USE OF THESES

This copy is supplied for purposes of private study and research only. Passages from the thesis may not be copied or closely paraphrased without the written consent of the author.
The author acknowledges the valuable contribution of academic supervisor Jeffrey Xu Yu; however, the results of this thesis are entirely based on the author's original research. Extended abstracts of Chapter 3 and Chapter 4 have appeared in [Liu and Yu 1998] and [Liu and Yu 1999], which were jointly prepared with Jeffrey Xu Yu based on research undertaken by the author. A preliminary version of Chapter 6 has appeared in [Liu and Chirathamajaree 1996]. This was jointly prepared with Chirathamajaree Chaiyaporn again based entirely on the author's own work. The contents of Chapter 5 and Chapter 7 have appeared in [Liu and Ramamohanarao 1994b], [Liu and Ramamohanarao 1994c] and [Liu and Ramamohanarao 1994a] and are based on research carried out and prepared solely by the author.

Except where otherwise indicated, this thesis is my own original work.

Hong-Cheu Liu
21 October 2000
This thesis is dedicated to my father, Shiang-Jiou Liu,
for his support and belief in me.
Acknowledgments

It is a great pleasure for me to acknowledge the assistance and contributions of a number of individuals to my PhD work. First, I would like to thank my supervisor, Dr Jeffrey Xu Yu, for his supervision and support during my PhD program at The Australian National University. He always encouraged me to be enthusiastic and to develop as a good researcher. I am sincerely grateful to him for his time and patience.

I would like to express my gratitude to my advisors, Dr Brian Molinari and Dr Vicki Peterson, for their many helpful comments and fruitful discussions on a draft version of this thesis. They spent much time carefully reading all chapters of the thesis. Their suggestions have greatly helped to improve the quality and presentation of this thesis. I am deeply indebted for their efforts and suggestions.

Many thanks go to Nick Craswell and Gail Craswell who helped me in the writing of this thesis. Ms Gail Craswell proof-read my papers and some chapters of this thesis. Nick Craswell suggested various improvements to the thesis. Lynda Lawson also assisted with corrections.

I am grateful for the support and assistance I received from colleagues at the Department of Computer Science. In particular, I would like to thank Weifa Liang, Steve Blackburn, Linda Wallace, Zhen He, Richard Walker and Hongxue Wang for their help in Latex questions, paper retrieval and other assistance.

I would also like to acknowledge the financial support of an Australian National University Postgraduate Scholarship. The Department of Computer Science, led by Dr Chris Johnson, provided a warm and excellent working environment including an enthusiastic technical service support group.

Finally, I would like to gratefully thank my wife and sons for their support, patience and encouragement. Most of all, I would like to express my sincere gratitude to the continuing support of my parents.
Abstract

This thesis considers the theory of database queries on the complex value data model extended with external functions. In modern intelligent database systems, we expect that query systems be able to handle a wide range of calculus formulas correctly and efficiently. Accordingly, they will require general query translators and efficient optimisers. Motivated by these concerns, this thesis undertakes a comprehensive study of query evaluation in the complex value model and investigates the following issues:

- identifying recursive sets of complex value formulas which define domain independent queries;
- implementing complex value calculus queries with the incorporation of functions;
- solving the problem of how to process join operation in complex value databases; and
- investigating some algebraic properties concerning nested relational operators.

The first part of this thesis extends some classical properties of the relational theory - particularly those related to query safety - to the context of complex value databases with fixed external functions and investigates the problem of how to implement calculus queries. Two notions of syntactic criteria for queries which guarantee domain independence, namely, embedded evaluable and embedded allowed, are generalised for this data model. This thesis shows that all embedded-allowed calculus (or fix-point) queries are external-function domain independent and continuous.

This thesis discusses the topic of “embedded allowed database programs” and proves that embedded allowed stratified programs satisfying certain constraints are embedded domain independent. It also develops an algorithm for translating embedded allowed queries into equivalent algebraic expressions as a basis for evaluating safe queries in all calculus-based query classes.
The second part of this thesis considers the issue of query optimisation for nested relational databases. Within a restricted set of nested schema trees, a join operator, called P-join, is proposed. The P-join operator does not require as many restructuring operators and combines the advantages of the extended natural join and recursive join for efficient data access. A P-join algorithm which takes advantage of a decomposed storage model and various join techniques available in the standard relational model to reduce the cost of join operation in nested relational databases is also proposed.

Finally, this thesis investigates some algebraic properties of nested relational operators which are useful for query optimisation in the nested relational model and outlines a heuristic optimisation algorithm for nested relational expressions by adopting algebraic transformation rules developed in this thesis and previous related work.
# Contents

Acknowledgments vii  
Abstract ix  
List of Figures xv  

## 1 Introduction  
1.1 Query Processing in the Standard Relational Model 2  
1.2 Query Processing in Extended Relational Models 5  
1.3 Related Work 7  
1.4 Overview of the Thesis 11  

## 2 Complex Value Databases and Query Languages 15  
2.1 Complex Value Databases 15  
2.2 Query Languages 18  
2.2.1 A Complex Value Algebra 18  
2.2.2 A Complex Value Calculus 20  
2.2.3 The Calculus (CALC\textsuperscript{cv}) + Fix-point 23  
2.2.4 Rule-based Languages for Complex Values 24  
2.3 Domain-Independent Computable Queries 25  
2.3.1 Notion of Genericity 26  
2.3.2 Domain Independence 26  
2.3.3 Computable Queries 27  
2.4 Query Translation 27  
2.5 Query Optimisation 30  

## 3 Domain-Independent Queries with External Functions 31  
3.1 Basic Concepts 32
## Contents

3.2 Evaluatable Queries .................................................. 35
   3.2.1 Embedded Evaluatable Formulas .......................... 36
   3.2.2 Embedded Allowed Formulas .............................. 39
3.3 Domain-independent Database Programs ......................... 42
3.4 Em-allowed, Finiteness and Domain Independence ............. 45

4 Translation of Complex Value Calculus Queries ................. 47
   4.1 Removing Universal Quantifiers and Renaming Variables ...... 48
   4.2 Transforming a Formula into Existential Normal Form ....... 49
   4.3 Transforming an ENF Formula into Complex Value Algebra Normal Form 55
   4.4 Transforming $\text{ALG}^{cn}\text{NF}$ Formulas into Algebra Expressions .... 60
   4.5 Proofs of Theorems 3.2, 3.3 and 3.4 .......................... 65

5 A Join Operator for Complex Value Databases .................... 69
   5.1 Basic Concepts .................................................. 70
      5.1.1 Nested Schema Tree .................................... 70
      5.1.2 Selection Operator and Projection Operator .......... 71
   5.2 The Path Join .................................................. 73
      5.2.1 Extended Cartesian Product and Path Cartesian Product .... 74
      5.2.2 P-join Operator ......................................... 77
   5.3 A Comparison of the P-join and Other Join Operators ....... 81
   5.4 Algebraic Equivalences ....................................... 83
   5.5 Decomposition P-join Operator .............................. 88
   5.6 Correctness of Decomposition P-Join ........................ 93

6 Implementation of the P-Join ...................................... 97
   6.1 Computation Mechanism for P-join ........................... 98
      6.1.1 Basic Assumptions ..................................... 98
      6.1.2 Storage Model .......................................... 99
      6.1.3 Mechanism of Computing P-join ........................ 99
      6.1.4 Size Estimate of P-join Output ....................... 102
   6.2 Cost Estimate of Computing P-join .......................... 102
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Representation of complex value</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>A database instance</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>The overall definition of the function $bd$</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of nested relation and nested schema tree</td>
<td>71</td>
</tr>
<tr>
<td>5.2</td>
<td>Examples of selection and projection</td>
<td>73</td>
</tr>
<tr>
<td>5.3</td>
<td>An example of extended Cartesian product</td>
<td>75</td>
</tr>
<tr>
<td>5.4</td>
<td>An example of path Cartesian product</td>
<td>77</td>
</tr>
<tr>
<td>5.5</td>
<td>An example of $\theta$ P-join</td>
<td>78</td>
</tr>
<tr>
<td>5.6</td>
<td>Natural P-join between $r$ and $q$</td>
<td>79</td>
</tr>
<tr>
<td>5.7</td>
<td>The schemas of $r$, $q$ and $r \bowtie q$</td>
<td>80</td>
</tr>
<tr>
<td>5.8</td>
<td>Natural P-join between $r$ and $s$</td>
<td>80</td>
</tr>
<tr>
<td>5.9</td>
<td>The schemas of $r$, $s$ and $r \bowtie s$</td>
<td>81</td>
</tr>
<tr>
<td>5.10</td>
<td>Relations $r_1$, $r_2$ and the result of the query in Example 5.7</td>
<td>82</td>
</tr>
<tr>
<td>5.11</td>
<td>Nested relations $r_1$, $r_2$ and $r_3$</td>
<td>88</td>
</tr>
<tr>
<td>5.12</td>
<td>The result of the expression 5.2 in Example 5.9</td>
<td>88</td>
</tr>
<tr>
<td>5.13</td>
<td>An example of decomposition P-join between $r$ and $q$</td>
<td>89</td>
</tr>
<tr>
<td>6.1</td>
<td>The storage structures of $R$ and $S$.</td>
<td>100</td>
</tr>
<tr>
<td>6.2</td>
<td>The relation Temp</td>
<td>101</td>
</tr>
<tr>
<td>6.3</td>
<td>The output of P-join</td>
<td>101</td>
</tr>
<tr>
<td>7.1</td>
<td>Examples of $\cup^e$ and $\cap^e$</td>
<td>111</td>
</tr>
<tr>
<td>7.2</td>
<td>Examples of $\bowtie^e$ and $\pi^e$</td>
<td>112</td>
</tr>
<tr>
<td>7.3</td>
<td>Product database</td>
<td>122</td>
</tr>
</tbody>
</table>
Database Management Systems (DBMSs) are being widely used to support new applications such as engineering design, image and voice data management and spatial information systems. In order to model these applications the complex value data model has been proposed as a significant extension of the relational one and in the last decade much research has been carried out in this areas. Complex values (nested relations, complex objects) are increasingly parts of these advanced database systems. Intuitively, complex values are relations in which the entries may be themselves tuples or relations. In this data model, we use hierarchical structures rather than flat tables to enable the representation of complex objects. Complex values allow the application of the tuple and set constructor recursively. A first normal form relation is a special kind of complex value.

The complex value data model lacks key features of the object-oriented paradigm such as objects and inheritance. However, it provides the core structure of object-oriented (object-relational) databases and comprises an important component of semantic models. Therefore languages that access such complex structures are important, and the capability and performance of query systems are worthy of study.

As in the case of the relational data model, query languages for complex values have been developed from three paradigms: extensions of relational calculus [Abiteboul and Beeri 1995; Jacobs 1982; Kuper and Vardi 1993; Roth et al. 1988]; extensions of relational algebra [Abiteboul and Bidoit 1986; Abiteboul and Beeri 1995; Kuper and Vardi 1984; Thomas and Fischer 1986; Schek and Scholl 1986]; and deductive languages [Abiteboul and Grumbach 1991; Beeri et al. 1987; Kuper 1988; Kuper 1990]. They all use higher order types. The variety of features and operations found in these languages
is quite complicated. Previous investigations of complex value query languages have focused on equivalent expressive power and complexity of various query classes, various means for obtaining expressive and tractable languages for complex value databases. However, the work done in the database community on query translation and optimisation for the complex value model is insufficient and needs further refinement.

1.1 Query Processing in the Standard Relational Model

A query that is expressed in a high-level, declarative style query language, such as SQL or QUEL, must first be scanned, parsed, and validated. An internal representation of the query is then created, usually as a tree or a graph expressed by relational algebra. The database system must then devise an execution strategy for retrieving the result of the query from the internal database files.

Relational calculus is declarative and suitable for user interface while relational algebra is procedural and suitable for database operations. The process of internally translating calculus-based expressions written by the user or generated by a layer of software into algebra expressions is called query translation. The query translator in modern sophisticated systems will handle software-generated queries which will surely be non-conjunctive and much more complex than the large majority of queries, falling into the class of conjunctive queries, posed by typical users of traditional databases. This translator is also able to distinguish reasonable queries which can be answered sensibly from unreasonable ones [Van Gelder and Topor 1991].

In the database field, to answer sensibly means that values of any correct answer lie within the active domain of the query or the input [Topor 1987]. There are only certain calculus queries (or formulas) which can be regarded as reasonable in this sense. Such queries are called domain independent as they yield the same answer no matter what the underlying domain of interpretation. The following are examples of unreasonable query phenomenon.

1. \( \{x|\neg \text{Movies("Cries and Whispers", "Bergman", } x)\} \)

2. Given two relations \( R(w, y) \) (\( w \) requires \( y \)) and \( S(x, y) \) (\( x \) supplies \( y \)). The ques-
§1.1 Query Processing in the Standard Relational Model

The set of correct answers for each of the above queries depends on the domains of the variables.

Many researchers have tackled these problems, including [Beeri and Milo 1997; Demolombe 1992; Escobar-Molano et al. 1993; Fagin 1982; Hull and Su 1994; Ramakrishnan et al. 1987; Topor 1987; Van Gelder and Topor 1991; Ullman 1982]. There have been several attempts to semantically or syntactically characterise the set of formulas that ensure domain independent [Abiteboul et al. 1995; Demolombe 1992; Escobar-Molano et al. 1993; Fagin 1982; Topor 1987; Ullman 1982; Ullman 1988; Van Gelder and Topor 1991]. The class of domain independent formulas defined by Fagin [1982] is the largest class having the property that the constants in the database and the query provide a sufficient domain for the values in the answer. This class was shown to be not solvable by Vardi [1981].

The power of query languages is based on the fact that they express a restricted class of declarative programs. Although relational calculus queries may not return finite results, as illustrated in the above examples, a natural subclass of the relational calculus does, namely, the class of range-restricted queries. This class gives guarantees of finite output and is complete in this respect: it captures all relational calculus queries whose outputs are always finite (the safe queries). The class of safe formulas, introduced by Ullman [1982], characterises those relational calculus formulas that have equivalent relational algebra expressions. A syntactic definition of safe appears in Ullman’s revised edition [Ullman 1988].

If the answers to relational calculus queries are possibly infinite sets or even undefined, it will be difficult to provide a well-defined procedure for translating user queries into equivalent relational algebra queries which can be evaluated by the underlying relational database systems. In the past, Van Gelder and Topor [1991] identified such problems in SQL and QUEL.

The practical integration of user-defined functions in the relational algebra is rela-
tively straightforward. However, it is significantly more difficult to support this in the relational calculus. The main reason is that the semantic of relational algebra queries evaluated on a database instance is independent of the underlying domains while the relational calculus is domain sensitive. Therefore, the semantics of calculus queries in the presence of functions need further investigation.

A query typically has many possible execution strategies, and the process of choosing a suitable execution strategy for processing a query is known as query optimisation.

A high-level relation query is more declarative in nature; it specifies what the intended result of the query is rather than the details of how the result should be obtained. The query optimiser is responsible for translating such high-level queries into equivalent queries or machine instruction programs that are arguably more efficient than a naive execution of the initial query.

A myriad of factors affect query processing, including storage and indexing techniques, page sizes and paging protocols, access routines, information about the stored data, statistic properties of anticipated queries and updates, and so on [Abiteboul et al. 1995]. Query optimisation can be performed at all levels of the three-level database architecture. Because of the myriad factors that play a role in query evaluation, most practically successful techniques rely heavily on heuristics.

There are two main techniques for query optimisation. The first technique, heuristic optimisation, is often referred to as query rewriting. Heuristic rules are applied to modify the internal representation of a query - which is usually in the form of a tree or a graph data structure - to improve its expected performance on execution. The rules typically reorder the operations in a query execution strategy. The second technique - systemic optimisation using cost estimates - is often referred to as plan generation. The query optimiser, possibly after using some heuristics, will estimate and compare the costs of different execution strategies and choose the execution plan with the lowest cost estimate. The two techniques are usually combined to some extent in a query optimiser.
1.2 Query Processing in Extended Relational Models

Most basic issues concerning object-oriented (or object-relational) databases, such as the design of query languages or the analysis of their expressive power, can be largely resolved using techniques already developed in connection with standard relational and complex value models. However, the presence of new features (such as object identifiers, typing, method calls) bring new issues and techniques. In particular, query processing is seriously affected by these new features and needs to be further investigated.

A DBMS provides data storage but data processing is provided by a host programming language with a relatively simple query language such as SQL embedded in it. Therefore, it is desirable to provide effective communication between the underlying programming language and the database query language. In particular, functions must be first-class citizens in query languages.

Support for both complex values and user-defined functions is important in a DBMS. In response to these requirements, SQL3 generalizes the relational model into an object model offering abstract data types and therefore allows users to define data types which suit their applications. Tables may then contain collections of objects. As for supporting the operational behaviours of any user defined type, query languages are extended in such a way that user-defined functions can be registered in a DBMS such as Informix and Postgres. For example, the following SQL3 statement [Stonebraker and Brown 1998] registers a function called "vesting" on a data type "date".

```
create function vesting (date) returns float as external name 'foo'
language C;
```

The function returns a float value and is written using the programming language C. The function has been compiled and is kept in the file foo. The DBMS can then use any user-defined types and functions in the same way as built-in types and functions. Alternatively, programming languages such as PASCAL/R and Persistent Java extend imperative languages to incorporate access to a particular database model.
The practical integration of external functions in query languages is generally well developed and understood, but the semantics of queries in the presence of external functions needs to be further investigated. Some attempts to extend the definition of domain independence for the complex value data model incorporated with external functions have already been made [Abiteboul and Beeri 1995; Suciu 1995]. Abiteboul and Beeri [1995] explored the incorporation of arbitrary interpreted functions and predicates into query languages for complex objects. They defined the notion of bounded-depth domain independence and showed that, with extended interpreted functions and predicates, the algebra, the bounded-depth domain independent calculus, the safe calculus and the Datalog-like language have equivalent expressive power. Suciu [1995] proposed a notion of domain independence (called external-function domain independence) for queries with external functions which can also be applied to query languages with fix-points or other kinds of iterators. None of these attempts, however, address the aspects of calculus queries with fix-points and deductive database programs. The first goal of this thesis is to identify recursive sets of higher order (complex value) formulas which define domain independent queries, investigate the notion of domain independent deductive database programs and implement domain independent queries.

Much theoretical and application research on query optimisation are still being carried out in various extended relational data models. The query optimiser is the heart of DBMS performance and must also be extended with knowledge about how to execute user-defined functions efficiently, take advantages of new index techniques, examine new query processing heuristics, and navigate among data using references. Object-oriented (object-relational) databases remedy the significant disadvantages of the standard relational model by borrowing a variety of data structuring construct from the complex value model. Query processing in the complex value model is very similar to that in object-oriented databases. For this reason, this thesis focuses on query optimisation in complex value databases. However, it is expected that the results in this thesis can be applicable to object (object-relational) databases with minor modification.

As in the relational model, the join operator is one of the most expensive and critical issues in nested relational query processing.
Example 1.1 Consider the following database which has nested relations Product and Part.

\[
\text{Product} = (\text{prodname}, \text{Warranty}(\text{premium}, \text{country}, \text{w-period}), \text{Composition}(\text{c-name}, \text{c-id}, \text{Parts}(\text{p-name}, \text{quantity})), \text{Distributor}(\text{company}, \text{fee}))
\]

\[
\text{Part} = (\text{p-name}, \text{weight}, \text{Warranty}(\text{country}, \text{w-period}), \text{Source}(\text{company}, \text{cost}))
\]

Consider also the following query.

Find those countries and the corresponding w-periods which are the same in some product and one of its parts, together with companies that are both a product distributor and a part source (Presumably, this information might be used to reduce premium). Group the findings on prodname and p-name.

Due to the fact that the intending join attributes “company”, “p-name” and “w-period” appear on different subschema levels, this query cannot be expressed by join operators proposed so far without restructuring operations. When the algebra expression of queries includes restructuring operators, they are not easily optimised. The second goal of this thesis is to solve the problem of how to optimise queries which include join operations in the nested relational model.

In modern intelligent database systems, we expect that query systems be able to handle a wider range of calculus formulas correctly and efficiently. Accordingly, they will require more general query translators and optimisers. This thesis undertakes a more comprehensive study of query evaluation in the complex value model.

1.3 Related Work

This section briefly reviews the research literature relating to this thesis. The evaluable formulas, originally proposed by Demolombe [1992] and discussed by Van Gelder and Topor [1991] in the context of standard relational databases, comprise the largest decidable subclass of the domain independent formulas. Demolombe [1992] proved that evaluable formulas are domain independent and showed that evaluable formulas have
the same expressive power as domain independent formulas. The allowed formulas proposed by Topor [1987] are a strict subclass of the evaluable formulas. Van Gelder and Topor [1991] investigated the properties of two such classes and developed algorithms to transform an evaluable formula into an equivalent allowed formula and from there into relational algebra. This thesis generalises the notions of evaluable and allowed for the complex value model.

Topor [1991] investigated the algebra, calculus and Datalog languages in which the underlying domain is partitioned into two sorts, namely, the sort \( \mathbb{Z} \) holding the integers and the sort, \( \text{dom} \), ranging over a disjoint set of uninterpreted constants. This thesis also considers query evaluation using domains based on subsets of \( \mathbb{Z} \) whereas Topor [1991] did not. Therefore, the notion of Topor’s domain independence is different from our notion of domain independence.

The notion of safe calculus queries developed in [Abiteboul and Beeri 1995] is based on “range-restriction” of variables occurring in calculus formulas. Each variable is attached to a range formula. The positive literal \( R(x), x \in t, x = t, x \subseteq t \) and \( \forall y (y \in x \rightarrow \varphi(y)) \), and formulas obtained from them by using \( \land, \lor \) are called range formulas. For example, \( R(z) \land y \in z \) restricts \( z \) and \( y \); if \( \varphi_1(x_1), ..., \varphi_n(x_n) \) restrict \( x_1, ..., x_n \) respectively, then \( \varphi_1(x_1) \land ... \land \varphi_n(x_n) \land f(x_1, ..., x_n) = y \) restricts \( y \).

Abiteboul and Beeri [1995] also presented a sketch showing the translation from safe calculus into algebra. This translation is based on associating, with each subformula, “range-restriction” for the free variables that are occurring in the formula. The translation is motivated from a primarily theoretical perspective whereas this thesis adopts the finiteness dependency approach which influences the choice of some translation steps in the algorithm. The issue of the order in which the conjunctions are evaluated is also considered in this thesis. The translation of [Abiteboul and Beeri 1995] is potentially less efficient than the algorithm developed in this thesis which adopts a heuristic to simplify the computation involving finite dependencies required to implement the translation.

Escobar-Molano, Hull, and Jacobs [1993] introduced the notion of embedded allowed, which generalised the “allowed” criteria to incorporate scalar functions and developed an algorithm for translating these embedded allowed queries into the rela-
§1.3 Related Work

tional algebra. The notion of range-restriction is weaker than the notion of embedded allowed. The notion of embedded domain independence developed in [Escobar-Molano et al. 1993] is restriction to the flat relational case of the notion of bounded depth domain independence in [Abiteboul and Beeri 1995]. Both notions are used only in conjunction with query languages without recursive queries (or any other kind of iterations). This thesis generalises the notion of embedded allowed for (recursive) queries for the complex value model. Their translation framework used the notion of finiteness dependencies [Ramakrishnan et al. 1987] which is analogous to functional dependencies and carries information about how sub-formula involving scalar functions can restrict the possible range of variables. This thesis also develops an algorithm for translating embedded allowed queries into equivalent algebra expressions as a basis for evaluating safe queries in all (complex value) calculus-based query classes.

The notion of external-function domain independence, proposed by [Suciu 1995], generalises those of generic and domain independent queries on databases without external functions and can also be applied to query languages with fix-points or other kinds of iterations. Suciu [1998] showed that all queries in an nested relational algebra language over a set of external functions, $\mathcal{NRA}(\Sigma) + fix$, are external-function domain independent and continuous, while this thesis shows that all “embedded allowed formulas” are external-function domain independent.

Beeri and Milo [1997] studied the issue of safe calculus and their translation to an algebra based on the perspective of algebraic specifications. The notion of “strict DB-domain independent” defined there is similar to the notion of embedded domain independent in [Escobar-Molano et al. 1993]. Their paper also provided a definition of “safe” calculus queries, showed that they are strict DB-domain independent and indicated how to translate these into algebra queries. However, the notion of safe used in [Beeri and Milo 1997] is strictly weaker than “embedded allowed” developed in [Escobar-Molano et al. 1993]. Also the translation is potentially less efficient than the algorithm developed in this thesis as it will in some cases involve a construction of the active domain of the input instance while the algorithm in this thesis avoids to use it.

This thesis extends the above notions for the complex value data model and explores the issue of the incorporation of external functions into query languages.
There is a growing interest among database researchers in constraint databases which generalise relational databases by finitely representable infinite relations [Kanelakis et al. 1995; Benedikt et al. 1998]. Each constraint relation is a quantifier-free first-order DNF (disjunctive normal form) formula in some constraint theory. Each disjunct of the DNF formula is a constraint tuple. The constraint data model and query languages provide an alternative approach to resolving the problem of safety, that permits query to have answers that are infinite but finitely representable. However, there is an analogy of the closed-form evaluation problem as in un-constraint setting, i.e., the class of constraints used in output queries should be the same as the class used to define input databases. This closed-form requirement has been difficult to meet in constraint query languages that contain the negation symbol. Many researchers have attempted to tackle this safe issue in constraint query languages [Revesz 1998; Benedikt and Libkin 1998].

The literature relating to query optimisation is now briefly reviewed. Roth, Korth, and Silberschatz [1988] defined an extended relational calculus as the theoretical basis for their nested database query language, defined a minimal extended relational algebra and proved its equivalence to the extended relational calculus. They also defined a class of nested relations, called partition normal form, with certain good properties and extended their algebra operators to work within this domain. However, the extended natural join defined in [Roth et al. 1988] limits the relations that can participate in the join to those whose only common attributes are elements of the top level schema. This thesis proposes a new join operator which combines the benefits of the extended natural join and recursive join for efficient data retrieval.

Colby [1990] developed a recursive algebra for nested relations that allows tuples at all levels of nesting in a nested relation to be accessed and modified without any special navigational operators and without having to flatten the nested relation. Colby also showed that most of the query optimisation techniques that have been developed for relational algebra can be easily extended for this recursive algebra. However, the operators of this algebra can only be applied at a fix level of a nested relation. For example, a relation can be joined to another relation at a fixed level by using standard join operator. The join operator proposed in this thesis can combine two nested relation
schemata at multi-levels (see Chapter 5).

1.4 Overview of the Thesis

This thesis considers a number of issues in query translation and optimisation for complex value databases. These include

- identifying recursive sets of complex value formulas which define domain independent queries;

- extending the notion of embedded allowed formula to a higher order logic (complex value calculus) with fix-points, such that all embedded allowed formulas define a domain independent, computable query;

- implementing complex value calculus queries with the incorporation of functions;

- solving the problem of how to efficiently express and optimise queries which include join operations in the complex value model; and

- designing algorithms for query optimisation in the complex value databases.

The first part of this thesis explores the issue of the semantics of complex value calculus queries in the presence of functions, and investigates the problem of how to implement complex value calculus queries. Chapter 2 gives a brief review of the main theoretical foundations and results from database systems which are used in this thesis. The review begins with some basic concepts from complex value databases and query languages. Several topics are discussed including notions of genericity, domain independence and computable queries. Finally query evaluation is briefly reviewed and the terminology is presented concerning query translation and optimisation.

Chapter 3 explores how external functions can be incorporated into the complex value queries to provide the power needed to express such queries in advanced database systems. Adding external functions in the calculus paradigm raises some serious problems that require effort and care to resolve satisfactorily. This chapter is developed in two stages. The first stage is to examine the problematic issue of “safety” and domain
independence in the context of the complex value model extended with external functions and develop some solutions for them. The second stage is to investigate the important problem of finding syntactic restrictions on the database programs (Datalogcv) that ensure domain independence. Finally, the chapter presents a brief discussion concerning the relationship between finite and domain independence: The main results in Chapter 3 have appeared in [Liu and Yu 1998; Liu and Yu 1999].

Chapter 4 describes a procedure to translate any embedded allowed formula into an equivalent algebra expression, by progressing through two forms, Existential Normal Form (ENF) and Complex Value Algebra Normal Form (ALGcvNF).

The translation procedure has four phases.

1. Replace all sub-formulas of the form $\forall y(y \in x \rightarrow \varphi(y))$ with $x \subseteq \{ y \mid \varphi(y) \}$, replace any remaining sub-formula of the form $\forall \varphi$ by $\lnot \exists \lnot \varphi$ and rename the quantified variables if necessary.

2. Transform the embedded allowed formula into ENF.

3. Transform the ENF formula into ALGcvNF.

4. Translate the ALGcvNF formula into an equivalent algebra expression.

Our goal in defining ALGcvNF is to ensure that each sub-formula that we need to evaluate is finite. An abstract of the work presented in Chapter 4 has appeared in [Liu and Yu 1999].

The second part of this thesis considers query optimisation issues in the complex value model. The join operation is one of the most critical issues in nested relational query processing. Chapter 5 introduces a new join operator (P-join) which merges two schema trees and combines the advantages of the extended natural join [Roth et al. 1988] and recursive join [Colby 1990] for efficient data access. The algebra expressions of queries, when expressed using P-join, are more succinct and more easily optimised than they are when using other join operators. This chapter also presents a series of algebraic equivalences which are useful for query optimisation in the nested relational model. This work offers a solution to the problem of how to efficiently process queries.
which include join operations in the nested relational model. The contents of Chapter 5 have appeared in [Liu and Ramamohanarao 1994b; Liu and Ramamohanarao 1994c].

Chapter 6 proposes an algorithm for computing the P-join and estimates the cost of using various join techniques developed in relational database systems. The complexity of the P-join algorithm is not more than other join algorithms with expensive restructuring operators involved and additional block shuffle for reading unnecessary data files. The contents of Chapter 6 have appeared in [Liu and Chirathamajaree 1996].

Chapter 7 considers the issue of algebraic optimisation which is both theoretically and practically important for query processing. This chapter investigates some algebraic properties concerning the nested relational operators, and outlines a heuristic optimisation algorithm for nested relational expressions by adopting algebraic transformation rules developed in this chapter and previous related work. The results of Chapter 7 have appeared in [Liu and Ramamohanarao 1994a].

Finally, the last chapter concludes this thesis by summarising the author's research work and presenting some open problems.
Chapter 2

Complex Value Databases and Query Languages

This chapter presents terminology for those previously studied concepts including complex values and their types, query languages, domain independence, and the notion of query translation and optimisation.

2.1 Complex Value Databases

The data structure in the relational model can be viewed as the result of applying to atomic values two basic constructors: a tuple constructor and a set constructor, to make instances of a relation (sets of tuples). The complex value data model is formed using the tuple and set constructors recursively. A fundamental characteristic of such complex values is that, in them, sets may contain members with arbitrarily deep nesting of tuple and/or set constructors.

This section briefly reviews the notions of complex values, complex value data types, and nested relation schema.

We assume the existence of a countably infinite set $\text{att}$ of attributes. Different attributes should have distinct domains. As the nature of the elements of the domains is irrelevant to our theoretical development in this thesis, it suffices to use the same domain of values for all of the attributes. Thus we now fix a countably infinite set $\text{dom}$, called the underlying domain. The elements of this domain are called atomic values. Complex values are constructed from them using the constructors.

We also assume a countably infinite set $\text{rename}$ of relation names disjoint from...
Complex Value Databases and Query Languages

the previous sets. We associate a sort (i.e., type) with each relation name and each (complex) value. The family of complex value data types is defined recursively from the underlying domain \( \text{dom} \) and the tuple and set constructors.

At the schema level, we specify a set of complex sorts (or types). These indicate the structure of the data. The following two definitions are adopted from [Abiteboul et al. 1995].

**Definition 2.1** The abstract syntax of sorts is given by

\[
\tau = \text{dom} | < B_1 : \tau, ..., B_k : \tau > | \{ \tau \},
\]

where \( k \geq 0 \) and \( B_1, ..., B_k \) are distinct attributes. A sort is flat if it has the form \( < B_1 : \text{dom}, ..., B_k : \text{dom} > \).

We assume that there is a function \( \text{sort} \) from \( \text{relname} \) to \( \mathcal{P}^{\text{fin}}(\text{att}) \) (the finitary powerset of \( \text{att} \)). The sort of a relation name is simply \( \text{sort}(R) \).

