Algebraic Optimization of Outerjoin Queries

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:25235126">http://nrs.harvard.edu/urn-3:HUL.InstRepos:25235126</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Algebraic Optimization of Outerjoin Queries

A thesis presented
by
César Alejandro Galindo-Legaria
to
The Division of Applied Sciences
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of
Computer Science.

Harvard University
Cambridge, Massachusetts
June, 1992
©1992 by César A. Galindo-Legaria.
All rights reserved.
Abstract

An advantage of relational database languages is that they allow “declarative” query specification: users pose queries as a set of conditions or properties on data to be retrieved, rather than by giving a procedure to obtain such data. The database system is then responsible for generating an efficient execution plan, depending on how information is physically stored. In this context, generation of efficient plans is known as database query optimization. Careful query analysis is justified during optimization, as it often brings considerable savings over the time that straightforward evaluation would take. These savings are sometimes of several orders of magnitude.

Query optimization has been an active area of research since the introduction of the first relational databases, and important heuristics have been developed for specific classes of queries. In particular, when relational join is the operator used to combine tables, the problem is well understood and it has been solved with considerable success.

Although joins cover a large number of common queries, database languages provide facilities that cannot be evaluated using joins only, and an increasing number of applications rely on these facilities. The approach taken by current query processors is to divide the query into various blocks that are optimized independently, but by discarding a global analysis of the query, many optimization opportunities are likely to be be missed.

The purpose of this thesis is to extend join optimization techniques to queries that contain both joins and outerjoins. The benefits of query optimization are thus extended to a number of important database applications, such as federated databases, nested queries, and hierarchical views, for which outerjoin is a key component.

Query optimization techniques are based on algebraic properties of relational operators, which are significantly harder to identify, prove, and apply for outerjoins than for joins. For this reason, previous work on outerjoin optimization has produced only partial results. This thesis presents new and more general supporting structures to model and understand the processing of outerjoin queries.

Our analysis of join/outerjoin queries is done in two parts. First, we investigate the interaction of outerjoin with other relational operators, to find simplification rules and associativity identities. Our approach is comprehensive and includes, as special cases, some outerjoin optimization heuristics that have appeared in the literature. Second, we abstract
the notion of feasible evaluation order for binary, join-like operators, considering associativity rules but not specific operator semantics. Combining these two parts, we show that a join/outerjoin query can be evaluated by combining relations in any given order—just as it is done on join queries, except that now we need to synthesize an operator to use at each step, rather than using always join. The purpose of changing the order of processing of relations is to reduce the size of intermediate results.

Our theoretical results are converted into algorithms compatible with the architecture of conventional database optimizers. For optimizers that repeatedly transform feasible strategies, the outerjoin identities we have identified can be applied directly. Those identities are sufficient to obtain all possible orders of processing. For optimizers that generate join programs bottom-up, we give a rule to determine the operators to use at each step.
Acknowledgements

I am indebted to the members of my research committee, Prof. Harry Lewis, Dr. Arnon Rosenthal, and Prof. Stuart Shieber, for all the help and support they have given me during the past few years. I feel fortunate to have had such able and concerned individuals supervise my work. Dr. Rosenthal suggested the problem that would become my thesis topic, and he patiently guided me in the development of my work. Prof. Lewis gave me apt and timely advise on all academic matters, and his example has shown me how to be to be a good professor and scholar. Prof. Shieber shared with me his sharp insight in many subjects, starting with an excellent course he taught two years ago. I want to acknowledge the extra effort they all made to read and comment this thesis in the middle of busy schedules.

I want to thank Dr. Alex Buchmann for all the help he has given me since I came to Boston. His advise shaped some of the most crucial career decisions I have taken in the past few years. I also thank Dr. Meichun Hsu, who believed in my potential and took me as her student when I first came to Harvard. I gratefully acknowledge the help of my friend Colin Gounden, who proofread the final version of this thesis.

Harvard provided me with a rich and stimulating environment for both my academic and personal growth. My affiliation with Lowell House was an important part of my experience here, and I thank Prof. William Bossert for giving me the opportunity of joining the House. I also learned a lot by working with Dr. Lewis Tyler, executive director of the Latin American Scholarship Program of American Universities (LASPAU).

My professors in Mexico helped me and encouraged me constantly, in particular Prof. Enrique Kortright from ITESM-CQ, and Profs. Renato Barrera and Manuel Guzman from CINVESTAV-IPN. During part of my graduate studies I had a CONACyT scholarship from the mexican government, number 52224.

Finally, my appreciation and sense of gratitude for my family are beyond words. My wife to be, Zandra, has been more patient and caring that I deserve. My parents, Elvira and Salvador, and my siblings, Karina and René, have been an unfealing source of love and have supported me in everything I have undertaken. My father was particularly close to me when I wrote this thesis, which is affectionately dedicated to him.
## Contents

1 Introduction  
1.1 The need for database query optimization .......................... 1  
1.2 Limitations of current query optimizers ............................ 3  
1.3 Outerjoin queries .................................................... 4  
1.4 Previous work on outerjoin processing ............................. 5  
1.5 Thesis overview and summary of contributions .................... 7  

2 Query processing in the relational model ............................ 9  
2.1 Relational algebra .................................................... 9  
2.1.1 Basic definitions .................................................. 9  
2.1.2 Relational operators .............................................. 13  
2.1.3 Join-like operators ................................................ 14  
2.1.4 Natural join and outerjoin ....................................... 18  
2.1.5 Other formulations of relational algebra ......................... 19  
2.2 Query optimization principles ...................................... 20  
2.2.1 Operators and their implementation ............................. 21  
2.2.2 Algebraic optimization ............................................ 22  
2.3 Select/project/join queries ......................................... 23  

3 Applications of outerjoin .............................................. 29  
3.1 Outerjoin in commercial query languages .......................... 29  
3.2 Hierarchical information ............................................. 30  
3.2.1 Hierarchical views ................................................. 30  
3.2.2 A query language for set- and entity-valued attributes ........ 33
3.3 Database merging .................................................. 35
3.4 Nested queries .................................................. 37
  3.4.1 Nested SQL blocks ........................................... 37
  3.4.2 Universal quantifiers ....................................... 38

4 Algebraic properties of relational operators .......................... 40
  4.1 Basics .......................................................... 40
    4.1.1 Ordering the domain of relations ....................... 41
    4.1.2 Properties of the basic relational operators .......... 42
  4.2 Preserving join expressions .................................. 47
  4.3 Associativity identities of join-like operators ............... 51
    4.3.1 Generalized outerjoin .................................. 53

5 Elements of a theory of reordering .................................. 56
  5.1 Trees, graphs, and query reordering .......................... 57
  5.2 A complete set of tree rewrite rules ......................... 60
  5.3 Conditions for reorderable expressions ...................... 66

6 Outerjoin simplification, reordering, and implementation .......... 67
  6.1 Freely-reorderable outerjoins ................................ 68
  6.2 Outerjoin simplification .................................... 71
    6.2.1 Strong predicate simplification ....................... 72
    6.2.2 Properties of simple trees ............................ 73
  6.3 Reordering of join/1-outerjoin/2-outerjoin .................. 74
  6.4 Modifying a relational optimizer to handle outerjoins ...... 83
    6.4.1 Enumeration of join-orderings .......................... 84
    6.4.2 Choosing the appropriate operators ................... 85
  6.5 Implementation algorithms .................................. 86

7 Concluding remarks ................................................. 91

A Proofs of algebraic identities ..................................... 93
  A.1 Reduction identities for joins and outerjoins ............... 93
A.2 Identities for preserved relations ......................... 96
A.3 Associativity of join-like operators ......................... 102
List of Figures

2.1 Table representation of relations. ........................................ 13
2.2 Basic operators of relational algebra. ................................. 15
2.3 Join-like operators. ....................................................... 17
2.4 Construction of natural full outerjoin. ............................... 20
2.5 Algebraic optimization steps. ............................................ 24
2.6 Query graph and relational expression for a select/project/join query. . . . 27

3.1 Hierarchical views of data. ............................................. 32
3.2 A relation in nested form. .............................................. 33
3.3 Database merging using full outerjoin. .............................. 36
3.4 Solving queries with universal quantifiers. ........................ 39

4.1 Preserve DAGs. ............................................................ 48
4.2 Application of PresR() on relations defining a preserve DAG. ....... 50
4.3 Preserve DAG for associativity conditions. .......................... 50
4.4 Generalized outerjoin. .................................................... 54

5.1 An operator tree and its query graph. ................................ 58
5.2 Association trees of a query graph. .................................. 59
5.3 Reordering with tree-rewrite rules. ................................... 61
5.4 Nodes v and w in the proof of lemma 5.1. ........................... 63
5.5 Moving v one level closer to the root in the proof of lemma 5.1. ....... 64

6.1 An annotated query graph. ............................................. 69
6.2 Topology of the annotated graph of a freely-reorderable query. ....... 70
6.3  Simplification of join/outerjoin expression. .................................. 74
6.4  Selection of the root operator — join/1-outerjoin reordering. .......... 77
6.5  Reordering with GOJ. ................................................................. 79
6.6  Selection of the root operator — join/1-/2-outerjoin reordering. ....... 80
6.7  Operator selection — join/1-outerjoin. ......................................... 85
6.8  Query graph and auxiliary values. ............................................... 86
6.9  Operator tree for an arbitrary processing order. ........................... 87
6.10 Operator selection for simple queries. ......................................... 88
Chapter 1

Introduction

1.1 The need for database query optimization

Relational database systems include some form of “declarative language” to manipulate information [Ullm82, Kort86]. To query information in a database, users specify conditions data must satisfy, rather than give an algorithm to retrieve such data. Besides ease of use, the advantage of declarative queries in database is that they give administrators some freedom to reorganize the physical data structures on which data is stored. It is then the responsibility of the system to find an efficient procedure or access plan to retrieve queried information.

In general, a large number of access plans exist for a given query, with ample differences in the amount of computer resources required. The following example illustrates two out of the many possible execution plans for a query. We will use the SQL2 ANSI standard [ANSI92].

Example 1.1 Queries and access plans.

For a database with tables containing information about customers and orders, the following SQL query finds all purchase orders on item “axz255,” and the customers who placed them.

Select All
From CUSTOMERS, ORDERS
Where \( \text{ORDERS.item#} = \text{"axz255"} \) and \\
\( \text{CUSTOMERS.cust#} = \text{ORDERS.cust#} \)

A general procedure to evaluate this query is to perform two nested loops, with loop variables taking values from each of the tables involved. At the inner loop, the procedure outputs the values of these variables whenever they satisfy the Where conditions. If we have 1,000 customers and 5,000 purchase orders, the condition will be evaluated \( 5 \times 10^6 \) times.

It is likely, however, that we have few orders for any one item. Assuming that we sell 2,000 different items and they are evenly distributed in orders, we estimate that there are 3 orders for item “axy255.”

A more efficient access plan is to retrieve first the purchase orders for item “ax255,” and then find the corresponding customers based on customer#. If the file system keeps an index or hash table on the ORDERS.item# and CUSTOMER.cust#, then the query is answered by six —expected— fast lookups on files.

Several issues are illustrated by the above example. To determine an efficient access plan, we need to know how data is stored and what are the indexes or hash tables available. We also need some statistical information on the content of data tables, including their size and expected distribution of values. Finally, we need to consider different orders in which to apply relational operators. For our previous example, the query basically consists of two operators: a relational selection to specify the orders we want, and a relational join to combine the two tables. The first access plan computes the join first —in a rather inefficient way— and then selects the item we want. The second access plan performs the selection first —reducing one of the tables significantly— and then computes the join.

Clearly, the selection of a good access plan is critical for performance. The automatic generation of efficient access plans for queries is the subject of database query optimization. The module that performs this task in a system is called the query optimizer. The problem is hard, and the term “optimization” is actually an exaggeration —exactly optimal plans can rarely be determined. Nonetheless, active research on the area for the past two decades
has produced a set of effective tools and techniques to process efficiently a large class of
common queries. These techniques have made practical the use of declarative, user-friendly
interfaces in databases.

Exploiting the physical organization of data is an important part of query optimization,
but dramatic performance improvements are the result of a change in the order in which
operations are applied, as illustrated in our example with selection and join. Improving a
query by changing the order of operator application is known as algebraic optimization, be-
because it depends on associativity or commutativity properties of those operators. Algebraic
optimization is much more powerful — and complex — than efficient use of auxiliary lookup
structures. The size of intermediate results, for instance, may vary greatly among the al-
ternative evaluation orders. Algebraic properties may even point to superfluous operations,
or files that need not be accessed at all.

1.2 Limitations of current query optimizers

In relational databases, current optimizers focus on the class of select/project/join queries,
which will be defined formally in chapter 2. For now, consider this class to be that which
follows the form of the SQL query of example 1.1. They are formulated as the join of several		
tables, some of which may be reduced by selections. A relational projection applied at the
de end can discard some table columns.

The algebraic properties of these queries are well-understood, and they are the basis of
effective optimization strategies for this class of queries [Smit75]. These strategies are an
integral part of query optimizers of most relational database systems [Astr76, Ullm82].

Queries outside this class are usually decomposed into “optimization blocks,” which are
treated separately. Optimization is rarely attempted between these blocks, thus discarding
a powerful performance-improvement tool.

Although an important proportion of queries in traditional database applications are
actually select/project/join, query languages do provide constructs that go beyond this well-
understood class, and recent applications rely increasingly on these constructs. In order to
extend the success of relational database systems to these new applications, it is necessary
to develop algebraic optimization techniques for wider classes of queries. Our focus in this
work is the class of queries containing joins and *outerjoins*, which are introduced next.

### 1.3 Outerjoin queries

Relational join combines the information from two tables by creating pairs of matching tuples. If a tuple in one relation has no corresponding tuple in the other, data in such will not appear in the join — e.g. in the join of CLIENTS and ORDERS, only clients who have placed purchase orders will be part of the join. In general, join loses part of the information contained in its arguments.

Outerjoin is a modification of join that preserves all information from one or both of its arguments. For instance, the outerjoin of tables $R_1$, $R_2$ preserving $R_1$ contains the join of those tables, plus the unmatched tuples of $R_1$. Since they have no corresponding $R_2$ tuples, unmatched tuples from $R_1$ are padded with null values on $R_2$ columns [Date83a, Date86]. The following example illustrates the use of outerjoin.

**Example 1.2** *Outerjoin preserves unmatched tuples.*

Using the same database of example 1.1, we now want to find all customers in Boston, and their current purchase orders. Since we want even those Boston customers with no purchase orders, a join is not appropriate. We use an outerjoin. The keyword *Left* below specifies which relation needs to be preserved.

```
Select All
From CUSTOMERS Left Outerjoin ORDERS on CUSTOMERS.cust# = ORDERS.cust#
Where CUSTOMERS.city = “Boston”
```

As we show later in chapter 6, implementation methods for outerjoin can be obtained by small modifications of those for join. But outerjoin is very different from join in two respects: (1) it serves to express a significantly larger class of practical applications, and (2) its algebraic manipulation is more complex.

Next we describe briefly some applications of outerjoin. They will be covered in more detail in chapter 3.
Hierarchical views. For some applications, it is convenient to think of information as hierarchically structured—a “parent” object has an associated set of “children”. For instance, each department in a company has an associated set of employees. We can store the information on two tables, one for data concerning departments and another for data on employees. If a join is used to combine information from these files, departments with no employees will be discarded. In general, to construct hierarchical views we need outerjoin rather than join, to preserve objects whose set of “children” may be empty [Roth85, Scho87, Rose90, Davi92].

Nested queries. Commercial languages such as SQL allow “query nesting”—that is, testing a predicate on a single “outer” record may involve computing a subquery. Query nesting is a natural abstraction tool, independent of its SQL implementation. But evaluating queries in this manner may be very inefficient, if a large number of subqueries are computed independently. Algebraic optimization of this class of queries was first studied in [Kim82]. An unusually large number of mistakes have been found in the work done in this area, which might be due, in part, to the lack of a formal framework beyond select/project/join. In the general case, outerjoin is necessary to represent these queries algebraically [Daya87, Mura89].

Database merging. In large companies it is common to find that independent databases have been developed by different groups within the organization. The goal of “federated databases” is to provide a global view that combines all information from these independent databases [Shet90]. There are many aspects to the problem, which has become the subject of active research. For query processing, the way tables need to be combined is using outerjoins, rather than joins [Daya84, Wang90].

Current query optimizers perform only a limited analysis of the queries generated by these applications, as they are beyond the well-studied select/project/join queries.

1.4 Previous work on outerjoin processing

In spite of the importance of outerjoin applications, most of the work on their efficient evaluation has been fragmentary.
In [Daya83], Dayal gives some initial rules on valid evaluation orders for joins and outerjoins, in the context of universally quantified queries — to be discussed in section 3. The focus of his paper is to propose an algebraic representation of quantified queries based on an operator called graft by the author. This graft operator is basically an outerjoin. Then, in [Daya87], he points out that outerjoin — and other operators that are useful for nested subqueries — can be implemented by minor modifications to join algorithms. He also gives some rules to change the order of processing for joins and outerjoin, in the context of nested SQL, but his rules have glitches and do not apply to all processing orders.

In [Mura89], Muralikrishna deals with the use of outerjoin to represent nested SQL queries. He points out mistakes in previous work in the subject, and shows how some of these mistakes can be corrected. He also gives a pipelining algorithm to evaluate the outerjoins that model nested SQL queries. His coverage of how to change the order of evaluation is limited, and does not handle arbitrary orders in which to process relations.

Some identities for processing natural outerjoins are given in [Ozso89], as part of the optimization strategies used by a system that stores and computes statistical information.

As for implementation algorithms of outerjoin, a general algorithm for several binary operators is given in [Kell89]. This general algorithm abstracts principles common to the implementation of relational binary operators, and defines a number of “hooks” to customize the algorithm. The paper describes how to use this general algorithm to compute some cases of outerjoins.

An early attempt to solve the general outerjoin optimization problem is found in [Rose84], where some rules are given to generate different orders of evaluation. This work decomposed outerjoin into very low-level operators, which do not correspond to the way conventional query optimizers — such as System R [Astr76] — work. In more recent, rule-driven optimizers, a large number of transformation rules would need to be applied in the proper sequence to generate meaningful changes in the order of processing. Furthermore, the set of queries obtainable by these rules is not characterized.

The first steps towards a more comprehensive treatment of the problem were the papers by Rosenthal and myself [Rose90, Gali92]. This thesis reformulates and extends the results of those papers.
1.5 Thesis overview and summary of contributions

This thesis develops the necessary theory and algorithms to process outerjoin queries efficiently. The main contributions can be summarized as follows:

- We give algorithms to simplify outerjoins and generate access plans for general join/outerjoin expressions. These algorithms are compatible with current optimizer architectures, so that extending those optimizers should be possible without major redesign or implementation effort.

- We include a complete analysis of the properties of outerjoin and how it interacts with other relational operators.

- The class of freely-reorderable join/outerjoin queries is identified. These queries do not need to specify an order of evaluation, so they allow for more declarative user interfaces. This class is important because it can be handled by minimal extensions of current optimizers, and it is the type of expressions generated by some applications.

- We present a declarative specification of “preserved relations,” which generalize the argument-preserving properties of outerjoins. They provide a convenient conceptual handle to understand join/outerjoin expressions. Also, they appear to be an appropriate framework to develop pipelining algorithms similar to the one presented in [Mura89].

