,z)) \vee (p(x,y) \wedge \exists z \in Z \bullet r(y,z))](X) \\
& (\textbf{distribute } \wedge) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet (p(x,y) \wedge \exists z \in Z \bullet q(x,z)) \wedge \\
& \qquad \nexists y \in Y \bullet (p(x,y) \wedge \exists z \in Z \bullet r(y,z))](X) \ (\textbf{distribute } \nexists) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet (p(x,y) \wedge \exists z \in Z \bullet q(x,z)) \wedge \\
& \qquad \nexists y \in (Y \underset{x,z:r(y,z)}{\ltimes} Z) \bullet p(x,y)](X) \ (\textbf{range nest}) \\
\equiv\ & \sigma[x : \nexists y \in Y \bullet (p(x,y) \wedge \exists z \in Z \bullet q(x,z))](X \underset{x,y:p(x,y)}{\triangleright} (Y \underset{x,z:r(y,z)}{\ltimes} Z)) \\
& (\textbf{unnest}) \\
\equiv\ & \sigma[x : \nexists z \in Z \bullet q(x,z) \vee \nexists y \in Y \bullet p(x,y)](X \underset{x,y:p(x,y)}{\triangleright} (Y \underset{x,z:r(y,z)}{\ltimes} Z)) \\
& (\textbf{descope}) \\
\equiv\ & \sigma[x : \nexists z \in Z \bullet q(x,z)](V) \cup \sigma[x : \nexists y \in Y \bullet p(x,y)](V) \ (\textbf{split}) \\
& \textbf{with } V = X \underset{x,y:p(x,y)}{\triangleright} (Y \underset{x,z:r(y,z)}{\ltimes} Z) \\
\equiv\ & (V \underset{x,z:q(x,z)}{\triangleright} Z) \cup (V \underset{x,y:p(x,y)}{\triangleright} Y) \ (\textbf{unnest}) \\
& \textbf{with } V = X \underset{x,y:p(x,y)}{\triangleright} (Y \underset{x,z:r(y,z)}{\ltimes} Z)
\end{aligned}
$$

The result involves four partial joins and a union, but no division or set difference. In this case, it is difficult to compare the respective results. The result of Codd can be improved easily, but which one of the expressions is the best is hard to tell.

Query 28, disjunction

Rewriting example B.53 (28), disjunctive, Codd

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists y \in Y \bullet \nexists z \in Z \bullet \neg p(x,y) \vee q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \pi_X(\sigma[v : \nexists z \in Z \bullet \neg p(v_X,v_Y) \vee q(v_X,z) \vee r(v_Y,z)](X \times Y))\\
&\quad\equiv\quad \pi_X((\sigma[v : p(v_X,v_Y) \wedge \neg q(v_X,v_Z) \wedge \neg r(v_Y,v_Z)]((X \times Y) \times Z)) \div Z)
\end{aligned}$$

The result of standard rewriting is the expression:

$$\pi_X((X \underset{p}{\bowtie} Y) \underset{q \vee r}{\triangleright} Z)$$

Further algebraic optimization yields a result which presumably cannot be improved:

$$\begin{aligned}
&\pi_X((X \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z) \vee r(y,z)}{\triangleright} Z)\\
&\quad\equiv\quad \pi_X(((X \underset{p(x,y)}{\bowtie} Y) \underset{r(y,z)}{\triangleright} Z) \underset{q(x,z)}{\triangleright} Z) \ (\textbf{split})\\
&\quad\equiv\quad \pi_X(((X \underset{r(y,z)}{\triangleright} Z) \underset{p(x,y)}{\bowtie} Y) \underset{q(x,z)}{\triangleright} Z) \ (\textbf{exchange})\\
&\quad\equiv\quad (((X \underset{r(y,z)}{\triangleright} Z) \underset{p(x,y)}{\ltimes} Y) \underset{q(x,z)}{\triangleright} Z) \ (\textbf{push})
\end{aligned}$$

Distribution yields the same result:

Rewriting example B.54 (28), disjunctive, distribution

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z) \wedge \nexists z \in Z \bullet r(y,z)](X) \ (\textbf{distribute})\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \exists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet r(y,z)](X) \ (\textbf{descope})\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \exists y \in (Y \underset{y,z:r(y,z)}{\triangleright} Z) \bullet p(x,y)](X) \ (\textbf{range nest})\\
&\quad\equiv\quad \sigma[x : \nexists z \in Z \bullet q(x,z) \wedge \exists y \in (Y \underset{y,z:r(y,z)}{\triangleright} Z) \bullet p(x,y)](X) \ (\textbf{range nest})\\
&\quad\equiv\quad (X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\ltimes} (Y \underset{y,z:r(y,z)}{\triangleright} Z) \ (\textbf{unnest})\\
&\quad\equiv\quad ((X \underset{x,z:q(x,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\ltimes} Y) \underset{y,z:r(y,z)}{\triangleright} Z) \ (\textbf{associate})
\end{aligned}$$