**Definition 2.2** The interpretation of a data type \( \tau \) (i.e., the set of values of \( \tau \)), denoted \( [\tau] \), is defined recursively as follows:

1. \( [\text{dom}] = \text{dom} \),
2. \( [[\tau]] = \mathcal{P}^{\text{fin}}([\tau]) = \{ X | X \subseteq [\tau] \text{ and } X \text{ finite} \} \), and
3. \( [< B_1 : \tau_1, ..., B_k : \tau_k >] = [\tau_1] \times \cdots \times [\tau_k] = \{ < B_1 : \nu_1, ..., B_k : \nu_k > | \nu_j \in [\tau_j], j \in [1, k] \} \).

**Example 2.1** Consider the sort of a complex value relation

\[
R: \{ < A : \text{dom}, B : \text{dom}, C : \{ < A : \text{dom}, E : \{ \text{dom} \} > \} > \}.
\]

A value of this sort is \( \{ < A : a, B : b, C : \{ < A : c, E : \{ e \} >, < A : d, E : \{ \} > \} >, < A : e, B : f, C : \{ \} > \} \). This value is shown in Figure 2.1.

A (complex value) relation of sort \( \tau \) is a finite set of values of sort \( \tau \) - that is, a finite subset of \( [\tau] \). A complex value database is an extension of a relational database. It consists of relations among complex objects of specified types.
Definition 2.3 A (complex value) relation schema is a relation name with an associated sort, i.e., an expression $R[T_1, ..., T_n]$, where $R \in \text{rename}$ and $T_1, ..., T_n$ are sorts. A (complex value) database schema is a sequence $R = < R_1 : \tau_1, ..., R_n : \tau_n >$, where

1. $R_i \in \text{rename}$ for $i \in [1...n]$;
2. $\tau_i$ is a sort for $i \in [1...n]$; and
3. $R_i \neq R_j$ if $i \neq j$.

An instance $r$ of a relation schema $R$ is a finite subset of $\text{sort}(R)$. An instance over a database schema $R$ is a mapping associating with every relation schema in $R$ an instance of that relation schema.

The principal variation of the complex value model is the nested relation model, which is to be discussed in Chapter 5. For nested relations, set and tuple constructors are required to alternate. A more fundamental constraint is imposed in the so-called Verso-relations [Scholl et al. 1989]. A Verso-relation is a nested relation such that each component may itself be a nested relation but at least one of them must be atomic. A further assumption for Verso-relations is that for each set of tuples, the atomic attributes form a key. The definition is given below.

Definition 2.4 Let $R$ be a nested relation schema with attributes $\text{att}_R$ containing atomic attributes $A_1, ..., A_k$ and non-atomic attributes $X_1, ..., X_l$. An instance $r$ over
schema $R$ is in partitioned normal form (PNF) if and only if the following two conditions hold$^1$:

1. $A_1 A_2 \ldots A_k \rightarrow \text{attr}_R$.

2. For each tuple $t \in r$ and for all $X_i : 1 \leq i \leq l$, $t[X_i]$ is in PNF.

### 2.2 Query Languages

Theoretical research on data models and languages for manipulating complex value data has grown out of three different but equivalent perspectives, namely, algebraic, calculus-based and logic programming oriented paradigms. This section briefly reviews notation that encompasses different formulations reflecting each of them.

#### 2.2.1 A Complex Value Algebra

A many-sorted algebra, denoted $\text{ALG}^{cv}$, for complex value databases is reviewed as follows. A family of core operators of the algebra is first presented and then an extended family of operators is described. The detailed description of this algebra can be found in [Abiteboul et al. 1995].

**The Core of $\text{ALG}^{cv}$**

Let $r, r_1, r_2, \ldots$ be relations of sort $\tau, \tau_1, \tau_2, \ldots$ respectively.

**Set operations:** Union ($\cup$), intersection ($\cap$), and difference ($-$) are binary set operations.

**Tuple operations:** Selection ($\sigma$) and projection ($\pi$) are defined in the natural manner.

**Powerset:** $\text{powerset}(r)$ is a relation of sort $\{\tau\}$ where

\[
\text{powerset}(r) = \{\nu \mid \nu \subseteq r\}.
\]

**Tuple Creation:** If $A_1, \ldots, A_n$ are distinct attributes, $\text{tup\_create}_{A_1, \ldots, A_n}(r_1, \ldots, r_n)$ is of sort $< A_1 : \tau_1, \ldots, A_n : \tau_n >$, and

\[
\text{tup\_create}_{A_1, \ldots, A_n}(r_1, \ldots, r_n) = \{< A_1 : \nu_1, \ldots, A_n : \nu_n > \mid \forall i (\nu_i \in \tau_i)\}.
\]

**Set Creation:** $\text{set\_create}(r)$ is of sort $\{\tau\}$, and $\text{set\_create}(r) = \{r\}$.

**Tuple Destroy:** If $r$ is of sort $< A : \tau' >$, $\text{tup\_destroy}(r)$ is a relation of sort $\tau'$ and

---

$^1$The symbol $\rightarrow$ denotes functional dependency.
\[ \text{tup.destroy}(r) = \{ v \mid < A : v > \in r \}. \]

Set Destroy: If \( \tau = \{ \tau' \} \), then \( \text{set.destroy}(r) \) is a relation of sort \( \tau' \) and
\[ \text{set.destroy}(r) = \cup r = \{ w \mid \exists v \in r, w \in v \}. \]

There are infinite possibilities in the choice of algebraic operations for complex values. There are several additional algebraic operations. It is important to note that all these operations can be simulated by the core of ALG\textsuperscript{cv} [Abiteboul and Beeri 1995]. Now two important ones, \textit{nest} and \textit{unnest}, are presented.

Restructuring operators:
The restructuring operators \textit{nest} and \textit{unnest} are used to add one level of nesting to a relation and to flatten a relation by one level, respectively. \textit{Nest}(\( \nu \)) takes a relation \( R \) and groups together tuples with common values in some subset of the attributes in \( R \). \textit{Unnest}(\( \mu \)), the inverse of the nest operator, takes a relation nested on some set of attributes and un-groups it by one level.

Suppose we have \( R \) and \( S \) with sorts
\[
\text{sort}(R) = < A_1 : \tau_1, \ldots, A_k : \tau_k, X : \{ < A_{k+1} : \tau_{k+1}, \ldots, A_n : \tau_n > \} > \\
\text{sort}(S) = < A_1 : \tau_1, \ldots, A_k : \tau_k, A_{k+1} : \tau_{k+1}, \ldots, A_n : \tau_n >
\]
Then for instances \( r \) of \( R \) and \( s \) of \( S \), we have
\[
\mu_X(r) = \{ < A_1 : x_1, \ldots, A_n : x_n > \mid \exists t \in r, \\
< A_1 : x_1, \ldots, A_k : x_k > = t[A_1, \ldots, A_k] \land < A_{k+1} : x_{k+1}, \ldots, A_n : x_n > \in t[X] \}
\]
\[
\nu_{\kappa=(A_{k+1}, \ldots, A_n)}(s) = \{ < A_1 : x_1, \ldots, A_k : x_k, X : u > \mid \exists t \in s, \\
t[A_1, \ldots, A_k] = t[A_1, \ldots, A_k] \\
\land u = \{ v[A_{k+1}, \ldots, A_n] \mid v \in s \land v[A_1, \ldots, A_k] = t[A_1, \ldots, A_k] \}
\]

Example 2.2 In Figure 2.2, we have
\[
\mu_X(R) = S \text{ and } \\
\nu_{\kappa=(B,C)}(S) = J
\]

An important subset of ALG\textsuperscript{cv}, denoted \( \mathcal{NRA} \), is formed from the core operations of ALG\textsuperscript{cv} by removing the \textit{powerset} operator and adding the \textit{nest} operator. The language
\[ \mathcal{NRA} \] is usually called the nested relation algebra in the context of nested relational databases. The complex value algebra \( \text{ALG}^{cv} \) can express queries which have hyper-exponential time data complexity. In contrast, the nested relation algebra \( \mathcal{NRA} \) has complexity in PTIME [Grumbach and Vianu 1995].

### 2.2.2 A Complex Value Calculus

In the complex value data model, the calculus is a many-sorted calculus. Calculus variables may denote sets so the calculus will permit quantification over sets. The complex value calculus, modeled after a standard first-order logic, is normally considered to be a second-order logic.

The calculus, denoted \( \text{CALC}^{cv} \), is a strongly typed extension of first order logic using the constructible types defined above. For each data type \( \tau \), the existence of an
infinite set \( \{ x, y, \ldots \} \) of variables of that type is assumed. The alphabet includes typed logical predicates for each type \( \tau \), equality \( =_{\tau} \), membership \( \in_{\tau} \) and containment \( \subseteq_{\tau} \) to manipulate sets. We generally omit the type indices when the types are understood. The vocabulary of the calculus language is defined as follows.

1. parentheses (, );

2. logical connectors \( \land, \lor, \neg, \rightarrow; \)

3. quantifiers \( \exists, \forall; \)

4. typed equality \( =_{\tau} \), membership \( \in_{\tau} \), and containment \( \subseteq_{\tau} \) symbols;

5. typed predicate symbols;

6. typed tuple functions \( <>_{\tau_1, \ldots, \tau_n} \), and typed set functions \( \{}_{\tau_1, \ldots, \tau_n} \).

The logical notations, such as terms, atomic formula, well-formed formula, interpretation and calculus query are introduced as follows.

**Definition 2.5** *Terms* of the complex value calculus language are defined as follows:

1. complex value constants of some type \( \tau \);

2. variables whose types can be inferred from the context, and

3. if \( x \) is a tuple variable and \( C \) is an attribute of \( x \), then \( x.C \) is a term.

**Definition 2.6** *Atomic formulas* (positive literals) are typed expressions of the form

\[
R(t_1, \ldots, t_n), \ t = t', \ t \in t', \text{ or } t \subseteq t',
\]

where \( R \in R, R \) is a database schema; and \( t_i, t, \text{ and } t' \) are terms with the obvious type compatibility restrictions.

*Formulas* are defined from atomic formulas using standard connectives and quantifiers.
Example 2.3 The collection of subsets of the second component of tuples of relation $R:\{ <A, B> \}$, which do not contain the values 5 or 10, is represented by the formula:

$$\{ y \mid \exists u(R(u) \land y \subseteq u.B \land 5 \notin y \land 10 \notin y) \}$$

Example 2.4 The replacement of the second component of tuples of relation $R:\{ <A, B> \}$ by its count number is represented by the formula:

$$\{ x \mid \exists y(R(y) \land x.A = y.A \land x.B = count(y.B)) \}$$

As in the case of the algebra, we also consider extensions of the calculus that can be simulated by the core syntax described above.

**Constructed Terms**

If $t_1, \ldots, t_k$ are terms and $B_1, \ldots, B_k$ are distinct attributes, then $< B_1 : t_1, \ldots, B_k : t_k >$ is a term; if the $t_i$ are of the same sort, $\{ t_1, \ldots, t_k \}$ is a term; and if $t_1$ is a tuple term with attribute $C$, then $t_1.C$ is a term.

To define the semantics of complex value calculus queries, it is convenient to introduce some notation. Given a database instance $I$, its active domain $\text{adom}(I)$ is the set of all constants occurring in $I$. The set of constants occurring in a query $q$ is denoted $\text{adom}(q)$. We use $\text{adom}(q, I)$ as an abbreviation for $\text{adom}(q) \cup \text{adom}(I)$. We define $\text{dom}(\tau, d)$, for some sort $\tau$ and set $d$ to be:

1. $\text{dom}(\text{dom}, d) \overset{\text{def}}{=} d$,
2. $\text{dom}(\{ \tau \}, d) \overset{\text{def}}{=} P^{\text{in}}(\text{dom}(\tau, d))$, and
3. $\text{dom}(< B_1 : \tau_1, \ldots, B_k : \tau_k >, d) \overset{\text{def}}{=} \text{dom}(\tau_1, d) \times \cdots \times \text{dom}(\tau_k, d)$

We now present the notion of *relativised interpretation*, which explicitly specifies the underlying domain of base values used. We write $\varphi(x_1, \ldots, x_n)$ to indicate that $x_1, \ldots, x_n$ is a listing of the variables occurring free in $\varphi$. A *valuation* $\nu$ over a finite set of variables is a ground substitution of those variables. A database over schema $R$ is $DB = (d, I)$, where $I$ is an instance over $R$, $d$ is an arbitrary set of elements containing $\text{adom}(\varphi, I)$, i.e., $\text{adom}(\varphi, I) \subseteq d \subseteq \text{dom}$. If $\nu$ is a valuation over free variables $x_i$ of sort $\tau_i$, $1 \leq i \leq n$, with range contained in $\text{dom}(\tau_i, d)$, then $I$ satisfies $\varphi$ for $\nu$ relative to $d$, denoted $I \models_d \varphi[\nu]$, is defined in the usual manner.
A query is an expression \( \{x_1, \ldots, x_n \mid \varphi(x_1, \ldots, x_n) \} \). We sometimes use \( \varphi(x_1, \ldots, x_n) \) to denote the query \( \{x_1, \ldots, x_n \mid \varphi(x_1, \ldots, x_n) \} \). If \( \nu \) is a valuation over \( \text{free}(\varphi) \) the answer to a query \( q \) on a database \( DB = (d, I) \), denoted \( q(DB) \), is defined by

\[
q(DB) = \{\nu([x_1, \ldots, x_n]) \mid I \models_d \varphi[\nu],
\nu \text{ is a valuation over } \text{free}(\varphi)\}
\]

Then in the evaluation of \( \varphi \) on \( I \), each variable of type \( \tau_i \) in \( \varphi \) ranges over \( \text{dom}(\tau_i, \text{dom}(I)) \).

### 2.2.3 The Calculus (CALC\textsuperscript{cv}) + Fix-point

Fix-point operators are redundant in the context of unrestricted higher-order logic [Grumbach and Vianu 1995]. However, a fix-point construct provides a tractable form of recursion, e.g., it can express transitive closure in polynomial space (time) and yield languages which are well-behaved with respect to expressive power [Grumbach and Vianu 1995]. We review inflationary and non-inflationary extensions of the calculus with recursion.

**Partial Fix-point Operator and Logic**

Let \( R \) be a database schema, and let \( T[\tau_1, \ldots, \tau_m] \) be a typed relation which is not in \( R \). Let \( \varphi(T) \) be a formula using \( T \) and relations in \( R \), with \( m \) free variables \( x_1 : \tau_1, \ldots, x_m : \tau_m \). Then \( \mu_T(\varphi(T)) \) denotes the relation of type \( [\tau_1, \ldots, \tau_m] \) which is the limit, if it exists, of the sequence \( \{\Phi_n\}_{n \geq 0} \) defined by

\[
\Phi_0 = \emptyset;
\]

\[
\Phi_n = \{ (x_1, \ldots, x_m) \mid \varphi(\Phi_{n-1}, x_1, \ldots, x_m) \}, \quad n > 0.
\]

where \( \varphi(\Phi_{n-1}, x_1, \ldots, x_m) \) denotes the result of evaluating \( \varphi \) on the instance \( I \) over \( R \) and the instance \( \Phi_{n-1} \) over \( T \).

The expression \( \mu_T(\varphi(T)) \) can be used as a term or as a relation in more complex formulas like any other relation. For example, if \( x \) is a variable of type \( [\tau_1, \ldots, \tau_m] \) then \( x = \mu_T(\varphi(T)) \) is a fix-point formula.

The extension of the calculus with \( \mu \) is called partial fix-point logic, denoted \( \text{CALC}^{cv} + \mu \). \( \text{CALC}^{cv} + \mu \) formulas are built by repeated applications of \( \text{CALC}^{cv} \) operators.
and the partial fix-point operator, starting from atoms. Partial fix-point operators can be nested. \( \text{CALC}^{\text{cv}} + \mu \) queries over a database schema \( \text{R} \) are expressions of the form

\[
\{ <x_1, \ldots, x_n> | \xi \}
\]

where \( \xi \) is a \( \text{CALC}^{\text{cv}} + \mu \) formula.

**Inflationary Fix-point Operators and Logic**

Adding \( \mu^+ \) instead of \( \mu \) to \( \text{CALC}^{\text{cv}} \) yields the inflationary fix-point logic, denoted by \( \text{CALC}^{\text{cv}} + \mu^+ \). The definition of \( \mu^+_T(\varphi(T)) \) is identical to that of the partial fix-point operator except that the sequence \( \{ \Phi_n \}_{n \geq 0} \) is defined as follows:

\[
\begin{align*}
\Phi_0 & = \emptyset; \\
\Phi_n & = \Phi_{n-1} \cup \varphi(\Phi_{n-1}), \quad n > 0.
\end{align*}
\]

Note that the sequence \( \{ \Phi_n \}_{n \geq 0} \) is increasing: \( \Phi_{i-1} \subseteq \Phi_i \) for each \( i > 0 \), and the sequence converges in all cases.

**Example 2.5** Let \( R \) be a binary relation. The following formula defines the \textit{nest} operation on the second argument, using a unary relation \( T \) and variables \( x : \text{dom}, s : \{\text{dom}\} \).

\[
\{ <x, s> | \exists z(R(x, z)) \land s = \mu^+_T((R(x, y) \lor T(y))) \}
\]

### 2.2.4 Rule-based Languages for Complex Values

Query languages based on the deduction paradigm are extensions of Datalog to incorporate complex values. These languages are based on the calculus and do not increase the expressive power of \( \text{ALG}^{\text{cv}} \) or \( \text{CALC}^{\text{cv}} \). However, certain queries can be expressed in this deduction paradigm more efficiently and with lower complexity than they can be by using the powerset operator in the \( \text{CALC}^{\text{cv}} \) [Grumbach and Vianu 1991; Grumbach and Vianu 1995]. A major difference between the various proposals of logic programming with a set construct lies in their approach to nesting: grouping in \( \mathcal{LDL} \) [Beeri et al. 1987], data functions in \( \text{COL} \) [Abiteboul and Grumbach 1991], and a form of universal quantification in [Kuper 1990]. We briefly review the concept of Datalog for complex values and queries.
Definition 2.7 A database clause (rule) is an expression of the form

\[ p(x_1, \ldots, x_i) \leftarrow L_1, \ldots, L_n, \]

where the head \( p \) is a derived predicate, and each \( L_i \) of the body is a literal. A program \( \mathcal{P} \) is a finite set of rules.

We distinguish between the intentional predicates and functions, which appear in heads of rules, and the extensional ones, which appear only in bodies.

Definition 2.8 A database is a finite set of database clauses. A query is a formula of the form \( \leftarrow W \), where \( W \) is a calculus formula (i.e., \( W \in \text{CALC}_{cv} \)) and any free variables in \( W \) are assumed to be universally quantified at the front of the query.

Definition 2.9 Let \( \mathcal{P} \) be a database program, \( Q \) a query \( \leftarrow W \). An answer to \( \mathcal{P} \cup \{\leftarrow W\} \) is a ground substitution \( \theta \) such that \( \forall(W\theta) \) is a logical consequence of \( \mathcal{P} \).

Definition 2.10 Let \( \mathcal{P} \) be a database program, \( W \) a formula, and \( S \) an interpretation. Then \( \text{ans}(\mathcal{P}, W, S) \) is the set of all answers to \( \mathcal{P} \cup \{\leftarrow W\} \) that are ground substitutions for all free variables in \( W \).

Example 2.6 The following is a Datalog\(_{cv}\) program:

\[ p(\{a\}) \leftarrow \]
\[ r(x) \leftarrow q(x) \land p(z) \land x \notin z \]

2.3 Domain-Independent Computable Queries

To formulate a precise definition of a query, we focus on a high level abstraction and thus on the mappings between input instances and output instances expressible by queries. Queries on complex value databases are defined by extending the classical definition for relational ones.

A query takes as input an instance over \( R \) and returns as answer a relation over some schema \( S \). It is understood that the arities of the input relations \( R_i \), as well as the schema of the output relation, are fixed for a given query. This assumption
is referred to *well-typeness*. In addition to this basic requirement, queries must be *generic*, *domain independent* and *computable*.

### 2.3.1 Notion of Genericity

A query is usually defined to be a *generic database transformation*, i.e., a function mapping input database instances to output relations which is invariant under isomorphisms. The idea is that queries must conform to the data independence principle. This principle is a database that provides an abstract interface and in which the internal representation of data has no effect on the results of queries. Essentially, this means that the query treats data values as uninterpreted objects.

**Definition 2.11** Let $R$ be database schema. A query $q$ is *generic* with respect to a domain $d \subseteq \text{dom}$ iff for each database instance $I$ over $R$, $\text{adom}(I) \subseteq d$, and each permutation $\rho$ of $d$, $\rho(q(I)) = q(\rho(I))$.

We will assume throughout this thesis that all queries are generic in this sense, i.e., they map isomorphic database instances to isomorphic outputs.

### 2.3.2 Domain Independence

A property usually imposed upon queries is *domain independence*, meaning that the answer of the query depends only on the active domain of the input instance and not on the underlying universe.

As mentioned in Chapter 1, the calculus paradigm, used without care, can easily express queries whose "answers" are infinite. To overcome these obstacles, a variety of approaches have been developed to resolve this problem based on the use of both semantic and syntactic restrictions. We focus on semantic restriction in this subsection and discuss syntactic restriction in Chapter 3.

Recall from relational database theory that an arbitrary calculus formula can be effectively transformed into a semantically equivalent formula in *prenex normal form* [Abiteboul et al. 1995]:

\[
\forall x_1, \ldots, \forall x_n \exists \psi(x_1, \ldots, x_n)
\] (2.1)
where each $\%$ is either $\forall$ or $\exists$ and $\Psi(\cdot)$ is a quantifier-free formula with free variables among $x_1, \ldots, x_n$.

There are two important interpretations of calculus queries. Under the active-domain semantics, all quantified variables range over the active domain of a database. That is, a sentence given by formula 2.1 defines the query $q$ such that the value of $q$ on a database $I$ is the value of $$\%_1 x_1 \in \text{adom}(q,I), \ldots, \%_n x_n \in \text{adom}(q,I) \Psi(x_1, \ldots, x_n)$$

Under the natural semantics, all quantified variables range over infinite set $d \subseteq \text{dom}$. That is, the sentence defines the query $q$ whose value on $I$ equals $$\%_1 x_1 \in d, \ldots, \%_n x_n \in d \Psi(x_1, \ldots, x_n)$$

It is easy to write queries with undefined output under natural interpretation. Although the active domain semantic has the advantage that the output is always defined, the active domain information is not readily available to users. One approach to resolving this problem is to consider the class of queries that yield the same output on all possible underlying domains [Demolombe 1992; Hull and Su 1994; Topor 1987; Ullman 1988; Van Gelder and Topor 1991].

Definition 2.12 A calculus query $q$ is domain independent if for any databases $DB_1 = (d_1, I)$ and $DB_2 = (d_2, I)$ (i.e., same instances, but different domains), then $q(DB_1) = q(DB_2)$.

2.3.3 Computable Queries

A query is computable if there exists a Turing Machine which, when started with a natural encoding of a database instance $I$ on its tape, halts with an encoding of $q(I)$ on the tape, or diverges, when $q(I)$ is undefined. That is, the query must be "implementable" by a Turing Machine.

2.4 Query Translation

As mentioned in Chapter 1, the class of relational calculus queries or formulas that have sensible answers, called the domain independent class, is known to be undecidable.
Definition 2.13 A variable \( x \) is range-restricted in a formula \( \varphi \) if either

- \( \varphi \) is an atomic formula \( R(t_1, \ldots, t_n) \), \( R \) is a relation name and \( x \) occurs in \( \varphi \); or
- \( \varphi \) is \( x = a \) or \( a = x \); or
- \( \varphi \) is \( \psi_1 \land \psi_2 \), and \( x \) is range-restricted in either \( \psi_1 \) or \( \psi_2 \); or
- \( \varphi \) is \( \psi_1 \lor \psi_2 \), and \( x \) is range-restricted in both \( \psi_1 \) and \( \psi_2 \); or
- \( \varphi \) is \( \varphi_1 \land x = y \), and \( x \) or \( y \) is range-restricted in \( \varphi_1 \); or
- \( \varphi \) is \( \exists y \phi \) or \( \forall y \phi \), and \( x \) is range-restricted in \( \phi \); or
- \( \varphi \) is \( \neg \phi \) and \( x \) is range-restricted in \( \text{pushnot}(\neg \phi) \), where \( \phi \) is not of the form \( R(t_1, \ldots, t_n) \).

Here the \( \text{pushnot} \) operator "pushes" negations one step towards the atoms in formulas [Van Gelder and Topor 1991].

Definition 2.14 Given a formula \( \neg \phi \) where \( \phi \) is not of the form \( R(t_1, \ldots, t_n) \), the function \( \text{pushnot}(\neg \phi) \) returns a formula as follows:

<table>
<thead>
<tr>
<th>( \neg \phi )</th>
<th>( \text{pushnot}(\neg \phi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\xi_1 \land \ldots \land \xi_n) )</td>
<td>( (\neg \xi_1) \lor \ldots \lor (\neg \xi_n) )</td>
</tr>
<tr>
<td>( (\xi_1 \lor \ldots \lor \xi_n) )</td>
<td>( (\neg \xi_1) \land \ldots \land (\neg \xi_n) )</td>
</tr>
<tr>
<td>( \exists \overline{x} \xi )</td>
<td>( \forall \overline{x} \neg \xi )</td>
</tr>
<tr>
<td>( \forall \overline{x} \xi )</td>
<td>( \exists \overline{x} \neg \xi )</td>
</tr>
<tr>
<td>( \neg \xi )</td>
<td>( \xi )</td>
</tr>
<tr>
<td>( \neg (t_1 = t_2) )</td>
<td>( t_1 \neq t_2 )</td>
</tr>
<tr>
<td>( \neg (t_1 \neq t_2) )</td>
<td>( t_1 = t_2 )</td>
</tr>
</tbody>
</table>
Informally, the variable $x$ is range-restricted in the formula $\varphi$ if, whenever $\varphi(c)$ is true in a database instance $I$, $c \in \text{adom}(\varphi, I)$.

**Definition 2.15** A formula (or calculus query) $\varphi$ is *safe-range* if

- every free variable in $\varphi$ is range-restricted in $\varphi$; and
- for every sub-formula $\exists y \phi$ in $\varphi$, the variable $y$ is range-restricted in $\phi$; and
- for every sub-formula $\forall y \phi$ in $\varphi$, the variable $y$ is range-restricted in $\neg \phi$.

**Example 2.7** The following formulas are safe-range.

\[
\begin{align*}
P(x, y) \land (Q(x) \lor R(y)) \\
P(x) \land \forall y (Q(y) \rightarrow R(x, y)) \\
P(x) \land \exists y (x = y \land \neg Q(y))
\end{align*}
\]

The translation of a relational calculus query that includes disjunction or negation is a solved problem. Van Gelder and Topor [1991] developed an algorithm to transform an evaluable formula into an equivalent relational algebra expression. This algorithm was generalised by Escobar-Molano, Hull, and Jacobs [1993] to translate allowed formulas with scalar functions into the extended relational algebra.

**Example 2.8** The safe-range queries in Example 2.7 are translated by applying Van-Gelder and Topor’s algorithm as follows.

\[
\begin{align*}
P(x, y) \land (Q(x) \lor R(y)) & \equiv (P(x, y) \land Q(x)) \lor (P(x, y) \land R(y)) \\
& \rightarrow \pi_{1,2}(P \bowtie_{1=1} Q) \cup \pi_{1,2}(P \bowtie_{2=1} R) \\
P(x) \land \forall y (Q(y) \rightarrow R(x, y)) & \equiv P(x) \land \neg \exists y (Q(y) \land \neg R(x, y)) \\
& \equiv P(x) \land \neg \exists y (P(x) \land Q(y) \land \neg R(x, y)) \\
& \rightarrow P - \pi_1(P \times Q - R)
\end{align*}
\]
\[ P(x) \land \exists y(x = y \land \neg Q(y)) \]
\[ \equiv P(x) \land \exists (P(x) \land x = y \land \neg Q(y)) \]
\[ \rightarrow P \bowtie_{1=1} \pi_1 (\Delta_{i_1} P \text{ diff } 2=1 Q) \]

where \( \Delta \) is a operator described below. Here \( \text{diff} \) is the generalised difference operator described in [Van Gelder and Topor 1991]. The expression \( \Delta_{i_1, ..., i_k} R \) is the relation that results from appending the result of applying the function \( \delta \) to the components \( i_1, ..., i_k \) of each tuple in \( R \) at the right of that tuple.

The translation of complex value queries that are incorporated with external functions is more difficult and complicated than that of relational calculus queries. This thesis undertakes a more comprehensive study of this issue.

2.5 Query Optimisation

As there are many equivalent transformations of the same high-level query, the DBMS has to choose the one that is efficient and that minimises resource usage. In standard relational database systems, heuristic rules that query optimisation can apply include:

- performing selection and projection operations as soon as possible,

- combining Cartesian product with a subsequent selection whose predicate represents a join condition into a join operation, and

- using associativity of binary operations to rearrange leaf nodes so that leaf nodes with the most restrictive selections are executed first.

Complex value query languages contains restructuring operations. It is difficult to optimise an algebra expression of query which includes restructuring operators. This thesis considers this issue and solve the problem of how to optimise queries which include join operations.
Chapter 3

Domain-Independent Queries with External Functions

This chapter explores the issue of the semantics of complex value calculus queries in the presence of functions. As mentioned in Chapter 1, some calculus queries cannot be answered sensibly. The database area emphasises finite structures and the database research community has developed notions of "domain independence" and "safety" to capture intuitive properties related to this finitude.

It is highly desirable to check whether a formula satisfies the safety property when we need to support any complex values and any user-defined functions in a query language. However, domain-independence is undecidable even for the flat relational data model without functions. There have been several attempts to identify such decidable subclasses of the domain independent formulas in the complex value model [Abiteboul and Beeri 1995; Suciu 1995; Liu et al. 1996].

This chapter investigates the notion of domain independence of complex value calculus queries and focuses on safe queries expressed in a higher order logic with external functions applied to finite databases rather than to the constraint databases which allow the finite representation of infinite databases based on the use of constraints. The notions of two syntactic criteria, called "embedded evaluable" and "embedded allowed", for queries which guarantee embedded domain independence, are generalised for the complex value model.

There has been considerable development in deductive databases that use the first-order language as a mathematical notation for describing data. Query evaluation in such a model is the process of proving theorems from logical formulas and explicit
facts. This chapter also considers the deduction paradigm for complex values. As in deductive databases, there are only certain complex value database programs which may be regarded as reasonable. Since the class of embedded domain-independent complex value database programs is recursively unsolvable, it is desirable to search for recursive subclasses with simple decision procedures. This chapter also considers this decision problem.

The structure of this chapter is organised as follows. Section 3.1 briefly reviews some notions and basic concepts. Section 3.2 investigates the properties of two large decidable subclasses of domain independent formulas, namely, *embedded evaluable* and *embedded allowed* formulas. Section 3.3 defines a recursive class of *embedded allowed database programs* and proves that embedded allowed stratified programs satisfying certain constraints are embedded domain-independent. Finally, relationships between properties such as embedded domain independence, finiteness and embedded allowed in various calculus-based query languages are shown in Section 3.4.

3.1 Basic Concepts

This section briefly reviews some basic concepts and two notions of domain independence.

Let $R = (R_1, ..., R_k)$ be a database schema and $I = (r_1, ..., r_k)$ be a database instance of $R$ over domain $d$. We focus on a fixed finite set $F$ of functions which are associated with signatures. An interpretation is $(d, F, I)$, where $d \subseteq \text{dom}, F \subseteq F$, $F = (f_1, ..., f_l)$, $f_i : \text{dom}(r_i, d) \rightarrow \text{dom}(r'_i, d)$ are functions. A database $DB$ is given by interpretation of each relation $r_i$ as a finite relation over $d$ and augmented with a number of external functions $f_1, ..., f_l$.

Let $\varphi$ be a formula with free variables $x_1, ..., x_n$. If $\sigma$ is a valuation over $\text{free}(\varphi)$, then the notion of the interpretation $(d, F, I)$ satisfying $\varphi$ under $\sigma$, denoted $I = (d, F) \varphi[\sigma]$, is defined in the usual manner [Escobar-Molano et al. 1993; Abiteboul et al. 1995].