- We give a general notion of candidate evaluation orders of a relational query with binary operators, and show how to obtain valid evaluation orders given an arbitrary set of associative identities. The concept provides a framework in which completeness results can be examined.

- We also give new formalisms that simplify the description and implementation of algebraic optimization in relational databases. They are the following:
  
  - The definition of a relation-containment order, useful to formulate optimization principles.
  
  - The algebraic formulation of argument reduction as a query optimization technique.
– The notion of argument preserving as the algebraic principle behind outerjoins.

– An abstract model for the selection of a processing order for binary operators more general than join.

Chapter 2 presents background material on relational algebra and query processing. We use a new formulation of relational algebra that simplifies the algebraic manipulation of null values introduced by outerjoins.

Chapter 3 reviews some of the applications of outerjoins, extending and complementing the brief descriptions given earlier in this chapter.

Chapter 4 presents algebraic properties of data operators. Some of the associativity identities of outerjoins appeared first in [Rose90, Gali92]. Rather than just presenting a list of algebraic identities, emphasis is given to general concepts and algebraic structures.

Chapter 5 abstracts some of the tools currently used to process join queries, to frame the problem of selecting a processing order for more general binary operators.

Chapter 6 combines the results of chapter 4 and 5 to show how join/outerjoin expressions can be evaluated for any processing order. These results are also presented as algorithms compatible with current optimizer architectures. We describe implementation algorithms for outerjoins.

Chapter 7 presents conclusions and suggests directions for future work.
Chapter 2

Query processing in the relational model

This chapter covers some background material for the problem of outerjoin optimization. We start by reviewing relational algebra. Our formulation of relational algebra is slightly more general than previous ones, with the purpose of simplifying the algebraic manipulation. Then we explain some principles used in the optimization of relational queries. Finally, we describe the class of select/project/join queries, which is the focus of current optimizers in relational databases.

2.1 Relational algebra

We will now present a variant of relational algebra that we use in our work. Our reformulation is aimed at an easier algebraic manipulation of the null values introduced by outerjoins. Rather than explicitly padding tuples with null values, as it is usually done [Date83b, Rose84], we allow relations to contain tuples defined on different sets of attributes. This simplifies the algebraic manipulation considerably.

2.1.1 Basic definitions

Tuples. We start by defining a relational tuple as a partial mapping from a finite set of names called attributes to a set of atomic values. Relational tuples were defined as
tuples in the mathematical sense in the original relational proposal [Codd70]. Although
the name has been kept, the definition of relational tuple as a mapping is commonly used
[Ullm82, Pare89]. We will usually refer to them simply as tuples, as a familiar database
term.

The *scheme* of a tuple $t$ is the set of attributes for which it is defined — i.e. the pre-image
of $t$. The scheme of $t$ is denoted $\text{sch}(t)$.

Tuples are closely related to file records, and we use the conventional dot notation for
the value assigned by a tuple to a given attribute. For tuple $t$ and attribute $a \in \text{sch}(t)$, $t.a$
denotes the value associated to $a$ by $t$.

Since tuples are defined on a finite domain, we can check whether or not two tuples have
equal extensions. For tuples $t_1$, $t_2$, we say $t_1 = t_2$ if $\text{sch}(t_1) = \text{sch}(t_2)$ and they assign the
same value to each attribute.

Besides comparing them, we can do some basic data manipulation with tuples. The
*tuple projection* of $t$ on attributes $A$ is denoted $t[A]$ and restricts the domain of $t$ as follows:

- $\text{sch}(t[A]) = A \cap \text{sch}(t)$; and

- $(\forall a)(a \in \text{sch}(t[A]) \Rightarrow t[A].a = t.a)$.

**Example 2.1 Tuples and tuple operations.**

Writing tuples as a set of attribute/value pairs, examples of tuples are:

$$
t_1 = \{(\text{customer#}, 228), (\text{name}, Jones), (\text{city}, Boston), (\text{phone}, 232-2092)\}
$$

$$
t_2 = \{(\text{customer#}, 350), (\text{name}, Smith), (\text{city}, Cambridge)\}
$$

$$
t_3 = \{(\text{item#}, axz225), (\text{description}, workstation), (\text{price}, $6400)\}
$$

The schemes of those tuples are $\text{sch}(t_1) = \{\text{customer#}, \text{name}, \text{city}, \text{phone}\}$;
$\text{sch}(t_2) = \{\text{customer#}, \text{name}, \text{city}\}$; and $\text{sch}(t_3) = \{\text{item#}, \text{description}, \text{price}\}$.

Examples of tuple projection and referencing of tuple attributes are as follows:

$$
t_1[\{\text{name, city}\}] = \{(\text{name}, Jones), (\text{city, boston})\}
$$

$$
t_3.\text{price} = $640
$$
**Predicates.** A *tuple predicate* $p$ is a total mapping from tuples to \{*True, False*\}; it must have a minimum set of attributes called the *scheme* of the predicate, and denoted $\text{sch}(p)$, such that

$$(\forall t_1)(\forall t_2)(t_1[\text{sch}(p)] = t_2[\text{sch}(p)] \Rightarrow p(t_1) = p(t_2)).$$

Note that the negation, conjunction, and disjunction of tuple predicates is a tuple predicate, as defined above.

We call a predicate $p$ *strong* on attributes $A$ when the following holds:

$$(\forall t)(\text{sch}(t) \cap A = \emptyset \Rightarrow p(t) = \text{False}).$$

That is, the predicate fails for tuples undefined on a given set of attributes. Clearly, if $p$ is strong on $A$, it is also strong on all supersets of $A$.

Intuitively, the scheme of a predicate is the set of attributes such predicate *may* examine in any given tuple. On the other hand, if the predicate is strong on a set of attributes, it means the predicate *must* examine those attributes.

Rather than requiring predicates to be based on comparison operators and logical connectives, we prefer to impose only those restrictions necessary for our algebraic work. This covers user-defined predicates and predicates involving arithmetic operations, which are usually left out in formal accounts of the relational model, yet widely available in commercial systems [Ullm82].

Tuple predicates are only allowed to examine a pre-determined set of attributes, and they always evaluate to either *True* or *False*, even if some of these attributes are not defined in the tuple. It is worth pointing out that, for instance, SQL predicates actually behave this way, and they are strong by default on each attribute they reference, since comparisons are unsuccessful when applied to a null value [Negr91, ANSI92].

**Example 2.2 Tuple predicates.**

Using C-like notation, a tuple predicate $p$ to test whether or not the city-attribute of a tuple is “Boston” may be defined as follows:

```c
#define p(t) ( (city ∈ sch(t)) && (t.city=="Boston") )
```

The operator `&&` evaluates the second part only if the first part succeeds. Since the predicate evaluates to *False* for tuples undefined on attribute city, it is strong
on \{\text{city}\}. Also, the result of the predicate depends only on attribute \text{city}, so its scheme is \text{sch}(p) = \{\text{city}\}.

For the tuples \(t_1, t_2, t_3\) of example 2.1.1, \(p(t_1) = \text{True}; p(t_2) = \text{False}; p(t_3) = \text{False}\).

**Relations.** A relation \(R\) is a pair \(\langle S, E \rangle\). \(S\) is a set of attributes called the scheme of the relation, and denoted \(\text{sch}(R)\). \(E\) is a set of tuples called the extension of the relation, and denoted \(\text{ext}(R)\). The extension of a relation is restricted as follows:

\[
(\forall t)(t \in E \Rightarrow \emptyset \subseteq \text{sch}(t) \subseteq S).
\]

Our definition of relation combines the notions of database schema and instance [Pare89]. The distinction is useful for other aspects of database management. For our purposes, it is convenient to package scheme and extension into a single unit.

The tuples in a relation may be defined on different sets of attributes, as long as they are a subset of the relation scheme. An alternative approach is to fill in with "null" values, and have all tuples defined on the same set of attributes [Date83b]. Our conventions simplify algebraic manipulation. To apply our results in other contexts, null filling, if necessary, can be used either in the low-level file manipulation or to translate relations to higher interface levels.

Also, we forbid the empty tuple to be part of the extension of a relation — the semantics of an empty tuple are unclear.

**Example 2.3 Relations.**

We use the familiar table representation to depict relations. A relation containing tuples \(t_1, t_2\) of example 2.1.1 is shown as a table in figure 2.1. The scheme of the relation gives labels for the table frame, and table rows are taken from the extension. There are no repeated rows, and the order of rows is immaterial.

**Databases.** Clearly, there is a natural correspondence between relations and data files. We will consider a database as a collection of relations, where each of these relations is stored as a data file. They are called the base relations of the database.
<table>
<thead>
<tr>
<th>customer#</th>
<th>name</th>
<th>city</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>228</td>
<td>Jones</td>
<td>Boston</td>
<td>232-2092</td>
</tr>
<tr>
<td>350</td>
<td>Smith</td>
<td>Cambridge</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: Table representation of relations.

### 2.1.2 Relational operators

Relational operators are used to combine information from relations, or to find tuples satisfying some conditions.

**Set union, intersection, and difference.** Since relations are basically a set of tuples, set union, intersection and difference are applicable. We do need to adapt these operators to take into account relation schemes. In the traditional formulation of relational algebra, the union of two relations is defined only for relations with equal scheme. We relax this restriction since, for our algebraic purposes, it adds unnecessary complexity in the manipulation of relational expressions. This “untyped union” is useful for the analysis of current relational queries containing outerjoins, and it need not be made available to users of the query language.

For relations \( R_1 = \langle S_1, E_1 \rangle \), \( R_2 = \langle S_2, E_2 \rangle \), we extend the definition of set union ( \( \cup \) ), intersection ( \( \cap \) ), and difference ( \( - \) ) to operate on relations, as follows:

\[
R_1 \cup R_2 := \langle S_1 \cup S_2, E_1 \cup E_2 \rangle.
\]

\[
R_1 \cap R_2 := \langle S_1 \cap S_2, E_1 \cap E_2 \rangle.
\]

\[
R_1 - R_2 := \langle S_1, E_1 - E_2 \rangle.
\]

**Set containment.** Set comparison is extended to relations by considering only their extension. For relations \( R_1 = \langle S_1, E_1 \rangle \), \( R_2 = \langle S_2, E_2 \rangle \), we say \( R_1 \subseteq R_2 \) whenever \( E_1 \subseteq E_2 \); and similarly for \( \supseteq \). Relations are compared based on their extensions, so the meaning of \( R_1 = R_2 \) is \( R_1 \subseteq R_2 \) and \( R_2 \subseteq R_1 \).

**Relational projection and selection.** Given a relation, we can extract some of its tuples or limit the domain of its tuples. Selection and projection do that.
The relational projection \( \pi \) of \( R = \langle S, E \rangle \) on a set of attributes \( A \) is

\[
\pi_A R := \langle S \cap A, \{t[A] \mid (t \in E) \land (\text{sch}(t) \cap A \neq \emptyset)\}\rangle.
\]

The relational selection \( \sigma \) of \( R = \langle S, E \rangle \) on a tuple predicate \( p \) is

\[
\sigma_p R := \langle S, \{t \mid t \in R \land p(t)\}\rangle.
\]

**Cartesian product.** The basic operator to combine information from two relations with different schemes is the cartesian product. For relations \( R_1 = \langle S_1, E_1 \rangle, R_2 = \langle S_2, E_2 \rangle \) such that \( S_1 \cap S_2 = \emptyset \), the cartesian product \( (\times) \) is defined as follows:

\[
R_1 \times R_2 := \langle S_1 \cup S_2, \{t \mid (\text{sch}(t) \subseteq (S_1 \cup S_2)) \land (t[S_1] \in E_1) \land (t[S_2] \in E_2)\}\rangle.
\]

Figure 2.2 examples of the application relational operators applied on tables \( R, S, \) and \( T \).

### 2.1.3 Join-like operators

The operators defined in this section are expressed in terms of the basic relational operators of the previous section. It is still useful to treat them as operators in their own right, because they corresponds to common conceptual operations, and they also turn out to be implementation units. We first define join —the most common operator to combine relations— and then some of its variants. In the case of these variants, the defining relational expressions are significantly more complex than that of regular join. A main goal of this thesis is to work out the algebraic difficulties that have prevented the adequate processing of these join variants. They still are conceptual units, as will be shown in the applications of next chapter. Later in chapter 6, we show how they all share a common implementation structure.

**Join.** *Relational join* applies some restrictions to the result of cartesian product. The join of relations \( R_1 = \langle S_1, E_1 \rangle, R_2 = \langle S_2, E_2 \rangle \) on tuple predicate \( p \) is defined as follows:

\[
R_1 \bowtie_p R_2 := \sigma_p(R_1 \times R_2).
\] (2.1)
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>f</td>
</tr>
<tr>
<td>c</td>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>g</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>a</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td>b</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>f</td>
</tr>
<tr>
<td>c</td>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td>b</td>
<td>f</td>
</tr>
</tbody>
</table>

**Figure 2.2:** Basic operators of relational algebra.
Predicate $p$ will usually test some form of “match” between tuples in $R_1$ and those in $R_2$, and it is sometimes called the \textit{match predicate}. Given tuples $t_1 \in R_1$ and $t_2 \in R_2$, $p(t_1, t_2)$ determines whether or not the pair will go into the result — i.e. if $t_1$ matches $t_2$.

**Semijoin and antijoin.** Since the join of relations $R_1$, $R_2$ applies some match predicate, it may not preserve all tuples from its arguments. If there is a proper subset $R'_1 \subset R_1$ such that $R'_1 \blacktriangleleft R_2 = R_1 \blacktriangleleft R_2$, some tuples in $R_1$ do not have a matching tuple in $R_2$. This observation is the basis of some query processing algorithms that “reduce” relations in a distributed environment before sending them to another network site [Bern81, Bern83, Aper83]. The reducer operator is called \textit{semijoin}, and it is defined as follows:

$$R_1 \blacktriangleleft R_2 := \pi_{\text{ch}(R_1)}(R_1 \blacktriangleleft R_2). \quad (2.2)$$

The complement of semijoin is called \textit{antijoin}, and it consists of the tuples of $R_1$ for which there is no matching tuple in $R_2$, for a given match predicate. The definition of antijoin has an embedded universal quantifier that is algebraically expressed as set difference. It is defined as follows:

$$R_1 \triangleright R_2 := R_1 - (R_1 \blacktriangleleft R_2). \quad (2.3)$$

**Outerjoins.** The outerjoin of two relations includes the result of join, plus all unmatched tuples from one or both of its arguments. This way it preserves all the information from its arguments. The operator was first examined by Date in [Date83a]. Its applications are the subject of next chapter. We define \textit{left outerjoin} and \textit{full outerjoin}, respectively, as follows:

$$R_1 \nrightarrow R_2 := (R_1 \nrightarrow R_2) \cup (R_1 \nleftarrow R_2). \quad (2.4)$$

$$R_1 \nrightarrow R_2 := (R_1 \nrightarrow R_2) \cup (R_2 \nrightarrow R_1). \quad (2.5)$$

We also use the \textit{right outerjoin}: $R_1 \nleftarrow R_2 := R_2 \nrightarrow R_1$. Left and right outerjoin preserve only one of its arguments, so they are called one-sided outerjoins or 1-outerjoins; full outer-join is also called two-sided outerjoin or 2-outerjoin, because it preserves information from both arguments.

Using relations $R, S$ of figure 2.2, figure 2.3 shows examples of join, semijoin, antijoin, and outerjoin.
Figure 2.3: Join-like operators.
2.1.4 Natural join and outerjoin

Join operators are based on cartesian product, and thus require that relations have disjoint schemes. This is natural at an implementation level—even if two records types have a common field name, there is always a way to disambiguate any field reference.

From a more conceptual point of view, using the same attribute name in two different relations may reflect some semantic correspondence. The natural join operates on two relations whose schemes intersect, and combines tuples that coincide in their common attributes. This operator can also be expressed using attribute renaming, relational join, and a final projection. Note how, in figure 2.3, all tuples in $R^{B=D} S$ have the same values for columns $B$ and $D$. Projecting out one of those attributes does not lose any information.

Natural join can be regarded as a conceptual abstraction directly implemented on relational join—attribute renaming requires no data manipulation at the level of files. The algebraic properties of natural join are basically those of join.

Natural one-sided outerjoin can also be obtained by renaming, computing the regular outerjoin, and then projecting out some attributes. For relation $R^{B=D} S$ of figure 2.3, we can discard column $D$ without losing any information. In contrast with join, discarding $B$ does lose information.

Natural full outerjoin cannot be expressed using the same strategy—i.e. projecting the result of regular full outerjoin. To see the difficulties, consider relation $R^{B\leftrightarrow D} S$ of figure 2.3. We lose information by projecting out column $B$ or $D$.

To obtain the natural outerjoin, tuple coalescing has been suggested; that is, “merging” the information in some attributes of each tuple [Date83b]. For our previous example, both columns $B$ and $D$ would be replaced by a new column that merges the information—merging is natural because either the values on $B$ and $D$ are equal, or one of them is not defined.

If we use this approach to define natural full outerjoin, then we need to explore the properties of a new coalescing operator to process natural full outerjoin. Instead of using tuple coalescing, we propose a new construction that uses attribute renaming and regular one-sided outerjoins. Algebraic properties of one-sided outerjoin can then be used to analyze natural full outerjoin.
To construct the natural full outerjoin, we use an auxiliary, virtual relation containing the common attributes of both arguments. For relations $R_1 = \langle S_1, E_1 \rangle$, $R_2 = \langle S_2, E_2 \rangle$ where $S_1 \cap S_2 = S_3 \neq \emptyset$, choose new sets of attributes $S_{31}$, $S_{32}$ of the same cardinality of $S_3$, disjoint with each other and with $S_1$, $S_2$. Compute the natural outerjoin as follows:

- Set $R_{aux}$ to $\pi_{S_3}R_1 \cup \pi_{S_3}R_2$.
- Rename $S_3$ as $S_{31}$ in $R_1$, obtaining $R'_1$.
- Rename $S_3$ as $S_{32}$ in $R_2$, obtaining $R'_2$.
- Get $R_1 \leftrightarrow R_2$ by computing $\pi_{S_1 \cup S_2}(R'_1 \upharpoonright_{\pi_{S_{31}}=\pi_{S_{32}}} R_{aux} \upharpoonright_{\pi_{S_{31}}=\pi_{S_{32}}} R'_2)$.

The construction is useful at the conceptual level for query analysis. At the implementation level, two-sided outerjoin and tuple coalescing are likely to be used.

**Example 2.4 Using one-sided outerjoins to construct natural full outerjoin.**

Figure 2.4 shows how the natural outerjoin is constructed using one-sided outerjoins. The data is the same as that of $R, S$ in figure 2.2, but now there is a common attribute $B$, to make natural outerjoin applicable. The common attribute $B$ is renamed to $B_r$ and $B_s$ to obtain $R'$ and $S'$, respectively.

### 2.1.5 Other formulations of relational algebra

Our formalization of relational algebra is a slight generalization of other formulations. Our purpose is to make algebraic manipulation easier, in particular for expressions containing outerjoins. Our results can be translated to the relational model, as it is usually defined, by padding with null values all tuples in each relation $R$ until they are defined for all attributes of the scheme of $R$.