Query 28, conjunction

Rewriting example B.55 (28), conjunctive, Codd

$$\begin{aligned}
&\sigma[x : \exists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z) \wedge r(y,z)](X)\\
&\quad\equiv\quad \sigma[x : \exists y \in Y \bullet \nexists z \in Z \bullet \neg p(x,y) \vee (q(x,z) \wedge r(y,z))](X) \ (\textbf{scope})\\
&\quad\equiv\quad \sigma[x : \exists y \in Y \bullet \forall z \in Z \bullet p(x,y) \wedge (\neg q(x,z) \vee \neg r(y,z))](X) \ (\textbf{PNF})\\
&\quad\equiv\quad \pi_X(\sigma[v : \forall z \in Z \bullet p(v_X,v_Y) \wedge (\neg q(v_X,z) \vee \neg r(v_Y,z))](X \times Y)) \ (\textbf{unnest})\\
&\quad\equiv\quad \pi_X(\sigma[v : p(v_X,v_Y) \wedge (\neg q(v_X,v_Z) \vee \neg r(v_Y,v_Z))]((X \times Y) \times Z) \div Z) \ (\textbf{unnest})\\
&\quad\equiv\quad \pi_X(\sigma[v : \neg q(v_X,v_Z) \vee \neg r(v_Y,v_Z)]((X \underset{x,y:p(x,y)}{\bowtie} Y) \times Z) \div Z) \ (\textbf{unnest})\\
&\quad\equiv\quad \pi_X((((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)}{\bowtie} Z) \cup ((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(r_Y,z)}{\bowtie} Z)) \div Z)
\end{aligned}$$

The result of standard rewriting is the expression:

$$\pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)\wedge r(v_Y,z)}{\triangleright} Z)$$

Query 32, disjunction

Rewriting example B.56 (32), disjunctive, Codd

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv \sigma[x : \nexists y \in Y \bullet \nexists z \in Z \bullet \neg p(x,y) \vee (q(x,z) \vee r(y,z))](X) \ (\textbf{scope})\\
&\quad\equiv (\sigma[v : \exists z \in Z \bullet \neg p(v_X,v_Y) \vee (q(v_X,z) \vee r(v_Y,z))](X \times Y)) \div Y \ (\textbf{division})\\
&\quad\equiv (\sigma[v : \neg p(v_X,v_Y) \vee (q(v_X,v_Z) \vee r(v_Y,v_Z))]((X \times Y) \times Z)) \div Y \ (\textbf{unnest})
\end{aligned}$$

The result achieved by standard rewriting is:

$$X - \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)\vee r(v_Y,z)}{\triangleright} Z)$$

Further algebraic rewriting yields:

$$\begin{aligned}
&X - \pi_X((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)\vee r(v_Y,z)}{\triangleright} Z)\\
&\quad\equiv X - \pi_X(((X \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:r(v_Y,z)}{\triangleright} Z) \underset{v,z:q(v_X,z)}{\triangleright} Z) \ (\textbf{split})\\
&\quad\equiv X - \pi_X(((X \underset{v,z:r(v_Y,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\bowtie} Y) \underset{v,z:q(v_X,z)}{\triangleright} Z) \ (\textbf{exchange})\\
&\quad\equiv X - (((X \underset{v,z:r(v_Y,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\ltimes} Y) \underset{v,z:q(v_X,z)}{\triangleright} Z) \ (\textbf{push})\\
&\quad\equiv X - (((X \underset{v,z:r(v_Y,z)}{\triangleright} Z) \underset{x,y:p(x,y)}{\ltimes} Y) \underset{v,z:q(v_X,z)}{\triangleright} Z) \ (\textbf{push})
\end{aligned}$$

Note that the result is literally the complement of that achieved for expression 28. Distribution with bypass processing leads to:

Rewriting example B.57 (32), disjunctive, distribution

$$\begin{aligned}
&\sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z) \vee r(y,z)](X)\\
&\quad\equiv \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet q(x,z) \wedge \nexists z \in Z \bullet r(y,z)](X) \ (\textbf{distribute})\\
&\quad\equiv \sigma[x : \exists z \in Z \bullet q(x,z) \vee \nexists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet r(y,z)](X) \ (\textbf{descope})\\
&\quad\equiv T + \sigma[x : \nexists y \in Y \bullet p(x,y) \wedge \nexists z \in Z \bullet r(y,z)](F)\\
&\qquad \textbf{with } (T,F) = X \underset{x,z:q(x,z)}{\ltimes^2} Z \ (\textbf{bypass})\\
&\quad\equiv T + F \underset{x,y:p(x,y)}{\triangleright} (Y \underset{y,z:r(y,z)}{\triangleright} Z)\\
&\qquad \textbf{with } (T,F) = X \underset{x,z:q(x,z)}{\ltimes^2} Z \ (\textbf{case 29})
\end{aligned}$$

Again, the set difference operator can be avoided by means of bypass processing, which is made possible by distribution of the quantification.

Appendix C

Example cost formulas

We have example expressions:

- $Q_1 = \sigma[v : \exists y \in x.c \bullet p_3(v_X, y) \wedge p_6(y, v_Z)](X \underset{x,z:p_5(x,z)}{\bowtie} Z)$
- $Q_2 = \sigma[x : \exists v \in (x.c \underset{y,z:p_6(y,z)}{\bowtie} Z) \bullet p_3(x, v_Y) \wedge p_5(x, v_Z)](X)$

  which is equivalent to:

  $\sigma[x : \exists v \in x.c \bullet p_3(x, v_Y) \wedge p_5(x, v_Z)](\alpha[x : x \ \mathbf{except} \ (c = (x.c \underset{y,z:p_6(y,z)}{\bowtie} Z))](X))$

For a nested-loop execution method we present cost formulas to compute the cost of the joins, expressed in block accesses, adapted from [ElNa89]. Let n_B denote the number of memory buffers available, b_X denotes the number of disk blocks occupied by X, and bfr_X is the blocking factor of file X, i.e. the number of values (tuples) that fit in one block. Below, the expression $|c|$ represents the average number of tuples present in attribute c per $x \in X$.
We assume that each value y that belongs to attribute c of X fits in one block, and one additional block is needed for the remaining attribute values of X, so the blocking factor of X is $1/1 + |c|$. The blocking factor of Z is assumed to be 1, so the blocking factors of the join results are $bfr_{XZ} = 1/2 + |c|$, $bfr_{cZ} = 1/2$. Also, $b_X = |X| + |X| \cdot |c|$, $b_Z = |Z|$.

Query	Read outer	Read inner	Write result
Q_1	b_X	$(b_X/n_B - 1) \cdot b_Z$	$(js_1 \cdot |X| \cdot |Z|)/bfr_{XZ}$
Q_2	b_X	$(b_X/n_B - 1) \cdot b_Z$	$|X| \cdot ((1 + (js_2 \cdot |c| \cdot |Z|)/bfr_{cZ})$

The blocking factors bfr of the join results are respectively $bfr_{XZ} = 1/12$, and $bfr_{cZ} = 1/2$. Let the number of buffers n_B be 6, and assume that in the joins each left-hand operand tuple joins with exactly one right-hand operand tuple, so the join selectivities are $js_1 = 1/|Z|$, and $js_2 = 1/|Z|$, respectively. Let $|X| = 100$, $|c| = 10$, and $|Z| = 100$.

Query	Read outer	Read inner	Write result
Q_1	$100 + (100 \cdot 10) =$ 1100	$(1100/5) \cdot 100 =$ 22000	$100/(1/12) =$ 1200
Q_2	1100	22000	$100 \cdot ((1 + 10/(1/2)) = 2100$

The costs for reading are the same for both queries. The costs for writing Q_2, and the difference between the two becomes larger as the number of tuples in c grows. Data replication then becomes more and more diadvantageous.
Note that for query Q_1 we assumed that the outer loop iterates over X. If we take Z as the outer loop operand, the costs of expression Q_1 become even smaller:

Query	Read outer	Read inner	Write result
Q_1	b_Z	$(b_Z/n_B - 1) \cdot b_X$	$(js_1 \cdot \|X\| \cdot \|Z\|)/bfr_{XZ}$
Q_1	100	$(100/5) \cdot 1100 = 22000$	1200

For Q_2, we are forced to take table X as the outer loop operand.

References

[ElNa89] Elmasri, R. and S.B. Navathe, **Fundamentals of Database Systems**, Benjamin/Cummings Publishing Company Inc., 1989.

Symbol Index

Concept Index