The notion of the answer $q(DB)$ to the query $q$ on the database $DB = (d, F, I)$ is defined by:

$$q(DB) = \{[\nu_1, ..., \nu_n] | \nu_i \text{ is of type } r_i, I = (d, F) \varphi(\nu_1, ..., \nu_n)\}$$
A database query can be viewed as a partial function $q$ mapping any database $DB$ with interpretation $(d, F, I)$ to $q(DB) \in \text{dom} \{\tau\}, \tau$ is some type for query result.

Before presenting our main results, we need to review two notions of domain independence. The notion of "embedded domain independence" was proposed to generalise "domain independence" to incorporate functions [Escobar-Molano et al. 1993]. The fundamental idea behind this notion is that, for any query $q$, there is a bound on the number of times functions (and their inverses) can be applied. The answer to $q$ on an input instance $I$ depends on the closure of $\text{adom}(q, I)$. This notion for complex objects is reviewed as follows.

Given a database instance $I$ and a query $q$, let $C_q$ be a set of constants that appear in $q$. Following [Suciu 1995], $\text{term}^n(DB)$ for some database $DB$ with interpretation $(d, F, I)$ is defined as follows:

$$\text{term}^0(DB) \overset{\text{def}}{=} \text{atom}(I, C_q)$$

$$\text{term}^{n+1}(DB) \overset{\text{def}}{=} \text{term}^n(DB) \cup \{\text{atom}(f_i(x)) \mid f_i \in F; x \in \text{dom}(\tau_i, \text{term}^n(DB)), i = 1,\ldots,l\}$$

where $\text{atom}(I, C_q)$ are all values in domain $d$ mentioned in the instance $I$ and $C_q$.

**Definition 3.1** Two databases $DB_1 = (d_1, F_1, I)$ and $DB_2 = (d_2, F_2, I)$ agree on $\text{atom}(I, C_q)$ to level $n$ if

- $\text{term}^{n+1}(DB_1) = \text{term}^{n+1}(DB_2)$ and
- $\forall x \in \text{dom}(\tau_i, \text{term}^n(DB)), f_i \in F_1, f_i' \in F_2, f_i(x) = f_i'(x)$, i.e., $f_i$ and $f_i'$ agree on any input whose atomic values are in $\text{term}^n(DB)$.

We now review the notion of embedded domain independence.

**Definition 3.2** A calculus query $q$ is *embedded domain independent at level $n$* if, for all interpretations $S_1 = (d_1, F_1, I)$ and $S_2 = (d_2, F_2, I)$ which agree on $\text{atom}(I, C_q)$ to level $n$, $q$ yields the same output on $S_1$ and $S_2$. The query $q$ is *embedded domain independent* if for some $n$ it is embedded domain independent at level $n$. 
Next we will review the notion of external-function domain independent queries proposed by Suciu [1995]. Let $DB_1$, $DB_2$ be two databases with interpretations $(d_1, F_1, I_1)$, $(d_2, F_2, I_2)$ respectively. A morphism $\xi : DB_1 \rightarrow DB_2$ is a partial injective function $\xi : d_1 \rightarrow d_2$, which can be lifted from the base type to partial functions at any type $\tau$, $\xi_\tau : \text{dom}(\tau, D) \rightarrow \text{dom}(\tau, D')$, such that

- for every $i$, $\xi(R_i)$ is defined and $\xi(R_i) = R'_i$, where $R_i \in I_1$, $R'_i \in I_2$, and

- for any $x \in \text{dom}(\tau, d_1)$, if $f_j^'(\xi(x))$ is defined then so is $\xi(f_j(x))$ and $f_j^'(\xi(x)) = \xi(f_j(x))$, where $f_j \in F_1$, $f_j^' \in F_2$. That is, the following diagram commutes for any morphism $\xi$.

\[
\begin{array}{ccc}
dom(\tau, d_1) & \xrightarrow{\xi_\tau} & \dom(\tau, d_2) \\
\uparrow f_j & & \uparrow f_j^' \\
dom(\tau, d_1) & \xrightarrow{\xi_\tau} & \dom(\tau, d_2)
\end{array}
\]

Let us write $e_1 \sqsubseteq e_2$ whenever expression $e_1$ is undefined or $e_1 = e_2$.

**Definition 3.3** A query $q$ is external-function domain independent (ef-domain independent) iff for every morphism $\xi : DB_1 \rightarrow DB_2$, $q(DB_2) \sqsubseteq \xi(q(DB_1))$.

As domain independence is an undecidable problem, it is desirable to find a simple syntactic condition on queries that implies domain independence. The formulas that are thereby restricted are called safe. Several decidable syntactic criteria for safe queries have been developed in the literature. The notion of safe calculus queries developed in [Escobar-Molano et al. 1993], called embedded allowed (em-allowed), is based on inferring from syntactic properties of a calculus formula that a variable ranges, for all practical purposes, over a bounded set of possible values. This chapter extends the definition of em-allowed to the complex value model and then introduces that all queries in em-allowed $\text{CALC}^{cv}$ (or $\text{CALC}^{cv} + \mu^+$) are ef-domain independent.
3.2 Evaluable Queries

This section proposes two large decidable subclasses of domain independent formulas, namely, embedded evaluable and embedded allowed formulas.

A key element in the notion of embedded evaluable and embedded allowed formulas is the definition of the bounding function $bd$ which associates finiteness dependencies (FinDs) to formulas. First we introduce FinD over basic type and complex value type variables.\(^1\)

Intuitively, a formula $\varphi$ satisfies the FinD $\{x_1, x_2\} \rightarrow \{y_1, y_2\}$ if for each database $DB$, $x_i$ range over $\text{dom}(\tau_i, \text{term}^k(DB))$, $i = 1, 2$, then $\varphi$ will be true only on assignments which map $y_i$ into $\text{dom}(\tau'_i, \text{term}^{k+l}(DB))$, for some $l$.

**Definition 3.4** Let $\varphi$ be a formula and $X \rightarrow Y$ a FinD over variable set $X$. A formula $\varphi$ satisfies the finiteness dependency $X \rightarrow Y$, denoted $\varphi \models X \rightarrow Y$, if for each database $DB = (\text{dom}, F, I)$ and each $k \geq 0$ there is some $l \geq 0$ such that $\forall y_i \in Y$, $\sigma(y_i) \in \text{dom}(\tau_{y_i}, \text{term}^{k+l}(DB))$ whenever $\sigma$ is a variable assignment for $X$ satisfying $\sigma(x_i) \in \text{dom}(\tau_{x_i}, \text{term}^k(DB))$, $\forall x_i \in X$ and $I \models \varphi[\sigma]$.

**Example 3.1** Let the sort of relation $R$ be $\{< z : \text{dom}, x : \{\text{dom}\} >\}$. Given $\varphi \equiv R(z, x) \land z \in x \land \neg Q(z) \land f(z) = y$

it can be shown that $\varphi \models \emptyset \rightarrow zx$, $\varphi \models x \rightarrow z$, $\varphi \models z \rightarrow y$, $\varphi \models x \rightarrow y$, and $\varphi \models \emptyset \rightarrow zxy$.

FinDs satisfy the properties of functional dependencies [Ullman 1988; Escobar-Molano et al. 1993]. We illustrate the following basic inference rules.

fd1 $XY \rightarrow X$,

fd2 $XW \rightarrow YU$, if $X \rightarrow Y$ and $U \subseteq W$, and

fd3 $X \rightarrow Z$, if $X \rightarrow Y$ and $Y \rightarrow Z$.

\(^1\)Note that the finiteness dependencies proposed in [Escobar-Molano et al. 1993], [Ramakrishnan et al. 1987] and [Topor 1991] only consider basic types.
where $X, Y, Z, U, W$ range over sets of variables.

If $\Gamma$ is a set of FinDs and $X \rightarrow Y$ is a FinD then $\Gamma \vdash X \rightarrow Y$ if $X \rightarrow Y$ can be derived from $\Gamma$ using the inference rules given above.

**Definition 3.5** If $\Gamma$ is a set of FinDs over a variable set $V$ then the closure of $\Gamma$ over $V$ is

$$\Gamma^{*,V} = \{X \rightarrow Y \mid XY \subseteq V \text{ and } \Gamma \vdash X \rightarrow Y\}$$

For a formula $\varphi$, $\Gamma^{*,\varphi}$ is a shorthand for $\Gamma^{*,\text{free}(\varphi)}$.

Given a formula $\varphi$, $bd(\varphi)$ returns the set of FinDs, as shown in Figure 3.1. In Figure 3.1, the operator $\otimes$ is defined as follows: given sets $\Gamma_1, ..., \Gamma_n$ of FinDs,

$$\Gamma_1 \otimes ... \otimes \Gamma_n = \{X_1...X_n \rightarrow Y \mid X_i \rightarrow Y \in \Gamma_i \text{ for } i \in [1, ..., n]\}$$

The formulas 1 to 11 and their associated functions $bd$ were presented in [Escobar-Molano et al. 1993]. We add formulas 12 to 19 for the complex value model.

### 3.2.1 Embedded Evaluable Formulas

This subsection identifies a large decidable subclass of domain independent formulas with external functions, called embedded evaluable formulas. To define embedded evaluable we need to define a certain relation between variables and (sub)formulas. It is called **constrained**. The following procedure $ct$ is proposed to generate the set of constrained variables of a formula.
§3.2  Evaluable Queries

### Table

<table>
<thead>
<tr>
<th>( \varphi )</th>
<th>( bd(\varphi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  ( R(t_1, \ldots, t_n) )</td>
<td>( { \emptyset \rightarrow X }^{*}^{\varphi} ) where ( X = ) set of variables that are members of ( { t_1, \ldots, t_n } ).</td>
</tr>
<tr>
<td>2  ( \neg R(t) )</td>
<td>( \emptyset^{*}^{\varphi} )</td>
</tr>
<tr>
<td>3  ( \neg \xi )</td>
<td>( bd(\text{pushnot}(\neg \xi)) ) for ( \xi ) not of the form ( R(t) ).</td>
</tr>
<tr>
<td>4  ( f(t_1, \ldots, t_n) = t )</td>
<td>( \emptyset^{*}^{\varphi} ) if ( t ) is not a variable, or ( t ) is a variable occurring in one of ( t_1, \ldots, t_n ).</td>
</tr>
<tr>
<td>5  ( f(t_1, \ldots, t_n) = t )</td>
<td>( { X \rightarrow t }^{*}^{\varphi} ) if ( t ) is a variable not occurring in any of ( t_1, \ldots, t_n ), where ( X = ) set of variables occurring in ( t_1, \ldots, t_n ).</td>
</tr>
<tr>
<td>6  ( x = y )</td>
<td>( { x \rightarrow y, y \rightarrow x }^{*}^{\varphi} )</td>
</tr>
<tr>
<td>7  ( t_1 \neq t_2 )</td>
<td>( \emptyset^{*}^{\varphi} )</td>
</tr>
<tr>
<td>8  ( \xi_1 \land \ldots \land \xi_n )</td>
<td>( (bd(\xi_1) \cup \ldots \cup bd(\xi_n))^{*}^{\varphi} )</td>
</tr>
<tr>
<td>9  ( \xi_1 \lor \ldots \lor \xi_n )</td>
<td>( (bd(\xi_1) \otimes \ldots \otimes bd(\xi_n))^{*}^{\varphi} )</td>
</tr>
<tr>
<td>10  ( \exists \vec{x} \xi )</td>
<td>( bd(\xi) - ) all FinDs in which some variable in ( \vec{x} ) occurs ( }^{*}^{\varphi} )</td>
</tr>
<tr>
<td>11  ( \forall \vec{x} \xi )</td>
<td>( bd(\xi) - ) all FinDs in which some variable in ( \vec{x} ) occurs ( }^{*}^{\varphi} )</td>
</tr>
<tr>
<td>12  ( x \subseteq { y \mid \phi(y) } )</td>
<td>( { X \rightarrow x } ) if ( x ) is a variable not occurring in ( \phi ), ( X = ) set of variables occurring in ( \phi ).</td>
</tr>
<tr>
<td>13  ( \forall y( y \in x \rightarrow \varphi(y) ) )</td>
<td>( { X \rightarrow x } ) if ( x ) is a variable not occurring in ( \phi ), ( X = ) set of variables occurring in ( \phi ).</td>
</tr>
<tr>
<td>14  ( t \in t' )</td>
<td>( { X \rightarrow t }^{*}^{\varphi} ) if ( t ) is a variable not occurring in ( t' ), where ( X = ) set of variables occurring in ( t' ).</td>
</tr>
<tr>
<td>15  ( t \in t' )</td>
<td>( \emptyset^{*}^{\varphi} ) if ( t ) is not a variable or ( t ) is a variable occurring in ( t' ).</td>
</tr>
<tr>
<td>16  ( t \subseteq t' )</td>
<td>( { X \rightarrow t }^{*}^{\varphi} ) if ( t ) is a variable not occurring in ( t' ), where ( X = ) set of variables occurring in ( t' ).</td>
</tr>
<tr>
<td>17  ( t \subseteq t' )</td>
<td>( \emptyset^{*}^{\varphi} ) if ( t ) is not a variable or ( t ) is a variable occurring in ( t' ).</td>
</tr>
<tr>
<td>18  ( x.i )</td>
<td>if ( x ) is of type ( &lt; t_1, \ldots, t_n &gt; ), then for each ( i, 1 \leq i \leq n, x \rightarrow x.i ).</td>
</tr>
<tr>
<td>19  ( x )</td>
<td>if ( x ) is of type ( &lt; t_1, \ldots, t_n &gt; ), ( \emptyset \rightarrow x.i ) for each ( i ) then ( \emptyset \rightarrow x ).</td>
</tr>
</tbody>
</table>

**Figure 3.1:** The overall definition of the function \( bd \)
procedure (constrained-variables)
input: a calculus formula $\varphi$
output: a subset of the free variables of $\varphi$

begin
(procedure is a predicate in \{\in, \subseteq\})
(In each of the cases following, $X$ denotes a set of variables that are members of $\bar{t}$)
($Y$ denotes a set of unconstrained variables generated during the process)
$Y := \emptyset$

case $\varphi$ of
$R(\bar{t})$ : $ct(\varphi) := X$;
$\xi \land \neg R(\bar{t})$ : $ct(\varphi) := (ct(\xi) - X) \cup \{z \mid bd(\xi) \models \emptyset \rightarrow z\};$
$Y := Y \cup X - \{z \mid bd(\xi) \models \emptyset \rightarrow z\}$
$\xi \lor \neg R(\bar{t})$ : $ct(\varphi) := (ct(\xi) - X); Y := Y \cup X$
$\neg \xi$ : $ct(\varphi) := ct(pushnot(\neg \xi)), \text{ for } \xi \text{ not of the form } R(\bar{t})$
$f(\bar{t}) = t$ : If $t$ is a variable and $X \subseteq \{z \mid bd(\varphi) \models \emptyset \rightarrow z\} , ct(\varphi) := t$
$t \ \text{pred} \ t'$ : If $t$ is a variable and $Z \subseteq \{z \mid bd(\varphi) \models \emptyset \rightarrow z\}$,
$ct(\varphi) := t, \text{ where } Z = \text{ set of variables that are members of } t'$
$\xi_1 \lor \xi_2$ : $ct(\varphi) := ct(\xi_1) \cup ct(\xi_2) - Y; \text{ where }$
$\xi_1 \text{ and } \xi_2 \text{ are not of the form } \neg R(\bar{t})$
$\xi_1 \land \xi_2$ : $ct(\varphi) := \{z \mid bd(\xi_1) \models \emptyset \rightarrow z\} \cup \{z \mid bd(\xi_2) \models \emptyset \rightarrow z\}$
$\cup (ct(\xi_1) \cup ct(\xi_2) - Y), \text{ where } \xi_1 \text{ and } \xi_2 \text{ are not of the form } \neg R(\bar{t})$
$\exists x \xi$ : $ct(\varphi) := ct(\xi) - \{x\}$
$\forall x \xi$ : $ct(\varphi) := ct(\xi) - \{x\}$

return $ct$
end

Definition 3.6 A variable $x$ is constrained in a formula $\varphi$ if $x \in ct(\varphi)$.

Consider a fixed database $DB$ and variable $x$ of type $\tau_i$. Intuitively, the fact that $x$ is constrained in a formula $\varphi$ tells us that if $\varphi(x, \bar{y})$ is true, then either

- $x \in \text{dom}(\tau_i, \text{term}^n(DB))$, or
- $\varphi(x, \bar{y})$ is true for all values of $x$, i.e., $x \in \text{dom}(\tau_i, \text{dom})$. 
Example 3.2 Consider the following formula

$$\varphi = (P(x, y) \lor Q(y)) \land x \in y \land (R(y, u) \lor \neg S(y)).$$

Let $A \overset{\text{def}}{=} (P(x, y) \lor Q(y))$, $B \overset{\text{def}}{=} x \in y$, $C \overset{\text{def}}{=} (R(y, u) \lor \neg S(y))$. By using the bd function listed in Figure 3.1, we get $bd(\varphi) \models \emptyset \rightarrow xy$. The set of constrained variables in formula $\varphi$ can be computed using the procedure $ct$. The process is described as follows.

1. $ct(A) = \{x, y\}$,

2. $ct(A \land B) = \{z \mid bd(A) \models \emptyset \rightarrow z\} \cup \{z \mid bd(B) \models \emptyset \rightarrow z\} \cup (ct(A) \cup ct(B) - Y).
   
   So $ct(A \land B) = \{y\} \cup \emptyset \cup (\{x, y\} \cup \{x\} - \emptyset) = \{x, y\}$, and

3. $ct((A \land B) \land C) = \{z \mid bd(A \land B) \models \emptyset \rightarrow z\} \cup \{z \mid bd(C) \models \emptyset \rightarrow z\} \cup (ct(A \land B) \cup ct(C) - Y)$. So $ct((A \land B) \land C) = \{x, y\} \cup \emptyset \cup (\{x, y\} \cup \{u\} - \emptyset) = \{x, y\} \cup \{x, u\}
   
   = \{x, y, u\}$.

Definition 3.7 A formula $\varphi$ is embedded evaluable if the following conditions hold:

(a) $bd(\varphi) \models \emptyset \rightarrow \text{free}(\varphi)$;

(b) for each sub-formula $\exists x \psi$ of $\varphi$, $x \in ct(\psi)$;

(c) for each sub-formula $\forall x \psi$ of $\varphi$, $x \in ct(\neg \psi)$.

Example 3.3 The following formula is em-evaluable.

$$\varphi(y, z) = \exists x [(p(x, y) \lor q(y)) \land z = f(y)]$$

Let $A \overset{\text{def}}{=} [(p(x, y) \lor q(y)) \land z = f(y)]$. We have $bd(\varphi) \models \emptyset \rightarrow yz$ and $ct(A) = \{x, y, z\}$, as required for $\varphi$ to be em-evaluable.

3.2.2 Embedded Allowed Formulas

This subsection presents a generalised notion of 'embedded allowed' for the complex value model and gives some of its properties.
Definition 3.8 A formula $\varphi$ is embedded allowed (em-allowed) if the following conditions hold:

(a) $bd(\varphi) \models \emptyset \rightarrow free(\varphi)$;

(b) for each sub-formula $\exists \bar{x}\psi$ of $\varphi$, $bd(\psi) \models free(\exists \bar{x}\psi) \rightarrow [\bar{x} \cap free(\psi)]$;

(c) for each sub-formula $\forall \bar{x}\psi$ of $\varphi$, $bd(-\psi) \models free(\forall \bar{x}\psi) \rightarrow [\bar{x} \cap free(\psi)]$;

(d) for each sub-formula $x \subseteq \{y \mid \phi(y)\}$, $bd(\phi) \models free(\phi) \rightarrow y$.

Example 3.4 Consider the formula

$$\varphi = p(x, y) \land \exists t(u = f(x, y) \land \neg q(u, t) \land t = f(u)).$$

We have $\emptyset \rightarrow xy, xy \rightarrow u$ and $u \rightarrow t$. We can get $\emptyset \rightarrow xuy$ which satisfies the condition (a). $u \rightarrow t$ satisfies the condition (b). So $\varphi$ is em-allowed.

Example 3.5 Consider the formula

$$\varphi(x, y) = \exists z \exists u \phi(x, y),$$

where $\phi(x, y) = (R(z) \land S(u) \land x \subseteq \{y \mid (y + z.A) \in u.C\})$.

$bd(\phi) = \{\emptyset \rightarrow zu, zu \rightarrow y, zuy \rightarrow x\}^*$. We have $bd(\varphi) \models \emptyset \rightarrow xy$ (i.e., $\emptyset \rightarrow free(\varphi)$) which satisfies the condition (a), $bd(\phi) \models \emptyset \rightarrow zu$, which satisfies the condition (b) (i.e., $free(\exists z \exists u \phi) \rightarrow \{z, u\}$) and $zu \rightarrow y$ which satisfies the condition (d). So $\varphi$ is em-allowed.

The following theorem shows the important relationship between the two classes of formulas.

Theorem 3.1 Every em-allowed formula is em-evaluable.

Proof: We consider the condition (b) of the em-allowed definition. As for each sub-formula $\exists \bar{x}\psi$, $bd(\psi) \models free(\exists \bar{x}\psi) \rightarrow [\bar{x} \cap free(\psi)]$, $\psi$ must not be only of the form $\neg R(\bar{t})$. If $\bar{x}$ occurs in the form $\neg R(\bar{x})$, it must also occur in some other form of $R(\bar{t})$ or $f(\bar{t}) = \bar{x}$ or $\bar{t}$ pred $t$. Therefore, by the constrained-variable procedure we get $\bar{x} \subseteq ct(\varphi)$, which satisfies the condition (b) of em-evaluable. Similarly, as the condition (c) of em-allowed holds, the condition (c) of em-evaluable holds as well. □
§3.2 Evaluable Queries

The safety condition and the equivalence of the domain-independent CALC\textsuperscript{eq}, the safe CALC\textsuperscript{eq} and the ALG\textsuperscript{eq} have been studied in [Abiteboul and Beeri 1995]. The syntactic condition known as safety ensures that each variable is range restricted, in the sense that relative to the given ordering, it is restricted by the formula to lie within the active domain of the query or the input. For example, in the case $x = f(x_1, \ldots, x_k)$ in a formula $F$, $x$ is restricted if all the $x_i$ precede $x$ in the ordering. A formula is safe relative to a given partial ordering if all the variables are restricted in it. It is easy to check that every safe formula defined in [Abiteboul and Beeri 1995] is em-allowed.

We now state two of the main results of the first part of this thesis. The proof is demonstrated in the course of translating em-allowed formulas into equivalent algebra queries; see Chapter 4.

**Theorem 3.2** Every em-allowed formula is embedded domain independent.

**Theorem 3.3** Every em-allowed formula is ef-domain independent.

By adopting the algorithm (con-to-gen) described in [Van Gelder and Topor 1991] with minor modification, every em-evaluable formula can be effectively transformed into an equivalent em-allowed formula. The relationship between these query classes we have discussed is summarised as follows:

```
safe  \longrightarrow  \text{em-allowed}  \longrightarrow  \text{em-evaluable}  \quad \leftrightarrow  \quad \text{embedded domain independent}  \quad \leftrightarrow  \quad \text{ef-domain independent}
```

The notion of em-domain independence is used only in conjunction with query languages without iterations and fails when extended to languages with fix-points [Suciu 1995]. The notion of ef-domain independence is more appropriate for queries with external functions than the notion of em-domain independence. For this reason the following theorem investigates the aspects of calculus queries with fix-points.

We first review the concept of continuous [Suciu 1995].

**Definition 3.9** A query $q$ is continuous if for any database $DB = (d, F, I)$ for which $q(DB)$ is defined, there is some finite approximation $DB_0 = (d_0, F_0, I)$ (i.e., $d_0$ is finite and $DB_0 \subseteq DB$) such that $q(DB_0) = q(DB)$. 
Let $\Sigma = \{f_1, ..., f_l\}$ be a signature. $NRA(\Sigma)$ is the nested relation algebra over $\Sigma$. Suciu [1998] proved that all queries in $NRA(\Sigma) + \text{fix}$ are ef-domain independent and continuous. It is expected that all em-allowed queries expressed in $CALC^{cv} + \mu^+$ are also ef-domain independent and continuous. The proof is shown at the end of Chapter 4 as we need first to demonstrate that every em-allowed calculus query can be translated into an equivalent algebra query.

**Theorem 3.4** All em-allowed queries in $CALC^{cv} + \mu^+$ are ef-domain independent and continuous.

### 3.3 Domain-independent Database Programs

Just as not all calculus queries are reasonable, so not all complex value database programs are reasonable. The set of correct answers to an acceptable query can depend on the language; that is, the answer to a query may not be domain-independent. The following two examples show this phenomenon.

**Example 3.6** Let $P$ be the database program:

\[
q(a) \leftarrow \\
r(x, y) \leftarrow [p(x, z) \land z = f(x)] \lor q(y)
\]

The set of answers to $P \cup \{\leftarrow r(x, y)\}$ depends on the interpretation, so $P$ is not a reasonable database.

**Example 3.7** Let $P$ be the database program:

\[
p(\{a\}) \leftarrow \\
q(x) \leftarrow q(x) \\
r(x) \leftarrow q(x) \land p(z) \land x \notin z
\]

Let $Q$ be the query $\leftarrow r(x)$. Then, if $a$ is the only constant in the domain of an interpretation, there are no answers for $P \cup \{Q\}$. But, if the domain contains any
constant \(b \neq a\), then \(\{x/b\}\) is an answer for \(\mathcal{P} \cup \{Q\}\). So \(\mathcal{P}\) is not a ‘reasonable’
database program.

In order to capture the concept of a ‘reasonable’ recursive program query, the notion
of an embedded domain-independent database program is introduced.

Let \(C_\mathcal{P}\) denote the set of constants appearing in the program \(\mathcal{P}\).

**Definition 3.10** A database program \(\mathcal{P}\) is embedded domain-independent at level \(i\) if
\(\text{ans}(\mathcal{P}, A, S_1) = \text{ans}(\mathcal{P}, A, S_2)\), for all interpretations \(S_1 = (d_1, F_1, I)\) and
\(S_2 = (d_2, F_2, I)\) that agree on \(\text{atom}(C_\mathcal{P}, I)\) to level \(i\), and for all atoms \(A\) in \(\mathcal{P}\).

Therefore, given a database \(I\) and an interpretation \(S\), a program \(\mathcal{P}\) is embedded
domain-independent if for every atom \(A\) in \(\mathcal{P}\) and for every interpretation \(S'\) which
agrees with \(S\) on \((C_\mathcal{P}, I)\), the set of answers for \(A\) is independent of the interpretation
\(S'\).

The decision problem for the class of embedded domain independent programs is
now considered. Unfortunately, the class of embedded domain independent programs
is recursively unsolvable. As for deductive databases, it is desirable to search for sub­
classes with simple decision procedures. We define the class of ‘em-allowed’ programs
and show that every em-allowed program that satisfies certain constraints is embedded
domain independent.

**Definition 3.11** A rule is em-allowed if each variable that appears in the head also
appears in the body and the body is em-allowed.

**Definition 3.12** A database program \(\mathcal{P}\) is em-allowed if each clause in \(\mathcal{P}\) is an em­
allowed formula.

**Example 3.8** The following database program is em-allowed.

\[
p(a) \leftarrow
s(x, z) \leftarrow r(x, y) \land z \subseteq y \land 5 \notin z
q(x, v) \leftarrow s(x, z) \land v = \text{count}(z)
t(x) \leftarrow p(x) \lor q(x, c)
\]
Not every \textit{em}-allowed database program with stratified negation is embedded domain-independent. The following example exhibits this phenomenon.

**Example 3.9** Let \( P \) be an \textit{em}-allowed stratified \( \mathcal{LDL} \) program:

\[
\begin{align*}
q(a) & \leftarrow \\
q(b) & \leftarrow \\
r(x,y) & \leftarrow r(x,y) \\
s(x,\{y\}) & \leftarrow r(x,y) \\
p(a) & \leftarrow \neg q(x) \land s(x,z) \land x \in z \\
t(a) & \leftarrow \neg p(a)
\end{align*}
\]

Suppose \( t \) is the selected derived relation. Then \( \{a\} \) is a set of answers for \( t \) if, and only if the domain of the interpretation contains only constants \( a \) and \( b \).

We prescribe the following two additional conditions for stratified programs using both negation and functions.

- **C1** If the rule \( p(x) \leftarrow q(y), \ldots, f(\ldots), \ldots \) is in stratum \( P_i \) and the rules defining \( q \) are in some stratum \( P_j \), then \( j < i \) if \( y \) appears in \( \text{bd}(f) \), otherwise \( j \leq i \).

- **C2** If the rule \( p(x) \leftarrow p(x) \land \ldots \) is in stratum \( P_i \) then it must include some predicate \( q \) such that the variable \( x \) appears in \( q, q \in P_j \) and \( j < i \).

The set of stratified programs satisfying the above two constraints is denoted as \( \text{Datalog}_{\text{em-strat}}^{\text{cv}} \).

**Theorem 3.5** Every query expressed in \( \text{Datalog}_{\text{em-strat}}^{\text{cv}} \) is embedded domain independent.

**Proof:** A query is expressible in \( \text{Datalog}^{\text{cv}} \) with stratified negation if and only if it is expressible in \( \text{CALC}^{\text{cv}} \) [Abiteboul and Beeri 1995]. As each rule in the program is \textit{em}-allowed, each variable is range restricted. For each stratum, the rules defining a predicate can be expressed as an \textit{em}-allowed \( \text{CALC}^{\text{cv}} \) formula. Constraints C1 and
3.4 Em-allowed, Finiteness and Domain Independence

This section discusses the relationship between em-allowed, domain independence and finiteness in the various calculus-based query languages we have described.

For \( CALC \) queries, em-allowed implies embedded domain independence, ef-domain independence and finiteness. It is not equivalent to finiteness, as the query
\[
\forall y p(x, y) \land z = f(x)
\]
is finite but not em-allowed. Clearly, the class of embedded domain independent queries is larger than that of em-allowed queries. For example, \( F(y, z) = \exists x [p(x, y) \lor q(y)] \land z = f(y) \) is embedded domain independent but not em-allowed. Although not every domain independent query is em-allowed, the embedded domain independent calculus and em-allowed calculus are equivalent. That is, for every embedded domain independent formula there exists an equivalent em-allowed formula.

For stratified \( Datalog \) queries, em-allowed does not imply embedded domain independence (see Section 3.3). Em-allowed does not imply finiteness as the query
\[
\{ p(0) \leftarrow p(n) \leftarrow p(m) \land succ(m) = n \}
\]
is em-allowed but not finite. However, em-allowed does imply a weaker form of finiteness. A database instance \( I \) can be regarded as a set of ground atomic formulas for the base predicates of a \( Datalog \) program \( P \). Let \( T^k_P \) be the set of facts about derived predicates in \( P \) that can be deduced from \( I \) by at most \( k \) applications of the rules in \( P \). A \( Datalog \) program is weakly finite if \( T^k_P(I) \) is finite for all \( k \geq 0 \) and all databases \( I \). Then em-allowed implies weak finiteness. However, em-allowed is not equivalent to weakly finite, as the query
\[
\{ p(0) \leftarrow m < 0 \land m > 0 \}
\]is weakly finite but not em-allowed.
For fix-point queries, $CALC^{cv} + \mu(\mu^+)$, em-allowed implies finite but it does not imply embedded domain-independence. By Theorem 3.4, embedded allowed formulas define $ef$-domain independent queries.
Chapter 4

Translation of Complex Value Calculus Queries

This chapter describes a non-trivial generalisation of the algorithm of [Escobar-Molano et al. 1993] for translating embedded allowed (complex value) formulas (Definition 3.8) into equivalent algebra queries. Our algorithm consists of the following four steps.

Step-1: First, replace all sub-formulas of the form $\forall y(y \in x \rightarrow \varphi(y))$ with $x \subseteq \{y \mid \varphi(y)\}$. Next, replace any remaining sub-formula of the form $\forall \phi$ by $\neg \exists \neg \phi$. Finally, rename the quantified variables if necessary.

Step-2: Transform the em-allowed formula $F$ obtained in Step-1 into an equivalent formula $F'$ in Existential Normal Form (ENF).

Step-3: Transform the formula $F'$ into an equivalent complex value algebra normal form $\psi$ (ALG$^{cv}$NF).

Step-4: Translate $\psi$ into an equivalent algebra expression $E_\psi$.