Below, we discuss our deviations and their benefits:

- We allow rows with no value assigned to some of the attributes in the relation scheme. This can be regarded as a particular way to manipulate null values, which necessarily appear with outerjoins. The algebraic manipulation becomes easier, in particular because we do not require relations to have the same scheme when computing unions.
We define tuple predicates as functions satisfying some general properties needed for algebraic optimization, rather than build them from comparison operators. This formulation captures a larger class of predicates, common in practice but seldom formalized. Also, our formulation allows the application of the same predicate on relations with different schemes—a detail that is not always clear in other formalizations.

We have made the relation scheme a part of the relation itself, instead of having a database schema and database instances. The separation is unnecessary for our algebraic work.

We also gave a new construction of natural full outerjoin based on attribute renaming and one-sided outerjoin. This is useful because the properties of one-sided outerjoin can be used in the analysis of natural full outerjoin.

### 2.2 Query optimization principles

In the physical implementation of relational databases, base relations are usually stored as data files, and user queries are translated into algebraic expressions with relational operators
to be evaluated on those data files. A wide spectrum of algorithms can be used to compute the result of a particular expression, and these algorithms may be very different in terms of their resource requirements. The task of the query optimizer is to find an efficient procedure to compute a query, according to the way data is physically stored.

2.2.1 Operators and their implementation

Given a collection of relations stored as data files, it is easy to find algorithms to compute the relational operators we have defined. For relational selection, for instance, we could scan a file sequentially and test each record, outputing to the result all tuples that satisfy the selection predicate; or, in some cases, we may be able to exploit indexes rather than scanning the complete file.

To evaluate relational expressions on a computer system we must determine, for each operator, an algorithm to implement it on files. These algorithms are called *implementation methods*. A given operator has usually several implementations, depending on its arguments and also on the physical paths available. To illustrate this, we can call *scan-select* and *index-select* the two algorithms briefly outlined above to evaluate relational selection. Scan-select can always be applied; index-select is faster for some predicates, but can only be applied if an index on the appropriate field exists.

An algebraic expression in which each operator is annotated with an implementation method is called an *access plan*. Detailed analyses of the cost of implementation methods appear in the literature [Ullm82].

**Join as an implementation unit.** Even though relational join can be expressed in terms of simpler relational operators, it is both a conceptual and an implementation unit. The implementation aspect is shown in the following example.

**Example 2.5 Cartesian product and join methods.**

For our database of clients and purchase orders, we can use the following query to obtain a listing of all clients and the purchase orders they have placed:

```
Select All
From CUSTOMERS, ORDERS
```
Where CUSTOMERS.cust# = ORDERS.cust#

We generate a big intermediate relation if we first compute the cartesian product of both relations and then apply the Where predicate. Computing this product takes time proportional to the product of the sizes of the relations involved.

If we implement directly the join operator, we may take advantage of indexes or other physical access paths. In particular, assume that both CUSTOMERS and ORDERS are sorted on cust#. Then we can do a simultaneous sequential scan of the files, in a merge-like fashion, and output in a single pass the result of the join. This takes time proportional to the sum of the sizes of the relations involved.

Implementation algorithms for join operators are described in section 6.5.

2.2.2 Algebraic optimization

Clearly, operator methods are necessary to evaluate relational expressions on physical files, and method selection is one tactic available to the query optimizer. But the most powerful optimization techniques come from what is called algebraic optimization, where algebraic properties of data operators are used to process operators in a different order than originally specified. The following example illustrates this.

Example 2.6 Algebraic optimization.

We return to the query of example 1.1, and show the algebraic manipulation done during optimization. The algebraic properties used are examined in chapter 4. To find all purchase orders on item “axz225,” and the customers who placed them, we use:

Select All  
From CUSTOMERS, ORDERS  
Where ORDERS.item# = “axz225” and  
     CUSTOMERS.cust# = ORDERS.cust#
The process of algebraic optimization on this query is shown in figure 2.5. First, since there is a selection conjunct that references only one of the two relations, we can use it to reduce the relation before the Cartesian product takes place. Then we replace the Cartesian product by a join, because a selection is being applied to the Cartesian product result.

These algebraic transformations were the basis of example 1.1. The Cartesian product of the original expression generates a big intermediate relation. Since the predicate (ORDERS.item# = “axz225”) is highly selective, the size of intermediate results in the optimized query is much smaller than in the original.

Changing the order in which operators are applied is based on the known algebraic properties of these operators. The size of intermediate results can be estimated before implementation methods are chosen [Yao77, Ceri85, Lipt90]. A relevant concept is that of an operator with high selectivity — one that is expected to reduce significantly the size of its arguments. As a general heuristic, it makes sense to try to evaluate first the selective operators of a query.

Relational selections are often highly selective. Joins are also selective sometimes. For instance, consider that the match predicates are equalities on keys, and the sizes of $R_1$, $R_2$, $R_3$ are 3 tuples, 1 million tuples, and 1 million tuples, respectively — $R_1$ might be the result of a reduced relation. Then, by selectivity arguments, evaluating $(R_1 \bowtie_1 R_2) \bowtie_1 R_3$ is likely to be much better than evaluating $R_1 \bowtie_1 (R_2 \bowtie_1 R_3)$.

Chapter 4 develops in detail algebraic properties useful for optimization.

2.3 Select/project/join queries

Select/project/join queries have been the focus of most research and development in relational query optimization. The algebraic structure of these queries is well understood, and it is the basis of most optimizers in commercial relational databases. We cover next some of the basic ideas behind this class of queries and its optimization.
\( \sigma(\text{ORDERS.item} \neq "\text{axz225}" \land (\text{CUSTOMERS.cust} \neq \text{ORDERS.cust})) \)

(a) Initial expression.

\( \sigma\text{CUSTOMERS.cust} = \text{ORDERS.cust} \)

\( \times \)

\( \times \)

CUSTOMERS

\( \sigma\text{ORDERS.item} = "\text{axz225}" \)

ORDERS

(b) Push selection down.

\( \text{CUSTOMERS.cust} = \text{ORDERS.cust} \)

\( \times \)

CUSTOMERS

\( \sigma\text{ORDERS.item} = "\text{axz225}" \)

ORDERS

(c) Use join instead of product.

Figure 2.5: Algebraic optimization steps.
The class of S/P/J queries. Select/project/join queries (S/P/J queries) are those expressed by a relational expression of the following form:

\[ \pi_A \sigma_p(R_1 \times R_2 \times \ldots \times R_n), \]

where \( A \) is a set of attributes from \( R_1, \ldots, R_n \), and \( p \) is a tuple predicate in conjunctive normal form. Disregarding the final projection, the algebraic form corresponds to:

\[ \{(t_1, t_2, \ldots, t_n) \mid t_1 \in R_1, t_2 \in R_2, \ldots, t_n \in R_n, p(t_1, t_2, \ldots, t_n)\} \]

S/P/J queries are also called existentially quantified conjunctive queries, or simply conjunctive queries [Ullm89]. Queries whose logical description involves universal quantifiers are harder to express algebraically [Ullm82]. We will come back to those queries in the next chapter.

Although the class of S/P/J queries is considerably less expressive than the complete relational algebra, it has the advantage of an identified set of algebraic optimization heuristics, and it does cover a large number of queries found in common business applications. In example 2.2.2, we used an S/P/J query to illustrate algebraic optimization.

Optimization techniques. The main optimization techniques for S/P/J queries are based on the following observations and properties —to be examined in detail in chapter 4:

- A selection on predicate \( p_1 \land p_2 \) can be replaced by two consecutive selections, first on \( p_1 \) and then on \( p_2 \).
- If predicate \( p \) references only attributes of \( R_1 \), then \( \sigma_p(R_1 \times R_2) = (\sigma_p R_1) \times R_2 \).
- Cartesian product is associative.
- Cartesian product followed by selection becomes join.

And the algebraic optimization heuristics consist basically of these two parts:

- First, apply all predicate conjuncts that refer to one relation only, to reduce the size of individual relations, if they are likely to do so. Predicates that are true for most of the tuples —e.g., salary less than \( 1 \times 10^6 \)— are delayed until the relation is access by other operator.
• Second, group cartesian products and selections to obtain joins, and select an order of evaluation of these joins that minimizes the size of intermediate results.

There are many details involved in the actual implementation and integration of the above two rules, and how they affect method selection, but they can be considered as the two basic guidelines for the algebraic optimization of S/P/J queries.

**SQL query blocks.** It is usually easy to express S/P/J queries in commercial database languages. In the case of SQL [Astr75], they are closely identified with *query blocks*. A query block in SQL has the following basic structure:

```
Select A 
From R_1, R_2, ..., R_n 
Where p 
```

A is a set of attributes from relations $R_1, ..., R_n$, and $p$ is a predicate. $t_1, ..., t_n$ are introduced as tuple variables associated to particular relations. It corresponds to $\pi_A \sigma_p(R_1 \times R_2 \times \cdots \times R_n)$ [Negr91]. We do not consider the **Having** and **Group-by** clauses of SQL, which implement operations not expressible in relational algebra [Ullm82].

Query blocks can then be combined in various ways, allowing, for instance, relational union and difference. There is also a particular form of nesting blocks, so that a predicate $p(t)$ in the **Where** clause computes a subquery for each value of $t$ — references to $t$ in the subquery get instantiated for each tuple. Nested query blocks are described in some detail in the next chapter.

Although SQL queries may be composed of several query blocks, the unit of optimization in most systems is still the query block. Algebraic transformations spanning several blocks are outside the scope of S/P/J queries. This area has been a subject of great interest [Kim82, Daya87, Mura89, Kris92], but a complete, implementable solution is still missing. The outerjoin processing strategies developed in this thesis provide solid grounds on which part of the issues involved in multi-block query processing can be developed.

We will examine nested queries in more detail in chapter 3.

**Query graphs** A useful auxiliary structure to explore alternative execution orders for joins is the *query graph*. For a conjunctive query in which all conjuncts reference two or
more relations—conjuncts on one relation may have been already applied to reduce base relations, or they are subsumed by conjuncts on more relations—we create a query graph as follows:

- For each relation \( R_i \) in the quantified query, create a graph node, labeled \( R_i \).
- For each conjunct \( p \) in the quantified query, add an edge in the graph connecting all relations referenced by \( p \), and label it \( p \). If more than two relations are referenced, hyperedges are used [Ullm82].

**Example 2.7** Query graphs for S/P/J queries.

Figure 2.6 shows the query graph and an algebraic expression to evaluate the following SQL query:

```
Select A
From  R_1, R_2, R_3, R_4
Where  p_1(R_1, R_2) and p_2(R_2, R_3) and p_3(R_2, R_4)
```

An edge between two relations indicates that they have to be joined as part of the query. To compute the query, we can replace a pair of nodes \( R_i, R_j \) of the graph by a new node \( R_k \) representing the result of the join—references to \( R_i, R_j \) in the remaining graph must then
be changed to refer to $R_k$. Hyperedges require multiway join [Ullm82], and disconnected nodes are processed by cartesian product. By successively reducing the graph in this way, we are actually determining an order in which to process joins.

Query graphs are a fundamental structure for the analysis of queries in database optimizers. Among its applications are processing queries in distributed databases [Bern81, Ceri85] and in the universal-relation model [Ullm89]. They are a central data structure in our work, and in chapter 5 we develop the issue of query graphs and algebraic expressions.

Another important application of query graphs is the simplification of user interfaces, since they provide a visual representation of relations and how they are being combined. This representation captures the query semantics because join is associative and commutative, and thus a query graph denotes a unique answer without specifying an execution order.
Chapter 3

Applications of outerjoin

In chapter 2 we reviewed the well known class of select/project/join queries, and how they
are used and optimized. We now present some important applications whose queries are
beyond that class, and show how they can be expressed using outerjoins. Later chapters
consider the efficient evaluation of outerjoin queries.

3.1 Outerjoin in commercial query languages

Date proposed an outerjoin construct for SQL in [Date83a]. The operator is available to-
day in a number of implementations of SQL, and it is included in the ISO-ANSI standard
draft for SQL2 [ANSI92]. Among the systems that currently support outerjoin are NonStop
SQL of Tandem [Tand90], SSQL of ShareBase [Shar91], and ORACLE/SQL [Hart92]. The
syntactic construct for outerjoin in the first two of these systems follow the ANSI recom-
mendation. In SQL2, outerjoins are specified in the From clause of query blocks, as shown
earlier in example 1.3. For instance, the following query

\[
\begin{align*}
& \text{Select} \quad A \\
& \text{From} \quad R_1 \text{ Left Outerjoin } R_2 \text{ on } p_1, R_3 \\
& \text{Where} \quad p_2 \\
\end{align*}
\]

corresponds to

\[
\pi_A \sigma_{p_2}( (R_1 \overrightarrow{p_1} R_2) \times R_3 ).
\]
The place of parentheses in the above expression is essential to the semantics of the query. As stated, if $R_3$ is empty then the above query evaluates to an empty relation; the result could be non-empty if we performed the cartesian product first and then the outerjoin. SQL2 semantics prescribe taking the cartesian product —associative and commutative— of all table expressions in the From clause, and then apply a selection on the conditions in the Where clause.

A single table reference can specify more than one outerjoin, and both one-sided and two-sided outerjoins are allowed. Natural outerjoins can also be specified.

### 3.2 Hierarchical information

#### 3.2.1 Hierarchical views

In database systems, a view is a virtual relation presented to a group of users, defined by means of a query on the base relations. Queries on views are well defined; translation of view updates into base relations is not defined, in general. Users treat views as if they were real relations, and the system replaces references to views by their defining expressions when processing queries. One purpose of views is to present information in the database in the most suitable form for each group of users, while keeping a single data repository. The view abstraction also allows changes in the way data is organized in base relations without altering the user views —all we need to do is update the mapping between views and base relations. Views are an important database abstraction with important applications and challenging problems [Daya79].

To illustrate hierarchical views, assume we have a database with information about courses and classrooms. Data is stored in three relations, COURSES, ROOMS, and ASSIGNMENTS —for room assignments. Sample data for these relations is shown in figure 3.1. The join of these relations gives all information for each course and room in a single line, which could be an appropriate database view for some users.

But we may have courses for which no classroom has been assigned yet. Information about those courses would be lost when we take the join of our base relations. A hierarchical view presents all courses, appending any available classroom information. This is actually an outerjoin COURSES → (ASSIGNMENTS ≀ ROOMS).
Similarly, we may have classrooms that have not been assigned to any course yet. For users interested in classrooms and all their related information, the hierarchical view is rooted on ROOMS, and it corresponds to the outerjoin ROOMS $\rightarrow$ (ASSIGNMENTS $\bowtie$ COURSES). These hierarchical views for sample data are shown in figure 3.1.

The idea of a hierarchical view is that each "parent" object has a set of "children." Outerjoin is used to combine data to preserve objects with no children. The hierarchy can be composed of several levels; for instance, we could have professors, courses, and classrooms, as an extension of our previous example. Outerjoin is used to combine information between any two levels.

It is well known that relational algebra cannot compute the transitive closure of a relation [Ullm82]. This limitation carries over to our outerjoin-constructed hierarchies. Given a relation of persons in which each person indicates two parents, we can define a hierarchical view of pairs of people (parent,children), or (parent, children or grandchildren), but we cannot express a view for (parent, descendant) for an unbounded number of generations.

**Nested relations.** The hierarchical view of data we have seen is closely related to the *nested relational model*, where relation attributes may be set-valued [Roth88, Scho87, Pare89]. The hierarchical view rooted at classrooms of figure 3.1 could be represented as a nested relation as shown in figure 3.2. Instead of having one tuple with parent-child information for each child, we present the children as a set associated to the parent. The relation in figure 3.1 is said to be in *first normal form* (1NF), while that in figure 3.2 is in *nested form*.

The algebra of nested relations includes operators *nest* and *unnest* that convert a 1NF relation to a nested relation, and vice-versa. Some form of null or empty values are needed in the 1NF relation to adequately model a set-valued attribute that happens to be empty [Roth85].

Nested relations can serve as the basis of user interfaces, even if data is stored in first normal form. As shown in our previous example, nested relations would be mapped to hierarchical views, which can in turn be obtained by means of outerjoins.
### COURSES

<table>
<thead>
<tr>
<th>course#</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS51</td>
<td>Intro CS</td>
</tr>
<tr>
<td>CS152</td>
<td>Prog Lang</td>
</tr>
<tr>
<td>CS162</td>
<td>Op Sys</td>
</tr>
<tr>
<td>CS175</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

### ASSIGNMENTS

<table>
<thead>
<tr>
<th>course#</th>
<th>room#</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS51</td>
<td>SCD</td>
<td>2pm</td>
</tr>
<tr>
<td>CS152</td>
<td>A101</td>
<td>3pm</td>
</tr>
<tr>
<td>CS175</td>
<td>A101</td>
<td>10am</td>
</tr>
</tbody>
</table>

### ROOMS

<table>
<thead>
<tr>
<th>room#</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCD</td>
<td>250</td>
</tr>
<tr>
<td>A101</td>
<td>70</td>
</tr>
<tr>
<td>G23</td>
<td>20</td>
</tr>
</tbody>
</table>

### COURSES → (ASSIGNMENTS ∝ ROOMS)

<table>
<thead>
<tr>
<th>course#</th>
<th>name</th>
<th>time</th>
<th>room#</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS51</td>
<td>Intro CS</td>
<td>2pm</td>
<td>SCD</td>
<td>250</td>
</tr>
<tr>
<td>CS152</td>
<td>Prog Lang</td>
<td>3pm</td>
<td>A101</td>
<td>70</td>
</tr>
<tr>
<td>CS175</td>
<td>Graphics</td>
<td>10am</td>
<td>A101</td>
<td>70</td>
</tr>
<tr>
<td>CS162</td>
<td>Op Sys</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### ROOMS → (ASSIGNMENTS ∝ COURSES)

<table>
<thead>
<tr>
<th>room#</th>
<th>capacity</th>
<th>time</th>
<th>course#</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCD</td>
<td>250</td>
<td>2pm</td>
<td>CS51</td>
<td>Intro CS</td>
</tr>
<tr>
<td>A101</td>
<td>70</td>
<td>3pm</td>
<td>CS152</td>
<td>Prog Lang</td>
</tr>
<tr>
<td>A101</td>
<td>70</td>
<td>10am</td>
<td>CS175</td>
<td>Graphics</td>
</tr>
<tr>
<td>G23</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Hierarchical views of data.
### 3.2.2 A query language for set- and entity-valued attributes

We now show how outerjoins can be used to evaluate extensions of SQL that handle relations with set- or entity-valued attributes. These extensions were devised by J. Bauer (unpublished), as part of an effort to design a new commercial DBMS with somewhat more than relational power. A. Nori and J. Hee helped refine the operators’ semantics. The “free reorderability” criterion in [Rose90] emerged from an effort to validate these designers’ intuition that this language would be easily optimizable.

#### The operators unnest and link

Two new operators, denoted $*$ and $->$, can be used in the **From** clause of an SQL query. We will use *entity* as a synonym for tuple, to emphasize that in this section’s model, tuples have identity, repeating fields, and entity-valued attributes.