Steps (1), (2) and (3) are accomplished using families of transformations which map sub-formulas to equivalent sub-formulas, and step (4) is accomplished using transformations which map sub-formulas to algebra expressions. The major difference between our algorithm and those of [Van Gelder and Topor 1991; Escobar-Molano et al. 1993; Abiteboul and Beeri 1995] are as follows.

- The algorithms of [Van Gelder and Topor 1991; Escobar-Molano et al. 1993] apply only to the relational model, while our algorithm extends their work to the complex value model.
The algorithm of [Van Gelder and Topor 1991] does not consider external functions. The algorithm of [Escobar-Molano et al. 1993] considers formulas incorporated with scalar functions. In our algorithm, we consider domain independent queries with external functions in a general setting, by allowing the inputs and outputs of the external functions to be scalar values, sets, nested sets, etc.

The paper [Abiteboul and Beeri 1995] provides a translation from range-restricted calculus formulas into the complex value algebra. This translation is motivated from a primarily theoretical perspective. Our algorithm adopts the finite dependency approach, not considered in [Abiteboul and Beeri 1995], which influences the choice of some transformation steps.

In step (3), we introduce transformations T17 and T18 not included in the relational model [Van Gelder and Topor 1991; Escobar-Molano et al. 1993].

A rather involved construction is now presented for translating em-allowed queries into the algebra. The translation involves four steps which are described in the following four sections. Finally, we conclude the chapter by presenting the proofs of Theorems 3.2, 3.3 and 3.4.

4.1 Removing Universal Quantifiers and Renaming Variables

This step replaces sub-formulas of the form \( \forall y (y \in x \rightarrow \varphi(y)) \) with \( x \subseteq \{ y \mid \varphi(y) \} \), replaces any remaining sub-formula of the form \( \forall \varphi \) by \( -\exists \neg \varphi \) and renames the quantified variables if necessary.

**Example 4.1** Consider the following formula

\[
\varphi = (f(y) = z \land x + 5 = y) \land \neg \exists z (\neg R(x, z) \lor S(y))
\]

Applying the two transformations we obtain

\[
\varphi = (f(y) = z \land x + 5 = y) \land \neg \exists w \neg (\neg R(x, w) \lor S(y))
\]
Example 4.2 Consider the following formula

$$\varphi = \exists z R(z) \land \forall y (y \in x \rightarrow y \in z)$$

This formula can be translated as follows:

$$\varphi = \exists z R(z) \land x \subseteq \{y \mid y \in z\}$$

As shown in the following lemma, applying these transformations preserves the em-allowed property.

Lemma 4.1 If $\varphi$ is em-allowed and $\varphi_t$ is the result of applying either of these transformations to $\varphi$, then $\varphi_t$ is em-allowed.

Proof: Applying the replacement of $\forall \varphi$ by $\exists \neg \varphi$ and renaming the quantified variables preserves the em-allowed property [Escobar-Molano et al. 1993]. The sub-formula $\forall y (y \in x \rightarrow \varphi(y))$ is equivalent to $x \subseteq \{y \mid \varphi(y)\}$. As shown in Figure 3.1,

$$bd(\forall y (y \in x \rightarrow \varphi(y))) = bd(x \subseteq \{y \mid \varphi(y)\}).$$

So applying this transformation preserves the em-allowed property. $\square$

We assume that all the formulas considered in the following three sections have already been transformed in this manner.

4.2 Transforming a Formula into Existential Normal Form

This section describes the algorithm that transforms a formula into Existential Normal Form (ENF). First we review a procedure, called "simplification of a formula" [Abiteboul et al. 1995; Van Gelder and Topor 1991; Escobar-Molano et al. 1993], which is an important component in the translation. Then we present additional transformations that are necessary to get the desired form and show that each transformation preserves the em-allowed property.

Definition 4.1 A formula is simplified if and only if the following properties hold.

- There is no occurrence of $\neg \neg \varphi$.
- There is no occurrence of $\neg (\tau_1 = \tau_2)$ or $\neg (\tau_1 \neq \tau_2)$, for terms $\tau_1$ and $\tau_2$. 

• The operators 'and' (\(\land\)), 'or' (\(\lor\)) and existential quantifier (\(\exists\)) are flattened; that is,
  
  - in subformula \(\varphi_1 \land ... \land \varphi_n\), no operand \(\varphi_i\) is itself a conjunction,
  - in subformula \(\varphi_1 \lor ... \lor \varphi_n\), no operand \(\varphi_i\) is itself a disjunction, and
  - in subformula \(\exists \exists \varphi\), \(\varphi\) does not begin with \(\exists\).

• In every subformula \(\exists \exists \varphi\), each variable \(x_i\) is actually free in \(\varphi\).

The simplification of a formula is achieved by using the following equivalence-preserving rewriting rules:

1. replace \(\lnot \varphi\) by \(\varphi\); \(\lnot (t_1 = t_2)\) by \(t_1 \neq t_2\); \(\lnot (t_1 \neq t_2)\) by \(t_1 = t_2\).

2. flatten 'and's, 'or's, and existential quantifiers.

It is easy to define the algorithm SIMPLIFY, which iterates over the above transformations (i.e., rules \(T_1 - T_7\) stated in [Escobar-Molano et al. 1993]) to produce simplified formulas.

Algorithm (SIMPLIFY)

Input: Formula without universal quantifiers, \(\varphi\)

Output: A simplified formula equivalent to \(\varphi\)

begin

while some sub-formula \(x_0\) of \(\varphi\) matches any one of the following do

case T1: \(x_0 = \lnot \lnot \varphi\); \(\varphi := \) replace \(x_0\) by \(\varphi\) in \(\varphi\)

T2: \(x_0 = \lnot (\tau_1 = \tau_2)\); \(\varphi := \) replace \(x_0\) by \(\tau_1 \neq \tau_2\) in \(\varphi\)

T3: \(x_0 = \lnot (\tau_1 \neq \tau_2)\); \(\varphi := \) replace \(x_0\) by \(\tau_1 = \tau_2\) in \(\varphi\)

T4: \(x_0 = \varphi_1 \land ... \land \varphi_n\), where operand \(\varphi_i\) is a conjunction;

\(\varphi := \) replace \(x_0\) in \(\varphi\) by \(\xi_1 \land ... \land \xi_m\), where no operand \(\xi_j\) is a conjunction

T5: \(x_0 = \varphi_1 \lor ... \lor \varphi_n\), where operand \(\varphi_i\) is a disjunction;

\(\varphi := \) replace \(x_0\) in \(\varphi\) by \(\xi_1 \lor ... \lor \xi_m\), where no operand \(\xi_j\) is a disjunction
§4.2 Transforming a Formula into Existential Normal Form

T6: $\chi_0 = \exists \bar{x} \phi$, $\phi$ begins with $\exists$;

$\varphi := \text{replace } \chi_0 \text{ in } \varphi \text{ by } \exists \bar{y} \xi$, where $\xi$ does not begin with $\exists$

T7: $\chi_0 = \exists \bar{x} \phi$;

$\varphi := \text{replace } \chi_0 \text{ in } \varphi \text{ by } \exists \bar{y} \phi$, where each variable $y_i$ is actually in $\phi$

end case

end while

if there is sub-formula $\chi_0 = x \subseteq \{ y \mid \phi(y) \}$ in $\varphi$ then

$\varphi := \text{replace } \chi_0 \text{ in } \varphi \text{ by } x \subseteq \{ y \mid \text{SIMPLIFY}(\phi(y)) \}$

return $\varphi$

end

As shown below, these transformations preserve em-allowedness.

**Lemma 4.2** Given an em-allowed formula $\varphi$, SIMPLIFY($\varphi$) terminates and yields an equivalent em-allowed simplified formula.

**Proof:**

(a) As stated in the Lemma 7.3 of [Escobar-Molano et al. 1993], the SIMPLIFY algorithm terminates on all inputs.

(b) Let $\chi_0$ be a sub-formula that matches a pattern in transformations T1 - T7 and let $\chi_t$ be the corresponding transformed sub-formula. Then by [Escobar-Molano et al. 1993],

$$bd(\chi_0) = bd(\chi_t); bd(\neg \chi_0) = bd(\neg \chi_t).$$

$\chi_t$ satisfies conditions (b) and (c) of the em-allowed definition. For each sub-formula $x \subseteq \{ y \mid \phi(y) \}$ in $\varphi$, Consider $\phi_t = \text{SIMPLIFY}(\phi)$. As described above, $bd(\phi) = bd(\phi_t)$. SIMPLIFY($\varphi$) satisfies condition (d) of the em-allowed definition. We conclude that the SIMPLIFY algorithm yields an equivalent em-allowed simplified formula. □
It is convenient to think of a calculus formula in terms of its parse tree where the leaves of the tree are atomic formulas. There is a sub-formula which corresponds to each internal node labelled by $\lor, \land, \neg, \exists x$ or a sub-formula $x \subseteq \{ y \mid \varphi(y) \}$.

We now introduce the concept of Existential Normal Form and give additional transformations to put a formula into this form. Our notion of ENF is slightly different from that of [Escobar-Molano et al. 1993], in that we add item 4.

As noted in [Escobar-Molano et al. 1993], the simplified formulas can be classified into two groups, positive and negative.

**Definition 4.2** A simplified formula $\varphi$ is **negative** if and only if $\varphi \equiv \neg \phi$ for some formula $\phi$ or $\varphi \equiv t_1 \neq t_2$ for some terms $t_1, t_2$. A simplified formula is **positive** if it is not negative.

**Definition 4.3** A formula $F$ is in **Existential Normal Form** (ENF) if and only if the following conditions hold.

1. It is simplified.

2. Each disjunction in the formula satisfies:
   - the parent of the disjunction, if it has one, is $\land$, and
   - each operand of the disjunction is a positive formula.

3. The parent, if any, of a conjunction of negative formulas is $\exists$.

4. For each sub-formula $x \subseteq \{ y \mid \varphi(y) \}$ in the formula $F$, $\varphi$ is in ENF.

In order to transform a formula into ENF, it is necessary to make some transformations in the sub-formulas that violate the conditions of the previous definition.

We present the algorithm ENF, that transforms a formula into an equivalent ENF formula by simplifying the formula then alternating between applying transformations T8 - T12 and simplifying the formula, until a fixed point is reached. The result of ENF on input $\varphi$ is denoted ENF($\varphi$). The algorithm is similar to that of [Escobar-Molano et al. 1993], except that our algorithm can be applied to complex value calculus formulas.
Algorithm (ENF)

Input: Formula without universal quantifiers, \( \varphi \)
Output: An ENF formula equivalent to \( \varphi \)

begin
\[ \varphi := \text{SIMPLIFY}(\varphi) \]
while some sub-formula \( \chi \) of \( \varphi \) matches any one of the following do
  case T8: \( \chi = \neg(\psi_1 \land \ldots \land \psi_n) \), where for each \( i \), \( \psi_i \) is negative and \( \psi_i \equiv \neg \xi_i \);
  \[ \varphi := \text{replace } \chi \text{ in } \varphi \text{ by } \xi_1 \lor \ldots \lor \xi_n \]
  T9: \( \chi = \psi \land \xi_1 \land \ldots \land \xi_n \), where \( \psi \) is negative and \( \psi \equiv \neg \xi \);
  \[ \varphi := \text{replace } \chi \text{ in } \varphi \text{ by } \neg(\xi \land \neg \xi_1 \land \ldots \land \neg \xi_n) \]
  T10: \( \chi = (\theta_1 \lor \ldots \lor \theta_k) \lor \xi_1 \lor \ldots \lor \xi_n \), where \( \theta_i \) is negative and \( \theta_i \equiv \neg \lambda_i \);
  \[ \varphi := \text{replace } \chi \text{ in } \varphi \text{ by } \neg((\lambda_1 \lor \ldots \lor \lambda_k) \land \neg \xi_1 \land \ldots \land \neg \xi_n) \]
  T11: \( \chi = \neg(\psi_1 \land \ldots \land \psi_n) \);
  \[ \varphi := \text{replace } \chi \text{ in } \varphi \text{ by } (\neg \psi_1 \lor \ldots \lor \neg \psi_n) \]
  T12: \( \chi = \exists \bar{x}(\psi_1 \lor \ldots \lor \psi_n) \);
  \[ \varphi := \text{replace } \chi \text{ in } \varphi \text{ by } (\exists \bar{x}_1 \psi'_1 \lor \ldots \lor \exists \bar{x}_n \psi'_n), \] where \( \bar{x}_i \) are variables not occurring in the formula and \( \psi'_i \) is the result of renaming \( \bar{x}_i \) by \( x_i \).
end case
if there is sub-formula \( \chi = x \subseteq \{ y \mid \phi(y) \} \) in \( \varphi \) then
  \[ \varphi := \text{replace } \chi \text{ in } \varphi \text{ by } x \subseteq \{ y \mid \text{ENF}(\phi(y)) \} \]
return \( \varphi \)
end

Example 4.3 Consider the formula
\[ \varphi = \neg(\neg f(x) = y \land x \in z) \land \neg T(x) \land S(z) \]
It can be translated into
\[ \varphi' = ((f(x) = y \land x \in z) \lor T(x)) \land S(z) \]
which is in ENF.
Lemma 4.3 Given an em-allowed formula $\varphi$, the algorithm ENF terminates and yields an equivalent formula in ENF that is em-allowed.

Proof: First we prove that algorithm ENF terminates on all inputs.

(a) By Lemma 7.7 of [Escobar-Molano et al. 1993], ENF($\varphi$) terminates if $\varphi$ contains no sub-formula $x \subseteq \{y \mid \phi(y)\}$.

(b) $\varphi$ contains sub-formula $x \subseteq \{y \mid \phi(y)\}$: as ENF($\phi$) also terminates, ENF($\varphi$) terminates.

We now establish that ENF($\varphi$) is em-allowed.

(a) $\varphi$ contains no sub-formula of the form $x \subseteq \{y \mid \phi(y)\}$:

Let $\chi_0$ be a sub-formula that at some point in the execution of ENF matches the pattern of original in the transformations $T_3 - T_{12}$ and let $\chi_t$ be the corresponding transformed sub-formula. As in the relational model, either $\chi_0 = \text{pushnot}(\chi_t)$ or $\chi_t = \text{pushnot}(\chi_0)$, $bd(\chi_0) = bd(\chi_t)$ and $bd(\neg\chi_0) = bd(\neg\chi_t)$. The conditions of the em-allowed property then follow. Applying these transformations preserves the em-allowed property in the context of the complex value model.

(b) $\varphi$ contains a sub-formula of the form $x \subseteq \{y \mid \phi(y)\}$:

Let $\chi_t = x \subseteq \{y \mid \text{ENF}(\phi(y))\}$. We prove that all conditions of the em-allowed property are satisfied. By (a), ENF($\phi(y)$) preserves the em-allowed property. $bd(\phi) = bd(\text{ENF}(\phi))$. As $\text{free}(\phi(y)) = \text{free}(\text{ENF}(\phi(y)))$,

$$bd(\chi_0) = [bd(\phi)\cup\text{free}(\phi(y)) \to x] = [bd(\text{ENF}(\phi))\cup\text{free}(\text{ENF}(\phi(y))) \to x] = bd(\chi_t).$$

$bd(\neg\chi_0) = bd(\neg\chi_t)$. Conditions (a), (b) and (c) of the em-allowed property are satisfied. Since $\chi_0$ satisfies condition (d) of the em-allowed property, $bd(\phi) \models \text{free}(\phi) \to y$. Because

$$bd(\phi) = bd(\text{ENF}(\phi)), \quad bd(\text{ENF}(\phi)) \models \text{free}(\text{ENF}(\phi)) \to y.$$

$\chi_t$ satisfies condition (d) of the em-allowed property.

We conclude that ENF($\varphi$) yields an equivalent formula in ENF that is em-allowed. $\square$
### 4.3 Transforming an ENF Formula into Complex Value Algebra Normal Form

This section presents the translation of an ENF formula into a formula in Complex Value Algebra Normal Form (ALGcvNF). We start by defining the complex value algebra normal form, and then give necessary transformation rules not present in the relational model.

First, following [Topor 1991], we introduce the concept of a *maximal* sub-formula in the context of the complex value model.

**Definition 4.4** A sub-formula $G$ of a formula $F$ is *maximal* if either

- $G$ is $F$, or
- $G$ is a positive non-arithmetic operand of the operator $\land$, or
- $G$ is a child of one of the operators $\exists, \forall, \neg$ or a range formula $x \subseteq \{y \mid \varphi(y)\}$.

**Example 4.4** Let $F$ be the formula

$$(p(x) \lor \neg q(x)) \land \neg r(x) \land x < y \land x \subseteq \{t \mid s(z) \land t \in z\}. $$

Then the maximal sub-formulas of $F$ are $p(x)$, $q(x)$, $\neg q(x)$, $(p(x) \lor \neg q(x))$, $r(x)$, $x \subseteq \{t \mid s(z) \land t \in z\}$, $s(z)$, $t \in z$, $s(z) \land t \in z$ and $F$ itself. The sub-formula $\neg r(x)$ and $x < y$ are not maximal.

Our aim is to transform a given em-allowed query into an equivalent query, all of whose maximal sub-queries can be effectively translated. Intuitively, for each maximal sub-formula, we hope to build an equivalent complex value algebra expression. If a sub-formula is not maximal, it will be used to augment an algebra query already constructed from its siblings. During this step, the function $bd$ is crucial to decide whether a sub-formula is em-allowed.

**Definition 4.5** An em-allowed formula $F$ is in *complex value algebra normal form* (ALGcvNF) if $F$ is in ENF and every maximal sub-formula of $F$ is em-allowed.
In addition to the rules T13 to T16 stated in [Escobar-Molano et al. 1993], we need the following rules to transform ENF em-allowed formulas to complex value algebra normal form.

Rule T17: \( \xi_1 \land ... \land \xi_m \land x \subseteq \{ y \mid \phi(y) \} \rightarrow \xi_1 \land ... \land \xi_m \land x \subseteq \{ y \mid \phi(y) \land \xi_{i_1} \land ... \land \xi_{i_k} \} \)

where \( \phi(y) \) is not em-allowed, but \( \phi(y) \land \xi_{i_1} \land ... \land \xi_{i_k} \) is em-allowed.

Rule T18: \( \psi \land x \subseteq \{ y \mid \phi(y) \land \xi_1 \land ... \land \xi_m \} \rightarrow \psi \land \xi_{i_1} \land ... \land \xi_{i_k} \land x \subseteq \{ y \mid \phi(y) \land \xi_{i_1} \land ... \land \xi_{i_k} \} \)

where \( \psi \) is not em-allowed, but \( \psi \land \xi_{i_1} \land ... \land \xi_{i_k} \) is em-allowed.

Now we present the algorithm ALG\( ^{cv} \)NF, that transforms an em-allowed formula in ENF into an equivalent formula which is in ALG\( ^{cv} \)NF. The output of ALG\( ^{cv} \)NF on input \( \varphi \) is denoted ALG\( ^{cv} \)NF(\( \varphi \)).

**Algorithm (ALG\( ^{cv} \)NF)**

*Input*: Formula without universal quantifiers, \( \varphi \)

*Output*: An ALG\( ^{cv} \)NF formula equivalent to \( \varphi \)

\[
\text{begin}
\text{while some sub-formula } \chi_0 \text{ of } \varphi \text{ matches any one of the following do}
\text{case T13: } \chi_0 = \exists y \phi \land \xi_1 \land ... \land \xi_n, \text{ where } \phi \text{ is not em-allowed;}
\quad \alpha := \text{ALG}^{cv} \text{NF(SIMPLIFY}(\phi \land \xi_{i_1} \land ... \land \xi_{i_k}))\),
\quad \text{where } \phi \land \xi_{i_1} \land ... \land \xi_{i_k} \text{ is em-allowed}
\quad \chi_t := \exists y \alpha \land \xi_{i_{k+1}} \land ... \land \xi_{i_n}
\quad \varphi := \text{replace } \chi_0 \text{ by } \chi_t \text{ in } \varphi
\text{T14: } \chi_0 = (\phi_1 \lor ... \lor \phi_m) \land \xi_1 \land ... \land \xi_n, \text{ where } \phi_1 \lor ... \lor \phi_m \text{ is not em-allowed;}
\text{for } i = 1, m \text{ do}
\quad \alpha_i := \text{ALG}^{cv} \text{NF(SIMPLIFY}(\phi_i \land \xi_{i_1} \land ... \land \xi_{i_k}))\),
\quad \text{where } \phi_i \land \xi_{i_1} \land ... \land \xi_{i_k} \text{ is em-allowed}
\quad \chi_t := (\alpha_1 \lor ... \lor \alpha_m) \land \xi_{i_{k+1}} \land ... \land \xi_{i_n}
\text{end}
\]
§4.3 Transforming an ENF Formula into Complex Value Algebra Normal Form

\varphi := \text{replace } \chi_0 \text{ by } \chi_t \text{ in } \varphi

\varphi := \text{ENF}(\varphi)

T15: \( \chi_0 = \neg \phi \land \xi_1 \ldots \land \xi_n \), where \( \phi \) is not em-allowed;

\[ \alpha := \text{ALG}^{cv}\text{NF}(\text{SIMPLIFY}(\phi \land \xi_{i_1} \land \ldots \land \xi_{i_k})), \]

where \( \phi \land \xi_{i_1} \land \ldots \land \xi_{i_k} \) is em-allowed

\[ \beta := \text{ALG}^{cv}\text{NF}(\xi_1 \land \ldots \land \xi_n) \]

\[ \chi_t := \neg \alpha \land \beta \]

\varphi := \text{replace } \chi_0 \text{ by } \chi_t \text{ in } \varphi

T16 \( \chi_0 = R(\tau_1, \ldots, \tau_n) \land \xi_1 \land \ldots \land \xi_m \), where \( R(\tau_1, \ldots, \tau_n) \) is not em-allowed;

\[ \chi_t := \exists z_1, \ldots, z_n (R(z_1, \ldots, z_n) \land z_1 = \tau_1 \land \ldots \land z_n = \tau_n \land \xi_1 \land \ldots \land \xi_m) \]

\varphi := \text{replace } \chi_0 \text{ by } \chi_t \text{ in } \varphi

T17 \( \chi_0 = \xi_1 \land \ldots \land \xi_m \land x \subseteq \{ y | \phi(y) \} \), where \( \phi \) is not em-allowed;

\[ \alpha := \text{ALG}^{cv}\text{NF}(\text{SIMPLIFY}(\phi(y) \land \xi_{i_1} \land \ldots \land \xi_{i_k})) \]

\[ \chi_t := \xi_1 \land \ldots \land \xi_m \land x \subseteq \{ y | \alpha \} \]

\varphi := \text{replace } \chi_0 \text{ by } \chi_t \text{ in } \varphi

T18 \( \chi_0 = \psi \land x \subseteq \{ y | \phi(y) \land \xi_1 \land \ldots \land \xi_m \} \), where \( \psi \) is not em-allowed;

\[ \alpha := \text{ALG}^{cv}\text{NF}(\text{SIMPLIFY}(\psi \land \xi_{i_k} \land \ldots \land \xi_{i_n})) \]

\[ \chi_t := \alpha \land x \subseteq \{ y | \phi(y) \land \xi_1 \land \ldots \land \xi_m \} \]

\varphi := \text{replace } \chi_0 \text{ by } \chi_t \text{ in } \varphi

end case

end while

return \( \varphi \)

definition 4.5 Consider the formula resulting from Example 4.1:

\[ \varphi_1 = (f(y) = z \land x + 5 = y) \land \exists w(R(x, w) \land \neg S(y)) \]

As \( R(z, w) \land \neg S(y) \) is not em-allowed, we have to apply T13 of the algorithm ALG^{cv}\text{NF}. We get

\[ \varphi'_1 = (f(y) = z \land x + 5 = y) \land \exists w(R(x, w) \land \neg S(y) \land x + 5 = y). \]
**Example 4.6** Consider the following formula

$$\exists u \exists s (S(u) \land R(s) \land x \subseteq \{y \mid (y + s.A) \in u.C \} \land (t \in x \land \neg Q(t)))$$

As $$((y + s.A) \in u.C)$$ is not em-allowed, we apply the rule T17 to obtain

$$\exists u \exists s (S(u) \land R(s) \land x \subseteq \{y \mid S(u) \land R(s) \land (y + s.A) \in u.C \} \land (t \in x \land \neg Q(t)))$$

which is in $\text{ALG}^{cv}\text{NF}$.

**Lemma 4.4** Given an ENF formula $\phi$ that is em-allowed, $\text{ALG}^{cv}\text{NF}(\phi)$ terminates and yields an equivalent formula in Complex Value Algebra Normal Form.

**Proof:** We first prove that algorithm ALG$^{cv}$NF terminates on all inputs. Given an ENF formula $\phi$, $N - em(\phi)$ is the number of non-em-allowed maximal sub-formulas in $\phi$. We show that each transformation decreases $N - em(\phi)$.

- T13, T14, T15, or T16 is applied:
  These transformations decrease $N - em(\phi)$. See [Escobar-Molano et al. 1993].

- T17 is applied:
  $N - em(\phi(y) \land \xi_1 \land \ldots \land \xi_k)$ is clearly less than $N - em(\phi(y))$. Because the only transformation of SIMPLIFY applicable to $\phi(y) \land \xi_1 \land \ldots \land \xi_k$ is T4, we have
  $N - em(\text{SIMPLIFY}(\phi(y) \land \xi_1 \land \ldots \land \xi_k)) < N - em(\phi(y))$. Hence T17 decreases $N - em(\phi)$.

- T18 is applied:
  The proof is same as that of T17.

We conclude that $\text{ALG}^{cv}\text{NF}$ terminates on all inputs.

We now show that the em-allowed property is preserved. Transformations T13 - T16 preserve the em-allowed property [Escobar-Molano et al. 1993]. Hence it suffices to consider the transformations T17 and T18. We now prove that transformation T17 preserves the em-allowed property. Let

$$\chi_0 = \xi_1 \land \ldots \land \xi_m \land x \subseteq \{y \mid \phi(y)\},$$

$$\chi_t = \xi_1 \land \ldots \land \xi_m \land x \subseteq \{y \mid \phi(y) \land \xi_{i_1} \land \ldots \land \xi_{i_k}\}.$$
We verify that all conditions of the em-allowed property are satisfied.

Condition (a):

We prove that \( bd(\chi_0) \subseteq bd(\chi_t) \). Let

\[
\Sigma_0 = bd(\xi_1) \cup \ldots \cup bd(\xi_m)
\]

\[
\Gamma_0 = \Sigma_0 \cup bd(x \subseteq \{ y \mid \phi(y) \})
\]

\[
= \Sigma_0 \cup bd(\phi) \cup \{ free(\phi) \rightarrow x \}
\]

\[
\Sigma_1 = bd(\xi_{i_1}) \cup \ldots \cup bd(\xi_{i_k})
\]

\[
\Gamma_t = \Sigma_0 \cup bd(x \subseteq \{ y \mid \phi(y) \land \xi_{i_1} \land \ldots \land \xi_{i_k} \})
\]

\[
= \Sigma_0 \cup bd(\phi) \cup \{ free(\phi \land \xi_{i_1} \land \ldots \land \xi_{i_k}) \rightarrow x \}
\]

\[
= \Sigma_0 \cup \Sigma_1 \cup bd(\phi) \cup \{ free(\phi \land \xi_{i_1} \land \ldots \land \xi_{i_k}) \rightarrow x \}
\]

\[
= \Sigma_0 \cup bd(\phi) \cup \{ free(\phi \land \xi_{i_1} \land \ldots \land \xi_{i_k}) \rightarrow x \} \quad \text{(because } \Sigma_1 \subseteq \Sigma_0)\]

Then, \( bd(\chi_0) = \Gamma_0 \) and \( bd(\chi_t) = \Gamma_t \). So, it suffices to show that \( \Gamma_0 \subseteq \Gamma_t \).

We prove \( ( free(\phi(y) \land \xi_{i_1} \land \ldots \land \xi_{i_k}) \rightarrow x ) \models free(\phi(y)) \rightarrow x \). \( \phi \) is not em-allowed, \( \emptyset \not\models free(\phi) \). \( \phi \land \xi_{i_1} \land \ldots \land \xi_{i_k} \) is em-allowed, so \( \emptyset \rightarrow free(\phi \land \xi_{i_1} \land \ldots \land \xi_{i_k}) \). Therefore it can imply that \( free(\xi_{i_1} \land \ldots \land \xi_{i_k}) \rightarrow free(\phi) \) must hold. Since \( \{ free(\xi_{i_1} \land \ldots \land \xi_{i_k}) \} \cap \{ x \} = \emptyset \), \( free(\xi_{i_1} \land \ldots \land \xi_{i_k}) \not\models x \). But \( free(\phi \land \xi_{i_1} \land \ldots \land \xi_{i_k}) \rightarrow x \). So \( free(\phi) \rightarrow x \) can be implied.

It is easily seen that: \( bd(\neg \chi_0) = \emptyset^{*} \chi_0 \subseteq bd(\neg \chi_t) \). Condition (a) is satisfied.

As \( bd(\xi_0) \subseteq bd(\xi_t) \) and \( bd(\neg \xi_0) \subseteq bd(\neg \xi_t) \), conditions (b) and (c) are satisfied.

For each sub-formula \( x \subseteq \{ y \mid \phi(y) \} \), \( bd(\phi) \models free(\phi) \rightarrow y \). By fd1,

\[
bd(\phi) \models free(\phi) \rightarrow y \Rightarrow free(\phi \land \xi_{i_1} \land \ldots \land \xi_{i_k}) \rightarrow y
\]

Condition (d) is satisfied.

The proof for transformation T18 is similar to that of T17. We conclude that ALGcvNF yields an equivalent em-allowed formula in Complex Value Algebra Normal Form. \( \square \)
4.4 Transforming ALG$^{cv}$NF Formulas into Algebra Expressions

This section describes the translation of ALG$^{cv}$NF formulas into equivalent complex value algebra queries. The idea is to translate each maximal sub-formula into an algebra expression that represents the values of the free variables in the formula that satisfies it.

In general, given an ALG$^{cv}$NF formula $\varphi$ with free variables $x_1, \ldots, x_n$, we shall construct an algebra expression $E_\varphi$ over attributes $x_1, \ldots, x_n$, such that for each database input $DB$,

$$E_\varphi(DB) = \{ x_1, \ldots, x_n \mid \varphi \}(DB)$$

As in the relational model, a crucial aspect in translating an ALG$^{cv}$NF formula $\varphi_1 \land \ldots \land \varphi_n$ into the algebra is sorting the conjuncts to a modified form so that each conjunct uses only variables from preceding conjuncts. For example, each free variable in a conjunct of the form $\neg \xi$ occurs in some preceding conjunct. Then translation of these modified formulas can be effectively performed sequentially starting from the first conjunct.

**Definition 4.6** A formula $\varphi$ is in modified ALG$^{cv}$NF if it is in ALG$^{cv}$NF and each polyadic "and" sub-formula, $\varphi_1 \land \ldots \land \varphi_n$, is ordered and for each $j \in [1, n]$, the prefix $\varphi_1 \land \ldots \land \varphi_j$ is em-allowed.

If $\varphi$ is a formula in Relational Algebra Normal Form, such an appropriate ordering can be found. There is similar result for formulas in Complex Value Algebra Normal Form.

**Lemma 4.5** Suppose that $\varphi$ is a formula in ALG$^{cv}$NF and $F = \varphi_1 \land \ldots \land \varphi_n$ is a sub-formula of $\varphi$. Then there is an ordering $i_1, \ldots, i_n$ such that for each $j \in [1, n]$, $F_j = \varphi_{i_1} \land \ldots \land \varphi_{i_j}$ is em-allowed.

**Proof:** See [Escobar-Molano et al. 1993]. □

Before presenting the translation of calculus formulas into equivalent algebra expressions, we review an algebra operator called the replace operator.
4.4 Transforming $\text{ALG}^{\text{cw}}\text{NF}$ Formulas into Algebra Expressions

**Definition 4.7** If $r$ is a relation and $f$ is a function, then $\rho < f >$ is a replace operation defined as:

$$\rho < f > (r) = \{ f(t) \mid t \in r \}$$

Translating an $\text{ALG}^{\text{cw}}\text{NF}$ formula into an equivalent algebra expression can be performed by applying the following method: translate conjunctions into joins or Cartesian products, negations into generalised differences (diff) [Van Gelder and Topor 1991], existential quantifiers into projections, inequalities into selections and equalities and arithmetic operations into *appends*. Append is a relational operator defined in [Topor 1991].