**UnNest or Flatten ($*$).** If $R$ has a set-valued attribute $Field$, $(R \ast Field)$ indicates the unnesting of the attribute. The result is a relation $R'$ with a scheme identical to that of $R$, except that $Field$ is now single-valued. Each tuple $r$ of $R$ will produce one or more tuples in $R'$: If $r.Field$ had $n > 0$ elements, then $R'$ has $n$ tuples for $r$, each tuple’s $Field$ containing a single value from the set; if $r.Field$ is empty then there is only one tuple for $r$ in $R'$ and it has a null value in $Field$. For example,

```sql
Select  All
From    EMPLOYEE*ChildName, DEPARTMENT
```

![Figure 3.2: A relation in nested form.](image)
Where  EMPLOYEE.D# = DEPARTMENT.D# and  

DEPARTMENT.Location = “Queretaro”

returns at least one tuple for each employee in a Queretaro department. For Queretaro employees with children, one tuple is returned for each child; otherwise, a tuple with null ChildName is returned.

Link via (→). If \( R \) has an entity-valued attribute \( Field \), \( (R→ Field) \) indicates the completion of each tuple \( r \) in \( R \) by concatenating to it the tuple referenced by \( r.Field \). The scheme of the result is \( \text{sch}(R) = Field \cup \text{sch}(\text{the entity type referenced by Field}) \). In the extension, for each tuple in \( R \) we will have exactly one tuple in the result. If \( r.Field \) is null, then a null tuple is concatenated to \( r \).

For example, suppose DEPARTMENT includes an EMPLOYEE-valued attribute Manager, and a REPORT-valued attribute Audit. The following query returns, for all Zurich departments, the department information, plus the employee attributes of the manager, and the report information from the audit:

\[
\text{Select All From DEPARTMENT→Manager→Audit} \\
\text{Where DEPARTMENT.Location = “Zurich”}
\]

Attributes obtained from the right side of → and * operators cannot appear in the Where-List predicates because the position of the restriction predicate would be ambiguous, either before or after unnesting. But they may be restricted in an enclosing query block.

Expressing link and unnest with outerjoins

We can make * and → special cases of outerjoins by defining traversable predicates on object identifiers. The reformulation is similar to that used in [Rose85] to apply relational optimization to SQL over Codasyl structures. Assume below that every entity, and also every field value, has a unique object identifier (e.g., a physical address on disk), denoted by the prefix @. The restatement in terms of outerjoins requires introducing special predicates NestedIn and LinkedTo that connect an entity to its repeating values or related entities.
These predicates are evaluated only as part of a join or outerjoin, and require the optimizer to generate code to traverse to the related value or entity. For example, given an EMPLOYEE instance denoted $e$, the access routines can find all DEPARTMENTS $d$ such that $\text{Manager}(d) = e$.

The operators are expressed in terms of outerjoin as follows:

- For UnNest, suppose we have an expression $(R \ast \text{Field})$. Define an abstract, one-column relation $\text{ValueOfField}$ containing every value appearing in $r.\text{Field}$ for any $r$ in $R$. Then define a predicate $\text{NestedIn}(\text{at} r, \text{at} \text{value})$ between identifiers of tuples of $R$ and $\text{ValueOfField}$, true whenever $\text{value}$ is in $r.\text{Field}$. The UnNest expression can be written using outerjoin ($\rightarrow$) as

$$R \rightarrow^{\text{NestedIn}(\text{at} r, \text{at} \text{value})} \text{ValueOfField}.$$  

- For Link, suppose we have an expression $(R \rightarrow \text{Field})$ where the entity-valued $\text{Field}$ points to a tuple in relation $\text{DomainOfField}$. Define a join predicate $\text{LinkedTo}(\text{at} r, \text{at} \text{value})$ between identifiers of tuples of $R$ and $\text{DomainOfField}$, true whenever $r.\text{Field}$ points to $\text{value}$. The Link-via expression can be written using outerjoin ($\rightarrow$) as

$$R \rightarrow^{\text{LinkedTo}(\text{at} r, \text{at} \text{value})} \text{DomainOfField}.$$  

### 3.3 Database merging

The integration of independently developed databases has been receiving increasing attention by the database community. For large organizations, it is common that different groups or departments develop their databases independently. At some point it becomes desirable to integrate the scattered information, but building a new database that would replace all the others is not feasible. The alternative is to build a system that operates as a higher layer, combining information from a collection of databases, but allowing their normal autonomous operation. The approach has been known as *multidatabases* or *federated databases* [Shet90].

There are many aspects to the problem. For instance, databases will probably be distributed in several machines, they may use different data models, and some information
may be partially duplicated. From the point of view of query processing, two sided outer-join has been identified as a key component for integrating information [Daya84, Wang90].

For example, assume we have two databases with information about checking accounts and loans. By combining these two relations we want to collect all information we have about the accounts of each client. Assuming that keys have been reconciled —i. e. clients are recorded with the same name in both relations— we use natural two-sided outerjoin to combine information. Tuples that agree on common attributes are merged into a single tuple, and we make sure all tuples from the original relations appear in the result, even if they provide only partial information. An example of this integration on sample data is shown in figure 3.3.

The result of integrating information from the individual databases is made available as a view, where queries may be posed. The principles of algebraic optimization still apply, although the cost model must also consider issues particular to the component databases and the integration strategies.
3.4 Nested queries

3.4.1 Nested SQL blocks

Test for empty/nonempty subquery. The simplest form of nesting SQL query blocks uses the \texttt{Exists} keyword. The \texttt{Exists} (subquery) predicate is \texttt{True} whenever the result of the subquery is nonempty. For instance, a query to find all customers who have ordered item “axz225” is shown next using \texttt{Exists} nesting:

\begin{verbatim}
Select  All
  From  CUSTOMERS
Where  Exists ( Select  All
                From  ORDERS
                Where  CUSTOMERS.cust# = ORDERS.cust# and
                              ORDERS.item# = "axz225"

SQL semantics prescribe that, for each \(c\) in \texttt{CUSTOMERS}, the subquery must be evaluated. Then \(c\) is in the result of the query whenever the result of the subquery is not empty. Actually, the query can be evaluated using semijoin [Daya87]:

\[ \texttt{CUSTOMERS \WHERE \texttt{ORDERS.cust#} = \texttt{ORDERS.cust#} \AND \texttt{ORDERS.item#} = "axz225" \ATOMIC JOIN \texttt{ORDERS}} \]

Nesting can also be done using \texttt{not Exists}. A query to find customers who have no purchase orders is

\begin{verbatim}
Select  All
  From  CUSTOMERS
Where  not Exists ( Select  All
                           From  ORDERS
                           Where  CUSTOMERS.cust# = ORDERS.cust#

In this case, the query can be computed using antijoin.

\end{verbatim}
Aggregates. Besides testing for empty/nonempty subqueries, we can compute an aggregate value from a subquery. This value can then be used for scalar comparisons. The following query uses the **Count** aggregate to find clients who have placed at most three purchase orders.

```sql
Select All
From CUSTOMERS
Where 3 \geq \text{Count (}

Select All
From ORDERS
Where CUSTOMERS.cust# = ORDERS.cust#
```

Other aggregates such as the sum or average of the elements in the subquery are also available. Rather than following a “tuple iteration semantics,” we would like to represent the query as an algebraic expression on which the query optimizer may work. The above query can be solved using the following steps:

- Using the hierarchical view concepts of section 3.2.1, create a nested relation $R_1$ for

  $\text{CUSTOMERS.cust#} \Rightarrow \text{ORDERS.cust#} \Rightarrow \text{ORDERS}$

- Replace the set-valued attribute of $R_1$ by the cardinality of the set, to create a 1NF relation $R_2$ — i.e. compute the aggregate.

- Output tuples for which the cardinality computed in the previous step is less than three.

These steps given should be taken as defining an algebraic expression subject to optimization, rather than as an actual algorithm. They are based on the work on processing nested queries in [Daya87].

### 3.4.2 Universal quantifiers

Dayal proposed in [Daya83] the evaluation of queries with universal quantifiers using a strategy similar to that outlined above for nested SQL blocks. A query of the form

$$\{t \mid t \in R_1 \land (\forall u \in R_2)(p_1(t,u) \Rightarrow p_2(t,u))\}$$

38
Figure 3.4: Solving queries with universal quantifiers.

is evaluated using the following strategy:

- Create a nested relation $R'_1$ with the result of $R_1 \overset{p}{\Rightarrow} R_2$. Each tuple in $R'_1$ has the form $(t, \{u_1, \ldots, u_n\})$, where $t \in R_1$, $u_i \in R_2$.

- Create a relation $R''_1$ as follows: if a tuple of the form $(t, \{u_1, \ldots, u_n\})$ is in $R'_1$, add tuple $(t, \{p_2(t, u) | u = u_i\})$ into $R''_1$—similar to computing an aggregate.

- The result of the query are those tuples in $R''_1$ with no value $False$ in their nested relations.

Figure 3.4 shows the application of the above steps to compute the query

$$Q(R_1, R_2) := \{t | t \in R_1 \land (\forall u \in R_2)(t.A = u.E \Rightarrow u.F = "c")\}$$
Chapter 4

Algebraic properties of relational operators

Algebraic properties of relational operators have been studied in the context of query processing [Smit75, Ullm82], as they constitute the basis of the most powerful query optimization techniques. Although significant insight was needed to devise these techniques, proving the necessary algebraic properties is not overly complicated. For outerjoin, however, the difficulties in their algebraic handling have long been recognized [Date83a, Date83b], and few results have been obtained to date. The algebraic properties of outerjoin need to be worked out if we are to process outerjoin queries efficiently. Such properties are examined in this chapter.

As we increase the number of operators and their complexity, it becomes harder to describe algebraic properties by presenting a list of identities to be proved independently — the number of combinations grows exponentially with the number of operators considered. Rather than attempting to give a comprehensive list of identities, we focus on general structuring concepts, in particular those relevant to query optimization techniques.

4.1 Basics

The basis of our formulation is a relation containment ordering over relations. This order simplifies the statement of some general query optimization strategies. We then review some properties of the basic relational operators according to our proposed framework. Proofs of
identities are given in appendix A.

### 4.1.1 Ordering the domain of relations

The notion of a relation “containing” another is frequently exploited in query processing. For instance, the optimization heuristic of early application of selections consists of finding “subrelations” whose information is sufficient to solve a given query. Also, the outerjoin intuitively “contains” the join and also its arguments. The notion is formalized next; it defines a partial order in the domain of relations:

**Definition.** A relation $R_1$ is said to be contained in $R_2$, denoted $R_1 \leq R_2$, if there exists a set of attributes $A$ such that:

$$R_1 \subseteq \pi_A R_2.$$  

$R_1$ is strictly contained in $R_2$, denoted $R_1 < R_2$, if $R_1 \leq R_2$ and $R_1 \neq R_2$.

The relation-containment order is useful to define general algebraic properties relevant for query optimization. We state these properties here for general functions on relations; later, we use them to examine familiar relational operators.

**Definition.** We say that a function $F$ on relations accepts a reduced argument $i$ if for some relation $R_i$ in the domain of $F$:

$$F(R_1, \ldots, R_i, \ldots, R_n) = F(R_1, \ldots, R_i', \ldots, R_n), \text{ for some } R_i' < R_i.$$  

Some systems, like SQL, allow duplicate tuples in relations, and projection does not remove duplicates. Our analysis of argument reduction does not consider duplicates, for which the notion needs to be revised and may not be applicable in many cases.

An important special case is when we can find a reducer $G_i$ for the arguments of the function. That is,

- for all $R_i$ in the domain, $G_i(R_i) \leq R_i$ and $F(R_1, \ldots, R_i, \ldots, R_n) = F(R_1, \ldots, G_i(R_i), \ldots, R_n)$; and
- for some $R_i$ in the domain, $G_i(R_i) < R_i$.  

41
For such functions, query optimizers will consider the application of the function reducer \( G_i \) before applying function \( F \), depending on the estimated evaluation cost. This strategy is applied in the semijoin-reduction technique used in distributed databases [Bern81, Bern83, Aper83].

Another property is that of argument preservation, defined as follows:

**Definition.** We say that a function \( F \) on relations preserves argument \( i \) if for all values of \( R_i \) in the domain,

\[
R_i \leq F(R_1, \ldots, R_i, \ldots, R_n).
\]

An important special case is when we can find an inverse \( F_i^{-1} \) for \( F \). That is, \( R_i = F_i^{-1}(F(R_1, \ldots, R_i, \ldots, R_n)) \). Clearly, a function that preserves argument \( i \) cannot accept a reduced argument \( i \).

For the composition of functions \( F_1, F_2 \), where the first preserves its arguments and the second accepts reduced arguments, an effective optimization strategy consists of finding a reducer \( G_2 \) of \( F_2 \) that is also an inverse of \( F_1 \), for some argument. If such \( G_2 \) exists, we can avoid computing \( F_1 \). That is, if reducer \( G_2 \) is the inverse of \( F_2 \) for argument \( k \), then

\[
F_2(R_{i_1}, \ldots, F_1(R_{j_1}, \ldots, R_{j_k}, \ldots, R_{j_m}), \ldots, R_{i_s}) = F_2(R_{i_1}, \ldots, R_{j_k}, \ldots, R_{i_s}).
\]

### 4.1.2 Properties of the basic relational operators

From a set of basic relational operators, others can be defined. A minimal generating set is not unique [Pare89], but five basic operators commonly used are selection, projection, cartesian product, union, and set difference [Ullm82].

Actually, for a given relation \( R_2 \), the set difference \( R_1 - R_2 \) can be expressed as a selection \( \sigma_p R_1 \), for some \( p \) that depends on \( R_2 \); namely, if the extension of \( R_2 \) is the set of tuples \( \{u_1, u_2, \ldots, u_n\} \),

\[
p(t) := \bigvee (t[sch(R_1)] = u_1) \vee (t[sch(R_1)] = u_2) \vee \cdots \vee (t[sch(R_1)] = u_n).
\]

The scheme of \( p \) is \( sch(R_1) \). The same argument applies to relational semijoin and antijoin—they can be reformulated as selection. For this reason, all the algebraic properties of selection apply also to set difference, semijoins, and outerjoins.
We are then left with four basic operators: Selection and projection reduce their arguments; union and cartesian product preserve their arguments.

\[ \sigma_p R \subseteq R. \]
\[ \pi_A R \subseteq R. \]
\[ R_1 \subseteq R_1 \cup R_2. \]
\[ R_1 \leq R_1 \times R_2, \text{ if } R_2 \neq \emptyset. \]

Unary and binary operators complement each other in terms of argument reduction/preservation:

\[ R \subseteq \pi_{A_1} R \times \pi_{A_2} R, \text{ if } A_1 \cup A_2 = \text{sch}(R), \text{ and } A_1 \text{ is disjoint from } A_2. \]  
(4.1)
\[ R = \sigma_p (R \cup R_2), \text{ for some } p. \]  
(4.2)

We first examine basic properties of our operators, and then work out some identities for argument reduction and preservation.

**Elementary properties**

**Idempotence.** Successive applications of projection or selection do not produce further reduction. Union is also idempotent.

\[ \sigma_p \sigma_p R = \sigma_p R. \]  
(4.3)
\[ \pi_A \pi_A R = \pi_A R. \]  
(4.4)
\[ R \cup R = R. \]  
(4.5)

**Identity elements and zero.** There is no identity for cartesian product in relational algebra, but there is one for union. Also, unary operators can behave as identity functions.

\[ \sigma_{\text{true}} R = R. \]  
(4.6)
\[ \pi_{\text{sch}(R)} R = R. \]  \hspace{1cm} (4.7)

\[ \pi_A R = \emptyset, \text{ if } A \text{ is disjoint from } \text{sch}(R). \]  \hspace{1cm} (4.8)

\[ R \cup \emptyset = R. \]  \hspace{1cm} (4.9)

\[ R \times \emptyset = \emptyset. \]  \hspace{1cm} (4.10)

**Commutativity of unary operators.** The application of a sequence of reducers may be changed, according to the following properties:

\[ \sigma_{p_1}\sigma_{p_2} R = \sigma_{p_1 \wedge p_2} R = \sigma_{p_2\sigma_{p_1}} R. \]  \hspace{1cm} (4.11)

\[ \pi_{A_1}\pi_{A_2} R = \pi_{A_1 \cap A_2} R. \]  \hspace{1cm} (4.12)

\[ \pi_A \sigma_p R = \sigma_p \pi_A R, \text{ if } \text{sch}(p) \subseteq A. \]  \hspace{1cm} (4.13)

**Distributivity of unary over binary operators.** Reducers have a distributive behavior with respect to union. This abstracts a “locality principle” of reducers — semantically, they work on one tuple at a time.

\[ \sigma_p (R_1 \cup R_2) = \sigma_p R_1 \cup \sigma_p R_2. \]  \hspace{1cm} (4.14)

\[ \pi_A (R_1 \cup R_2) = \pi_A R_1 \cup \pi_A R_2. \]  \hspace{1cm} (4.15)

They behave as follows with cartesian product:

\[ \sigma_p (R_1 \times R_2) = \sigma_p R_1 \times R_2, \text{ if } \text{sch}(p) \subseteq \text{sch}(R_1). \]  \hspace{1cm} (4.16)

\[ \pi_A (R_1 \times R_2) = \pi_A R_1, \text{ if } A \subseteq \text{sch}(R_1), R_2 \neq \emptyset. \]  \hspace{1cm} (4.17)

\[ \pi_{A_1 \cup A_2} (R_1 \times R_2) = \pi_{A_1} R_1 \times \pi_{A_2} R_2, \text{ if } A_1 \subseteq \text{sch}(R_1) \land A_2 \subseteq \text{sch}(R_2). \]  \hspace{1cm} (4.18)

**Properties of binary operators.** Finally, these commutativity, associativity, and distributivity properties of cartesian product and union also follow from the definitions:

\[ R_1 \cup R_2 = R_2 \cup R_1. \]  \hspace{1cm} (4.19)
\[(R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3). \quad (4.20)\]
\[R_1 \times R_2 = R_2 \times R_1. \quad (4.21)\]
\[(R_1 \times R_2) \times R_3 = R_1 \times (R_2 \times R_3). \quad (4.22)\]
\[R_1 \times (R_2 \cup R_3) = (R_1 \times R_2) \cup (R_1 \times R_3). \quad (4.23)\]

**Monotonicity.** Monotonicity properties follow directly from the definition of basic operators, subject to some restrictions. Assume \(R'_1 \leq R_1\), and in particular \(R'_1 \cup R''_2 \subseteq \pi_{A1}R_1\).

Then
\[
\sigma_p R'_1 \leq \sigma_p R_1, \text{ if } \text{sch}(p) \subseteq A_1. \quad (4.24)
\]
\[
\pi_{A2} R'_1 \leq \pi_{A2} R_1. \quad (4.25)
\]
\[
R'_1 \cup R_2 \leq R_1 \cup R_2, \text{ if } A_1 \subseteq \text{sch}(R_1) \cap \text{sch}(R_2). \quad (4.26)
\]
\[
R'_1 \times R_2 \leq R_1 \times R_2. \quad (4.27)
\]

**Argument reduction**

**Selection and projection.** We start by looking at argument reduction for selection and projection. For \(F(R) := \sigma_p R\), we know that \(F(R) \leq R\). Consider a relation \(R'\) such that \(F(R) \leq R' \leq R\). Using monotonicity and idempotence of selection, we have:
\[
\sigma_p R \leq R' \leq R.
\]
\[
\sigma_p \sigma_p R \leq \sigma_p R' \leq \sigma_p R.
\]
\[
\sigma_p R \leq \sigma_p R' \leq \sigma_p R.
\]

so \(\sigma_p R = \sigma_p R'\). Then, \(F(R) = F(R')\) for any \(R'\) containing \(\sigma_p R\) and contained in \(R\).

Naturally, \(\sigma_p R\) is the minimal argument to compute \(F(R)\) since, for \(R'' \prec \sigma_p R, \sigma_p R'' \prec \sigma_p R\).

By similar arguments, if \(F(R) := \pi_A R\), then \(F(R) = F(R')\) for all \(R'\) such that \(\pi_A R \leq R' \leq R\).

When selection is followed by projection, we have \(F(R) := \pi_A \sigma_p(R)\). We can reduce the argument using projection, \(F(R) = F(\pi_{A \cup \text{sch}(p)}R)\), since

45
\[ \pi_A \sigma_p R = \pi_A \sigma_p \pi_A \cup \text{sch}(p) R. \]

When the argument of the selection or projection take the form of an outerjoin we can work out specific reductions. If \( A \subseteq \text{sch}(R_1) \), then

\[ \pi_A(R_1 \bowtie R_2) = \pi_A(R_1 \bowtie P R_2) = \pi_A R_1. \] (4.28)

For the case of selection we first need to define a special predicate.

**Definition.** Given a set of attributes \( A \), the predicate *is-defined*, denoted \( \text{def}_A \), is true for tuples that are defined for some attributes in \( A \); that is

\[ \text{def}_A(t) := (A \cap \text{sch}(t) \neq \emptyset). \]

And now, we have

\[ \sigma_{\text{def}_A \text{sch}(R_2)}(R_1 \bowtie P R_2) = R_1 \bowtie P R_2. \] (4.29)

\[ \sigma_{\text{def}_A \text{sch}(R_2)}(R_1 \bowtie P R_2) = R_1 \bowtie P R_2. \] (4.30)

**Join and outerjoin.** It is known that join arguments can be reduced by means of semi-joins. That is

\[ R_1 \bowtie P R_2 = (R_1 \bowtie P R_2) \bowtie P R_2. \] (4.31)

The reduction can also be applied on one-sided outerjoins

\[ R_1 \bowtie P R_2 = R_1 \bowtie P (R_2 \bowtie P R_1). \] (4.32)
Argument preservation

Outerjoin. Outerjoin preserves one or both of its arguments, which can actually be recovered by projection. For \( F(R_1) := R_1 \xrightarrow{p} R_2, R_1 \leq F(R_1) \), since

\[
\pi_{\text{sch}(R_1)}(R_1 \xrightarrow{p} R_2) = R_1. \tag{4.33}
\]

For two-sided outerjoin, if \( F(R_1) := R_1 \xrightarrow{p} R_2 \) then also \( R_1 \leq F(R_1) \), since

\[
\pi_{\text{sch}(R_1)}(R_1 \xrightarrow{p} R_2) = R_1. \tag{4.34}
\]

Actually, identity 4.28 is the result of combining argument reduction for projection with argument preservation of join.

4.2 Preserving join expressions

The simplest way to construct a function that preserves its arguments is using union. Clearly, \( R_i \subseteq (R_1 \cup \cdots \cup R_n) \), for \( i = 1, \ldots, n \). The union \((R_1 \bowtie R_2 \cup R_1)\) preserves both \( R_1 \bowtie R_2 \) and \( R_1 \), and it is very similar to the outerjoin. The difference is that outerjoin eliminates from \( R_1 \) the information already present in \( R_1 \bowtie R_2 \), but we have

\[
R_1 \bowtie R_2 \leq R_1 \xrightarrow{p} R_2; \\
R_1 \leq R_1 \xrightarrow{p} R_2.
\]

We now define a function that follows the outerjoin preservation strategy. It takes the union of its arguments, reduced by eliminating the information contained in other arguments.

Definition. The nonredundant preserved relation of a collection of relations \( R_1, \ldots, R_n \) is defined as follows:

\[
\text{PresR}(R_1, \ldots, R_n) := R'_1 \cup \cdots \cup R'_n,
\]

where

\[
R'_i = R_i - \pi_{\text{sch}(R_i)} \left( \bigcup_{j \neq i} R_j \right).
\]
The function PresR() resembles symmetric difference. Since $R_1$ has no tuples in common with $R_1 \bowtie R_2$, the function PresR() serves to express outerjoins as follows:

\[
R_1 \xrightarrow{p} R_2 = \text{PresR}(R_1 \bowtie R_2, R_1).
\]

\[
R_1 \xrightarrow{p} R_2 = \text{PresR}(R_1 \bowtie R_2, R_1, R_2).
\]

**Preserve DAG.** The function PresR() does not preserve its arguments in all cases. For instance, PresR($R, R$) = $\emptyset$. It does preserve its arguments when they have a certain structure. The basic intuition is that as relations grow “horizontally,” their projection on a set of attributes diminishes “vertically;” that is, if $\text{sch}(R_1) \subseteq \text{sch}(R_2)$, then $\pi_{\text{sch}(R_1)} R_2 \subseteq R_1$.

**Definition.** A collection of relations $R_1, \ldots, R_n$ define a preserve directed-acyclic-graph (DAG), when the following holds

- If $i \neq j$, then $\text{sch}(R_i) \neq \text{sch}(R_j)$
- There is a relation, say $R_r$, called the root, such that $\text{sch}(R_i) \subseteq \text{sch}(R_r)$, for $i = 1, \ldots, n$.
- If $\text{sch}(R_i) \subseteq \text{sch}(R_j)$, then $\pi_{\text{sch}(R_i)} R_j \subseteq R_i$. We call $R_j$ an ancestor of $R_i$.
- If $\text{sch}(R_i) \nsubseteq \text{sch}(R_j)$, then $\pi_{\text{sch}(R_i)} R_j \cap R_i = \emptyset$.

Figure 4.1 shows three examples of preserve DAGs. In all three, the root is $R_1 \bowtie R_2 \bowtie R_3$. We omit the join predicates for readability, but joins in the ancestors must apply all predicates in their descendants. In formulation of outerjoins using PresR(), the arguments define a preserve DAG.
Preservation properties. When we restrict our attention to relations defining preserve DAGs, we find some useful properties of $\text{PresR}()$. First, each of the arguments is contained in the result; that is, if $R_1, \ldots, R_n$ define a preserve DAG,

$$R_i \subseteq \pi_{\text{sch}(R_i)}\text{PresR}(R_1, \ldots, R_n).$$ \hfill (4.37)

The projection of $\text{PresR}(R_1, \ldots, R_n)$ contains relation $R_i$ but in general it may contain some other elements — they come from descendants of $R_i$ in the preserve DAG, which are also contained in $\text{PresR}(R_1, \ldots, R_n)$. In case $R_1, \ldots, R_{n-1}$ are all ancestors of $R_n$, then

$$R_n = \pi_{\text{sch}(R_n)}\text{PresR}(R_1, \ldots, R_n).$$ \hfill (4.38)

Suppose $R_1, \ldots, R_n$ define a preserve DAG, and $A$ denotes a set of attributes such that $A \cap \text{sch}(R_i) \neq \emptyset$, for $i = 1, \ldots, k$; and $A \cap \text{sch}(R_i) = \emptyset$, for $i = k+1, \ldots, n$, then

$$\pi_A\text{PresR}(R_1, \ldots, R_n) = \pi_A\text{PresR}(R_1, \ldots, R_k).$$ \hfill (4.39)

**Example 4.1** Function $\text{PresR}()$ applied to relations that define a preserve DAG.

Figure 4.2 illustrates the result of $\text{PresR}()$ on relations that define a preserve DAG. It also shows the result of applying a projection to the result of $\text{PresR}()$. Using identities (4.39) and (4.38) on the relations of figure 4.2, we have

$$\pi_{\text{sch}(S)}\text{PresR}(RST, ST; S, T) = \pi_{\text{sch}(S)}\text{PresR}(RST, ST; S) = S.$$

**Associativity properties.** There is a property corresponding to associativity for the function $\text{PresR}()$, for some groupings of its arguments. Figure 4.3 illustrates the preserve DAG specified by the associativity conditions.

If the following properties hold

- $R_1, \ldots, R_n$ define a preserve DAG, with root $R_1$;
- $S_1, \ldots, S_m$ define a preserve DAG, with root $S_1$;
- $R_1, \ldots, R_n, S_1, \ldots, S_m$ define a preserve DAG;

49
<table>
<thead>
<tr>
<th>$R$</th>
<th>$S$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>f</td>
</tr>
<tr>
<td>c</td>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
<td>e</td>
</tr>
</tbody>
</table>

$ST = S^{F=H} T$

| $RST = R^{B=D} S^{F=H} T$ |
|-----------------------------|--|
| D | E | F | G | H | I |
| b | g | a | b | a | f |
| b | g | a | b | a | f |

$P = \text{PresR}(RST, ST, S, T)$

<table>
<thead>
<tr>
<th>$\pi_{(D,E,F,G,H,I)}^P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

Figure 4.2: Application of PresR() on relations defining a preserve DAG.

Figure 4.3: Preserve DAG for associativity conditions.
• there is no \( i, j \) such that \( S_i \) is ancestor of \( R_j \); and

• if for some \( j \) \( \text{sch}(S_1) \cap \text{sch}(R_j) = A \neq \emptyset \), then there is an \( S_k \) such that \( \text{sch}(S_k) = A \)

—condition not captured by figure 4.3:

\[
\text{PresR}(\text{PresR}(R_1, \ldots, R_n), \text{PresR}(S_1, \ldots, S_m)) = \text{PresR}(R_1, \ldots, R_n, S_1, \ldots, S_m). \tag{4.40}
\]

**Example 4.2** **Associativity of PresR().**

Relations \( R_1 \sqsupset R_2 \sqtriangleright R_3, R_1 \sqsupset R_2, R_2 \sqtriangleright R_3, R_2 \) define the preserve DAG of figure 4.1 (b). Equation 4.40 can be applied as follows:

\[
\text{PresR}(R_1 \sqsupset R_2 \sqtriangleright R_3, R_1 \sqsupset R_2, R_2 \sqtriangleright R_3, R_2)
\]

\[
= \text{PresR} (\text{PresR}(R_1 \sqsupset R_2 \sqtriangleright R_3, R_1 \sqsupset R_2), \text{PresR}(R_2 \sqtriangleright R_3, R_2))
\]

\[
= \text{PresR} (\text{PresR}(R_1 \sqsupset R_2 \sqtriangleright R_3, R_2 \sqtriangleright R_3), \text{PresR}(R_1 \sqsupset R_2, R_2)).
\]

Associativity of PresR() is an important property. We can convert expressions containing outerjoins into a function \( \text{PresR}(R_1, \ldots, R_n) \), using identities (4.35), (4.36), and then the associativity property we just showed. The \( \text{PresR}(R_1, \ldots, R_n) \) obtained in this process plays the role of a normal form for outerjoin expressions. The strategy used in appendix A to prove associative identities is to obtain the same PresR() function from both the left and right hand side of identities.

### 4.3 Associativity identities of join-like operators

We first treat the application of selections on the result of join and outerjoin. The identities also cover semijoins and antijoins, since they behave as selections.

Assume that \( \text{sch}(p_1) \subseteq \text{sch}(R_1) \).

\[
\sigma_{p_1}(R_1 \sqsupset R_2) = R_1 \sqsupset (\sigma_{p_1} R_1) \sqsupset R_2. \tag{4.41}
\]

\[
\sigma_{p_1}(R_1 \sqsupset R_2) = (\sigma_{p_1} R_1) \sqsupset R_2. \tag{4.42}
\]

51
\[ \sigma_{p_j}(R_1 \uparrow \downarrow R_2) = (\sigma_{p_j} R_1) \downarrow \downarrow R_2. \] 

In the following, call \( p_{ij} \) a predicate such that \( \text{sch}(p) \subseteq (\text{sch}(R_i) \cup \text{sch}(R_j)). \)

\[ (R_1 \uparrow \downarrow R_2) \uparrow \downarrow R_3 = R_1 \uparrow \downarrow (R_2 \uparrow \downarrow R_3). \] \hspace{1cm} (4.44)

\[ (R_1 \uparrow \downarrow R_2) \uparrow \downarrow R_3 = R_1 \uparrow \downarrow (R_2 \uparrow \downarrow R_3). \] \hspace{1cm} (4.45)

\[ (R_1 \arrow R_2) \arrow R_3 = R_1 \arrow (R_2 \arrow R_3), \] \hspace{1cm} if \( p^{23} \) is strong with respect to \( \text{sch}(R_2). \)

\[ (R_1 \arrow R_2) \arrow R_3 = R_1 \arrow (R_2 \arrow R_3). \] \hspace{1cm} (4.46)

\[ (R_1 \arrow R_2) \arrow R_3 = R_1 \arrow R_2 \arrow R_3, \] \hspace{1cm} if \( p^{23} \) is strong with respect to \( \text{sch}(R_2). \)

\[ (R_1 \arrow R_2) \arrow R_3 = (R_1 \arrow R_2) \arrow R_3, \] \hspace{1cm} if \( p^{23} \) is strong with respect to \( \text{sch}(R_2). \)

\[ (R_1 \arrow R_2) \leftrightarrow R_3 = R_1 \leftrightarrow (R_2 \leftrightarrow R_3), \] \hspace{1cm} if \( p^{23} \) and \( p^{12} \) are strong with respect to \( \text{sch}(R_2). \)

\[ (R_1 \arrow R_2) \leftrightarrow R_3 = R_1 \leftrightarrow (R_2 \leftrightarrow R_3), \] \hspace{1cm} if \( p^{23} \) is strong with respect to \( \text{sch}(R_2). \)

Predicate strongness is necessary where indicated for the above associative identities. The following example shows how non-strong predicates affect the associativity of outerjoins.
Example 4.3 Predicate strongness is necessary.

Consider the relations $A$ with one tuple $\{(A_1, a)\}$, $B$ with one tuple $\{(B_1, b)\}$, and $C$ with one tuple $\{(C_1, c)\}$. Let $p^{ab}(t)$ denote $(t.A_1 = t.B_1)$, and $p^{bc}(t)$ denote $(B_2 \notin \text{sch}(t) \lor t.B_2 = t.C_1)$. Then

$$A \rightarrow (B \rightleftharpoons C) \text{ yields tuple } \{(A_1, a)\}, \text{ but }$$

$$(A \rightarrow B) \rightleftharpoons C \text{ yields tuple } \{(A_1, a), (C_1, c)\}.$$

4.3.1 Generalized outerjoin

The associative identities above do not allow all join/outerjoin expressions to be reassociated. In particular, $R_1 \overset{p_{12}}{\rightarrow} (R_2 \overset{p_{23}}{\rightleftharpoons} R_3) \neq (R_1 \overset{p_{12}}{\rightarrow} R_2) \overset{p_{23}}{\rightleftharpoons} R_3$, since relation $R_1$ is not necessarily preserved in the right hand side. This section introduces a new operator that can be used to reorder join and outerjoin, in the general case.

To alter the processing order in the example above we need the generalized outerjoin operator (GOJ). GOJ was first defined in [Rose90], repairing a small error in the generalized join from [Daya87]. GOJ preserves a projection of one of its operands, much like outerjoin preserves a complete relation. For some cases, we need to preserve several projections of the arguments [Gali92].

**Definition.** The generalized outerjoin of relations $R_1, R_2$, with disjoint sets of attributes $A_{11}, \ldots, A_{1n}, A_{21}, \ldots, A_{2m}$, such that $A_{1i} \subseteq \text{sch}(R_1)$, $A_{2i} \subseteq \text{sch}(R_2)$ is:

$$R_1 \overset{p}{\text{GOJ}}[A_{11}, \ldots, A_{1n}, A_{21}, \ldots, A_{2m}] R_2 := \text{PresR}(R_1 \overset{p}{\rightleftharpoons} R_2, \pi_{A_{11}} R_1, \ldots, \pi_{A_{1n}} R_1, \pi_{A_{21}} R_2, \ldots, \pi_{A_{2m}} R_2). \quad (4.52)$$

**Example 4.4 Generalized outerjoin.**

Figure 4.4 shows relations $R, S$, an example of outerjoin and one of generalized outerjoin.

If there is only one set of attributes to preserve, say $A \subseteq \text{sch}(R_1)$, GOJ can be implemented by a modification to one-sided outerjoin algorithms, when relation $R_1$ is clustered.
**Figure 4.4: Generalized outerjoin.**

<table>
<thead>
<tr>
<th>R</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>b</td>
<td>g</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>b</td>
<td>g</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>f</td>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>g</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ R \overset{B=D}{\bowtie} S \]

\[ R \text{ GOJ}[B = D, \{A\}] S \]
on attributes $A$. When multiple attributes are preserved, auxiliary files and access paths need to be created. Algorithms for two sided outerjoin face similar problems, and the same strategies should be applicable in the implementation of both.

Using GOJ we can reorder the remaining join/outerjoin cases:

$$R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\leftrightarrow} R_3) = (R_1 \overset{p^{12}}{\rightarrow} R_2) \text{ GOJ}\[p^{23}, \text{sch}(R_1)] R_3, \quad (4.53)$$

if $p^{23}$ is strong with respect to $\text{sch}(R_2)$.

$$R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\leftrightarrow} R_3) = (R_1 \overset{p^{12}}{\leftarrow} R_2) \text{ GOJ}\[p^{23}, \text{sch}(R_1)] R_3, \quad (4.54)$$

if $p^{23}$ is strong with respect to $\text{sch}(R_2)$.

$$R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\leftrightarrow} R_3) = (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \text{ GOJ}\[p^{23}, \text{sch}(R_1), \text{sch}(R_3)] R_3, \quad (4.55)$$

if $p^{23}$ and $p^{12}$ are strong with respect to $\text{sch}(R_2)$.

Finally, if $p^{23}$ is strong with respect to $\text{sch}(R_2)$ and $p^{12}$ is strong with respect to a set of attributes $A_s$, and $A_s$ is disjoint with $A_{21}, \ldots, A_{2n}$, we have

$$R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \text{ GOJ}\[p^{23}, A_{21}, \ldots, A_{2n}, A_{31}, \ldots, A_{3m}] R_3) = (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \text{ GOJ}\[p^{23}, \text{sch}(R_1), A_{21}, \ldots, A_{2n}, A_{31}, \ldots, A_{3m}] R_3. \quad (4.56)$$
Chapter 5

Elements of a theory of reordering

When several files have to be consulted to evaluate a query, the order in which such files are considered makes a big performance difference, since the size of intermediate results may be very different. This is recognized and exploited in typical database query optimizers whenever files are combined using joins. Multiple orders of evaluation of those joins are generated by the optimizer, and then one is chosen, depending on its expected evaluation cost. Generating alternatives is commonly called the enumeration of join orders, and we call each of those alternatives a reordering of the original query.

Associative properties of binary operators are the basis on which different reorderings can be obtained, but the applicability of those properties depends on the specific form of match predicates used on the operators —see identities 4.44 through 4.51. Given a query with several joins, we need to examine what relations are being referenced by each predicate to determine what are the possible evaluation orders, and also what predicates can be applied at each step. Query graphs are used to represent information about the relations referenced by each predicate, as shown in section 2.3.