**Definition 4.8** If $r$ is a relation of $l$-tuples, then the append operator, $\Delta_{\delta(i_1,\ldots,i_k)}(r)$ is a set of $l + 1$ tuples, $k \leq l$, where $\delta$ is an arithmetic operator on the components $i_1,\ldots,i_k$. The last component of each tuple is the value of $\delta(i_1,\ldots,i_k)$.

Attribute renaming is also needed for query translation. A renaming operator for a finite set $U$ of attributes is an expression $\zeta_f$, where $f$ is an attribute renaming for $U$. $f$ is usually written as $A_1,\ldots,A_n \rightarrow B_1,\ldots,B_n$ to indicate $f(A_i) = B_i$.

We first consider atoms that can be translated independently of a surrounding formula.

<table>
<thead>
<tr>
<th>Sub-formula to translate</th>
<th>Algebra expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(i)$</td>
<td>$\pi_x(\sigma_f(R))$</td>
</tr>
<tr>
<td>$x \in t$</td>
<td>$(E_t)$</td>
</tr>
<tr>
<td>$x = t$</td>
<td>${ &lt; t &gt; } \mid (d, F)$</td>
</tr>
<tr>
<td></td>
<td>i.e., the expression denoting the unary relation containing a single tuple with value $t$ as evaluated under $(d, F)$.</td>
</tr>
<tr>
<td>$x \subseteq t$</td>
<td>$\text{powerset}(E_t)$</td>
</tr>
<tr>
<td>$x \subseteq { y \mid \phi(y) }$</td>
<td>$\rho &lt; \text{powerset} \geq \text{nest}<em>{X=y}(E</em>\phi)$</td>
</tr>
</tbody>
</table>

We next consider non-atomic maximal sub-formulas.
Sub-formula to translate

\[ \xi_1 \lor \ldots \lor \xi_n \]  
\[ \exists \exists \varphi \]  
\[ \neg \xi \]

Algebra expression

\[ E_{\xi_1} \cup \ldots \cup E_{\xi_n} \]  
\[ \pi_{\text{free}(\varphi)} - \{x\}(E_{\varphi}) \]  
\[ \{<>\} - E_{\xi} \]

Finally, we give the translation for formulas of the form \( \varphi_1 \land \ldots \land \xi_n \), where \( n > 1 \). We only consider the case of \( \psi \land \xi \) here, as each prefix \( \xi_j = \xi_1 \land \ldots \land \xi_j \) is em-allowed.

Sub-formula to translate

\[ \psi \land \xi \]

Algebra expression

\[ \sigma_{\theta}(E_{\psi}), \theta \text{ is a selection list derived from } t_1 = t_2. \]  
\[ \text{if } \xi \text{ has the form } t_1 = t_2 \text{ or } x \neq t, \text{ where } t_1 \text{ and } t_2 \text{ have only variables in } \psi. \]  
\[ \Delta_{\delta}(E_{\psi}), \delta \text{ is a function derived from term } t. \]  
\[ \text{if } \xi \text{ has the form } t = x, \text{ variable } x \text{ is not free in } \psi, \text{ and all variables in } t \text{ are free in } \psi. \]  
\[ E_{\psi} \land E_{\xi} \]  
\[ \text{if } \psi \text{ and } \xi \text{ are both constructive and have overlapping sets of free variables.} \]  
\[ E_{\psi} \times E_{\xi} \]  
\[ \text{if } \psi \text{ and } \xi \text{ are both constructive and do not have overlapping sets of free variables.} \]  
\[ \psi \land \neg \xi' \]  
\[ E_{\psi} \text{ diff } E_{\xi'} \]  
\[ \text{if } \text{free}(\xi') \subset \text{free}(\psi). \]  
\[ E_{\psi} - E_{\xi'} \]  
\[ \text{if } \text{free}(\xi') = \text{free}(\psi). \]

Let ALG\(^{\text{nf}}\) formula \( \varphi \) be fixed. The construction of \( E_{\varphi} \) is inductive, from leaf to root, and is sketched in the following algorithm.
Algorithm (Translation into the Algebra)

Input: a formula $\varphi$ in modified ALG$^{cv}$NF

Output: an algebra query $E_\varphi$ equivalent to $\varphi$

begin

case $\varphi$ of

$R(\vec{t})$ 
$E_t$ 

$x \in t$ 
$\{<t> | (d, F)\}$ 

$x \subseteq t$ 
powerset($E_t$) 

\begin{align*}
x &\subseteq \{y \mid \phi(y)\} \\
\rho &< \text{powerset} > (nest_{X=y}(E_\varphi))
\end{align*}

\begin{align*}
\psi \land \xi \\
\text{case } \xi \text{ of} \\
\begin{align*}
t_1 = t_2, & \text{ free}(t_1) \subseteq \text{ free}(\psi) \text{ and free}(t_2) \subseteq \text{ free}(\psi) \\
& \sigma_\theta(E_\psi), \theta \text{ is a selection list derived from } t_1 = t_2 \\
t = x, & \text{ x } \notin \text{ free}(\psi) \text{ and free}(t) \subseteq \text{ free}(\psi) \\
& \Delta_\delta(E_\psi), \delta \text{ is a function derived from term } t \\
x \neq t, & \text{ free}(x) \subseteq \text{ free}(\psi) \text{ and free}(t) \subseteq \text{ free}(\psi) \\
& \sigma_\theta(E_\psi), \theta \text{ is a selection list derived from } x \neq t \\
x \in t, & x \text{ is not free in } \psi \text{ and all variables in } t \text{ are free in } \psi \\
& E_\psi \times \text{set\_destroy}(\pi_X(\psi)), \text{ where } X = \text{ free}(t) \\
\neg \xi', & E_\psi \text{ diff } E_\xi', \text{ if free}(\xi') \subset \text{ free}(\psi) \\
& E_\psi - E_\xi' \text{ if free}(\xi') = \text{ free}(\psi)
\end{align*}

end case

if free(\psi) \cap free(\xi) = \emptyset, \text{ then } E_\psi \times E_\xi \\
else, E_\psi \bowtie E_\xi

\begin{align*}
\neg \xi & \{<>\} - E_\xi \\
\xi_1 \lor \ldots \lor \xi_n & E_{\xi_1} \cup \ldots \cup E_{\xi_n} \\
\exists \vec{x} \psi & \pi_{\text{free}(\psi) - \{\vec{x}\}}(E_\psi)
\end{align*}

end case

end
We conclude the section by presenting some examples.

**Example 4.7** The formula

$$\varphi = f(x) = y \land \neg P(x, z) \land g(x, y) = z \land R(x)$$

can be re-ordered as

$$\varphi' = R(x) \land f(x) = y \land g(x, y) = z \land \neg P(x, z)$$

In turn this can be translated into an algebra expression as follows:

$$
egin{align*}
E_1 & := R \\
E_2 & := \Delta_{\delta(1)}(E_1) \\
E_3 & := \Delta_{\delta(1,2)}(E_2) \\
E_4 & := E_3 \text{ diff } P
\end{align*}
$$

**Example 4.8** Consider the formula

$$\varphi = \exists z(r(z) \land x \subseteq \{y \mid y \in z\})$$

The algorithm $ALG^{cv}NF$ will apply $T_{17}$ to obtain

$$\varphi' = \exists z(r(z) \land x \subseteq \{y \mid r(z) \land y \in z\})$$

The range formula for $x$ contains the free variables $z$ and $y$. So the type of the corresponding algebra query is a set of pairs, $(z, y)$-values. By the above algorithm (translation into the algebra), we need $nest$ and $powerset$ operators. Finally, we join $r$ with the algebra query we obtained for the range formula for $x$, and then perform projection for the existential quantifier $z$. The equivalent algebra query is obtained using the program

$$
egin{align*}
E_1 & := r \times \zeta_{z \rightarrow y \text{set.destroy}(r)} \\
E_2 & := \text{nest}_{X=y}(E_1) \\
E_3 & := \rho < \text{powerset}(X) > (E_2) \\
E_4 & := r \Join E_3 \\
E_5 & := \pi_X(E_4)
\end{align*}
$$
Example 4.9 Consider the formula

$$\exists z \exists u (R(z) \land S(u) \land t \in \{ y \mid y + z.A \in u.C \} \land \neg Q(t))$$

Transforming it into $\text{ALG}^{cv}$NF yields

$$\exists z \exists u (R(z) \land S(u) \land t \in \{ y \mid R(z) \land S(u) \land y + z.A \in u.C \} \land \neg Q(t))$$

The equivalent algebra query is obtained using the following program

$$
E_1 := \text{tup} \cdot \text{create}\_C' (\text{set} \cdot \text{destroy}(\text{tup} \cdot \text{destroy}(\pi_C(S))));
$$

$$
E_2 := \pi'_{A'}(\sigma'_{\in C}(S \times E_1));
$$

$$
E_3 := E_2 \times (\pi_A(R));
$$

$$
E_4 := \Delta_{(2,3)}(E_3);
$$

$$
E_5 := \text{nest}_{C'}(\pi_{Y \rightarrow X}(E_4));
$$

$$
E_6 := \pi_{A', X, Y}(E_5);
$$

$$
E_7 := \text{replace} < [A', X, Z = (\zeta_{Y \rightarrow t}(X)) \ \text{diff} \ Q] > (E_6)
$$

$$
E_8 := \pi_{X, Z}(E_7)
$$

where $\delta(2, 3) = \text{column } 2 - \text{column } 3$. Note that $E_2$ is equivalent to $\text{unnest}_C(S)$.

4.5 Proofs of Theorems 3.2, 3.3 and 3.4

We have presented the main results of this chapter (namely, the translation of em-allowed complex value formulas into equivalent algebra queries.) In this section, we present the proofs of the main results of Chapter 3. We first show that every em-allowed formula is embedded domain independent and external-function domain independent. We then prove that all em-allowed queries in $\text{CALC}^{cv} + \mu^+$ are external-function domain independent and continuous.

Theorem 3.2 Every em-allowed formula is embedded domain independent.

Proof: By lemmas 4-1, 4-2, 4-3 and 4-4, any em-allowed formula can be effectively
translated into an equivalent algebra query. Because the algebra is embedded domain
independent, \( \varphi \) is embedded domain independent. \( \Box \)

As described in Chapter 2, the complex value algebra \( \text{ALG}^{cv} \) is a functional lan-
guage based on a small set of operations. An important subset of \( \text{ALG}^{cv} \), denoted
\( \mathcal{NRA} \), is formed from the core operators of \( \text{ALG}^{cv} \) by removing the powerset operator
and adding the nest operator\(^1\).

**Theorem 3.3** Every em-allowed formula is ef-domain independent.

*Proof:* By Theorem 3.2, every em-allowed formula can be translated into an equiva-
lent complex value algebra (\( \text{ALG}^{cv} \)) query with external functions. All queries in the
nested relational algebra \( \mathcal{NRA} + \text{fix} \) are ef-domain independent [Suciu 1995]. \( \mathcal{NRA} \)
is equivalent to \( \text{ALG}^{cv} \) without powerset with external functions.

Now we prove that the query powerset is ef-domain independent. Consider two
databases \( \text{DB}_1 = (d_1, F_1, I_1) \) and \( \text{DB}_2 = (d_2, F_2, I_2) \). For any morphism
\( \xi : \text{DB}_1 \rightarrow \text{DB}_2 \), for every \( i \), \( \xi(R_i) \) is defined and \( \xi(R_i) = R'_i \), where \( R_i \in I_1, R'_i \in I_2 \).
\( \xi(\text{powerset}(q(\text{DB}_1))) = \text{powerset}(q(\text{DB}_2)) \), for any query \( q \). So powerset is ef-domain
independent. Therefore, every em-allowed formula is ef-domain independent. \( \Box \)

If all external functions are computable, it is easy to get the following result.

**Corollary** Every em-allowed formula defines an ef-domain independent, computable
query.

**Theorem 3.4** All em-allowed queries in \( \text{CALC}^{cv} + \mu^+ \) are ef-domain independent
and continuous.

*Proof:* We first prove that all em-allowed queries \( \in \text{CALC}^{cv} \) are ef-domain inde-
dependent and continuous. As demonstrated in the course of translating em-allowed
formulas in \( \text{CALC}^{cv} \) into equivalent algebra queries in \( \text{ALG}^{cv}(\Sigma) \), every em-allowed
query in \( \text{CALC}^{cv} \) is equivalent to an algebra query in \( \text{ALG}^{cv}(\Sigma) \). We know that all
queries in the nested relational algebra \( \mathcal{NRA}(\Sigma) + \text{fix} \) are ef-domain independent and

\(^1\)See [Breazu-Tannen et al. 1992] for a detailed description.
continuous [Suciu 1995]. \( \mathcal{NRA}(\Sigma) \) is equivalent to \( \text{ALG}^{cv}(\Sigma) \) without powerset with external functions.

By Theorem 3.3, the query \textit{powerset} is \( ef \)-domain independent. As \textit{powerset} is finite, \textit{powerset} is continuous. Therefore CALC\(^ {cv} \) is \( ef \)-domain independent and continuous.

CALC\(^ {cv} + \mu^+ \) is equivalent to CALC\(^ {cv} \) [Abiteboul et al. 1995]. Therefore, every \( \text{em} \)-allowed queries in CALC\(^ {cv} + \mu^+ \) is \( ef \)-domain independent and continuous. \( \Box \)
Translation of Complex Value Calculus Queries
A Join Operator for Complex Value Databases

The definitions of join which have been proposed for the nested relational model are of four types: standard natural join ($\bowtie$), extended natural join ($\bowtie^e$) [Roth et al. 1988], recursive join [Colby 1990] (we denote it by $\bowtie^r$ to distinguish it from standard join $\bowtie$) and unnest-join ($\bowtie^u$) [Korth 1988]. The first three types of join are limited in power due to the fact that none of definitions is equivalent to a re-nested join of fully un-nested relations (that is, $\mu^*(r \otimes s) \neq \mu^*(r) \bowtie \mu^*(s)$, where $\mu^*$ denotes the complete un-nesting of a relation; and $\otimes$ is $\bowtie$, $\bowtie^e$ or $\bowtie^r$). However, although the fourth type of join - the unnest-join ($\bowtie^u$) - is equivalent to a re-nested join of fully un-nested relations, the necessary restructuring operations still have to be performed after this unnest-join operation.

This chapter will introduce, within a restricted set of nested schema trees, a new join operator called path join (P-join), which does not require as many restructuring operators and combines the advantages of the extended natural join and recursive join for efficient data access. In addition, this chapter compares this P-join operator with other join operators and derives some algebraic equivalences related to P-join operator. These results can be used for query optimisation in the nested relational model.

The structure of this chapter is organised as follows. Section 5.1 briefly reviews some well-known concepts. Section 5.2 presents the P-join. Section 5.3 compares the P-join operator with other join operators. Some algebraic equivalences and query optimisation related to the P-join operator are illustrated in Section 5.4. The P-join operator with multiple join-paths and algebraic properties related to it are presented in Section 5.5.
Finally, Section 5.6 establishes the correctness of P-join for nested relations.

5.1 Basic Concepts

We introduce some notational conventions. We assume all attributes of our relations are contained in \( \text{att} \). The elementary values are in the countable set \( \mathbf{d} \subseteq \text{dom} \). We also assume relations considered in this chapter do not contain null values. However the results of this chapter can be extended to handle relations with null values.

5.1.1 Nested Schema Tree

A nested relation schema is a relation name with an associate sort,

\[ \tau_{B_1, \ldots, B_n} \],

where \( \tau_{B_i} \) is the sort of \( B_i \) and set and tuple constructors are required to alternate in any given sort. A nested relation over a relation name \( R \) is a finite set of values of sort \( \text{sort}(R) \) (see Chapter 2).

Example 5.1 The following

\[ \tau_1 = \langle A, B, C : \{ < D, E : \{ < F, G > \} > \} > \rangle \] and
\[ \tau_2 = \langle A, B, C : \{ < E : \{ < F > \} > \} > \rangle \]

are nested relation sorts whereas

\[ \tau_3 = \langle A, B, C : < D, E : \{ < F, G > \} > > \rangle \] and
\[ \tau_4 = \langle A, B, C : \{ < F, G > \} > \rangle \]

are not.

The projection of relation \( r \) onto attributes \( N \) is denoted \( r[N] \), and similarly, the projection of tuple \( t \in r \) onto attributes \( N \) is denoted \( t[N] \). A relation structure \( \mathcal{R} \) consists of a relation schema \( R \) and an instance \( r \) defined on \( R \), and is denoted \( \langle R, r \rangle \).

A nested relation schema is structured as a rooted tree in which the nodes are labelled with elements of \( \text{att} \). Such a tree is called a schema tree. We write \( T_R \) to represent the schema tree of the relation schema \( R \). When there is no ambiguity, we
simply denote it by $T$. The set of nodes of schema tree $T$ is denoted by $\text{node}(T)$. The set of leaf nodes of schema tree $T$ is denoted by $\text{leaf}(T)$. Figure 5.1 shows a relation structure and the corresponding schema tree.

The following definitions are standard.

**Definition 5.1** Consider a schema tree $T$.

- If $N_1, N_2, ..., N_k$ is a sequence of nodes in $T$ such that $N_i$ is the parent of $N_{i+1}$, for $1 \leq i < k$, then this sequence is called the *path* from node $N_1$ to node $N_k$.

- The *length* of a path is one less than the number of nodes in the path.

- If there is a path from node $N_i$ to node $N_j$, $1 \leq i < j \leq k$, then $N_i$ is an *ancestor* of $N_j$ and $N_j$ is a *descendant* of $N_i$.

### 5.1.2 Selection Operator and Projection Operator

We need to extend the basic algebra operators, selection and projection, to work on nested relations without having to restructure the relation.

The selection and projection operators defined in [Abiteboul et al. 1995] can only be applied to attributes at the top level. The recursive algebra defined in [Colby 1990] can extract and manipulate data at all levels of nested relations. However, the selection condition can only specify attributes at the same schema level. It is expected that in practical database systems query languages can extract data at all schema levels and perform comparison on data at different schema levels.
The motivation for defining a more generalised selection operator is to allow the selection condition to be specified on attributes at different schema levels. The P-join operator (defined in the next section) combines the extended Cartesian product and this new selection operator. For the purpose of defining the selection operator, the concept of selection-comparable nodes is introduced.

**Definition 5.2** Given a schema tree $T$, for all nodes $N_a, N_b \in \text{leaf}(T)$, where $N_a \neq N_b$, if node $N_a$ is a child of an ancestor of node $N_b$, then $N_a$ and $N_b$ are called selection-comparable nodes. We denote it by $N_a \rightarrow N_b$.

For example, in Figure 5.1(b), $B \rightarrow D$, $A \rightarrow C$, but $C$ and $E$ are not selection-comparable nodes.

The expression of a node $N$ in the schema tree $T_R$ is denoted by $R_i \cdot P_i$, where $R_i \in \text{att}_R$. For example, the expression of node $D$ in the schema tree $T_S$ in Figure 5.1 is $X \cdot Z \cdot D$.

Before presenting the new selection operator, some notations are explained as follows. A predicate involved in a selection condition is expressed by $e_1 \phi e_2$, where $\phi$ is a comparison operator; $e_1$ and $e_2$ are the expressions for selection-comparable nodes in the schema tree, or one is an expression of a node and the other is a constant value (including a relation-valued constant). Comparison operators include set comparison and set membership operators, in addition to the usual arithmetic operators, that is, $\phi \in \{<, \leq, =, >, \geq, \in, \notin, \subseteq, \supseteq, \exists, =\}$.

**Definition 5.3** Let $r$ be a relation with relation schema $R$. Let $\theta = e_1 \phi e_2$ be a selection condition and let $c$ be a constant. Then $\sigma_\theta(r)$ is defined as follows:

1. $\sigma_\theta(r) = \{t \mid (t \in r) \land (\theta(t) = \text{true})\}$ if $\theta$ is a condition on $\text{att}_R$.
2. $\sigma_\theta(r) = \{t \mid (\exists t_r \in r), (t[E_R - R_i] = t_r[E_R - R_i]) \land (t[R_i] = \sigma_{\theta'}(t_r[R_i]))\}$

where $\theta' = \begin{cases} P_i^i \phi c & \text{if } \theta = R_i \cdot P_i^i \phi c \\ P_i^i \phi t_r[R_j] & \text{if } \theta = R_i \cdot P_i^i \phi R_j \\ P_i^i \phi P_i^j & \text{if } \theta = R_i \cdot P_i^i \phi R_i \cdot P_i^j \\ \end{cases}$

$R_i, R_j \in \text{att}_R$; $P_i^{i_1}, P_i^{i_2}$ are the expressions for selection-comparable nodes in subtree $T_{R_i}$. 
For example, Figure 5.2(b) shows a recursive selection, $\sigma_{X.C=c_1}$, on the relation $s$. Note that this selection operator is more general than the selection operators defined in [Colby 1990; Scholl 1986]. The selection operation $\sigma_{B=X.D}(s)$ cannot be expressed by Colby's selection operator as attributes $B$ and $D$ appear at the different schema levels.

**Definition 5.4** Let $r$ be a relation with relation schema $R$. The project-list $L$ of $R$ has the form (1) $L$ is empty, or (2) $L = (R_1L_1, \ldots, R_nL_n)$, where $R_i \in \text{attr}_R$; $L_i$ is a project-list of $R_i$. Then $\pi_L(r)$ is defined as follows:

1. $\pi(r) = r$

2. $\pi_{(R_1L_1, \ldots, R_nL_n)}(r) = \{ t \mid (\exists t_r \in r) \land (t[R_i] = \pi_{L_i}(t_r[R_i])); i \in \{1, \ldots, n\} \}$

For example, Figure 5.2(c) shows a recursive projection, $\pi_{(A,X(Z(D)))}$, on $s$. Note that the definition of this projection operator has the same meaning as the projection operator defined in [Colby 1990].

![Figure 5.2: Examples of selection and projection](image)

### 5.2 The Path Join

As mentioned in Chapter 1, the extended natural join ($\bowtie^e$) limits the relations that can participate in the join to those whose only common attributes are elements of the top level schema. For example, let $r_1$ be a relation on $R_1 = (A, X(B, C))$ and
$r_2$ be a relation on $R_2 = (B, D)$. Then $r_1 \bowtie^e r_2$ is the Cartesian product of $r_1$ and $r_2$. It is not equivalent to a re-nested join of fully un-nested relations, that is, 

$\mu_X (r_1 \bowtie^e r_2) \neq \mu_X (r_1) \bowtie \mu_X (r_2)$.

The problem with an recursive join ($\bowtie^r$) can be illustrated as follows. By the definition of $\bowtie^r$, a relation can only be joined to another relation at a fixed level. The recursive join can not work on two nested relations with common attributes at different schema levels. For example, the query of Example 1.1 cannot be expressed by recursive join without restructuring operations, as the common attributes “company”, “p-name” and “w-period” appear on different subschema levels.

The join operation is difficult to define in the nested relational model due to the possibility of different nesting paths for the attributes. The main motivation for defining the path join (P-join) is to provide a new nested relational operator which fills the role of join in the nested relational model without (or with fewer) restructuring operations.

In order to define the P-join operator, we also need to extend the Cartesian product to work on nested relations.

5.2.1 Extended Cartesian Product and Path Cartesian Product

First we extend the Cartesian product definition which operates on two relations with common higher-order attributes. The idea behind this extended Cartesian product is that we form the Cartesian product by combining two operands not only at the top level but also at the sub-schema levels.

Let "\( \circ \)" denote the concatenation operator for tuples as well as relation schemas. For example, if $t_1 = (1, 2)$ and $t_2 = (2, 4)$, then $t_1 \circ t_2 = (1, 2, 2, 4)$. Similarly, if $\text{att}_R = (A, X)$, $\text{att}_Q = (B, X)$ are two schemas of relations $r$, $q$ respectively, then $\text{att}_R \circ \text{att}_Q = (A, X, B, X)$. We distinguish between the two $X$s with suffixes $X_r$, $X_q$.

**Definition 5.5** Let $r$ and $q$ be two nested relations, with schemas $R$ and $Q$ respectively. Then the extended Cartesian product, $\times^e$, is defined as follows.
1. if \( r \) and \( q \) contain no common higher-order attributes

\[
 r \times^e q = r \times q
\]

2. if \( r \) and \( q \) contain common higher-order attributes

\[
 r \times^e q = \{ t \mid (\exists t_r \in r, \exists t_q \in q), \ t[(\text{attr}_R - \bar{X}) \circ (\text{attr}_Q - \bar{X})] \\
= t_r[\text{attr}_R - \bar{X}] \circ t_q[\text{attr}_Q - \bar{X}] \\
\land t[X_i] = t_r[X_i] \times^e t_q[X_i], \forall 1 \leq i \leq k \}
\]

where \( \bar{X} = \{X_1, ..., X_k\} \) are common higher-order attributes.

The above definition provides the base domain for the extended natural join operator. This is like Cartesian product which provides a base domain for the natural join operator.

**Example 5.2** Let \( r \) and \( q \) be the two relations given in Figure 5.3(a) and (b) respectively. Figure 5.3(c) shows the extended Cartesian product of \( r \) and \( q \).

<table>
<thead>
<tr>
<th>A</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>(a) ( r )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>(b) ( q )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Bq</td>
<td>Cq</td>
<td>Dr</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(c) ( r \times^e q )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.3:** An example of extended Cartesian product

We need the auxiliary concept of join-path. Let \( R \) be a relation schema, \( T \) is the schema tree of \( R \). A path \( P_r = (N_1 \cdot N_2 \cdots N_k) \) is a join-path of \( R \) if \( N_1 \) is a child of \( \text{root}(T) \) and \( N_k \) is a non-leaf node of \( T \).
A Join Operator for Complex Value Databases

Since every node in the schema tree is labelled by an attribute name over \( \text{att} \), a join-path \( P_r \) can be expressed by the form (1) \( P_r = \emptyset \) or (2) \( P_r = R_i \cdot P'_i \), where \( R_i \in \text{att}_R; P'_i \) is a join-path of \( R_i \).

Now we define the path Cartesian product, which applies extended Cartesian products to the inside of sub-schemas along the specified paths.

**Definition 5.6** Let \( r \) and \( q \) be two relations with schemas \( R \) and \( Q \) respectively and with paths \( P_r \) and \( P_q \) respectively. Then the path Cartesian product, \( \times^p \), is defined as follows.

1. if \( P_r = P_q = \emptyset \),
   \[
   r(P_r) \times^p q(P_q) = r \times^e q
   \]
2. if \( P_r = R_i \cdot P'_i \) and \( P_q = Q_j \cdot P'_q \) are two join-paths of \( R \) and \( Q \) respectively,
   \[
   r(P_r) \times^p q(P_q) = \begin{cases} 
   
   \{ t | (\exists t_r \in r) \land (t[\text{att}_R - \{R_i\}] = t_r[\text{att}_R - \{R_i\}]) \\
   \land (t[R_i] = t_r[R_i](P'_i) \times q(P_q)) \} & \text{if length of } P_r > \text{length of } P_q \\
   
   \{ t | (\exists t_r \in r, t_q \in q) \\
   \land (t[(\text{att}_R - \{R_i\}) \circ (\text{att}_Q - \{Q_j\}]) = t_r[\text{att}_R - \{R_i\}] \times^e t_q[\text{att}_Q - \{Q_j\}]) \\
   \land (t[R_iQ_j] = t_r[R_i](P'_i) \times t_q[Q_j](P'_q))) \} & \text{if length of } P_r = \text{length of } P_q \\
   
   q(P_q) \times^p r(P_r) & \text{if length of } P_r < \text{length of } P_q
   \end{cases}
   \]

Note that the extended Cartesian product is a special case of the path Cartesian product, that is, \( r \times^e q = r \times^e q \) when two join-paths \( P_r \) and \( P_q \) are the empty set.

**Example 5.3** Let \( r \) and \( q \) be two relations of Figure 5.4(a) and (b) respectively. Figure 5.4(c) shows the path Cartesian product of \( r(X \cdot Y) \) and \( q(Z) \).
§5.2 The Path Join

5.2.2 P-join Operator

For every node in the schema tree $T_R$, there is only one path from the root to that node. So we can denote $r(N_k)$ for $r(P_r)$ if the join-path $P_r = (N_1 \cdots N_k)$. $N_k$ is called the path-determining node of $P_r$. Suppose $N_R$ and $N_Q$ are the path-determining nodes of $P_r$ and $P_q$ respectively. Then the schema of relation $r(N_R) \Join q(N_Q)$ is denoted by $R(N_R) \Join Q(N_Q)$. In order to take the join of two nested relations $r$ and $q$ we require that the attributes in the join condition can appear only on the selection-comparable nodes in the schema tree of $R(N_R) \Join Q(N_Q)$.

**Definition 5.7** Let $r$, $q$ be two relations with two join-paths $P_r$ and $P_q$ respectively. $\theta$ is the predicate on selection-comparable nodes. The $\theta$ P-join, $\Join^\theta$, is defined by

$$r(N_R) \Join^\theta q(N_Q) = \sigma_\theta(r(N_R) \Join^p q(N_Q)),$$

where $N_R$, $N_Q$ are the path-determining nodes of two join paths $P_r,P_q$ respectively.

Note that in the above definition the join condition could be a conjunction of predicates: $\theta = \theta_1 \land \theta_2 \land \cdots \land \theta_k$; where all $\theta_i, 1 \leq i \leq k$, are predicates on selection-comparable nodes in the schema tree of $R(N_R) \Join Q(N_Q)$. Thus the $\theta$ P-join can be expressed by

$$r(N_R) \Join^\theta q(N_Q) = \sigma_\theta(r(N_R) \Join^p q(N_Q)) = \sigma_{\theta_1}(\cdots(\sigma_{\theta_k}(r(N_R) \Join^p q(N_Q))))).$$

Let us consider the following example to illustrate this $\theta$ P-join concept.

![Figure 5.4: An example of path Cartesian product](image)

(a) $r$

<table>
<thead>
<tr>
<th>A</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Y</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

| 15 | 10 | 1 | 1 |
| 25 | 20 | 3 | 2 |

(b) $q$

<table>
<thead>
<tr>
<th>C</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>G</td>
</tr>
</tbody>
</table>

| 20 | 2 | 2 |
| 3 | 3 |
| 40 | 2 | 2 |

(c) $r(X \cdot Y) \Join^p q(Z)$

<table>
<thead>
<tr>
<th>A</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

| 15 | 10 | 20 | 1 | 1 | 2 | 2 |
| 10 | 40 | 1 | 1 | 2 | 2 |
| 1 | 1 | 3 | 3 |
| 2 | 2 | 3 | 3 |
| 25 | 20 | 20 | 3 | 2 | 2 | 2 |
| 3 | 2 | 3 | 3 | 3 |
| 20 | 40 | 3 | 2 | 2 | 2 |
| 2 | 2 | 40 | 2 | 2 | 40 |

Let us consider the following example to illustrate this $\theta$ P-join concept.
A Join Operator for Complex Value Databases

Example 5.4 Figure 5.5 shows the θ P-join between the relations r and q, shown in Figure 5.4(a) and (b).

The natural P-join, written \( r(N_R) \bowtie_P q(N_Q) \), is applicable only when the same atomic attributes appear on the selection-comparable nodes in the schema tree of \( R(N_R) \times Q(N_Q) \). To compute \( r(N_R) \bowtie_P q(N_Q) \) we use the following informal algorithm.

1. Compute \( r(N_R) \times q(N_Q) \).

2. For each atomic attribute with the same name \( A \) which has the same parent node, select from \( r(N_R) \times q(N_Q) \) those tuples whose values agree in columns \( P \cdot A_r \) and \( P \cdot A_q \), i.e., \( \sigma_{P \cdot A_r = P \cdot A_q} (r(N_R) \times q(N_Q)) \), where \( P \) is the expression of the parent node of \( A \). Recall that \( A_r \) corresponds to the attribute \( A \) in \( r \) and \( A_q \) is defined analogously.

3. For each attribute \( A \) above, project out \( A_q \) and rename \( A_r \) to \( A \).

4. For each atomic attribute with the same name \( B \) but with different parent nodes, apply the selection operator with selection condition \( P_1 \cdot B_r = P_2 \cdot B_q \) to \( r(N_R) \times q(N_Q) \), where \( P_1, P_2 \) are the expressions of the parent nodes of \( B_r, B_q \) respectively.