The problem of finding valid reorderings of a query is complicated significantly if we are using different binary operators, and more so when some associative identities not only change the position of predicates but even the operator used —see identities 4.53 through 4.55. Given a desired order of processing, it is not immediate from a collection of associative identities whether or not relations can be processed in the desired order, and what is the operator to use at each step.
In this chapter, we study the problem of valid query reordering given some associative identities.

## 5.1 Trees, graphs, and query reordering

**Operator trees.** In this chapter we consider relational expressions formed by binary, join-like operators. We will represent these expressions as operator trees, where inner nodes are labeled by a binary operator with a match predicate, and leaves are labeled by relations.

An operator tree with left subtree \( T_1 \), right subtree \( T_2 \), and root \( \circ \), where \( p \) is a predicate, is denoted \(( T_1 \circ T_2 )\). In join-like operators, the match predicate references attributes from a relation in \( T_1 \) and one in \( T_2 \)—if we compute the join of \( R \) with itself, we need to rename the attributes to be able to distinguish each instance, thus creating two “different” relations.

We impose the following restrictions on operator trees:

- Leaves of the tree are labeled by different relations.
- For any subtree \(( T_1 \overset{p_1 \wedge \cdots \wedge p_n}{\circ} T_2 )\), for \( i = 1, \ldots, n \) there are relations \( R_{i_1}, R_{i_2} \) such that \( \text{sch}(p_i) \subseteq (\text{sch}(R_{i_1}) \cup \text{sch}(R_{i_2})) \), where \( R_{i_1} \in \text{leaves}(T_1) \) and \( R_{i_2} \in \text{leaves}(T_2) \).

The evaluation of an operator tree \( T \) is denoted \( \text{eval}(T) \), and it is computed bottom-up.

**Query graphs.** We now extend the definition of query graph given in chapter 2. For an operator tree \( T \), a *query graph* \( G = (V, E) = \text{graph}(T) \) is constructed as follows:

- The set of nodes \( V = \text{leaves}(T) \).
- For each inner node \( \overset{p}{\circ} \), where \( p = R_{i_1}^{R_{i_1}} \wedge \cdots \wedge R_{i_n}^{R_{i_n}} \), for each \( R_{i_1}^{R_{i_1}}, R_{i_2}^{R_{i_2}} \) there is an edge \(( R_{j_1}^{R_{j_1}}, R_{j_2}^{R_{j_2}} ) \in E \) labeled by \( p_i \); \( p_i \) references attributes in relations \( R_{i_1}, R_{i_2} \) and only in those two relations.

For the operator trees we are considering, query graphs are connected. Figure 5.1 shows an operator tree and its corresponding query graph; predicate \( p_i^{ij} \) references attributes of relations \( R_i, R_j \).

The two representations of queries—trees and graphs—emphasize different aspects of the query. An operator tree specifies unambiguously the inputs to each operation, and
can be evaluated. A query graph represents a collection of relations and the predicates that connect them, but it does not possess an evaluation rule. Operator trees and their relationship to query graphs can be used to define a suitable notion of “associativity” for join-like operators.

**Association trees.** To specify an order of processing of operands we use association trees. Given a query graph $G = (V, E)$, we define an association tree $T$ for $G$ as a binary tree such that:

- **leaves($T$) = $V$, and no relation appears in more than one leaf.**
- **For any subtree $T'$ of $T$, $G|_{\text{leaves}(T')}$ is connected$^1$.**

The set of association trees for a given graph $G$ is denoted $\text{assoc}(G)$. An association tree $T$ with left subtree $T_1$ and right subtree $T_2$ is denoted $T = (T_1 \odot T_2)$. An association tree specifies an order in which relations will be processed, but does not give operators to apply at each step; consequently, it cannot be evaluated.

Figure 5.2 shows all association trees of the query graph in figure 5.1, up to symmetric forms — e.g., $(T_1 \odot T_2)$ and $(T_2 \odot T_1)$.

---

$^1$Given a graph $G = (V, E)$ and $V' \subseteq V$, we denote the induced subgraph as $G|_{V'} = (V', E')$, where $E' = \{(u, v) \mid (u, v) \in E, u \in V', v \in V'\}$. 

---

58
Figure 5.2: Association trees of a query graph.
Association trees can also be obtained by discarding the inner node labels of operator trees. If we obtain the association tree \( T \) by discarding labels in operator tree \( T' \), then such \( T \in \text{assoc}(\text{graph}(T')) \), and we say that \( T = \text{assoc}(T') \).

**Reordering of join-like operators.** The general scheme to explore different orders in which to process relations for a given expression \( T \) is

- Generate candidate association trees \( T_i \in \text{assoc}(\text{graph}(T)) \).
- For each \( T_i \), place an operator at each node to obtain an operator tree \( T'_i \) such that \( \text{eval}(T'_i) = \text{eval}(T) \).
- Select a \( T'_i \) with the least estimated cost of evaluation.

If operator trees \( T'_i \) exist for all \( T_i \) above, we say that \( T \) is *fully reorderable*. This concept appears to adapt appropriately the idea of associativity to arbitrary expressions of join-like operators.

### 5.2 A complete set of tree rewrite rules

In the previous section we showed how a query graph is constructed from an operator tree, and how different association trees can be derived from the query graph. We now examine transformations between operator trees.

**Rewrite rules.** Given an operator tree \( T \) and an association tree \( T_i \in \text{graph}(T) \), we can apply tree rewrite rules on \( T \) to obtain a \( T'_i \) such that \( \text{assoc}(T'_i) = T_i \). \( T_i \) is viewed as a *goal association* describing an order of processing to compute \( T \).

Figure 5.3 shows how to transform the operator tree \( T \) of figure 5.1 (a) into another operator tree, using as a goal the association tree of figure 5.2 (c).

We limit the form of rewrite rules to the following two, called *reversal* and *reassociation* rules, respectively. These rules are not meant for the association of cartesian products, so \( \text{sch}(p^{T_1 T_2}) \) must actually intersect the scheme of leaves in \( T_1 \) and also in \( T_2 \); \( \text{sch}(p^{T_2 T_3}) \) intersects the scheme of relations in \( T_2 \) and also in \( T_3 \).

\[
(T_1 \circ_1 T_2) \xrightarrow{p} (T_2 \circ_1' T_1), \text{ where } \circ_1' = f_0(\circ_1)
\]
Figure 5.3: Reordering with tree-rewrite rules.
(T_1^{p_1 T_2} \land p_1^{T_3}) \circ (T_2^{p_2 T_3} \circ T_3) \sim ((T_1^{p_1 T_2} \land p_1^{T_3}) \circ (T_2^{p_2 T_3} \circ T_3),
\text{where } \circ'_1 = f_1(\circ_1, \circ_2, p_1^{T_3}), \circ'_2 = f_2(\circ_1, \circ_2, p_1^{T_3})

\) for some functions \( f_0, f_1, f_2 \). When \( T' \) is obtained from \( T \) by applying one of the above tree rewrite rules, we say that \( T \sim T' \). The transitive closure of relation \( \sim \) is denoted \( \sim^* \).

When the functions \( f_0, f_1, f_2 \) are such that \( T \sim T' \) implies \( \text{eval}(T) = \text{eval}(T') \), we call the rewrite rules \textit{result-preserving}.

For result-preserving reversal, the function \( f_0 \) gives the “symmetric form” of each operator, depending on its commutativity properties; for instance \( f_0(\rightarrow) = \leftarrow \), and \( f_0(\rightarrow\leftarrow) = \leftarrow \rightarrow \).

For result-preserving reassociation, the functions \( f_1 \) and \( f_2 \) depend on the algebraic properties of the operators \( \circ_1, \circ_2 \). For instance, from identity \( 4.45 \), we should chose \( f_1(\leftarrow, \rightarrow) = \leftarrow \rightarrow \) and \( f_2(\leftarrow, \rightarrow) = \rightarrow \).

\textbf{Generation of arbitrary associations.} The reversal and reassociation tree rewrite rules are complete, in the sense that for any association tree \( T' \in \text{assoc}(\text{graph}(T)) \), \( T \sim^* T'' \) such that \( T'' = \text{assoc}(T'') \).

For a given a goal association \( T' \), the strategy to construct \( T'' \) is shown by the pseudocode below. This pseudocode is given only as an outline of the following proof that tree rewrite rules are complete for reordering operator trees. \( T'' = \text{reassociate}(T, T') \). For a set of leaves \( L \) of \( T \) whose induced subgraph in \( \text{graph}(T) \) is connected, \( \text{sortleaves}(T, V) \) returns an operator tree \( T_n = (T_n^p \circ T_n^o) \) such that \( T \sim^* T_n \) and leaves\((T_n^o) = V \).

\textbf{reassociate}\((T, (T'_1, T'_2))\)

1. if \( T \) is a one node tree, return \( T 
2. \ (T_1 \circ T_2) = \text{sortleaves}(T, \text{leaves}(T'_2))
3. return (\text{reassociate}(T_1, T'_1) \circ \text{reassociate}(T_2, T'_2))

\textbf{sortleaves}\((T, L)\)

1. Set \( T_0 = T \), \( V = L \), and \( n = \text{size}(L) \)
2. For \( i = 1 \) to \( n \)
2.1. Pick \( v \) from \( V \), depending on \( V \) and \( L \)
2.2. Remove \( v \) from \( V \)
2.3. Get $T_i = (T_{i_1} \odot T_{i_2})$ from $T_{i-1}$, such that $\text{leaves}(T_{i_2}) \cup V = L$

3. return $T_n$

Observation. Note that $\text{graph}(T)$ depends only on the predicates of $T$. Since the set of predicates is not changed by our tree rewrite rules, if $T \sim T'$ then $\text{graph}(T) = \text{graph}(T')$. Also, note that if $T'$ is a subtree of $T$ and $G = \text{graph}(T)$, then $\text{graph}(T') = G_{\text{leaves}(T')}$. 

Lemma 5.1. If leaf $v$ is at level $n > 1$ in an operator tree $T$, and removing $v$ from $\text{graph}(T)$ leaves a connected graph, then there is a $T'$ such that $T \sim T'$ and $v$ is at level $n - 1$ in $T'$.

Proof. Find the subtree $T_s$ of $T$ in which $v$ appears at level 2. Apply reversal rules on $T_s$, if necessary, to put it in the form $(T_1 \odot_1 (T_2 \odot_2 v))$. If there is an edge in $E$ connecting a leaf of $T_1$ with one in $T_2$ then the application of one reassociation rule moves $v$ one level closer to the root.

Otherwise, there must be edges connecting $v$ with $\text{leaves}(T_1)$ and also connecting $v$ with $\text{leaves}(T_2)$. But since removing $v$ from $G$ leaves a connected graph, there must be another path connecting some leaf of $T_1$ with one in $T_2$ that does not include $v$. Choose a node $w$ of this alternate path, that is not a leaf of $T_1$ or $T_2$, and is connected to a leaf of $T_2$ by means of an edge $e$, as shown in figure 5.4.

Figure 5.5 shows a sequence of rule applications that move $v$ one level closer to the root. To simplify the presentation, instead of writing complete operators we only mark that one of its conjuncts is the label of edge $e$ and use $\odot$ as an anonymous variable. We also annotate subtree $T_k$ by indicating that it contains leaf $w$. 

Figure 5.4: Nodes $v$ and $w$ in the proof of lemma 5.1.
Figure 5.5: Moving $v$ one level closer to the root in the proof of lemma 5.1.
Lemma 5.2. If $V_2 \subseteq V$ for $G = (V, E)$, and both $G|_{V_2}$, $G|_{V\setminus V_2}$ are connected, then there is a node $v \in V_2$ such that removing $v$ from $G$ leaves a connected graph and removing $v$ from $G|_{V_2}$ also leaves a connected graph.

Proof. Call $G_2 = G|_{V_2}$. Note that for any connected graph $H$, the last node marked in a depth-first search can be removed from $H$ and leave a connected graph.

If $V_2$ has a single element then $V_2 = \{v\}$. Otherwise, select a node $u \in V_2$ such that there is an edge $(u, w) \in E$, but $w \not\in V_2$. Use depth-first search on $G_2$, starting at node $u$, to find a node $v \in V_2$ whose removal from $G_2$ leaves the graph connected. Removing $v$ from $G$ also leaves a connected graph, because all nodes can be reached from $u$.

Lemma 5.3. If $V_2 \subseteq V$ for $G = (V, E) = \text{graph}(T)$, and both $G|_{V_2}$, $G|_{V\setminus V_2}$ are connected, then $T \xrightarrow{\ast} (T_1 \circ T_2)$, where $\text{leaves}(T_2) = V_2$.

Proof. By induction on the size of $V_2$. Assume $G_2 = G|_{V_2}$ and $G_1 = G|_{V\setminus V_2}$.

Base. If $V_2 = \{v\}$, then removing $v$ from $G$ leaves $G_1$, which is a connected graph.

Then, by lemma 5.1, $T \xrightarrow{\ast} (T_1 \circ v)$.

Induction. By lemma 5.2, there is a node $v \in V_2$ such that removing $v$ from $G$ leaves a connected graph $G'$, and removing $v$ from $G_2$ leaves a connected graph $G'_2$.

By induction hypothesis, $T \xrightarrow{\ast} (T_1 \circ_i v)$.

Also by induction hypothesis, $T_i \xrightarrow{\ast} (T_1 \circ_j T_j)$, where $\text{leaves}(T_j) = V_2 - \{v\}$.

Since there is an edge connecting $v$ to some node in $V_2$, we can apply a reassociation rewrite rule as the last step in the following series of rule applications:

$$T \xrightarrow{\ast} (T_i \circ_i v) \xrightarrow{\ast} ((T_1 \circ_j T_j) \circ_i v) \xrightarrow{\ast} (T_1 \circ_k (T_j \circ_l v)),$$

where $\text{leaves}(T_j \circ_l v) = V_2$.

Theorem 5.1. If $T' \in \text{assoc}(G)$ for $G = \text{graph}(T)$, then there is an operator tree $T''$ such that and $T \xrightarrow{\ast} T''$ and $T' = \text{assoc}(T'')$.

Proof. By induction on the number of inner nodes of $T'$.

Base. If $T'$ has zero inner nodes, then $T'' = T$.
**Induction.** Suppose $T' = (T'_1, T'_2)$. Since $G_{\text{leaves}(T'_2)}$ is connected, by lemma 5.3, $T \xrightarrow{p_r} (T_1 \circ_r T_2)$, where leaves($T_2$) = leaves($T'_2$).

Then $T'_2 \in \text{assoc}(\text{graph}(T_2))$, and $T'_1 \in \text{assoc}(\text{graph}(T_1))$. By induction hypothesis, $T_1 \xrightarrow{p_r} T''_1$, $T_2 \xrightarrow{p_r} T''_2$, where $T''_1 = \text{assoc}(T'_1)$ and $T''_2 = \text{assoc}(T'_2)$.

Then $T \xrightarrow{p_r} T'' = (T''_1 \circ_r T''_2)$, and $T' = \text{assoc}(T'')$.

### 5.3 Conditions for reorderable expressions

We know that tree rewrite rules are sufficient to generate any reassociation of a given expression. This could be called a “syntactical” result, because it does not deal with the actual semantics of operators. The functions $f_0, f_1, f_2$ provide a “customization hook” to deal with the semantics of specific operators.

From the algebraic properties of operators, we can determine functions $f_0, f_1, f_2$ such that the tree rewrite rules are result-preserving for some operators. A class of reorderable expressions is determined by these functions.

**Theorem 5.2.** Given tree rewrite rules with functions $f_0, f_1, f_2$, a class $C$ of expressions is fully reorderable if

- for expressions in $C$, any applicable tree rewrite rule is result preserving; and
- $C$ is closed under tree rewrite rules.

**Proof.** Take an expression $T \in C$ and an association tree $T' \in \text{assoc}(\text{graph}(T))$.

By theorem 1, $T = T_0 \xrightarrow{p_r} T_1 \xrightarrow{p_r} \cdots \xrightarrow{p_r} T_n = T''$, such that $T' = \text{assoc}(T'')$.

Since $C$ is closed under tree rewrite rules, $T_i \in C$, for $i = 0, \ldots, n$. Since tree transforms are result preserving for expressions in $C$, $\text{eval}(T_i) = \text{eval}(T_{i+1})$. By transitivity, $\text{eval}(T) = \text{eval}(T'')$.

In the next chapter we apply the theory developed here to various classes of expressions involving joins and outerjoins. These classes have different restrictions on the pattern of operators allowed in operator trees. We will examine the set of result-preserving rewrite rules —and strategies for their systematic application—to make these classes closed.
Chapter 6

Outerjoin simplification, reordering, and implementation

In this chapter we combine the results of chapter 4 and 5 to find optimization techniques for queries containing joins and outerjoins.

First, we present a class of join/outerjoin queries that can be handled by minimal extensions on current optimizers. We will see how queries generated by some applications fall in this class. Next, we consider simplification rules for outerjoins. Depending on operations that will be applied to the outerjoin result, the operator can be replaced by a join or eliminated altogether. These simplification rules are powerful, and easy to describe and implement. Then, for reordering in the general case, we extend the join-order enumeration algorithm of conventional optimizers to handle joins and outerjoins. For every order of evaluation chosen by the enumerator, it needs to lookup in a table what operator will be used, instead of always using join, as it is currently done. Finally, we describe how the typical implementation algorithms for join can be modified to compute outerjoins.

As a notational convention in this section, if a tuple predicate $p$ is such that $\text{sch}(p) \subseteq (\text{sch}(R_i) \cup \text{sch}(R_j))$, we annotate it as $p^{ij}$.

The sections on simplification and implementation are applicable to arbitrary outerjoins, but the material on reordering is slightly restricted. The associative identities for join/outerjoin of chapter 4 cannot handle arbitrary tuple predicates. Some of them require that the predicate be strong with respect to some attributes, and others restrict the relations...
that may be referenced. The reordering results of this chapter are applicable on trees for which outerjoin predicates reference exactly two base relations, and join predicate conjuncts reference two base relations. Most outerjoin applications satisfy the restrictions we require for reordering. The complete reordering of these queries is a significant improvement over join-only reordering, because it appears naturally in most of the queries of the applications we reviewed in chapter 3. Reordering with arbitrary predicates seems to require a more sophisticated approach, and it may not work in all cases —see example 4.3

6.1 Freely-reorderable outerjoins

The associative identities for join and one sided outerjoin that do not change the operator after reassociation define a class of reorderable expressions. The corresponding tree rewrite rules will move predicates between operators when reordering, but operators remain unchanged. We use identities 4.44 through 4.47:

\[
\begin{align*}
(R_1 \bowtie R_2) \bowtie R_3 & = R_1 \bowtie (R_2 \bowtie R_3) \\
(R_1 \bowtie R_2) R_3 & = R_1 (R_2 R_3) \\
(R_1 \rightarrow R_2) R_3 & = R_1 (R_2 R_3) \\
(R_1 \leftarrow R_2) R_3 & = R_1 (R_2 R_3)
\end{align*}
\]

We use an annotated query graph to characterize a class of freely reorderable queries. The annotated query graph of \(T\) contains information not only about the predicate that connects two relations, but also the operator that used this predicate. If the conjunct \(p_{ij}\) is used in a join in \(T\), then there is an undirected edge \((R_i \rightarrow R_j)\) in graph(\(T\)). If \(p_{ij}\) is a predicate in an outerjoin \((T_1 \leftarrow T_2)\) and \(R_i \in \text{leaves}(T_1)\) then there is a directed edge \((R_i \rightarrow R_j)\) in graph(\(T\)). For two sided outerjoin we use bidirected edges. An operator tree and its annotated query graph is shown in figure 6.1.

**Theorem 6.1.** A join/1-outerjoin operator tree \(T\) is reorderable when the annotated graph(\(T\)) consists of a connected set of undirected edges, from which directed trees grow outward.
Proof. Call $C$ the set of operator trees such that if $T' \in C$ then $\text{graph}(T') = \text{graph}(T)$, for annotated graphs.

- Using reversal rewrite rules, convert subtrees of $T$ into the form $(T_1 \circ_1 (T_2 \circ_2 T_3))$.

For a query graph satisfying the restrictions of theorem 6.1, each subtree must have one of the following forms:

- $(T_1 \overset{p_{12}}{\circ_1} T_2 \overset{p_{23}}{\circ_2} T_3))$.
- $(T_1 \overset{p_{12}}{\circ_1} T_2) \overset{p_{23}}{\rightarrow} T_3))$.
- $(T_1 \overset{p_{12}}{\rightarrow} T_2 \overset{p_{23}}{\rightarrow} T_3))$.
- $(T_1 \overset{p_{12}}{\rightarrow} (T_2 \overset{p_{23}}{\rightarrow} T_3))$.

Then all applicable tree rewrite rules on $T$ are result preserving

- Since reassociation rules do not change operators, if $T \sim T_i$, then they have the same annotated query graph, and $T_i \in C$.

By theorem 5.2, $C$ is a class of reorderable expression.

Figure 5.1 shows a graph satisfying the conditions of theorem 6.1. An equivalent characterization of those graphs is:

- The graph is connected;

- no cycles contain directed edges;
Figure 6.2: Topology of the annotated graph of a freely-reorderable query.

- there is no path of the form $R_1 \rightarrow R_2 \rightarrow R_3$; and

- there is no path of the form $R_1 \rightarrow R_2 \leftarrow R_3$.

A query satisfying the conditions of theorem 6.1 is not only reorderable as defined in chapter 5; an appropriate operator tree can be generated by looking at the annotated query graph only. These queries are called *freely reorderable* or *freely associative*.

Information about the original operator tree is needed for other queries, since rewrite rules may change operators.

Freely-reorderable outerjoins go substantially beyond select/project/join queries, but impose relatively little burden on language syntax or optimization. Our theoretical foundation allows us to determine other freely-reorderable classes, based on a set of associative identities.

**Query language issues.** Languages that generate freely-reorderable queries do not require the user to indicate priority among join operations, and hence may present a simplified syntax. Actually, any specification language — e.g., graphical — from which the annotated graph can be constructed is appropriate as a query language. As an example, we now show that the SQL extensions for set- and entity-valued attributes described in section 3.2.2 generate freely-reorderable expressions.

In the outerjoin formulation of nest and unlink constructs, relations to the right of an outerjoin $R_1 \xrightarrow{p} R_2$ are those introduced by the attributes in the right side of operands *
and \( \rightarrow \). Each time a relation is obtained from a field, it is considered independent, i.e., a new relation. Therefore there is no way to create two edges directed into a given node — a cycle. Furthermore, these relations could not appear in the \textbf{Where} clause of the same query block, creating a join edge. Thus none of the forbidden subgraphs has arisen. Finally, note that the special outerjoin predicates are strong, as they are matching object identifiers and succeed only if the two objects exist. So, from Theorem 6.1, table references using \( \ast \) and \( \rightarrow \) are freely reorderable.

\section*{Query optimization issues}
For designers of query optimizers, freely-reorderable queries are much simpler than the general case. For language constructs that are known to generate only freely-reorderable outerjoins, the extension seems small, at least conceptually. Optimizers already implement a query graph and generate expression trees with different associations of the graph edges; now they must fill in join or else outerjoin, preserving the operator direction. There is no need to use generalized outerjoin, or perform a subtle analysis.

\section{Outerjoin simplification}
In chapter 4 we showed that outerjoins can sometimes be converted into joins in the context of particular expressions, as a case of argument reduction. This is an important query processing property because it reduces the size of intermediate results. In the case of outerjoin simplification, it also gives more freedom to choose implementation methods, e.g. in choosing the “outer-loop” operand.

There are basically three types of outerjoin simplification:

\textbf{Strong predicates.} If a later predicate will discard the null-padded tuples introduced by an outerjoin, then we can convert such outerjoin into a regular join without altering the result. For example, since the selection will reject tuples for which \( \text{Age} \) is undefined,

\[ \sigma_{\text{Age} > 50} (\text{DEPT} \rightarrow \text{EMPLOYEE}) = \sigma_{\text{Age} > 50} (\text{DEPT} \bowtie \text{EMP}). \]

\textbf{Projection on a preserved set.} If we are interested only in the projection of some attributes that are part of a relation being preserved and if duplicates are not significant,
there is no need to compute the outerjoin. For instance, to compute a list of department names:

\[ \pi_{\text{Dept-name}}(\text{DEPT} \rightarrow \text{EMPLOYEE}) = \pi_{\text{Dept-name}} \text{DEPT}. \]

**Integrity constraint.** If the database enforces, as a referential integrity constraint, that there is no department without employees, then the result of the outerjoin is the same as that of join.

\[ (\text{DEPT} \rightarrow \text{EMPLOYEE}) = (\text{DEPT} \bowtie \text{EMP}). \]

Simplification by strong predicates was developed in [Gali92]; simplification by projection, in [Chen90]. Simplification by integrity constraints is mentioned in [Rose90].

It seems unlikely that a user would write a join or outerjoin, and then project only some of the preserved attributes from one of the inputs. As shown in chapter 3, however, queries on base files are obtained after a number of mappings between query specifications. \((\text{DEPT} \rightarrow \text{EMPLOYEE})\) may well be a view that a user treats as a single virtual table.

Identities for outerjoin simplification by selection and projection are given in chapter 4. In this section we focus on simplification by strong predicates and the properties of the resulting expressions. The structure of simplified queries allows a systematic use of generalized outerjoin to reorder join/outerjoin expressions.

### 6.2.1 Strong predicate simplification

Recall that \(\text{def}_A(t)\) — introduced in page 46 — denotes the predicate that tests whether \(t\) is defined for at least one attribute in \(A\). We will use simplification identities 4.42 and 4.30:

\[
\sigma_{\text{def}_{\text{sch}(R_2)}}(R_1 \overset{p}{\rightarrow} R_2) = R_1 \overset{p}{\bowtie} R_2 \\
\sigma_{\text{def}_{\text{sch}(R_2)}}(R_1 \overset{p}{\rightarrow} R_2) = R_1 \overset{p}{\leftarrow} R_2
\]

We refer to the simplifications achieved by the \(\text{def}_A\) predicate as removing the arrow on \(A\).

To simplify a join/1-outerjoin/2-outerjoin query, we first add \(\text{def}_A\) as an explicit conjunct on any predicate that is strong with respect to the scheme \(A\) of some leaf relation.
This conjunct is then pushed down the operator tree as a selection, and used to simplify outerjoins. When no more simplifications are possible, the added $\text{def}_A$ conjuncts are removed from the tree. Identities for creating and pushing $\text{def}_A$ selections for join and outerjoin follow from the basic distributivity properties of selection. If $\text{sch}(p_1) \subseteq \text{sch}(R_1)$:

$$\sigma_{p_1}(R_1 \overset{p_2}{\bowtie} R_2) = R_1 \overset{p_1 \wedge p_2}{\bowtie} R_2 = (\sigma_{p_1} R_1) \overset{p_2}{\bowtie} R_2$$

$$\sigma_{p_1}(R_1 \overset{p_2}{\rightarrow} R_2) = (\sigma_{p_1} R_1) \overset{p_2}{\rightarrow} R_2$$

$$R_1 \overset{p_1 \wedge p_2}{\leftarrow} R_2 = (\sigma_{p_1} R_1) \overset{p_2}{\leftarrow} R_2$$

**Outerjoin Simplification Rule.** Suppose $T$ is an operator tree where join conjuncts and outerjoin predicates are strong with respect to the scheme of the relations they reference. Visit in preorder each node $\odot$ in $T$, and do the following:

- If $\odot = \overset{p_i \wedge p_j}{\bowtie}$, then remove arrows on $\text{sch}(R_i)$ on operators in the path from $\odot$ to $R_i$ in $T$; and remove arrows on $\text{sch}(R_j)$ on operators in the path from $\odot$ to $R_j$ in $T$.

- If $\odot = \overset{p_i \rightarrow}{\rightarrow}$, where $(T_l \overset{p_i \rightarrow}{\rightarrow} T_r)$ is a subtree of $T$ and $R_j \in \text{leaves}(T_r)$, then remove arrows on $\text{sch}(R_j)$ on operators below $\odot$ in the path from $\odot$ to $R_j$.

**Definition.** A tree is called *simple* when no arrow removals are possible.

Figure 6.3 shows the result of simplifying a query.

### 6.2.2 Properties of simple trees

The following properties hold on simple trees:

1. Application of a result-preserving rewrite on a simple operator tree yields a simple operator tree.

2. Subtrees of simple operator trees are simple.

3. Any simple operator tree $T$ can be transformed by a sequence of result-preserving rewrite rules into $T'$ such that:
Ancestors of 2-outerjoins in $T'$ are 2-outerjoins.

- Ancestors of 1-outerjoins in $T'$ are 1- and 2-outerjoins only.

4. If $T$ is simple, then a result-preserving rewrite can be applied to interchange any two adjacent joins, any two adjacent 1-outerjoins, and any two adjacent 2-outerjoins.

5. If $T$ is a simple operator tree, then removing a directed or bidirected edge from graph($T$) disconnects the graph. Also, all paths of the form $R_i \rightarrow \cdots \leftarrow R_j$ in graph($T$) include a bidirected edge.

Using property 4 above, a simple query can be evaluated in three phases: compute first all joins, then 1-outerjoins, and finally 2-outerjoins. Within each phase the order of computation is immaterial. It should be clear that this is not the whole space of evaluation orders, and it does not necessarily include the optimal cost strategy.

6.3 Reordering of join/1-outerjoin/2-outerjoin

We started this chapter by showing the class of join/outerjoin expressions that are freely reorderable. This section deals with reordering in the general case—subject only to the restriction that outerjoin predicates and join conjuncts reference exactly two base relations, and predicates are strong on the attributes of each relation they reference.
To reorder an expression, we first simplify it, using the simplification rule of section 6.2.1, and then apply the results presented in this section. Generalized outerjoin (GOJ) is needed to reorder, in the general case. The cost of implementing generalized outerjoin is justified by the savings it achieves for some reorderings, as is shown in the next example.

**Example 6.1 GOJ reorderings are useful**

Suppose we have relations $R_1, R_2, R_3$ with 3 tuples, 1 million tuples, and 1 million tuples, respectively. By identity 4.53, $R_1 \rightarrow (R_2 \bowtie R_3) = (R_1 \rightarrow R_2) \text{GOJ}[\text{sch}(R_1)] R_3$. If match predicates are equalities on keys, then the GOJ expression produces an intermediate result of cardinality 3, while the intermediate result generated by the other expression has cardinality 1 million. Even though GOJ is more complex than outerjoin, the overall query is evaluated more efficiently using the reordering with GOJ.

Once an association tree has been selected to evaluate a given join/outerjoin query, we need to select operators for the inner nodes of the tree. The operator at each step may be join, one- or two-sided outerjoin, or generalized outerjoin. We now give some definitions that will be useful to determine the operator at each step.

**Definition.** If $T$ is a simple operator tree and $e_0 = (R_i \rightarrow R_j)$ is a join edge in $\text{graph}(T)$, the closest conflicting outerjoin of $e_0$, denoted $\text{ccoj}(e_0)$, is an outerjoin edge $e_1 = (R_{k_1} \rightarrow R_{l_1})$ such that:

- There is a path $R_{k_1} \overset{e_1}{\rightarrow} \cdots \overset{e_0}{\rightarrow} R_i$ with no 2-outerjoin edge.
- For any other 1-outerjoin edge $e_2 = (R_{k_2} \rightarrow R_{l_2})$, if there is a path $R_{k_2} \rightarrow \cdots \rightarrow R_i$ with no 2-outerjoin edge, such path includes $e_1$.

The intuition behind this definition is that the outerjoin for $e_1 = \text{ccoj}(e_0)$ cannot be a descendant of the join for $e_0$ in the simple tree $T$ — the strong predicate in the join would have simplified the outerjoin into a join. If we want to reorder the query so that the outerjoin for $e_1$ is now a descendant of $e_0$, we need to use GOJ instead of join for $e_0$.

Property 5 of simple queries is violated if more than one outerjoin satisfies the definition of $\text{ccoj}(e_0)$, for a given join edge $e_0$ in a join/1-outerjoin simple query. So, the closest
conflicting outerjoin of a join edge, if it exists, is unique. Also, if removing a set of join edges \( e_0, \ldots, e_n \) from \( G \) leaves two connected graphs, then all of those join edges have the same closest conflicting outerjoin. Property 5 is violated otherwise.

The same idea is used to define the set of closest conflicting two-sided outerjoins.

**Definition.** If \( T \) is a simple operator tree and \( e_0 = (R_i \rightarrow R_j) \) is an outerjoin edge in graph(\( T \)), then the set of closest conflicting 2-outerojins for \( e_0 \) is defined as \( \text{cc2oj}(e_0) = \{ e_h \mid e_h = (R_{k_h} \leftrightarrow R_{l_h}) \text{ and there is a path } R_{k_h} \overset{e_h}{\rightarrow} \cdots \overset{e_0}{\rightarrow} R_i \text{ in graph}(T) \text{ in which } e_h \text{ is the only 2-outerjoin} \} \).

For a join edge \( e_0 = (R_i \rightarrow R_j) \), \( \text{cc2oj}(e_0) = \{ e_h \mid e_h = (R_{k_h} \leftrightarrow R_{l_h}) \text{ and there is a path } R_{k_h} \overset{e_h}{\rightarrow} \cdots \overset{e_0}{\rightarrow} R_i \text{ in graph}(T) \text{ in which } e_h \text{ is the only 2-outerjoin} \} \).

**Definition.** Let \( e_0 = (R_i \rightarrow R_j) \) be a 1-outerjoin edge in a graph \( G \). Removing \( e_0 \) from \( G \) leaves two connected graphs, say \( G_1 \) containing \( R_i \) and \( G_2 \) containing \( R_j \). The set of preserved attributes of \( e_0 \), denoted \( \text{pres}(e_0) \), is the union of attributes of relations in \( G_1 \).

Let \( e_0 \) be a 2-outerjoin edge and \( e_1 \) be an edge in a graph \( G \). Removing \( e_0 \) from \( G \) leaves two connected graphs, say \( G_1 \) containing \( e_1 \) and \( G_2 \). The set of preserved attributes of \( e_0 \) with respect to \( e_1 \), denoted \( \text{pres}_{e_1}(e_0) \), is the union of attributes of relations in \( G_2 \).

The next lemma shows how to choose the operator at the root, for any desired goal association, for simple join/1-outerjoin queries.

**Lemma 6.1.** Suppose \( T \) is a simple join/1-outerjoin operator tree in which all predicates are strong on the attributes of each of the relations they reference. If \( T' = (T'_a, T'_b) \) is an association tree of \( G = \text{graph}(T) \), then we can apply tree rewrites on \( T \) to obtain \( T'' = (T''_a \odot T''_b) \) such that:

- \( \text{eval}(T'') = \text{eval}(T) \);
- \( T''_a, T''_b \) are simple join/1-outerjoin operator trees;
- \( T'_a, T'_b \) are association trees of \( \text{graph}(T''_a) \), \( \text{graph}(T''_b) \), respectively; and
<table>
<thead>
<tr>
<th>$E_{ab}$</th>
<th>conditions</th>
<th>$\odot$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${R_i \rightarrow R_j}$</td>
<td>$p_{R_i R_j}$</td>
<td></td>
</tr>
<tr>
<td>${R_i \overset{p_1}{\rightarrow} R_j}$, $\exists \text{coj}(R_i, p_1, R_j)$</td>
<td>$\text{GOJ}[p_1 \land \cdots \land p_n, \text{pres}(\text{coj}(R_i, p_1, R_j))]$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>else</td>
<td>$p_1 \land \cdots \land p_n$</td>
</tr>
<tr>
<td>$(R_i, p_n)$</td>
<td>$\text{Edges connecting leaves of } T_a^i \text{ with leaves of } T_b^i \text{ in } G.$</td>
<td></td>
</tr>
<tr>
<td>$\odot$</td>
<td>$\text{Operator for } (T_a^i \odot T_b^i); \text{ assume that } R_i \in \text{leaves}(T_b^i)$.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.4: Selection of the root operator — join/1-outerjoin reordering.

- **operator $\odot$ is determined by the table in figure 6.4.**

**Proof.** Call $E_{ab}$ the set of edges connecting nodes in leaves$(T_a^i)$ with nodes in leaves$(T_b^i)$, in $G$.

**Case 1.** If there is at least one outerjoin edge in $E_{ab}$ then, by property 5 of simple trees, $E_{ab}$ contains exactly one edge, say $(R_i \rightarrow R_j)$. Then there is an outerjoin $p_{R_i R_j}$ in $T$. By properties 3 and 4 of simple trees, result-preserving rewrites transform $T$ into $(T_a^i \overset{p_{R_i R_j}}{\rightarrow} T_b^i)$. By properties 1 and 2 of simple trees, $T_a^i, T_b^i$ are simple operator trees.

**Case 2.** Suppose $E_{ab}$ is the set of join edges $e_1, \ldots, e_n$, and these join edges have no conflicting outerjoins. Find the set of nodes $V_f$ containing all nodes in paths that

- contain at least one edge $e_i \in R_{ab}$; and
- contain only join edges.

Find the set of outerjoin edges of the form $(v \rightarrow w)$, where $v \in V_f$; call them $\odot_1, \ldots, \odot_m$. Using case 1 of this lemma, we can transform $T$ into

$$T''' = (T_1 \odot_1 (\cdots (T_m \odot_m T_{m+1}) \cdots)),$$

where $\odot_1$ is the root, $\odot_{i+1}$ is a child of $\odot_i$, and $\text{graph}(T_{m+1}) = G|_{V_f}$.

By theorem 6.1, $T_{m+1}$ is freely reorderable, so it can be transformed into $T'''_{m+1}$ where the operator at the root is a join whose conjuncts $p_1, \ldots, p_n$ are taken from
edges \( e_1, \ldots, e_n \). Since each \( \circ_i \) is of the form \( (v \to w) \) for \( v \in \text{leaves}(T_{m+1}) \), the application of \( n \) result preserving rewrite rules on \( T'' \) moves the root of \( T'_{m+1} \) to the root of the tree.