We give a formal definition as follows. First, consider predicate \( \Phi(T_R) \) to hold when the condition \( \forall N_a, N_b \in \text{leaf}(T_R) , (N_a \ceq N_b) \lor (N_b \ceq N_a) \) holds.
Definition 5.8 Let $r$ and $q$ be two relations with two join-paths $P_r$, $P_q$ respectively. If (1) $A_1, \ldots, A_k$ are the names used for both schemas $R$ and $Q$, and $A^i_r$, $A^i_q$ have the same parent node, $\forall 1 \leq i \leq k$, (2) $B^1, \ldots, B^l$ are the names used for both $R$ and $Q$ and $B^j_r$, $B^j_q$ have different parent nodes, $\forall 1 \leq j \leq l$, and (3) $\Phi(T_R(N_R), Q(N_Q))$, then the natural $P$-join is given by

$$r(N_R) \bowtie^p q(N_Q) = \sigma_{\theta_B} (\pi_X[\sigma_{\theta_A} (r(N_R) \bowtie q(N_Q))]),$$

where $N_R$, $N_Q$ are the path-determining nodes of paths $P_r$, $P_q$ respectively and

$$\theta_A = (P^1 \cdot A^1_r = P^1 \cdot A^1_q) \land (P^2 \cdot A^2_r = P^2 \cdot A^2_q) \land \cdots \land (P^k \cdot A^k_r = P^k \cdot A^k_q),$$

$\bar{X}$ is the schema of $r(N_R) \bowtie q(N_Q)$ except that the components of $P^1 \cdot A^1_q, \ldots, P^k \cdot A^k_q$, and

$$\theta_B = (P^1 \cdot B^1_r = P^1 \cdot B^1_q) \land (P^2 \cdot B^2_r = P^2 \cdot B^2_q) \land \cdots \land (P^l \cdot B^l_r = P^l \cdot B^l_q).$$

Note that the choice of join-paths is highly dependent on the relation schema structure and queries. Also note that the extended natural join is a special case of the natural $P$-join, that is, it is a $P$-join without specified paths.

Example 5.5 Figure 5.6(c) shows the natural $P$-join between $r$ and $q$, which are given in Figure 5.6(a), (b). Explicitly

$$r(Y) \bowtie^p q(Z) = \sigma_{X \cdot B_r = B_q} (\pi_{A,X(B_r,C), B_q, YZ(D_r,E,F)} [\sigma_{YZ \cdot D_r = YZ \cdot D_q} (r(Y) \bowtie q(Z))]).$$

Here by Definition 5.8, $\sigma_{\theta_A} = \sigma_{YZ \cdot D_r = YZ \cdot D_q}$; $\pi_{\bar{X}} = \pi_{A,X(B_r,C), B_q, YZ(D_r,E,F)}$; and $\sigma_{\theta_B} = \sigma_{X \cdot B_r = B_q}$. The schemas are shown in Figure 5.7.

<table>
<thead>
<tr>
<th>(a) $r$</th>
<th>(b) $q$</th>
<th>(c) $r(Y) \bowtie^p q(Z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>$q$</td>
<td>$YZ$</td>
</tr>
<tr>
<td>A</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>$r(Y)$</td>
<td>$q(Z)$</td>
<td>$YZ$</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 5.6: Natural $P$-join between $r$ and $q$
A Join Operator for Complex Value Databases

Figure 5.7: The schemas of \( r, q \) and \( r \bowtie^p q \)

Example 5.6 Figure 5.8(c) shows the natural P-join between \( r \) and \( s \), which are given in Figure 5.8(a), (b). The schemas are shown in Figure 5.9. Explicitly

\[
\begin{align*}
(r(Z) \bowtie^p s(Z)) &= \pi_{A_r,XX'}(B,C_r,ZZ' (I_r,J,r,K)), Y(D_r,E_r) \\
&= \sigma_{A_r=A_r \land XX'.C_r=XX'.C_r \land XX'.ZZ'.I_r=XX'.ZZ'.I_r \land Y.D_r=Y.D_r \land Y.E_r=Y.E_r} \pi_{X} (r(Z) \bowtie^p s(Z'))
\end{align*}
\]

By Definition 5.8, \( \sigma_{\theta_a} = \sigma_{A_r=A_r \land XX'.C_r=XX'.C_r \land XX'.ZZ'.I_r=XX'.ZZ'.I_r \land Y.D_r=Y.D_r \land Y.E_r=Y.E_r} \) and \( \pi_{X_r} = \pi_{A_r,XX'(B,C_r,ZZ'(I_r,J,K)), Y(D_r,E_r)} \)

Figure 5.8: Natural P-join between \( r \) and \( s \)
§5.3 A Comparison of the P-join and Other Join Operators

The relational query language SQL has become widely accepted as a standard by the database community, and thus many of the languages proposed for nested relational database systems, such as NF2 [Pistor and Andersen 1986], SQL/NF [Roth et al. 1987] and TQL [Thom et al. 1992], have been designed to have an SQL flavor. These SQL-like languages have a recursively defined syntax which allows a relation to be joined at any level within a hierarchical structure. Queries which can be formulated using P-join therefore can be expressed in the TQL (or SQL/NF) language. This section compares the P-join operator with other join operators.

As mentioned earlier, the join operators $\Join^1, \Join^e, \Join^r$ have shortcomings in their expressive power. When applied to nested relations, the standard natural join ($\Join$) is not a reasonable nested counterpart of the standard natural join on an equivalent set of 1NF relations. The extended natural join ($\Join^e$) operator limits the relations that can participate in the join to those whose only common attributes are elements of the top level schema [Roth et al. 1988]. In most cases, even the recursive join ($\Join^r$) cannot be applied directly without restructuring operations due to the fact that common attributes appear on different nesting paths.

We give example queries to compare $\Join^p$ with $\Join^r$ (Example 5.7) and $\Join^\mu$ (Example 5.8).

Note that the extended natural join, recursive join and unnest-join operators cannot be used to directly compute the relation of Figure 5.8(c) without using restructuring operators.

Figure 5.9: The schemas of $r$, $s$ and $r \Join^p s$
A Join Operator for Complex Value Databases

Example 5.7 Consider the example database shown in Figure 5.10(a),(b). Let us suppose that we want to list the result of the join between \( r_1 \) and \( r_2 \) by schema \( A, X(B, Z(I, J)) \) under conditions \( r_1 \cdot X \cdot B = r_2 \cdot B \) and \( r_1 \cdot X \cdot U \cdot I = r_2 \cdot V \cdot I \). The result is shown in Figure 5.10(c).

This query can be expressed as a P-join operation using the TQL language.

```
SELECT A,
    ( SELECT B,
        ( SELECT I, J
            FROM U, V
            WHERE U \cdot I = V \cdot I
        ) AS Z
    FROM X, r2
    WHERE X \cdot B = r2 \cdot B
    ) AS X
FROM r1
```

Note that a reference to a nested relation refers to the relation within the current tuple. For example X refers to the X relation within the current \( r_1 \) tuple.

The above SELECT-FROM-WHERE expression corresponds to the following algebraic formula using the P-join operator

\[
\pi_{A,X(B,Z(I,J))}[(r_1)(U) \bowtie^P r_2(V)]_{UV \rightarrow Z}
\]

where \( UV \rightarrow Z \) denotes the renaming of attributes of the joined node \( UV \). In the recursive algebra defined in [Colby 1990], the same query has to be expressed using nest and unnest:

\[
\nu_{Z=(I,J)}(\pi_{A,X(B,I,J)}(X) \bowtie^r \mu_U(r_1)) | \mu_V(r_2)).
\]
Example 5.8 Suppose we want to list the result of the query of Example 1.1. The algebraic expression using P-join is:

\[ \pi_{\text{prodname, pname, Distributor + Source(\text{company})}}(\text{Product(Distributor) } \bowtie^p \text{ Part(Source)}) \]

The same query can be expressed as an algebraic formula using \( \bowtie^\mu \):

\[ \nu_{\text{Distributor + Source=(\text{company})}} \pi_{\text{prodname, pname, company}}(\text{Product } \bowtie^\mu \text{ Part}) \]

Thus relational expressions of queries are more succinct when expressed using P-joins than when they are expressed using other join operators (\( \bowtie^r, \bowtie^\rho \)), which require restructuring operations in order to satisfy their definitions. As mentioned earlier the extended natural join and recursive join are special cases of P-join. We claim that the P-join operator is better suited for the nested relational model than the other join operators which have been proposed for this model.

5.4 Algebraic Equivalences

This section lists some algebraic equivalences of the P-join operator. These properties are very useful for query optimisation in the nested relational model. Note that every new node \( N_{R_iQ_j} \) corresponding to new attribute name \( R_iQ_j \) defined in Definition 5.6 is called a joined node.

Theorem 5.1 Let \( r_i \) be three relations with schemas \( R_i, 1 \leq i \leq 3 \) respectively. Let \( N'_{ij} = N_iN_j \), where \( i, j \in \{1, 2, 3\} \) and \( i \neq j \). Let \( N' \) be the path-determining node in \( R_1(N_1) \bowtie^p R_2(N_2) \). Then

- **Commutativity law**
  \[ r_1(N_1) \bowtie^p r_2(N_2) = r_2(N_2) \bowtie^p r_1(N_1) \]

- **Associativity law**
  \( (a) \) Let \( N' \) not be a joined node in \( T_{R_1(N_1)\bowtie R_2(N_2)} \). Then
  \[ [r_1(N_1) \bowtie^p r_2(N_2)](N') \bowtie^p r_3(N_3) = \]
  \[ 1. \ [r_1(N') \bowtie^p r_3(N_3)](N_1) \bowtie^p r_2(N_2), \text{ if } N' \in r_1 \]
2. \[ [r_2(N') \bowtie_p r_3(N_3)](N_2) \bowtie_p r_1(N_1), \text{ if } N' \in r_2 \]

(b) Let \( N' \) be a joined node in \( T_{R_1(N_1)\bowtie_p R_2(N_2)} \). Then

\[ [r_1(N_1) \bowtie_p r_2(N_2)](N') \bowtie_p r_3(N_3) = \]

1. \( r_1(N_1) \bowtie_p [r_2(N_2) \bowtie_p r_3(N_3)](N_{23}^*) \)
   \[ = [r_1(N_1) \bowtie_p r_3(N_3)](N_{13}^*) \bowtie_p r_2(N_2), \text{ if } N' = N_{12}^* \]

2. \( r_1(N_1) \bowtie_p [r_2(N_2) \bowtie_p r_3(N_3)](N_2) \)
   \[ = [r_1(N'_1) \bowtie_p r_3(N_3)](N_1) \bowtie_p r_2(N_2), \text{ if } N' = N_1N_2' \neq N_{12}^*, \text{ where} \]
   \[ N'_1 \in r_1; N'_2 \in r_2 \]

- Distributivity of unnest over P-join

(a) Let \( X \) be not a joined node. Then

\[ \mu_X(r_1(N_1) \bowtie_p r_2(N_2)) = \]

1. \( \mu_X(r_1)(N_1) \bowtie_p r_2(N_2), \text{ if } X \in \text{att}_R_1 \)

2. \( r_1(N_1) \bowtie_p \mu_X(r_2)(N_2), \text{ if } X \in \text{att}_R_2 \)

(b) Let \( X \) be a joined node. Then

\[ \mu_X(r_1(N_1) \bowtie_p r_2(N_2)) = \]

1. \( \mu_{N'_1}(r_1)(N_1) \bowtie_p \mu_{N'_2}(r_2)(N_2), \text{ if } X = N_1N_2' \neq N_{12}^* \)

2. \( \mu_{N_1}(r_1) \bowtie_p \mu_{N_2}(r_2), \text{ if } X = N_{12}^* \)

Proof:

- Commutativity law

\[ r_1(N_1) \bowtie_p r_2(N_2) = \pi_X[\sigma_g(r_1(N_1) \times p r_2(N_2))] \]

\[ = \pi_X[\sigma_g(r_2(N_2) \times p r_1(N_1)) \quad \text{(by symmetric property of } \times \text{)}] \]

\[ = r_2(N_2) \bowtie_p r_1(N_1) \]

- Associativity law

(a)(1): First consider the schema level.

By the definition of the path Cartesian product, the schema trees of the resulting relations on both sides are equivalent. By the definition of P-join, the schemas on both sides are equal after applying selection operator
on those selection-comparable nodes with the same attribute names and projecting out duplicate attributes.

Now consider the instance level.

We first prove

\[ [r_1(N_1) \times r_2(N_2)](N') \times r_3(N_3) = [r_1(N') \times r_3(N_3)](N_1) \times r_2(N_2) \]  

(5.1)

Consider \( t \in LHS \) of 5.1. There exist \( t' \in r_1(N_1) \times r_2(N_2) \) and \( t_3 \in r_3 \) such that \( t = t'(N') \times t_3(N_3) \). Since \( t' \in r_1(N_1) \times r_2(N_2) \), there must be tuples \( t_1 \in r_1, t_2 \in r_2 \) such that \( t' = t_1(N_1) \times t_2(N_2) \). So

\[ t = (t_1(N_1) \times t_2(N_2))(N') \times t_3(N_3). \]

By the associativity property of the Cartesian product, we get

\[ [t_1(N_1) \times t_2(N_2)](N') \times t_3(N_3) = [t_1(N') \times t_3(N_3)](N_1) \times t_2(N_2) \in RHS \] of 5.1. Since \( t \) is an arbitrary element, this implies that \( LHS \) of 5.1 \( \subseteq RHS \) of 5.1. Similarly, \( RHS \) of 5.1 \( \subseteq LHS \) of 5.1. Then, applying the selection operator on those selection-comparable nodes with the same attribute names on both sides of equation 5.1, we get same results, i.e.,

\[ [r_1(N_1) \bowtie r_2(N_2)](N') \bowtie r_3(N_3) = [r_1(N') \bowtie r_3(N_3)](N_1) \bowtie r_2(N_2). \]

(a)(2): The argument for (a)(1) applies equally in this case.

(b)(1): First consider the schema level.

Similarly to (a), it is obvious that both sides are equal at the schema level. Now consider the instance level.

Similarly to (a), we can get

\[ [r_1(N_1) \times r_2(N_2)](N_{12}) \times r_3(N_3) = r_1(N_1) \times [r_2(N_2) \times r_3(N_3)](N_{23}) \]

\[ = [r_1(N_1) \times r_3(N_3)](N_{13}) \times r_2(N_2) \]

Applying the selection operator on those selection-comparable nodes with the same attribute names in the above equation, we can get the result.

(b)(2): The argument for (b)(1) applies equally in this case.
Distributivity of unnest over P-join:

(a)(1): It is obvious that the schemas of resulting relations on both sides are equal. By Theorem 8.1 of [Roth et al. 1988],

\[ \mu_X(r \bowtie e s) = \mu_X(r) \bowtie e \mu_X(s) = \mu_X(r \bowtie e s). \]

By the P-join definition, the extended natural join is applied to each schema level. Therefore, if \( X \in \text{att}_R \),

\[ \mu_X(r_1(N_1) \bowtie_P r_2(N_2)) = \mu_X(r_1(N_1) \bowtie_P \mu_X(r_2(N_2)) = \mu_X(r_1(N_1) \bowtie_P r_2(N_2)). \]

(a)(2): The proof is similar to that of part (1).

(b)(1): First we consider the schema level.

Assume \( N'_1 = (M, Y(...N_1)) \); \( N'_2 = (N, Y(...N_2)) \), where \( M \) and \( N \) are sets of attributes. Let \( R_1 = (...N'_1(M, Y(...N_1))...); R_2 = (...N'_2(M, Y'(...N_2))...). \)

Then

\[ R_1 \bowtie_P R_2 = (...N'_1N'_2(M, N, YY'(...N_1N_2)...)) \]
\[ \mu_X(R_1 \bowtie_P R_2) = (...M, N, YY'(...N_1N_2)...). \]

Now we look at the right hand side. \( \mu_X(R_1) = (...M, Y(...N_1),...); \mu_X(R_2) = (...N, Y(...N_2),...). \) So

\[ \mu_X(R_1) \bowtie_P \mu_X(R_2) = (...M, N, YY'(...N_1N_2)...) = \mu_X(R_1 \bowtie_P R_2). \]

Next we consider the instance level.

\( \subseteq \) Assume \( X = (A_1, ..., A_l) \). We partition \( \mu_X(r_1 \bowtie_P r_2) \) on \( \text{att}_R - X \) and then show that all tuples \( t_1, t_2, ..., t_n \) in any partition are in \( \mu_X(r_1(N_1) \bowtie_P r_2(N_2)) \). The tuples \( t_1, t_2, ..., t_n \) must have been unnested from a set of tuples \( u_1, ..., u_k \) which form a partition on \( \text{att}_R - X \) in \( (r_1 \bowtie_P r_2) \), where for all \( i, 1 \leq i \leq n \), there exists \( j, 1 \leq j \leq k \) such that \( t_i[A_1, ..., A_l] \in u_j[X] \). We then have

\[ \bigcup_{j=1}^{k} u_j[X] = \{t_i[A_1, ..., A_l] | 1 \leq i \leq n \}. \]
Each $u_j$ was created by applying P-join on $u_j^1$ in $r_1$ and $u_j^2$ in $r_2$. So we have $\bigcup_{j=1}^{k}u_j[X] \subseteq \bigcup_{j=1}^{k}(u_j^1 \bowtie^p u_j^2)[X]$. When we unnest $r_1$ on $N_1'$ the tuples $u_j^1$ unnest into tuples $w_l^1$, $1 \leq l \leq p$. Similarly, we unnest $r_2$ on $N_2'$ the tuples $u_j^2$ unnest into tuples $w_l^2$, $1 \leq l \leq q$. Then we have

\[
\{t_i[A_1, ..., A_t] \mid 1 \leq i \leq n\} \subseteq \bigcup_{j=1}^{k}u_j[X] \subseteq \bigcup_{j=1}^{k}(u_j^1 \bowtie^p u_j^2)[X] \\
\subseteq (\bigcup_{l=1}^{p}w_l^1 \bowtie^p \bigcup_{l=1}^{q}w_l^2)[A_1, ..., A_t]
\]

Therefore $\mu_{N_1'}(r_1)(N_1) \bowtie^p \mu_{N_2'}(r_2)(N_2)$ includes all tuples $t_1, ..., t_n$.

\[\vDash\text{We show that if } t \in [(\mu_{N_1'}(r_1)(N_1) \bowtie^p \mu_{N_2'}(r_2)(N_2))] \text{ then } t \in \mu_X(r_1(N_1) \bowtie^p r_2(N_2)). \text{ } t \text{ must be the result of the P-join of some } t_1 \text{ in } \mu_{N_1'}(r_1) \text{ and some } t_2 \text{ in } \mu_{N_2'}(r_2). \text{ Also } t_1 \text{ was unnested from some } u_1 \text{ in } r_1 \text{ and } t_2 \text{ was unnested from some } u_2 \text{ in } r_2. \text{ In the P-join of } r_1 \text{ and } r_2, \text{ } u_1 \text{ and } u_2 \text{ will be joined to produce } w, \text{ where } w[X] = u_1[X] \bowtie^p u_2[X]. \text{ } w \text{ will then unnest to include } t. \text{ So we have that } t \in \mu_X(r_1(N_1) \bowtie^p r_2(N_2)).\]

(b)(2): The proof is similar to that of part (1). ∎

As shown above, the commutativity and associativity laws for the P-join operator are a natural extension of the laws for the standard join operator of the (1NF) relational algebra. We give an example to show the associativity law of P-join.

**Example 5.9** Consider the expression 5.2 in nested relational algebra including $\bowtie^p$ given below, where $r_1, r_2$ and $r_3$ are the relations shown in Figure 5.11.

\[
(r_1(X) \bowtie^p r_2(Y))(XY') \bowtie^p r_3(Z)
\] (5.2)

The relation $r_4$ in Figure 5.12 shows the result of this expression. If $r_3$ is likely to have far less matches with $r_1$ as compared to $r_2$, then reordering the operands $r_2$ and $r_3$ will be more efficient than the expression 5.2. We can easily derive the following equivalent expression 5.3 from expression 5.2 by applying associativity law (b).

\[
((r_1(X) \bowtie^p r_3(Z))(XZ) \bowtie^p r_2(Y)
\] (5.3)
A Join Operator for Complex Value Databases

<table>
<thead>
<tr>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$X$</td>
<td>$A$</td>
</tr>
<tr>
<td>$B$</td>
<td>$C$</td>
<td>$B$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$b_1$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_3$</td>
<td>$c_3$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
</tbody>
</table>

**Figure 5.11:** Nested relations $r_1$, $r_2$ and $r_3$

<table>
<thead>
<tr>
<th>$r_4$</th>
<th>$r_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$XYZ$</td>
</tr>
<tr>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
</tr>
</tbody>
</table>

**Figure 5.12:** The result of the expression 5.2 in Example 5.9

### 5.5 Decomposition P-join Operator

Due to the fact that the same attribute names in two join relations may appear in multiple subtrees, we can extend P-join with multiple join-paths to exploit the more general situation.

- Decompose each schema into sub-schemas.

- Select one sub-schema from each relation to make pairs which contain the same attributes and satisfy the P-join condition $\Phi$, and then apply P-join on each pair.

- Combine these joined relations from each pair and the remaining sub-relations which correspond to the remaining sub-schemas.

Of course, the choice of join paths is highly dependent on the relation schema structure and queries. We give the following example to illustrate this decomposition P-join concept.
§5.5 Decomposition P-join Operator

Example 5.10 Figure 5.13(c) shows an example of decomposition P-join between \( r \) and \( q \), shown in Figure 5.13(a), (b).

The mechanism of computing this decomposition P-join is:

1. We decompose schema \( R \) into two sub-schemas \( R_1(A, X, Y) \), \( R_2(A, U) \); schema \( Q \) into two sub-schemas \( Q_1(I, Y') \), \( Q_2(I, V) \).

2. The two pair of sub-schemas \((R_1, Q_1)\) and \((R_2, Q_2)\) satisfy P-join condition \( \Phi(T_{R_1(Y) \times Q_1(Y')}) \) and \( \Phi(T_{R_2(U) \times Q_2(V)}) \) respectively. We apply P-join on each pair of sub-relations \((r_1[A, X, Y], q_1[I, Y'])\), \((r_2[A, U], q_2[I, V])\).

3. Combine two joined relations using the standard natural join ie.,

\[
(r_1(Y) \Join^P q_1(Y')) \Join (r_2(U) \Join^P q_2(V)).
\]

The formal definition of the decomposition P-join follows.

### Figure 5.13

An example of decomposition P-join between \( r \) and \( q \)

<table>
<thead>
<tr>
<th>(a) ( r )</th>
<th>(b) ( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>( X )</td>
</tr>
<tr>
<td>( B )</td>
<td>( C )</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
</tr>
<tr>
<td>( b_2 )</td>
<td>( c_2 )</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( b_2 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( I )</th>
<th>( Y' )</th>
<th>( V )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D )</td>
<td>( F )</td>
<td>( K )</td>
</tr>
<tr>
<td>( i_1 )</td>
<td>( d_1 )</td>
<td>( f_1 )</td>
</tr>
<tr>
<td>( i_2 )</td>
<td>( d_1 )</td>
<td>( f_2 )</td>
</tr>
</tbody>
</table>

| (c) \( r(Y, U) \Join^P q(Y', V) \) |
|---|---|---|---|---|
| \( A \) | \( I \) | \( X \) | \( Y \) | \( Z \) | \( D \) | \( E \) | \( F \) | \( K \) | \( G \) |
| \( B \) | \( C \) | \( I \) | \( J \) |
| \( a_1 \) | \( i_1 \) | \( b_1 \) | \( c_1 \) | \( i_1 \) | \( j_1 \) | \( d_1 \) | \( e_1 \) | \( k_1 \) |
| \( b_2 \) | \( i_2 \) | \( b_2 \) | \( c_2 \) | \( i_2 \) | \( j_2 \) |
| \( a_1 \) | \( i_2 \) | \( b_1 \) | \( c_1 \) | \( i_2 \) | \( j_2 \) | \( d_1 \) | \( e_1 \) | \( f_2 \) | \( k_2 \) | \( g_2 \) |
| \( a_2 \) | \( i_2 \) | \( b_2 \) | \( c_2 \) | \( i_2 \) | \( j_2 \) | \( d_1 \) | \( e_1 \) | \( f_2 \) | \( k_2 \) | \( g_2 \) | \( d_3 \) | \( e_3 \) | \( f_3 \) |
Definition 5.9 Let \((R_1, \ldots, R_n)\) and \((Q_1, \ldots, Q_n)\) be lossless-join decompositions on schemas \(R\) and \(Q\) respectively. The intersection of all sub-schemas in each decomposition contains at least one common zero-order attribute. That is, \(\text{att}_R = \bigcup_{i=1}^{n} R_i\); \(\text{att}_Q = \bigcup_{i=1}^{n} Q_i\), and \(R_i \cap Q_i = A_i\), where \(A_r\), \(A_q\) are zero-order attributes of \(R\), \(Q\) respectively. Consider \(\forall r \in I_R, \forall q \in I_Q, L_r = (N_1^r, \ldots, N_n^r)\) and \(L_q = (N_1^q, \ldots, N_n^q)\) are lists of join-paths of \(r\) and \(q\) respectively. If \(\Phi(T_{R_i,N_j^r})p_{Q_i,N_j^q}\), \(\forall i \in \{1, \ldots, n\}\), then the decomposition \(P\)-join of two relations \(r\) and \(q\) is:

\[
r(L_r) \bowtie_p q(L_q) = \sigma_{\Phi} (\mathcal{M}(J_1, \ldots, J_n))
\]

where \(J_i = r[R_i](N_{i}^r) \bowtie_p q(Q_i)(N_{i}^q)\), \(1 \leq i \leq n\), \(\bowtie\) is the standard natural join, \(\Phi\) is the predicate, with "\(=\)" comparison operator, on those selection-comparable nodes with the same attribute names in the schema tree of \(\mathcal{M}(J_1, \ldots, J_n)\).

Query optimisation is an important issue in any database system since a good strategy can make query processing faster. We list some algebraic equivalences related to the \(P\)-join operator with multiple join paths.

Theorem 5.2 Let \(r\), \(q\), \(s\) be three relations with lists of join-paths \(L_r\), \(L_q\) and \(L_s\) respectively. Then

- **Commutativity law:**
  \[r(L_r) \bowtie_p q(L_q) = q(L_q) \bowtie_p r(L_r)\]

- **Associativity law:**
  Suppose \(L_r = (N_1^r, \ldots, N_i^r)\), \(L_s = (N_1^s, \ldots, N_i^s)\) are lists of join-paths of \(r\), \(s\) respectively; \(L_{rs} = (N_1^{rs}, \ldots, N_o^{rs})\), \(L_q = (N_1^q, \ldots, N_o^q)\) are lists of join-paths of \(r(L_r) \bowtie_p s(L_s)\), \(q\) respectively. We assume \(1 \leq k < m < n < o\) and \(m \leq l\). If

  (a) \(N_1^{rs}, \ldots, N_k^{rs}\) are joined nodes and \(N_i^{rs} = N_i^r N_i^s\), \(1 \leq i \leq k < l\)
  
  (b) \(N_{k+1}^{rs}, \ldots, N_m^{rs}\) are joined nodes and \(N_i^{rs} = \bar{N}_i^r \bar{N}_i^s\), \(k + 1 \leq i \leq m \leq l\), \(N_i^r \neq N_j^s\), \(1 \leq j \leq l\),
  
  (c) \(N_{m+1}^{rs}, \ldots, N_n^{rs}\) are not joined nodes and all \(N_i^{rs} \in r\), \(m + 1 \leq i \leq n\)
  
  (d) \(N_{n+1}^{rs}, \ldots, N_o^{rs}\) are not joined nodes and all \(N_i^{rs} \in s\), \(n + 1 \leq i \leq o\), then
[\tau(L_r) \bowtie^P s(L_s)](L_{rs}) \bowtie^P q(L_q) = [\tau(L_r) \bowtie^P q(L_q)](L_{rq}) \bowtie^P s(L_s)

where \( L'_r = (N'_1, ..., N'_i, \tilde{N}_k, ..., \tilde{N}_m, N'^{rs}_{m+1}, ..., N'^{rs}_{n}) \); \( L'_q = (N'^q_1, ..., N'^q_n) \);
\( L_{rq} = (N'^q_1, ..., N'^q_k, N'^{rs}_{k+1}, ..., N'^{rs}_{n+1}, ..., N'^{rs}_{n}) \);
\( L'_s = (N'^s_1, ..., N'^s_i, N'^{rs}_{i+1}, ..., N'^{rs}_{o}) \).

- **Distributivity of unnest over P-join:**
  Suppose \( L_r = (N'^r_1, ..., N'^r_i) \), \( L_s = (N'^s_1, ..., N'^s_i) \) are the lists of join-paths of \( r \) and \( q \) respectively.

  (a) Let \( X \) be not a joined node.
  \[
  \mu_X(\tau(L_r) \bowtie^P s(L_s)) = 
  \begin{align*}
  1. & \quad \mu_X(\tau(L_r) \bowtie^P s(L_s)), \text{ if } X \in \text{attr}_R. \\
  2. & \quad \tau(L_r) \bowtie^P \mu_X(s)(L_s), \text{ if } X \in \text{atts}.
  \end{align*}
  \]

  (b) Let \( X \) be a joined node.
  \[
  \mu_X(\tau(L_r) \bowtie^P s(L_s)) = 
  \begin{align*}
  1. & \quad \mu_{X'}(\tau(L_r) \bowtie^P \mu_X(s)(L_s)), \text{ if } X = X' \neq N'^r_i N'^s_i, 1 \leq i \leq l. \\
  2. & \quad \tau(N'^r_i)(L'_r) \bowtie^P \mu_{X'}(s)(L'_s), \text{ if } X = N'^r_i N'^s_i, i \in \{1, ..., l\}, \\
  \text{where } & \quad L'_r = (N'^r_1, ..., N'^r_{i-1}, N'^r_{i+1}, ..., N'^r_l); L'_s = (N'^s_1, ..., N'^s_{i-1}, N'^s_{i+1}, ..., N'^s_l).
  \end{align*}
  \]

**Proof:**

- **Commutativity law:**
  By Definition 5.9, \( \tau(L_r) \bowtie^P q(L_q) = \sigma_{\theta}[\bowtie (J_1, ..., J_n)] \), where
  \[
  J_i = r[R_i](N'^r_i) \bowtie^P q(Q_i)(N'^q_i), 1 \leq i \leq n.
  \]
  By the commutativity law of Theorem 5.1, \( J_i = q[Q_i](N'^q_i) \bowtie^P r[R_i](N'^r_i), 1 \leq i \leq n. \) So \( \tau(L_r) \bowtie^P q(L_q) = q(L_q) \bowtie^P r(L_r). \)

- **Associativity law:**
  Multiple path join decompose each schema into sub-schemas and select one sub-schema from each relation to make pairs. Then apply P-join on each pair. There
are four cases as stated in the associativity law of Theorem 5.1. We consider 
\[ L_{rs} = (N_{r_1}^{rs}, ..., N_{o}^{rs}) \].

1. \( N_{m+1}^{rs}, ..., N_n^{rs} \) are not joined nodes and all \( N_i^{rs} \in r, m + 1 \leq i \leq n \).
   These nodes must be in the list of \( L'_{r} \).

2. \( N_{n+1}^{rs}, ..., N_0^{rs} \) are not joined nodes and all \( N_i^{rs} \in s, n + 1 \leq i \leq o \).
   Similarly, these nodes must be in the list of \( L'_{s} \).

3. \( N_k^{rs}, ..., N_i^{rs} \) are joined nodes and \( N_i^{rs} = N_i^r N_i^s, 1 \leq i \leq k < l \).
   By (2) (b) of Theorem 5.1, \( N_i^r, 1 \leq i \leq k \), must be in the list of \( L'_{r} \).

4. \( N_{k+1}^{rs}, ..., N_m^{rs} \) are joined nodes and \( N_i^{rs} = \tilde{N}_{i}^{r} \tilde{N}_{i}^{s}, k + 1 \leq i \leq m \leq l \),
   \( N_i^{rs} \neq N_j^{r} N_j^{s}, 1 \leq j \leq l \). By Theorem 5.1, \( \tilde{N}_i^r, k + 1 \leq i \leq m \), must be in the list of \( L'_{r} \).

Similarly, we get \( L_{rq} \) and \( L'_{q} \).

- **Distributivity of unnest over P-join:**

  (a): The proof is similar to that of item 3 (a) of Theorem 5.1.

  (b)(1): As \( X = \tilde{X}^r \tilde{X}^s, \tilde{X}^r \in r, \tilde{X}^s \in s \). We consider this joined node and its corresponding join paths. By item 3 of Theorem 5.1, we get
  \[
  \mu_X(r(L_r) \bowtie s(L_s)) = \mu_{\tilde{X}^r}(r)(L_r) \bowtie \mu_{\tilde{X}^s}(s)(L_s)
  \]

  (b)(2): The proof is similar to that of (1). \( \square \)

Due to the fact that we can reorder the path determining nodes according to a corresponding one-to-one mapping in the two join-path lists without changing the result of the P-join, we assume, for simplicity, all joined nodes in the equivalence of associativity law are in consecutive order. The equivalence can be similarly applied to joined nodes in the case of arbitrary order.

The following example shows how an expression can be derived from another expression using the associativity law of P-join. This can be used to derive more efficient expressions.
Example 5.11 Let r, s and q be three relations with schemas

\[ R = (A, X(B, C), Y(E, F), Z(I, J)), \]
\[ S = (A, X'(B, D), Y'(E, G), U'(M, P)), \]
\[ Q = (A, X''(C, D), Y''(E), Z''(I, K), U''(P)). \]

Consider the following nested relational algebraic expression, which includes P-join:

\[ [r(L_r) \bowtie^p s(L_s)](L_{rs}) \bowtie^p q(L_q)] \tag{5.4} \]

where \( L_r = (X, Y), L_s = (X', Y'), L_{rs} = (XX', YY', Z, U'), L_q = (X'', Y'', Z'', U''). \)

If q is likely to have far less matches with r compared to s, then reordering the operands s and q will be more efficient than evaluating expression 5.4. We can easily derive the following equivalent expression 5.5 from expression 5.4 by applying the associative law.

\[ [r(L'_r) \bowtie^p q(L'_q)](L_{rq}) \bowtie^p s(L'_s)] \tag{5.5} \]

where \( L'_r = (X, Y, Z), L'_q = (X'', Y'', Z''), L_{rq} = (XX'', YY'', U''), L'_s = (X', Y', U'). \)

5.6 Correctness of Decomposition P-Join

This section adapts the criteria defined by Roth et al. [Roth et al. 1989], to establish the correctness of our P-join for nested relations. The criteria for correctness of an extended operator are that it is faithful and precise.

We state the formal definition of faithfulness from [Roth et al. 1989].

Definition 5.10 Let \( P \) and \( P' \) be classes of relations and \( \psi \) and \( \psi' \) binary operators on \( P \) and \( P \cup P' \) respectively. We say that \( \psi' \) is faithful to \( \psi \) if \( r\psi'q = r\psi q \) for every \( r, q \in P \) for which \( r\psi q \) is defined.

Proposition 5.1 P-join is faithful to standard natural join.

Proof: By definition, the extended natural join \((\bowtie^0)\) is the P-join with no specified
join-path. So P-join is faithful to extended natural join. Since the extended natural join is faithful to the standard natural join [Roth et al. 1989], so P-join is faithful to standard natural join. □

The following definition of preciseness is from [Roth et al. 1989].

**Definition 5.11** Let $P$ and $P'$ be classes of relations and $\psi$, $\psi'$ binary operators on $P$ and $P'$ respectively. Let $\alpha, \beta$ be operators on $P \cup P'$. We say that $\psi'$ is a precise generalization of $\psi$ relative to $\alpha, \beta$ if one of the following two conditions holds:

1. $\alpha(\beta(r\psi' q)) = \alpha(\beta(r))\psi\alpha(\beta(q))$ for every $r, q \in P'$ for which $r\psi' q$ is defined.
2. $\beta(\alpha(r\psi' q)) = \beta(\alpha(r))\psi\beta(\alpha(q))$ for every $r, q \in P'$ for which $r\psi' q$ is defined.

Let $\xi$ be the duplicate attribute elimination operator. We prove that P-join is precise as follows.

**Proposition 5.2** P-join is a precise generalization of the standard natural join with respect to unnesting and the function $\xi$ i.e.,

$$\xi(\mu^*(r(L_r) \bowtie_P q(L_q))) = \xi(\mu^*(r)) \bowtie \xi(\mu^*(q))$$

for every $r, q$ for which $r(L_r) \bowtie_P q(L_q)$ is defined, where $L_r, L_q$ are the lists of join-paths of $r, q$ respectively.

**Proof:** We show inclusion both ways.

$\subseteq$: Let $t \in LHS$. There is a tuple $\hat{t} \in \mu^*(r(L_r) \bowtie_P q(L_q))$ such that $t = \xi(\hat{t})$. Also, there must be a tuple $u \in (r(L_r) \bowtie_P q(L_q))$ such that $\hat{t} \in \mu^*(u)$. Since $u \in (r(L_r) \bowtie_P q(L_q))$, there exist $t_r \in r$ and $t_q \in q$ such that $u = t_r(L_{t_r}) \bowtie_P t_q(L_{t_q})$, where $L_{t_r} = L_r$; $L_{t_q} = L_q$. By the $\bowtie_P$ definition, $u$ must have identical values on those selection-comparable nodes with the same attribute names. So $\hat{t}$ has identical values on those duplicate attributes. Therefore $t = \xi(\hat{t}) \in \xi(\mu^*(u)) = \mu^*(t_r) \bowtie \mu^*(t_q) \subseteq \mu^*(r) \bowtie \mu^*(q) = \xi(\mu^*(r)) \bowtie \xi(\mu^*(q))$.

(by distributivity of unnest over P-join illustrated in Theorem 5.2) We get $t \in RHS$. 


\[ \triangleright: \text{Let } t \in \text{RHS}. \text{ Since } \xi(\mu^*(r)) = \mu^*(r) \text{ and } \xi(\mu^*(q)) = \mu^*(q), \text{ so } t \in \mu^*(r) \bowtie \mu^*(q). \]

Now \( t \) must be the natural join of some \( t_1 \) in \( \mu^*(r) \) and some \( t_2 \) in \( \mu^*(q) \). Also, \( t_1 \) was unnested from some \( t_r \) in \( r \) and \( t_2 \) was unnested from some \( t_q \) in \( q \). In the join \( (\bowtie^p) \) of \( r(L_r) \) and \( q(L_q) \), \( t_r \) and \( t_q \) will combine to produce \( w \). By the \( \bowtie^p \) definition, in \( w \), the same attribute names can appear only on those selection-comparable nodes and have identical values. So \( t = t_1 \bowtie t_2 \in \xi(\mu^*(t_r \bowtie^p t_q)) = \xi(\mu^*(w)) \).

Since \( w = t_r(L_{t_r}) \bowtie^p t_q(L_{t_q}) \in r(L_r) \bowtie^p q(L_q) \), we have

\[ t \in \xi(\mu^*(r(L_r) \bowtie^p q(L_q))) = \text{LHS}. \quad \square \]

With these results (propositions 1 and 2) we conclude that P-join is correct for every \( r \in I_R, q \in I_Q \) for which \( r(L_r) \bowtie^p q(L_q) \) is defined.
A Join Operator for Complex Value Databases
This chapter considers algorithms for computing the P-join and estimates the cost of using various join techniques developed in relational database systems. We first establish a cost model and then consider several methods for computing the P-join, and evaluate their estimated cost based on this model.

There are many possible ways to implement the P-join, including the following.

**nested loops with index (or hash) search techniques:** This approach consists of nested loops over the tuples in the two relations. Along the specified join-paths, apply the iteration algorithm to the two corresponding nested sub-relations with the same values on the same attributes at each outer level. The index (or hash) lookup techniques are in use at each level on each available indexed attribute.

**sort-merge join:** The implementation of the P-join algorithm is an extension of the join algorithm in first normal form (1NF) databases. In computing \( r(N_1) \bowtie s(N_2) \), if \( r \) and \( s \) have some attributes which are the same at the top level, we can sort the relations \( r \) and \( s \) using these attributes. This is similar to the join algorithm based on sort-merge. We can recursively apply the above method to each level along the join-paths.

**semi-join:** If one of two join relations (or nested sub-relations) has many tuples that do not participate in the P-join, we can apply semi-join techniques used in distributed systems to the two relations (or nested sub-relations). The idea behind the semi-join is to reduce the number of tuples in a relation before transferring it to another site. This strategy could be an efficient solution to minimizing communication costs. In computing the P-join of \( r \) and \( s \), we adopt a similar technique,
Implementation of the P-Join

i.e., project out one (or more) joining attributes of relation \( r \) and join it with the corresponding nested sub-relation where the same attribute exists. Hence those tuples of \( s \) that potentially participate in the P-join can be identified. We then invoke the nested loops join algorithm (noted in 1) or sort-merge join algorithm (noted in 2) to get the result. This approach is similar to the semi-join algorithm used in computing the unnest join. The detailed description of the concepts of participating tuples can be found in [Korth 1988].

The determination of the complexity of these join algorithms requires the knowledge of storage structure, index types, and indexed attributes available.

We focus on the nested loop (with index) and sort-merge approaches in this chapter.

6.1 Computation Mechanism for P-join

This section establishes a cost model which we shall use to evaluate the estimated cost for the join methods described in the next section.

We shall adopt the basic index techniques and join methods proposed for flat relational databases. Good surveys regarding this topic can be found in [Ullman 1989; Graefe 1993].

6.1.1 Basic Assumptions

We give cost estimates on operations based on a statistical model of relations, that is, we assume that the values are equally likely to occur in relations and all sub-relations.

The cost estimate notations are as in [Ullman 1989]. Given a relation \( R \), \( T_R \) is the number of tuples in \( R \) and \( B_R \) is the number of block accesses required to read \( R \) if \( R \) is stored packed. Let \( l_R \) be the number of bytes needed for a tuple of \( R \). The index technique is a common tool utilised to retrieve tuples of a relation quickly. The image size of an index on attribute \( A \) of relation \( R \) is denoted as \( I_{R,A} \), or simply \( I_A \).

Let \( U \) stand for the number of blocks needed to store the output of the computation. For join operations, \( U \) often dominates the total cost. Let \( M \) be the number of blocks available in main memory at any one time.
For a natural join or equi-join, we first estimate the size of the output relation. Suppose we need to compute \( R \bowtie S \). Assuming the joined domain of one relation is contained by the corresponding domain of the other relation, the output size is \( T_RT_S(l_R + l_S)/bI \), where \( b \) is the number of bytes on a block; \( I \) is the product of the domain size for each attribute shared by \( R \) and \( S \).

### 6.1.2 Storage Model

Several storage structures for nested relational databases exist, such as the flattened storage model, the decomposed storage model, and the partial decomposed storage model [Ozsoyoglu and Hafez 1993]. The choice of a specific storage structure depends on query distribution. This section presents our P-join methods based on the decomposed storage model. We briefly describe this storage model.

The decomposed storage model partitions a nested relation into several 1NF relations. The attributes combined with instances at the same level in the scheme tree are stored together in one relation. For example, the relation \( S(A, X(B, C, Y(D, E))) \) is stored as three relations: \( S_1(A), S_2(B, C), S_3(D, E) \).

### 6.1.3 Mechanism of Computing P-join

A variety of strategies is available for computing the join of two nested relations, and a query optimiser must minimise the cost of joins whenever possible. In this subsection, we consider how to compute the P-join based on the decomposed storage model. Let \( R(A, X(B, C)) \) and \( S(A, Y(B, D)) \) be two nested relations. Suppose we need to compute \( r(X) \bowtie^P S(Y) \). In what follows, we shall assume that \( R \) and \( S \) are stored packed in \( R_1(A, P_1), R_2(B, C) \) and \( S_1(A, P_2), S_2(B, D) \), where \( P_1 \) and \( P_2 \) are pointers. Their storage structures are shown in Figure 6.1.

The basic idea is that we first perform a join on \( R_1 \) and \( S_1 \) and then perform a join on subsets of \( R_2 \) and subsets of \( S_2 \) using the information contained in pointer fields resulting from \( R_1 \bowtie S_1 \). A simple P-join algorithm for computing \( r(X) \bowtie^P s(Y) \) based on the decomposed storage model is given as follows.
Implementation of the P-Join

**Figure 6.1:** The storage structures of $R$ and $S$. 

<table>
<thead>
<tr>
<th>R_1</th>
<th>R_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>P_1</td>
</tr>
<tr>
<td>a_1</td>
<td></td>
</tr>
<tr>
<td>a_2</td>
<td></td>
</tr>
<tr>
<td>a_3</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S_1</th>
<th>S_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>P_2</td>
</tr>
<tr>
<td>a_1</td>
<td></td>
</tr>
<tr>
<td>a_2</td>
<td></td>
</tr>
<tr>
<td>a_3</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
</tbody>
</table>
§6.1 Computation Mechanism for P-join

### Temp

<table>
<thead>
<tr>
<th>A</th>
<th>P₁</th>
<th>P₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>P₁(a₁)</td>
<td>P₂(a₁)</td>
</tr>
<tr>
<td>a₂</td>
<td>P₁(a₂)</td>
<td>P₂(a₂)</td>
</tr>
<tr>
<td>a₃</td>
<td>P₁(a₃)</td>
<td>P₂(a₃)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6.2:** The relation Temp

### Out₁

<table>
<thead>
<tr>
<th>A</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₂</td>
<td></td>
</tr>
<tr>
<td>a₃</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Out₂

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>b₃</td>
<td>c₃</td>
<td>d₃</td>
</tr>
<tr>
<td>b₄</td>
<td>c₄</td>
<td>d₄</td>
</tr>
<tr>
<td></td>
<td>c₃</td>
<td>d₃</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6.3:** The output of P-join

**Algorithm 1:**

1. Compute $R₁ \bowtie S₁$. The result which we call Temp is shown in Figure 6.2.

2. Read in each block $B$ of relation Temp, and for each tuple $t_i = (a_j, p_1(a_j), p_2(a_j))$ of $B$, we read in $X_i$ and $Y_i$ according the information $p_1(a_j)$, $p_2(a_j)$ contained in the pointers $P_1$ and $P_2$. Then compute $X_i \bowtie Y_i$.

3. Upon completing the execution of $X_i \bowtie Y_i$, if it is not empty, store the result into output relation $Out₂ = (B, C, D)$. Also, store the tuple $(a_j, p(a_j))$ in the output relation $Out₁ = (A, P)$, where $p(a_j)$ is the address of the first tuple of $X_i \bowtie Y_i$ in
the relation Out2. The relations Out1 and Out2 are shown in Figure 6.3.

4. Repeat steps 2 and 3 until the whole relation Temp has been processed.

6.1.4 Size Estimate of P-join Output

We can estimate the output size $U$ in terms of the parameters for $R$ and $S$. We assume attributes $A, B, C, D$ take about $l_A$, $l_B$, $l_C$, and $l_D$ bytes respectively; pointers $p_1$, $p_2$ occupy $l_p$ bytes.

The expected size of the output of step 1 in the Algorithm 1 is $\frac{T_R T_S}{l_R A}$, assuming $R_1.A \subseteq S_1.A$. For convenience, when we estimate the sizes of the resulting tuples, we assume it is an equi-join which retains both joined attributes. Thus, the number of blocks needed to store the output of $R_1 \bowtie S_1$ is approximately $U_1$, where

$$U_1 = \frac{T_R T_S (l_{R_1} + l_{S_1})}{l_A} = \frac{(T_R S_{\frac{1}{I_X}} + T_S S_{\frac{1}{I_Y}})}{I_A}$$

We now estimate the output size of step 2. The estimate size of $X_i \bowtie Y_i$ is $\frac{T_{X_i} T_{Y_i}}{I_{X_i} B}$, assuming $X_i.B \subseteq Y_i.B$. We assume that there are $m$ tuples in the output relation Temp. The total number of blocks needed to store the output Out2 is approximately $U_2$, where

$$U_2 = \sum_{i=1}^{m} \frac{T_{X_i} T_{Y_i} (l_{B_i} + l_C + l_D)}{I_{B_i}}$$

where $I_{B_i}$ is $I_{X_i}.B$ or $I_{Y_i}.B$.

6.2 Cost Estimate of Computing P-join

Three classes of join methods on 1NF relations have been described and analysed in the literature (refer to [Ullman 1989; Mishra and Eich 1992] for details). They are the nested loop, sort-merge and join using index methods. This section estimates the cost of executing the P-join algorithm described in the previous section by using these three methods.

6.2.1 Nested Loop Method

The input cost of the nested loop method on flat relations $R$ and $S$ is $B_R \cdot (\frac{B_S}{M - 1}) + B_S$. We denote it by $NL(R, S, M)$. We estimate the cost of the P-join using the nested
In step 1, the cost of the input is $NL(R_1, S_1, M)$; the cost of the output is $U_1$. In step 2, we read each block of the relation Temp in turn, and we discard it when we read the next block of Temp. For each block of Temp and for each tuple of that block, we perform the join on $X_i \bowtie Y_i$, for some $i, 1 \leq i \leq m$. The input cost of steps 2 - 4 is

$$U_1 + \sum_{i=1}^{m} NL(X_i, Y_i, M - 1)$$

Note that the number of blocks available in the main memory now is $M - 1$, because one block has been used for reading the Temp.

If $X_i \bowtie Y_i$ is not empty, we write it into the relation Out2. We also update the pointers $p_1$ and $p_2$ to only one pointer $p'$ which contains the address of the first tuple of $X_i \bowtie Y_i$ in the relation Out2. This update process is carried out in main memory with no extra block access. Once the whole block of Temp has been updated in memory, we write it into the relation Out1. As we know $l_p < l_A$, if $A$ represents multiple attributes concatenated, the estimated output cost of Out1 is $U_1$. Thus, the output cost of steps 2 - 4 is $U_1 + U_2$. The total cost of the nested loop method is

$$C_{NL} = NL(R_1, S_1, M) + 3U_1 + \sum_{i=1}^{m} NL(X_i, Y_i, M - 1) + U_2$$

### 6.2.2 Sort-merge Method

A more sophisticated method for join is the sort-merge join, which offers savings if the joining relations are large. The input cost of sort-join for two flat relations $R$ and $S$ is $2B R \log M B_R + 2B S \log M B_S + B_R + B_S$ [Ullman 1989]. We denote it by $SJ(R, S, M)$. Similarly to analysis of the nested loop method, we can estimate the total cost for the sort-merge method to be

$$C_{SJ} = SJ(R_1, S_1, M) + 3U_1 + \sum_{i=1}^{m} SJ(X_i, Y_i, M - 1) + U_2$$

### 6.2.3 Join Using Index Method

It appears to be a considerable advantage to join with a clustering index on each joining attribute. Let $R(A, B)$ and $S(B, C)$ be two flat relations. Suppose that there is a clustering index on $B$ in both relations. Let $I = I_{S.B}; J = I_{R.B}$. Assuming $S.B \subseteq R.B$, 
the input cost for $R \bowtie S$ is $(I + 2J\log_M J) \times \max(I, \frac{B_k}{J}) + \max(I, B_S)(I + 2\log_M I)$ [Ullman 1989]. We denote it by $ID(R, S, M)$.

We estimate the cost of the P-join using index method. Let $R(A, X(B, C))$ and $S(A, Y(B, D))$ be two nested relations which are stored packed in $R_1(A, p_1)$, $R_2(B, C)$ and $S_1(A, p_2)$, $S_2(B, D)$, where $p_1$ and $p_2$ are pointers.

Suppose that $R_1$ and $S_1$ have a clustering index on $A$; $R_2$ and $S_2$ have a clustering index on $B$. The total cost for P-join of $R$ and $S$ is

$$C_ID = ID(R_1, S_1, M) + 3U_1 + \sum_{i=1}^{m-1} ID(X_i, Y_i, M - 1) + U_2$$

The performance of this method could be improved by using various indexing techniques.

6.3 A P-join Algorithm

We will now develop an algorithm for the P-join with a single join path according to the join-paths specified in P-join operation expression. It recursively applies Algorithm 1 to the two relations from the top level of joining nodes.

Let relation $r$ be stored as $r_1, \ldots, r_m$; $q$ be stored as $q_1, \ldots, q_n$. We assume that the decomposed relations in the joining branch are ordered in the rear of the relation sequences.

For example, let $r$ be a relation on $R = (A, X(B, C), Y(D, E))$ and $q$ be a relation on $Q(B, Z(D, F))$. We have $r_1(A)$, $r_2(B, C)$, $r_3(D, E)$ and $q_1(B)$, $q_2(D, F)$. Suppose we need to compute $r(N_R) \bowtie q(N_Q)$. We start from the top level of the joining nodes and read the first pair of relations which have some same attribute name. (In the above example, this pair of relations is $(r_3(D, E), q_2(D, F))$) We recursively apply algorithm 1 to $(r_i, q_j)$ according to the join-paths until it reaches the path-determining nodes.

We now consider the decomposed relations in the other branches. The join operation needs to be performed in any pair of $(r_l, q_s)$, $1 \leq l \leq i - 1$; $1 \leq s \leq j - 1$, which share the same join attribute name. All join attribute names appear in selection-comparable nodes. The forward and backward pointer addresses should be updated after each join operation.
A single path P-join algorithm based on the techniques described in the Algorithm 1 is as follows.

**Algorithm spj:**

1. According to the join path, we first read in \((r_i, q_j)\), which is the first pair of relations sharing some common attribute name(s) and the parents of \(r_i\) and \(q_j\) are joining nodes.

2. Recursively apply algorithm 1 according to the join-path until it reaches the end of paths.

3. Update the pointer addresses in the upper level sub-relations.

4. Starting from the top level, join any pair of relations \((r_l, q_s)\), \(1 \leq l \leq i - 1; 1 \leq s \leq j - 1\), in which \(r_l\) and \(q_s\) have some common attribute(s).

5. Update the related forward and backward pointer addresses in the resulting scheme tree.

As joining attributes can be joined together at any level of schema trees the cost estimation of the algorithm is very complex. Therefore we are not going to conduct such an analysis here. We give a simple example as follows.

**Example 6.1** Let \(R(A, X(B, Y(C, D)), Z(E, U(F, G)))\), and \(Q(C, Z'(E, U'(F, H)))\) be two relations. Suppose we want to compute \(r(U) \bowtie q(U')\). By the decomposed storage model, \(r\) is stored as \(r_1(A), r_2(B), r_3(C, D), r_4(E), r_5(F, G)\) and \(q\) is stored as \(q_1(C), q_2(E), q_3(F, H)\). According to Algorithm spj, the P-join is computed as follows.

1. As the joining paths are \(r.Z.U\) and \(q.Z'.U'\), the first pair of join relations are \(r_4(E)\) and \(q_2(E)\). Use Algorithm 1 to join \(r_4\) and \(q_2\) and then join \(r_5\) and \(q_3\).

2. Update the pointer addresses in \(r_1(A)\) and \(q_1(C)\).

3. Join \((r_l, q_s)\), where \(l = 3, s = 1\), i.e., join \(r_3\) and \(q_1\).

4. Update the pointers in the following sequence: \(r_2 \rightarrow r_1 \rightarrow r_4q_2 \rightarrow r_5q_3\) and update \(q_1\). The symbol \(r_4q_2\) denotes the resulting joined relation of \(r_4\) and \(q_2\).
5. Project the necessary attributes according to the query.

Now consider the case of a multiple paths join. Let \((R_1, \ldots, R_n)\) and \((Q_1, \ldots, Q_n)\) be lossless join decompositions on schemes \(R\) and \(Q\) respectively. The intersection of all subschemas in each decomposition contains at least one common atomic attribute. That is, \(\bigcap_{i=1}^{n} R_i = A_r; \bigcap_{i=1}^{n} Q_i = A_q\) where \(A_r, A_q\) are atomic attributes of \(R, Q\) respectively. Let \(L_r = (N_1^r, \ldots, N_n^r)\) and \(L_q(N_1^q, \ldots, N_n^q)\) be lists of join-paths of \(r\) and \(q\) respectively. Suppose we need to compute the decomposition P-join of \(r\) and \(q\), i.e., \(r(L_r) \bowtie_q q(L_q)\). We perform the join on each pair \((R_i, Q_i)\) by using the Algorithm spj.

We briefly state a multiple paths P-join algorithm (mpj) as follows.

**Algorithm mpj:**

1. Read in \(A_r\) and \(A_q\).

2. For \(i = 1\) to \(n\), perform each single path join \(r[R_i](N_i^r) \bowtie q[Q_i](N_i^q)\) using Algorithm spj.

3. If \(r[R_i](N_i^r) \bowtie q[Q_i](N_i^q) = \emptyset\) then stop and output empty result, otherwise continue to \(i = n\).

4. Update the pointer addresses in relations \(A_r\) and \(A_q\).

### 6.4 Discussion

The join which facilitates the retrieval of the information from two different relations is one of the most difficult operations to implement efficiently. This chapter has proposed a P-join algorithm which takes advantage of a decomposed storage model and various join techniques available in the 1NF relation model to reduce the cost of join operations in the nested relational model.

At the implementation level, some researchers have focused on storage management, while others have investigated retrieval methods for operations possible with single scan [Deshpande and Van Gucht 1988; Schek and Scholl 1986]. The parallel join algorithm of [Deshpande and Larson 1992] could be applied to the P-join operator in parallel computing environments. However, solution with acceptable performance for
optimising join queries on two nested relations which have different nesting depths for joining attributes has not yet been achieved.

P-join is more efficient than the join technique which is used in existing nested relational databases (e.g., Atlas [Sacks-Davis et al. 1995]). P-join does not need to read unnecessary data.

The semantics of explicit join of classes in object-oriented data models is similar to that of P-join in the nested relational data model [Cattell 1994; Kifer et al. 1992; Tanaka and Chang 1989]. The query model of nested relational databases does not consider the core object-oriented concepts, such as class hierarchy, methods, and inheritance. However, research into complex value databases is relevant to understanding the impacts of hierarchical structures on the query model and query processing for object-oriented databases. The fundamental techniques required for query processing in the object-oriented models, which capture complicated semantics, do not significantly differ from the techniques used for (nested) relational query processing [Cattell 1997; Kifer et al. 1992].
Implementation of the P-Join
Algebraic optimisation is both theoretically and practically important for query processing in (nested) relational databases. This chapter considers this issue and investigates some algebraic properties concerning nested relational operators.

The nested relational model plays the role of an intermediate stage in an evolutionary path from the relational model to object-oriented data models and query languages. As noted in [Scholl and Schek 1990], we can define a nested relation for each class. Each object of a class can be represented by a tuple in the relation for that class. By transforming object queries into an object algebra in the spirit of nested relational algebra, the nested relational optimisation techniques can be applied to query processing in object-oriented systems [Korth 1988; Scholl and Schek 1990].

For the most part, techniques developed for the relational model can be directly applied to the nested relational model. However, some differences exist and caution needs to be taken when we optimise queries in the nested relational model [Jan 1990; Liu and Ramamohanarao 1994a]. Some algebraic equivalences of the nested relational operators have been shown in [Liu and Ramamohanarao 1992].

As mentioned in Chapter 1, many natural queries cannot be expressed by other join operators without restructuring operations. It is difficult to optimise an algebraic expression of a query which includes restructuring operators [Liu and Ramamohanarao 1992; Liu and Ramamohanarao 1994a]. The powerset algebra proposed by Gyssens and Gucht [1988] is not helpful in solving the problem of optimising join queries on two nested relations which have different nesting depths for joining attributes. The P-join
operator does not require as many restructuring operators and combines the advantages of the extended natural join and the recursive join for data access. For this reason, this Chapter undertakes a more comprehensive examination of algebraic properties concerning the P-join.

The P-join operator has properties that the standard join operator possesses, such as the commutative and associative law, and commuting a projection with P-join and distributivity of unnest over P-join. This chapter gives new algebraic equivalences of the P-join operator and extended relational operators [Roth et al. 1988] which can be used for query optimisation in nested relational databases.

Finally, we outline an algorithm that transforms an initial query tree into an optimised tree that is more efficient to execute. This algorithm forms the basis for evaluating queries in nested relational databases.

7.1 Extended Relational Operators

This section briefly reviews some nested relational operators proposed in [Roth et al. 1988] which are used throughout this chapter. These nested relational operators include extended traditional set operators, extended natural join and extended projection operators.

**Definition 7.1** Let $r_1$ and $r_2$ be two nested relations with the same schema $R$. Let $A$ range over the zero-order names in $\text{att}_R$ and $X$ range over the higher-order names in $\text{att}_R$. The extended union $\cup^e$ is given by

$$r_1 \cup^e r_2 \overset{\text{def}}{=} \{ t \mid (\exists t_1 \in r_1, \exists t_2 \in r_2 : \left( \forall A, X \in \text{att}_R : t[A] = t_1[A] = t_2[A] \land t[X] = (t_1[X] \cup^e t_2[X]) \right)$$

$$\lor (t \in r_1 \land (\forall t' \in r_2 : (\forall A \in \text{att}_R : t[A] \neq t'[A])))$$

$$\lor (t \in r_2 \land (\forall t' \in r_1 : (\forall A \in \text{att}_R : t[A] \neq t'[A]))).$$

**Definition 7.2** Let $r_1$ and $r_2$ be two nested relations with the same schema $R$. Let $A$ range over the zero-order names in $\text{att}_R$ and $X$ range over the higher-order names in $\text{att}_R$. The extended intersection $\cap^e$ is given by
### §7.1 Extended Relational Operators

**Figure 7.1**: Examples of $\cup^e$ and $\cap^e$

\[
\begin{align*}
\text{(a) } r_1 & \quad \text{(c) } r_1 \cup^e r_2 \\
\begin{array}{cccc}
A & X & B & Y \\
& & C & D \\
an_1 & b_1 & c_1 & d_1 \\
& & c_1 & d_2 \\
b_2 & c_1 & d_1 & c_2 & d_2 \\
& c_3 & d_1 & & \\
an_2 & b_3 & c_2 & d_2 \\
& b_4 & c_1 & d_2 \\
\end{array} \\
\begin{array}{cccc}
A & X & B & Y \\
& & C & D \\
an_1 & b_1 & c_1 & d_1 \\
& & c_1 & d_2 \\
b_2 & c_1 & d_1 & c_2 & d_2 \\
& c_3 & d_1 & & \\
an_2 & b_3 & c_2 & d_2 \\
& b_4 & c_1 & d_2 \\
\end{array}
\end{align*}
\]

**Definition 7.3** Let $r_1$ and $r_2$ be two nested relations with schemas $R_1$ and $R_2$. Let $X$ be the higher-order attributes in $\text{att}_{R_1} \cap \text{att}_{R_2}$, $M = \text{att}_{R_1} - X$, and $N = \text{att}_{R_2} - X$. Then the extended natural join is a relation $r$ with schema $R$, where

- $R = (M, X, N)$ is the resulting schema, and

\[
r_1 \cap^e r_2 \overset{\text{def}}{=} \{ t \mid \exists t_1 \in r_1 \land \exists t_2 \in r_2 : \\
( \forall A, X \in \text{att}_R : t[A] = t_1[A] = t_2[A], \\
\land t[X] = (t_1[X] \cap^e t_2[X]), t[X] \neq \emptyset) \}\n\]
### Algebraic Optimisation for Nested Relational Databases

#### Definition 7.4

Let \( r \) be a relation over schema \( R \), and \( L \) be a project-list of \( R \). The extended projection operator is defined as:

\[
\pi_L^e(r) \overset{\text{def}}{=} \bigcup_{t \in \pi_L(r)} (t)
\]

If \( L = R \), we simply shorten \( \pi_L^e(r) \) to \( \pi^e(r) \). Figure 7.2(d) shows an example of extended projection.

#### 7.2 Equivalences of Algebraic Expressions

The essence of query optimisation is to find an execution plan that minimises a cost function. The optimisation process involves two deeply connected levels that are classified as heuristic optimisation and systematic cost estimation. The first level is based on heuristic rules for ordering the operations in a query execution strategy to find an
equivalent expression with improved performance expected. The second level uses a cost model based on system information to choose the execution plan with the lowest cost estimate. This section investigates heuristic rules for transforming algebraic expressions into equivalent ones.

A series of algebraic equivalences are presented in this section. First the properties of commutativity and associativity regarding the extended natural join and the property of commuting a projection with extended natural join are presented in Theorem 7.1 and Theorem 7.2 respectively. Then the properties of commuting projection (π or π^e) with the P-join are presented in Theorem 7.3 and Theorem 7.4. We also consider commuting a selection with the P-join in Theorem 7.5. Finally, we present the commutativity of projection with the unnest and the selection operators in Theorem 7.6 and Theorem 7.7 respectively.

Roth, Korth, and Silberschatz [1988] have shown the property of distributivity of unnest over extended natural join (extended intersection), i.e.,

\[ \mu_X(r \Join^e s) = \mu_X(r) \Join^e \mu_X(s) \]
\[ \mu_X(r \cap^e s) = \mu_X(r) \cap^e \mu_X(s) \]

The following theorem shows that the laws of commutativity and associativity hold for the extended natural join.

**Theorem 7.1** (1) \( r \Join^e s = s \Join^e r \) (2) \( r_1 \Join^e (r_2 \Join^e r_3) = (r_1 \Join^e r_2) \Join^e r_3 \)

**Proof:** (1) Under the extended natural join two tuples contribute to the join if the extended intersection of their projection over common attributes is not empty. The commutative property of extended natural join is implied by the commutative property of the extended intersection operator.

(2) We show inclusion both ways.

\( \subseteq \): \( \forall t \in LHS \), there exist a tuple \( t_1 \in r_1 \) and a tuple \( u \in r_2 \Join^e r_3 \) such that \( t \) is the extended natural join of \( t_1 \) and \( u \). That is, \( t = t_1 \Join^e u \). Similarly, there must be tuples \( t_2 \) and \( t_3 \) in \( r_2 \) and \( r_3 \) respectively such that \( u = t_2 \Join^e t_3 \). So, \( t = t_1 \Join^e u = t_1 \Join^e (t_2 \Join^e t_3) \).

Now we divide the following proof into two parts.
(a) Let \( A^* \) be the set of zero-order attributes in the schema of the relation of LHS.

\[
\begin{align*}
t[A^*] & = (t_1 \Join t_2 \Join t_3)[A^*] \\
& = t_1[E_{R_1} \cap A^*] \Join (t_2 \Join t_3)[E_{R_2 \cap R_3} \cap A^*] \\
& = t_1[E_{R_1} \cap A^*] \Join (t_2[E_{R_2} \cap A^*] \Join t_3[E_{R_3} \cap A^*]) \\
& = t_1[E_{R_1} \cap A^*] \Join (t_2[E_{R_2} \cap A^*] \Join t_3[E_{R_3} \cap A^*]) \\
& = (t_1[E_{R_1} \cap A^*] \Join t_2[E_{R_2} \cap A^*]) \Join t_3[E_{R_3} \cap A^*] \\
& = (t_1 \Join t_2)[E_{R_1 \cap R_2} \cap A^*] \Join t_3[E_{R_3} \cap A^*] \\
& = ((t_1 \Join t_2) \Join t_3)[E_{(R_1 \cap R_2) \cap R_3} \cap A^*] \\
& = ((t_1 \Join t_2) \Join t_3)[A^*]
\end{align*}
\]

(b) Let \( X^* \) be the set of higher-order attributes in the schema of the relation of LHS. There are three cases:

* \( X \in X^* \) and \( X \) precisely belongs to one relation schema, say \( r_i \), then \( r_i[X] \) do not participate in the join with the other two relations. We get \( t[X] = (t_1 \Join t_2 \Join t_3)[X] = t_i[X] = ((t_1 \Join t_2) \Join t_3)[X] \)

* \( X \in X^* \) and \( X \) is a common attribute of two relations, say \( r_i, r_j \), then

\[
\begin{align*}
t[X] & = (t_1 \Join t_2 \Join t_3)[X] \\
& = t_1[X] \Join t_2[X] \Join t_3[X] \\
& = ((t_1 \Join t_2) \Join t_3)[X]
\end{align*}
\]

* \( X \in X^* \) and \( X \) is a common attribute of three relations, then

\[
\begin{align*}
t[X] & = t_1[X] \Join t_2[X] \Join t_3[X] \\
& = t_1[X] \Join (t_2[X] \Join t_3[X]) \quad \text{(by \( \Join \) definition)} \\
& = (t_1[X] \Join t_2[X]) \Join t_3[X] \quad \text{(by associativity of \( \Join \))} \\
& = (t_1[X] \Join t_2[X]) \Join t_3[X] \\
& = ((t_1 \Join t_2) \Join t_3)[X]
\end{align*}
\]

So \( t[X^*] = (t_1 \Join t_2 \Join t_3)[X^*] = ((t_1 \Join t_2) \Join t_3)[X^*] \)
By a and b we imply that
\[ t = t_1 \bowtie^e (t_2 \bowtie^e t_3) = (t_1 \bowtie^e t_2) \bowtie^e t_3 \in (r_1 \bowtie^e r_2) \bowtie^e r_3 = RHS. \]
Since \( t \) is arbitrary element, we conclude \( LHS \subseteq RHS \).

\( \supseteq \): This proof is similar to that of part "\( \subseteq \)". □

Due to the fact that in the P-join operation the extended natural join \((\bowtie^e)\) is applied to each level of the schema, the laws of commutativity and associativity can be similarly valid for the P-join. The detailed description of these results has been presented in Chapter 5.

The following theorem concerns commuting a projection \((\pi)\) with extended natural join operators \((\bowtie^e)\). This equivalence is direct extension of that of the flat relational model.

**Theorem 7.2** Let \( r \) and \( s \) be nested relations with schemas \( R \) and \( S \). Consider \((M \subseteq \text{attr}_R; N \subseteq \text{attr}_S)\). Then \( \pi_{MN}(r \bowtie^e s) = \pi_M(r) \bowtie^e \pi_N(s) \leftrightarrow (M \cap N = \text{attr}_R \cap \text{attr}_S) \).

**Proof:**

\((\Leftarrow\Rightarrow)\) We show inclusion both ways under given conditions.

\(\supseteq\): \( \forall t \in LHS \), there is a tuple \( t' \in r \bowtie^e s \) such that \( t = \pi_{MN}(t') \). Since \( t' \in r \bowtie^e s \), there must be tuples \( t_r \in r; t_s \in s \) such that \( t' = t_r \bowtie^e t_s \).

Because \( M \subseteq \text{attr}_R, N \subseteq \text{attr}_S \) and \( M \cap N = \text{attr}_R \cap \text{attr}_S \), so \( t = t'[MN] = (t_r \bowtie^e t_s)[MN] = t_r[M] \bowtie^e t_s[N] \in \pi_M(r) \bowtie^e \pi_N(s) \). Hence \( t \in RHS \).

\(\subseteq\): \( \forall t \in RHS \), there exists a tuple \( \hat{t}_r \in \pi_M(s) \) and a tuple \( \hat{t}_s \in \pi_N(s) \) such that \( t = \hat{t}_r \bowtie^e \hat{t}_s \). Since \( \hat{t}_r \in \pi_M(r) \), there is a \( t_r \in r \) such that \( \hat{t}_r = t_r[M] \).

Similarly, there is a \( t_s \in s \) such that \( \hat{t}_s = t_s[N] \). By assumption \( M \cap N = E_R \cap E_S, t = \hat{t}_r \bowtie^e \hat{t}_s = t_r[M] \bowtie^e t_s[N] = (t_r \bowtie^e t_s)[MN] \in \pi_{MN}(r \bowtie^e s) \).

Hence \( t \in LHS \).

\((\Rightarrow\Leftarrow)\) Suppose \( M \cap N \neq \text{attr}_R \cap \text{attr}_S \). There exists attributes \( Q \in \text{attr}_R \cap \text{attr}_S \) such that \( Q \notin M \cap N \). We prove that \( t \in \pi_M(r) \bowtie^e \pi_N(s) \Rightarrow t \in \pi_{MN}(r \bowtie^e s) \) does not always hold. Since \( t \in \pi_M(r) \bowtie^e \pi_N(s) \) there must be \( \hat{t}_r \in \pi_M(r) \)
and $i_s \in \pi_N(s)$ such that $t = \hat{i}_r \bowtie_i \hat{i}_s$. Let $t^*_r = \{t' \mid t' \in r, t[M] = \hat{i}_r\}$ and $t^*_s = \{t'' \mid t'' \in s, t''[N] = \hat{i}_s\}$. We assume $t^*_r [Q] \cap t^*_s [Q] = \emptyset$. Because $Q \in \text{attr}_R \cap \text{attr}_S$, the relation $r \bowtie e s$ contains no tuple having $t$ as its $MN$ component. That is, $t \notin \pi_{MN}(r \bowtie e s)$. So $\pi_M(r) \bowtie e \pi_N(s) \neq \pi_{MN}(r \bowtie e s)$.

We conclude that if $(M \cap N \neq \text{attr}_R \cap \text{attr}_S)$ then $(\pi_{MN}(r \bowtie e s) = \pi_M(r) \bowtie e \pi_N(s))$ does not hold. $\square$

Now we consider commuting an extended projection ($\pi^e$) with extended natural join.

First, projecting the whole relation in the extended projection operator is examined.

**Lemma 7.1** $\pi^e(r_1 \bowtie e r_2) = \pi^e(r_1) \bowtie e \pi^e(r_2)$.

**Proof:** Let $\bar{X}$ be the higher-order attributes in $\text{attr}_{R_1} \cap \text{attr}_{R_2}$. Let $M = \text{attr}_{R_1} - \bar{X}$ and $N = \text{attr}_{R_2} - \bar{X}$. Then we show inclusion both ways to prove equivalence at the instance level.

We partition $r_1 \bowtie e r_2$ on zero-order attributes. By the definition of $\pi^e$, $\forall t \in LHS$, $t$ is the extended union of those tuples in some block $B$ of this partition. That is, $\exists$ integer $k$, such that $t = \bigcup \{t_i \mid t_i \in r_1 \bowtie e r_2, 1 \leq i \leq k, t_i \in B\}$. There exists $s_1 \in \pi^e(r_1)$ and $s_2 \in \pi^e(r_2)$ such that $s_1[M] = t[M], s_2[N] = t[N]$.

Now we prove $t = s_1 \bowtie e s_2$ as follows.

1. For each $t_i \in r_1 \bowtie e r_2$, there must exist $t^1_i \in r_1$ and $t^2_i \in r_2$ such that $t_i = t^1_i \bowtie e t^2_i, 1 \leq i \leq k$, and $t[M] = t_i[M] = t^1_i[M], t[N] = t_i[N] = t^2_i[N]$.

2. $\forall X \in \bar{X}$. $t[X] = \bigcup \pi^e t_i[X] = \bigcup \pi^e(t^1_i[X] \cap e t^2_i[X]) = (\bigcup \pi^e t^1_i[X]) \cap e (\bigcup \pi^e t^2_i[X]) = (\bigcup \pi^e t^1_i[X]) \cap e (\bigcup \pi^e t^2_i[X]) = s_1[X] \cap e s_2[X]$.

3. We claim that $t[X] = s_1[X] \cap e s_2[X]$. If it is not then there is some $w \in s_1[X] \cap e s_2[X]$ and $w \notin t[X]$. There must exist $w_1 \in r_1$ and $w_2 \in r_2$ such that $w \in (w_1 \bowtie e w_2)[X]$, where $w_1[M] = s_1[M] = t[M]; w_2[N] = s_2[N] = t[N]$. So $(w_1 \bowtie e w_2)[X] \notin t[X] = \bigcup t_i[X]$, that is, $w_1 \bowtie e w_2 \notin \bigcup t_i$. But $w_1 \bowtie e w_2$ have the same values on zero-order attributes with $t$. This implies $\bigcup t_i$ is not a partition on zero-order attributes of $r_1 \bowtie e r_2$. This is a contradiction. So $t[X] = s_1[X] \cap e s_2[X]$. We conclude that $t = s_1 \bowtie e s_2 \in (\pi^e(r_1) \bowtie e \pi^e(r_2))$. 


∀t ∈ RHS, there exists t’ ∈ π\(e\)(r\(_1\)) and t” ∈ π\(e\)(r\(_2\)) such that t = t’ ⊙ s”.

By the definition of π\(e\) t’ = ∪\(_{1\leq i \leq k}\) s\(_i\), where s\(_i\) ∈ r\(_1\) and all s\(_i\) have values on zero-order attributes. Similarly, t” = ∪\(_{1\leq j \leq l}\) s\(_j\), where s\(_j\) ∈ r\(_2\) and all s\(_j\) have values on zero-order attributes.

∀X ∈ \(\bar{X}\), t[X] = t’[X] ∩\(e\) t”[X]

\[
= (\cup_{1\leq i \leq k} s_i)[X] \cap\(e\) (\cup_{1\leq j \leq l} s_j)[X]
= (\cup_{1\leq i \leq k} s_i[X]) \cap\(e\) (\cup_{1\leq j \leq l} s_j[X])
= ∪\(_{1\leq i \leq k, 1\leq j \leq l}\) (s\(_i\)[X] \cap\(e\) s\(_j\)[X])
= ∪\(_{1\leq i \leq k, 1\leq j \leq l}\) (s\(_i\) \(\bowtie\) s\(_j\))[X]
\]

In r\(_1\) \(\bowtie\) r\(_2\), only those tuples which are joined by s\(_i\) and s\(_j\) can have the same values on zero-order attributes with t. So P = ∪\(_{1\leq i \leq k, 1\leq j \leq l}\) (s\(_i\) \(\bowtie\) s\(_j\)) forms a partition on zero-order attributes in r\(_1\) \(\bowtie\) r\(_2\). We get t ∈ LHS.

The following theorem shows that the property of commuting an extended projection with extended natural join holds.

**Lemma 7.2** \(\pi^e_{MN}(r_1 \bowtie e \ r_2) = \pi^e_M(r_1) \bowtie e \pi^e_N(r_2)\) iff \((M \cap N = att_R \cap att_S)\).

**Proof:** \(\pi^e_{MN}(r_1 \bowtie e \ r_2) = \cup_{t \in \pi_{MN}(r_1 \bowtie e \ r_2)}(t)

= \cup_{t \in (\pi_M(r_1) \bowtie e \pi_N(r_2))}(t) \quad \text{(by Theorem 7.2)}
\]

\[= \pi^e(\pi_M(r_1) \bowtie e \pi_N(r_2))
= \pi^e(\pi_M(r_1)) \bowtie e \pi_N(r_2) \quad \text{(by Lemma 7.1)}
\]

The next theorem investigates commuting projection with the P-join operator. If the project-list L can be split into L\(_1\) and L\(_2\) such that they contain attributes of r and s in L respectively, and they each contain all common attributes involved in the join, then we can get the following results regarding commuting a projection (\(\pi\) or \(\pi^e\)) with P-join.

**Lemma 7.3** \(\pi_L[r(N_r) \bowtie P \ s(N_s)] = \pi_L[\pi_{L_1}(r)(N_r) \bowtie P \ \pi_{L_2}(s)(N_s)]\), where \(N_r, \ N_s\) are the path-determining nodes of r and s respectively.
Proof: At each level of schema, P-join is defined in terms of $\bowtie^e$. We obtain this result by applying Theorem 7.2 recursively to each level of schema of P-join. That is,

$$\pi_L[r(N_r) \bowtie^p s(N_s)] = \pi_L[\sigma_{\theta_B}(\pi_L'(\pi_X'(r(N_r) \bowtie^P s(N_s))))]$$

$$= \pi_L[\sigma_{\theta_B'}(\pi_X'(\pi_L'(r(N_r) \bowtie^P \pi_L(s)(N_s))))]$$

$$= \pi_L[\pi_{L'}(r(N_r) \bowtie^p \pi_{L'}(s)(N_s))]$$

Here $\theta_A$, $\theta_B$, $\pi_X$ are selection conditions and projection operator defined in the P-join, $\theta_{A'}$, $\pi_{X'}$, $\theta_{B'}$ are $\theta_A$, $\pi_X$, $\theta_B$ restricted to the schema of $\pi_{L'}(r(N_r) \bowtie^P \pi_{L'}(s)(N_s))$. $L'$ is the project-list $L$ augmented with attributes in $B$. □

The following theorem is the generalisation of Lemma 7.3 with single path replaced by multiple paths. We present it without proof because the proof is similar to that of Lemma 7.3.

Theorem 7.3 $\pi_L[r(L_r) \bowtie^p s(L_s)] = \pi_L[\pi_{L_1}(r(L_r)) \bowtie^p \pi_{L_2}(s(L_s))], \text{ where } L_r, L_s$ are the lists of join-paths of $r$ and $s$ respectively.

We now consider commuting extended projection with the P-join.

Lemma 7.4 $\pi^e_L(r) = \pi^e_L(r)$

Proof: By the definition of $\pi^e$, the equivalence holds obviously.

Theorem 7.4 $\pi^e_L(r(L_r) \bowtie^p s(L_s)) = \pi^e_L(r(L_r)) \bowtie^p \pi^e_L(s(L_s)), \text{ where } L_r, L_s$ are the lists of join-paths of $r$ and $s$ respectively.

Proof:

$LHS = \pi^e_L(r(L_r) \bowtie^p s(L_s))$

$$= \pi^e L(r(L_r) \bowtie^p s(L_s))$$

$$= \pi^e L(\pi_{L_1}(r)(L_r) \bowtie^p \pi_{L_2}(s)(L_s))$$

(by Theorem 7.3)

$$= \pi^e L_1(\pi_{L_1}(r)(L_r) \bowtie^p \pi_{L_2}(s)(L_s))$$

$$= \pi^e L_1(\pi_{L_1}(r)(L_r) \bowtie^p \pi_{L_2}(s)(L_s))$$

(by Theorem 7.3)

$$= \pi^e L_1(\pi_{L_1}(r)(L_r) \bowtie^p \pi_{L_2}(s)(L_s))$$

(by Lemma 7.4)

$$= \pi^e L_1(r)(L_r) \bowtie^p \pi^e_{L_2}(s)(L_s)$$

$$= RHS \ □$$
We now consider commuting a selection (σ) with the extended natural join and the P-join.

**Lemma 7.5** (1) \( \sigma_\theta(r \bowtie^e s) = \sigma_\theta(r) \bowtie^e s \), if all the attributes mentioned in \( \theta \) are attributes of \( r \). (2) \( \sigma_\theta(r \bowtie^e s) = \sigma_{\theta_1}(r) \bowtie^e \sigma_{\theta_2}(s) \), if \( \theta \) is of the form \( \theta_1 \wedge \theta_2 \), where \( \theta_1 \) involves only attributes of \( r \), and \( \theta_2 \) involves only attributes of \( s \).

**Proof**: (1) (a) If the attributes mentioned in \( \theta \) are atomic attributes:

it is straightforward to get \( \sigma_\theta(r \bowtie^e s) = \sigma_\theta(r) \bowtie^e s \).

(b) Consider the attributes mentioned in \( \theta \) to contain higher-order attributes.

Because \( \sigma_\theta(r_1 \cap^e r_2) = \sigma_\theta(r_1) \cap^e r_2 \) for any two relations with the same schema, we get

\( \sigma_\theta(r \bowtie^e s) = \sigma_\theta(r) \bowtie^e s \).

(2) \( \sigma_\theta(r \bowtie^e s) = \sigma_{\theta_1 \wedge \theta_2}(r \bowtie^e s) \)

\[ = \sigma_{\theta_2} \left[ \sigma_{\theta_1}(r \bowtie^e s) \right] \]

\[ = \sigma_{\theta_2}(\sigma_{\theta_1}(r) \bowtie^e s) \]

\[ = \sigma_{\theta_1}(r) \bowtie^e \sigma_{\theta_2}(s). \]

By the definition of the P-join, we know that \( \bowtie^e \) is performed at each level in the schema of the P-join. We can generalise the result of Lemma 7.5 to the case of P-join as follows.

**Lemma 7.6** (1) \( \sigma_\theta(r(N_r) \bowtie^p q(N_q)) = \sigma_\theta(r)(N_r) \bowtie^p q(N_q) \), if \( \theta \) contains no attributes in common with \( q \). (2) \( \sigma_\theta(r(N_r) \bowtie^p q(N_q)) = \sigma_{\theta_1}(r)(N_r) \bowtie^p \sigma_{\theta_2}(q)(N_q) \), if \( \theta \) is of the form \( \theta_1 \wedge \theta_2 \), where \( \theta_1 \) and \( \theta_2 \) involve only attributes which are selection-comparable nodes in \( r \) and \( q \) respectively.

**Proof**:

(1) \( LHS = \sigma_\theta[r(N_r) \bowtie^p q(N_q)] \)

\[ = \sigma_\theta[\pi_X(\sigma_{\theta_A}(r(N_r) \times q(N_q)))] \]

\[ = \sigma_{\theta_B}(\pi_X(\sigma_{\theta_A}(r(N_r) \times q(N_q)))) \]

\[ = \sigma_{\theta_A}(r(N_r) \bowtie^p q(N_q)) \]

where \( \theta_A, \theta_B, \pi_X \) are selection conditions and projection operator defined in the P-join.
(2) $\sigma_\theta(r(N_r) \bowtie^p q(N_q)) = \sigma_{\theta_1 \land \theta_2}(r(N_r) \bowtie^p q(N_q))$

$= \sigma_{\theta_2}[\sigma_1(r(N_r) \bowtie^p q(N_q))]$

$= \sigma_{\theta_2}[\sigma_1(r(N_r)) \bowtie^p q(N_q)]$

$= \sigma_{\theta_1}(r(N_r)) \bowtie^p \sigma_{\theta_2}(N_q) \Box$

**Theorem 7.5** (1) $\sigma_\theta(r(L_r) \bowtie^p q(L_q)) = \sigma_\theta(r)(L_r) \bowtie^p q(L_q)$, if $\theta$ contains no attributes in common with $q$. (2) $\sigma_\theta(r(L_r) \bowtie^p s(L_s)) = \sigma_{\theta_1}(r)(L_r) \bowtie^p \sigma_{\theta_2}(q)(L_q)$, if $\theta$ is of the form $\theta_1 \land \theta_2$, where $\theta_1$ and $\theta_2$ involve only attributes which are selection-comparable nodes in $r$ and $q$ respectively.

**Proof**: By the definition of the decomposition P-join,

$$r(L_r) \bowtie^p q(L_q) = \sigma_\theta[\land (J_1, ..., J_n)],$$

where $J_i = r[R_i](N_i^r) \bowtie^p q[Q_i](N_i^q)$, $1 \leq i \leq n$. By Lemma 7.6,

$$\sigma_\theta(J_i) = \sigma_\theta(r[R_i])(N_i^r) \bowtie^p q[Q_i](N_i^q), 1 \leq i \leq n.$$

So $\sigma_\theta(r(L_r) \bowtie^p q(L_q)) = \land (\sigma_\theta(J_1, ..., J_n))$

$= \land_{i=1}^n (\sigma_\theta(r[R_i])(N_i^r) \bowtie^p q[Q_i](N_i^q))$

$= \sigma_\theta(r)(L_r) \bowtie^p q(L_q). \Box$

The following theorem concerns the commutativity of projection (extended projection) with the unnest operator.

**Lemma 7.7** $\pi_{L \mu_X}(r) = \mu_X \pi_{L'}(r)$ if $X \subseteq L'$ and $L = \mu_X(L')$

**Proof**: See [Thomas and Fischer 1986].

**Theorem 7.6** $\pi_{L'}^e \mu_X(r) = \mu_X \pi_{L'}^e(r)$ if $X \subseteq L'$ and $L = \mu_X(L')$

**Proof**: $LHS = \pi^e(\pi_{L \mu_X}(r))$

$= \pi^e(\mu_X \pi_{L'}(r))$ (by Lemma 7.7)

$= \mu_X(\pi^e(\pi_{L'}(r)))$

$= \mu_X(\pi_{L'}^e(r)) \Box$
The following theorem illustrates the commutativity of projection (extended projection) with the selection operator.

**Theorem 7.7** If condition $\theta$ involves only attributes of $L$, then

1. $\pi_L(\sigma_\theta(r)) = \sigma_\theta(\pi_L(r))$,
2. $\pi^c_L(\sigma_\theta(r)) = \sigma_\theta(\pi^c_L(r))$.

**Proof:** (1) By the definition of $\pi$ and $\sigma$, it is straightforward to get this result.

(2) $\pi^c_L(\sigma_\theta(r)) = \bigcup_{t \in \pi_L(\sigma_\theta(r))} \{t\}$

   $\quad = \bigcup_{t \in \sigma_\theta(\pi_L(r))} \{t\}$

   $\quad = \sigma_\theta(\bigcup_{t \in \pi_L(r)} \{t\})$

   $\quad = \sigma_\theta(\pi^c_L(r))$  $\Box$

### 7.3 Outline of a Heuristic Optimisation Algorithm

This section discusses optimisation techniques that can use the equivalence rules of Section 7.2 to optimise nested relational algebraic expressions. We can now outline the steps of an algorithm that transforms an initial query tree into an optimised tree that is more efficient to execute. The main ideas behind this algorithm are similar to those discussed in 1NF relational databases, except that restructuring operators and path-dependent nested relational operators are the new operators considered in the algorithm. The steps of the algorithm are as follows:

- Analyse the parse tree according to the input algebraic expression which includes standard operations as well as nested relational operators.

- Using the rule of *cascade of selection*, separate each `SELECT` operation with conjunctive conditions into a cascade of `SELECT` operations.

- Using the rules concerning associativity of binary operations $\cup^e$, $\cap^e$, $\setminus$ and $\bowtie$, rearrange the leaf nodes of the tree.
Algebraic Optimisation for Nested Relational Databases

(a) Product

<table>
<thead>
<tr>
<th>prodname</th>
<th>Warranty</th>
<th>Composition</th>
<th>Distributor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>premium</td>
<td>country</td>
<td>w-period</td>
</tr>
<tr>
<td>prod-A</td>
<td>$120</td>
<td>U.S.A.</td>
<td>5 yrs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prod-B</td>
<td>$200</td>
<td>Aust.</td>
<td>3 yrs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Part

<table>
<thead>
<tr>
<th>p-name</th>
<th>weight</th>
<th>Source</th>
<th>company</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>200g</td>
<td>comp-A</td>
<td>$10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>comp-C</td>
<td>$12</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>350g</td>
<td>comp-B</td>
<td>$18</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3: Product database

- Using the rules concerning the cascading of PROJECT and commuting of PROJECT with other operations, move projection as far down the tree as possible.
- Apply the knowledge of functional dependency, multivalued dependency and mutual data dependency [Jan 1990; Liu and Ramamohanarao 1992] to move the restructuring operations UNNEST and NEST down the binary operations, but not down the unary operations (σ, π).
- Identify subtrees that represent groups of operations that can be executed by a single access routine.

Let us introduce a database example and illustrate the algebraic optimisation techniques that apply to the corresponding query.

Example 7.1 Consider the following database which has nested relations Product and Part, shown in Figure 7.3.

Product = (prodname, Warranty(premium, country, w-period), Composition(c-name, c-id,
§7.3 Outline of a Heuristic Optimisation Algorithm

Parts((p-name, quantity)), Distributor(company, fee))

\[ Part = (p-name, weight, Source(company, cost)) \]

Consider the following query.

Find those products whose warranty period is under three years, their parts, together with those companies that are both the distributor and parts source, and their corresponding delivery fees and costs. Group the result on prodname and p-name.

Note that we denote \( X + Y \) for the new attribute \( XY \). We could express this query as:

\[
\pi_{prodname, p-name, Distributor + Source(company, fee, cost)} (\sigma_{Warranty.w-period \leq 3} (Product(Distributor) \bowtie_p Part(Source)))
\]

The algebraic transformation of this query is as follows.

\[
\pi_{prodname, p-name, Distributor + Source(company, fee, cost)} (\sigma_{Warranty.w-period \leq 3} (Product(Distributor) \bowtie_p Part(Source)))
\]

\[
= \pi_{prodname, p-name, Distributor + Source(company, fee, cost)}
\]

\[
((\sigma_{Warranty.w-period \leq 3} Product)(Distributor) \bowtie_p Part(Source))
\]

\[
= \pi_{prodname, p-name, Distributor + Source(company, fee, cost)}
\]

\[
((\pi_{prodname, Composition(Parts(p-name))}(\sigma_{Warranty.w-period \leq 3} Product))(Distributor)
\]

\[
\bowtie_p [\pi_{p-name, Source(company, cost)} Part](Source))
\]

The resulting equivalent expression is more efficient than the original one as relations \( Product \) and \( Part \) have fewer tuples before applying P-join on them.
Chapter 8

Conclusion and Further Research

This thesis has presented a theory of database queries on the complex value data model and investigated the issues of query translation and optimisation for this data model. The issues involve what kinds of structural properties of relational calculus queries remain when we consider queries on the complex value model extended with external functions; how to implement the complex value calculus queries, and how to optimise queries that include join operation in complex value databases.

First, this thesis extended some of classical properties of the relation theory - particularly those related to query safety - to the context of complex value databases with fixed external functions. The binding function proposed in [Escobar-Molano et al. 1993] has been extended and the notions of “evaluable” and “allowed” have been generalised to incorporate external functions in complex value databases. Significantly, this thesis showed that all em-allowed complex value calculus (or fix-point) queries are external-function-domain independent and continuous. The problem of whether a broader subclass of embedded domain independent formulas can be recognised efficiently remains open. This thesis also showed the relationship between properties such as embedded domain independence, finiteness and em-allowedness in various calculus-based query languages.

Second, this thesis investigated the issue of how to implement calculus queries with the incorporation of functions. The translation of relational calculus queries that support both user-defined functions and complex values into the corresponding relational algebra queries is challenging, because the class of domain independent is known to be undecidable even for DBMSs that don’t support user-defined functions and complex values. An algorithm for translating embedded allowed queries into equivalent
algebraic expressions has been developed. The algorithm is still open to optimisation.

Third, the issue of query optimisation has been investigated in this thesis. Chapter 5 proposed, within a restricted set of nested schema trees, the P-join operator which does not require as many restructuring operators and combines the advantages of the extended natural join and recursive join for efficient data access. The correctness of P-join using the criteria of faithfulness and precision of generalisation have been proved. This generalised P-join operator has properties that the standard join operator possesses, such as commutativity and associativity. This work offers a solution to the problem of how to efficiently express and optimise queries which include join operations in the nested relational model.

Fourth, this thesis proposed a P-join algorithm which takes advantage of a decomposed storage model and various join techniques available in the standard relational model to reduce the cost of join operation in nested relational databases. The complexity of the P-join algorithm developed in Chapter 6 is not more than other join algorithms with expensive restructuring operators involved, which have been proposed for nested relational database systems. Nonetheless, the issue of how to apply the P-join to the object-oriented models with minor changes, the design and implementing parallel algorithms for the P-join and how to optimise recursive queries including P-join operation requires further research.

The main advantages of the P-join operator are summarised as follows.

- P-join is a generalisation of other join operators proposed for the 1NF model.

- Most queries in extended SQL can be written naturally in terms of P-join and therefore can be optimised more efficiently.

- Compared to other join operators, P-join is more powerful in terms of expression and optimisation.

- The performance of methods of computing join described in this thesis is more efficient than that of the current technology available in existing nested relational database systems.

Fifth, this thesis investigated some algebraic properties of nested relational operators
which are useful for query optimisation in the nested relational model. Many instances of object-oriented queries are structurally similar to nested relational queries. Most essential techniques for nested relational query processing are directly applicable to object-oriented query processing. The semantics of the explicit join of classes in object-oriented data model is similar to that of P-join in the nested relational data model. Therefore, we hope that the theoretical results obtained for optimisation of nested relational algebra can be carried over to an object algebra.

This thesis examined the issues of safe queries, query translation and query optimisation in the complex value model. However, the issues of query translation with optimisation [Nakano 1990] and combining several algebra operators during their execution are worthwhile investigating in the future. There is also the need to address the complexity issue of query evaluation for different query classes.
Abiteboul, S. and Beeri, C. 1995. The power of languages for the manipulation of complex values. The Very Large Data Bases Journal 4, 727–794. (pp. 1, 6, 8, 9, 19, 31, 41, 44, 47, 48)


COLBY, L. 1990. A recursive algebra and query optimization for nested relations. *Information Systems* 15, 5, 567–582. (pp.10, 12, 69, 71, 73, 82)


Kuper, G. 1990. Logic programming with sets. *Journal of Computer and System Sciences* 41, 1, 44–64. (pp.1, 24)


LIU, H.-C. AND RAMAMOHANARAO, K. 1994c. Path-dependent join for nested relations. In *Proceedings of 5th International Hong Kong Computer Society Database Workshop* (Hong Kong, 1994), pp. 111–123. (pp. iii, 13)


OZSOYOGLU, G. AND HAFEZ, A. 1993. Near-optimum storage models for nested relations based on workload information. *IEEE Transactions on Knowledge and Data Engineering* 5, 6, 1018–1038. (p. 99)

PISTOR, P. AND ANDERSEN, F. 1986. Designing a generalized NF² model with an


STONEBRAKER, M. AND BROWN, P. 1998. *Object-Relational DBMSs - tracking the next great wave* (Second ed.). Morgan Kaufmann Publisher, Inc.  (p.5)