We applied only result preserving rewrite rules so, by properties 1 and 2 of simple trees, we obtained a simple tree \( T'' = (T''_1 \bowtie \cdots \bowtie T''_n) \) whose subtrees are also simple. Assume without loss of generality that at least one leaf from association tree \( T'_{m} \) is in \( T''_i \). Removing edges \( E_{ab} \) from \( G \) leaves connected graphs \( G|_{\text{leaves}(T'_{m})} = \text{graph}(T''_i) \), \( G|_{\text{leaves}(T'_{m})} = \text{graph}(T''_i) \). Then \( T'_a \) is an association tree of graph \( (T''_i) \) and \( T'_b \) is an association tree of graph \( (T''_j) \).

**Case 3.** Suppose \( E_{ab} \) is the set of join edges \( e_1, \ldots, e_n \), with predicates \( p_1, \ldots, p_n \), and \( p_0 = \text{coj}(e_1) \). We will apply the transformation strategy illustrated in figure 6.5.

Move \( p_0 \) to the root of the tree using result-preserving rewrites to obtain \( T_1 = (T_c \bowtie \cdots \bowtie T_f) \). Now, note that joins for \( e_1, \ldots, e_n \) are in the simple tree \( T_d \), and have no conflicting outerjoin in \( T_d \). By case 2 of this lemma, result-preserving rewrites can be used to obtain \( T_i = (T_c \bowtie \cdots \bowtie T_f) \). Finally, apply identity (4.48) to reorder \( T_i \) and obtain \( T_{i+1} = (T_c \bowtie \cdots \bowtie p_n, \text{sch}(T_c) \bowtie T_f) \), where \( \text{sch}(T_c) = \text{pres}(p_0) \).

Trees \( (T_c \bowtie \cdots \bowtie T_f), T_f \) are simple. \( T'_a, T'_b \) are association trees of graph \( (T_c \bowtie \cdots \bowtie T_f) \), graph \( T_f \).

In the next lemma we extend the result to consider also two-sided outerjoins. The proof of how the reordered tree is obtained is similar to the previous proof, but it has more cases.

**Lemma 6.2.** Suppose \( T \) is a simple join/1-outerjoin/2-outerjoin operator tree in which all predicates are strong on the attributes of each of the relations they reference. If \( T' = (T'_a, T'_b) \) is an association tree of \( G = \text{graph}(T) \), then we can apply tree rewrites on \( T \) to obtain \( T'' = (T''_1 \bowtie T''_r) \) such that:

- \( \text{eval}(T'') = \text{eval}(T) \);
- \( T'_{1}', T'_{r}' \) are simple join/1-outerjoin operator trees;
- \( T'_{a}', T'_{b}' \) are association trees of graph \( (T''_1) \), graph \( (T''_r) \), respectively; and

78
Figure 6.5: Reordering with GOJ.
<table>
<thead>
<tr>
<th>$E_{ab}$</th>
<th>conditions</th>
<th>$\odot$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${(R_i \leftrightarrow R_j)}$</td>
<td>$p_{R_i R_j}$</td>
<td>${(R_i \leftrightarrow R_j)}$</td>
</tr>
<tr>
<td>${(R_i \rightarrow R_j)}$</td>
<td>$\exists \text{ cc2oj}(R_i \rightarrow R_j)$</td>
<td>$\text{GOJ}<em>{\exists p</em>{R_i R_j}, \text{pres}(e_0), \text{pres}<em>{e_i}(e_1), \ldots, \text{pres}</em>{e_n}(e_n)}$ for $e_0 = (R_i \rightarrow R_j); e_1, \ldots, e_n \in \text{cc2oj}(e_0)$</td>
</tr>
<tr>
<td>else</td>
<td>$p_{R_i R_j}$</td>
<td>${(R_i \rightarrow R_j)}$</td>
</tr>
<tr>
<td>${(R_i, p_1 \rightarrow R_{j_1})$</td>
<td>$\exists \text{ coj}(R_i, p_1 \rightarrow R_{j_1})$</td>
<td>$\text{GOJ}<em>{\exists p_1 \land \cdots \land p_n, \text{pres}((\text{coj}(e_0)), \text{pres}</em>{e_0}(e_1), \ldots, \text{pres}<em>{e_n}(e_n)}$ for $e_0 = (R_i, p_1 \rightarrow R</em>{j_1}); e_1, \ldots, e_n \in \text{cc2oj}(\text{coj}(e_0))$</td>
</tr>
<tr>
<td>$\ldots$ (R_{i_n}, p_n \rightarrow R_{j_n})$</td>
<td>$\text{else} \exists \text{cc2oj}(R_{i_n}, p_n \rightarrow R_{j_n})$</td>
<td>$\text{GOJ}<em>{\exists p_1 \land \cdots \land p_n, \text{pres}</em>{e_0}(e_1), \ldots, \text{pres}<em>{e_n}(e_n)}$ for $e_0 = (R</em>{i_n}, p_n \rightarrow R_{j_n}); e_1, \ldots, e_n \in \text{cc2oj}(e_0)$</td>
</tr>
<tr>
<td>else</td>
<td>$p_1 \land \cdots \land p_n$</td>
<td>$\text{else} \exists \text{cc2oj}(R_{i_n}, p_n \rightarrow R_{j_n})$</td>
</tr>
</tbody>
</table>

$E_{ab}$: Edges connecting leaves of $T_a'$ with leaves of $T_b'$ in $G$.
$\odot$: Operator for $(T'_l \odot T'_r)$; assume that $R_i \in \text{leaves}(T'_l)$.

Figure 6.6: Selection of the root operator —join/1-/2-outerjoin reordering.

- **operator $\odot$ is determined by the table in figure 6.6.**

**Proof.** Call $E_{ab}$ the set of edges connecting nodes in leaves($T_a'$) with nodes in leaves($T_b'$), in $G$.

**Case 1.** If there is at least one two-sided outerjoin edge in $E_{ab}$ then, by property 5 of simple trees, $E_{ab}$ contains exactly one edge, say $(R_i \leftrightarrow R_j)$. Then there is an outerjoin $p_{R_i R_j}$ in $T$. By properties 3 and 4 of simple trees, result-preserving rewrites transform $T$ into $(T_l' p_{R_i R_j} T_r')$. By properties 1 and 2 of simple trees, $T_l', T_r'$ are simple operator trees.

**Case 2.** If there is at least one one-sided outerjoin edge in $E_{ab}$ then, by property 5 of simple trees, $E_{ab}$ contains exactly one edge, say $(R_i \rightarrow R_j)$. Assume $(R_i \rightarrow R_j)$ has no conflicting two-sided outerjoins.

Apply result-preserving rewrites on $T$ to obtain

$$T''' = (T_1 p_1 \cdots (T_m p_m (T_{m+1} p_{R_i R_j} T_{m+2}) \cdots))$$

80
this can be done by properties 3 and 4 of simple trees. Since \((R_i \rightarrow R_j)\) has no conflicting outerjoins, applying \(m\) result preserving rewrites moves \(o_{ij}\) to the root of the tree.

**Case 3.** If there is at least a one-sided outerjoin edge in \(E_{ab}\) then, by property 5 of simple trees, \(E_{ab}\) contains exactly one edge, say \(e_0 = (R_i \rightarrow R_j)\). Assume \(cc2oj(R_i \rightarrow R_j) = e_1, \ldots, e_m\) are the conflicting two-sided outerjoins, with predicates \(p_1, \ldots, p_m\).

Apply result-preserving rewrites on \(T\) to obtain
\[
T''' = (T_1 \overset{p_1}{\rightarrow} (\cdots (T_m \overset{p_m}{\rightarrow} T_{m+1}) \cdots)),
\]
where the outerjoin \(p^{R_iR_j}\) is in \(T_{m+1}\); this can be done by properties 3 and 4. Since \((R_i \rightarrow R_j)\) has no conflicting two-sided outerjoins in \(\text{graph}(T_{m+1})\), case 2 of this lemma applies, and we can transform \(T_{m+1}\) into \(T'_{m+1} = (T'_{m+2} \overset{p^{R_iR_j}}{\rightarrow} T'_{m+3})\). We then do the following:

\[
\begin{align*}
T''' &= (T_1 \overset{p_1}{\rightarrow} (\cdots (T_m \overset{p_m}{\rightarrow} T_{m+1}) \cdots)) \\
&\sim (T_1 \overset{p_1}{\rightarrow} (\cdots (T_m \overset{p_m}{\rightarrow} (T'_{m+3} \overset{p^{R_iR_j}}{\rightarrow} T'_{m+2})) \cdots)) \\
&\sim (T_1 \overset{p_1}{\rightarrow} (\cdots (T_m \overset{p_m}{\rightarrow} \text{GOJ}_{[p^{R_iR_j}], \text{pres}(e_0), \text{pres}_{e_0}(e_1), \ldots, \text{pres}_{e_0}(e_m)] T'_{m+2}) \cdots)) \\
&\sim (T'_1 \text{GOJ}_{[p^{R_iR_j}], \text{pres}(e_0), \text{pres}_{e_0}(e_1), \ldots, \text{pres}_{e_0}(e_m)] T'_{m+2}) = T''.
\end{align*}
\]

The tree \(T''\) has the properties specified by the lemma.

**Case 4.** Suppose \(E_{ab}\) is the set of join edges \(e_1, \ldots, e_n\), and these join edges have no conflicting two-sided outerjoins. Then the query does not have two-sided outerjoins, and lemma 6.1 can be used to obtain the desired operator tree \(T''\).

**Case 5.** Suppose \(E_{ab}\) a set of join edges with predicates \(p_{j1}, \ldots, p_{jn}\), with no closest conflicting outerjoin, but with conflicting two-sided outerjoins \(e_1, \ldots, e_m\), with predicates \(p_1, \ldots, p_m\). By properties 3 and 4, apply result preserving rewrite rules on \(T\) to obtain
\[
T''' = (T_1 \overset{p_1}{\rightarrow} (\cdots (T_m \overset{p_m}{\rightarrow} T_{m+1}) \cdots)),
\]
where the join for edges \(E_{ab}\) are in tree \(T_{m+1}\), which contains no two sided outerjoins, and no conflicting outerjoin for edges in \(E_{ab}\). Then, by lemma 6.1, \(T_{m+1}\) can be
transformed into \((T_{m+2} \bowtie T_{m+3})\). Call \(p_j = p_{j_1} \wedge \ldots \wedge p_{j_n}\). Assume \(e_0\) is one of the edges \(E_{ab}\). We transform \(T'''\) as follows:

\[
T''' = (T_1 \overset{p_1}{\rightarrow} \cdots (T_m \overset{p_m}{\rightarrow} (T_{m+1} \overset{p_j}{\rightarrow} T_{m+2} \overset{p_{j_1}}{\rightarrow} T_{m+3})) \cdots))
\]

Case 6. Suppose \(E_{ab}\) a set of join edges with predicates \(p_{j_1}, \ldots, p_{j_n}\), with a closest conflicting outerjoin \(e_0 = (R_i \rightarrow R_j)\). Assume \(e_1, \ldots, e_m\) are the conflicting two-sided outerjoins of \(e_0\), with predicates \(p_1, \ldots, p_m\). Proceed as in case 3 to obtain \(T'''\) from \(T\) such that

\[
T''' = (T_1 \overset{p_1}{\rightarrow} \cdots (T_m \overset{p_m}{\rightarrow} (T_{m+1} \overset{p_{j_1}}{\rightarrow} T_{m+2} )) \cdots)).
\]

Joins for edges \(E_{ab}\) must be in tree \(T'_{m+3}\), which contains no two sided outerjoins, and no conflicting outerjoin for edges in \(E_{ab}\). Then, by lemma 6.1, \(T'_{m+3}\) can be transformed into \((T_{m+4} \overset{p_{j_1} \wedge \ldots \wedge p_{j_n}}{\bowtie} T_{m+3})\). Call \(p_j = p_{j_1} \wedge \ldots \wedge p_{j_n}\). We transform \(T'''\) as follows:

\[
T''' = (T_1 \overset{p_1}{\rightarrow} \cdots (T_m \overset{p_m}{\rightarrow} (T_{m+1} \overset{p_{j_1}}{\rightarrow} T_{m+2} )) \cdots))
\]

The tree \(T''\) has the properties specified by the lemma.

**Theorem 6.2.** Suppose \(T\) is a simple join/1-outerjoin/2-outerjoin operator tree in which all predicates are strong on the attributes of each of the relations they reference. For any association tree \(T'\) of \(\text{graph}(T)\), we can apply tree rewrite rules on \(T\) to obtain \(T''\) such that
\[
\begin{align*}
\text{eval}(T'') &= \text{eval}(T); \quad \text{and} \\
\text{assoc}(T'') &= T'.
\end{align*}
\]

**Proof.** By induction on the number of leaves in \( T \).

**Base.** If \( T \) is a single relation, \( T'' = T \).

**Induction.** Assume \( T' = (T'_a, T'_b) \) and call \( G = (V, E) = \text{graph}(T) \). Then, by lemma 6.2, we can apply tree rewrites on \( T \) to find \( T''' = (T''_i \odot T''_r) \) such that

- \( \text{eval}(T''') = \text{eval}(T) \);
- \( T''_i, T''_r \) are simple join/1-outerjoin operator trees; and
- \( T'_a, T'_b \) are association trees of \( \text{graph}(T''_i), \text{graph}(T''_r) \).

Now, by induction hypothesis, we can apply tree rewrite rules on \( T''_i, T''_r \) to find \( T''_i, T''_r \) such that \( \text{eval}(T''_i) = \text{eval}(T''_i'), \text{eval}(T''_r) = \text{eval}(T''_r'), \text{assoc}(T''_i) = T'_a \), and \( \text{assoc}(T''_r) = T'_b \). Then \( T''' = (T''_i \odot T''_r) \) is such that

- \( \text{eval}(T''') = \text{eval}(T) \); and
- \( \text{assoc}(T''') = T' \).

**Observation.** Given an operator tree \( T \) and an association tree \( T' \) for \( \text{graph}(T) \), theorem 6.2 gives a rule to determine the operators that should be placed at the inner nodes of \( T' \) to obtain a new operator tree \( T'' \).

Call \( G = \text{graph}(T) \). Note that, if \( T'_1 = (T'_2, T'_3) \) is a subtree of the association tree \( T' \), the operator that will be placed at the root of \( T'_1 \) to obtain \( T'' \) depends only on the graph \( G|_{\text{leaves}(T'_1)} \).

### 6.4 Modifying a relational optimizer to handle outerjoins

To be of practical use with conventional relational DBMSs, we need to show how these results are integrated into current optimizers.

The optimizer should first simplify its input operator tree. It then computes the auxiliary functions \( \text{ccoj}(\cdot) \), \( \text{cc2oj}(\cdot) \), and also \( \text{pres}(\cdot) \). Finally, it should use an enumeration algorithm on the graph of the simple query to generate evaluation alternatives.
A general form of this enumeration algorithm is described in section 6.4.1. Restrictions on the order and extent to which the space of alternatives is explored can be easily imposed to make the algorithm compatible with enumeration strategies of a given optimizer. Section 6.4.2 gives tables indicating how to choose operators for the trees generated.

### 6.4.1 Enumeration of join-orderings

Conventional optimizers enumerate operator trees as alternative strategies for evaluating a query. These trees are built bottom-up from the query graph. Each step combines two trees to produce a tree with one more join. A conventional algorithm for enumerating association trees is as follows:

**Algorithm A.** Enumeration of association trees.

**Input.** A query graph $G$ of $n$ relations.

**Output.** A set of association trees for $G$.

**Procedure.**

**A-1.** For each node in $G$ create a 1-leaf tree.

**A-2.** For $k = 2, \ldots, n$: Choose $T_l, T_r$ such that

- $\text{leaves}(T_l) \cap \text{leaves}(T_r) = \emptyset$.
- $|\text{leaves}(T_l)| + |\text{leaves}(T_r)| = k$.
- $G|_{\text{leaves}(T_l) \cup \text{leaves}(T_r)}$ is connected.

Create a tree $(T_l, T_r)$.

**A-3.** Output trees with $n$ leaves.

Particular optimizers impose restrictions on the trees to be considered, e. g. that $T_r$ in step **A-2** be a one-leaf tree. Most optimizers insist that every join involve an edge in $G$. For us this is more than a mechanism to speed the optimizer — our reordering results of section 6.3 make this assumption on graph connectivity.

---

<sup>1</sup>Actually, the implementation of the algorithm would produce a graph that embeds all the trees, rather than a set of trees [Rose86].
\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
$e_0$ & conditions & $\oplus$ \\
\hline
$(R_i \rightarrow R_j)$ & $p^{R_i,R_j}_{\rightarrow}$ & \\
$\left(R_i \leftarrow R_j\right)$ & if $c\mathrm{coj}(e_0) \in E'$ & $G\mathrm{OJ}[p^{R_i,R_j}, \mathrm{pres}(c\mathrm{coj}(e_0))]$ \\
& else & $p^{R_i,R_j}_{\bowtie}$ \\
\hline
\end{tabular}
\end{table}

$e_0$: Edge connecting graph$(T_l)$ and graph$(T_r)$ in $G$.

$\oplus$: Operator for $(T_l \odot T_r)$, assume that $R_i \in \text{leaves}(T_l)$.

$E'$: Edges found in either graph$(T_l)$ or graph$(T_r)$.

Figure 6.7: Operator selection — join/1-outerjoin.

6.4.2 Choosing the appropriate operators

Algorithm $A$ enumerates association trees. To generate executable operator trees we need to determine an operator in step $A\cdot 2$. We now show how to choose this operator and its parameters.

Join/1-outerjoin simple queries

The table in figure 6.7 shows how to choose the operators for association trees created by algorithm $A$, based on the proof of theorem 6.2. When algorithm $A$ combines two trees $T_l$, $T_r$, if $e_0$ is the edge connecting leaves$(T_l)$ with leaves$(T_r)$ in $G$, the table shows how to choose the appropriate operator $\oplus$ for the new tree $(T_l \odot T_r)$.

Figure 6.8 shows the annotated query graph of a simple query, and the values of the auxiliary functions $c\mathrm{coj}()$ and $\mathrm{pres}()$. Figure 6.9 shows the operators selected for an execution order, according with the table in figure 6.7.

Join/1-outerjoin/2-outerjoin simple queries

For queries with both one- and two-sided outerjoins, it seems a good heuristic to delay the evaluation of two-sided outerjoins. Two-sided outerjoins are not selective, and their reordering requires generalized outerjoins to preserve multiple sets of attributes, in general. Following this heuristic, algorithm $A$ would produce trees in which two-sided outerjoins are evaluated only after joins and one-sided outerjoins. Choosing an operator is done as for
join/1-outerjoins, using the table in figure 6.7. When there is a bidirected edge connecting the leaves of subtrees, the operator to use is two-sided outerjoin.

Evaluating two-sided outerjoins before joins and 1-outerjoins requires a more sophisticated analysis. The table in 6.10 shows how to choose operators for any reordering of join/1-outerjoin/2-outerjoin queries. Both the table in figure 6.7 and in 6.10 are derived from theorem 6.2. The way we use them here is justified by the observation in page 83 of how theorem 6.2 constructs the new operator tree by looking at subgraphs only.

### 6.5 Implementation algorithms

We now describe in some detail the main strategies used to implement cartesian products and join, and how they can be modified to compute outerjoins.

**Nested loops.** The basic implementation method for cartesian product is *nested-loops*:

```plaintext
for t_1 in R_1 do
    for t_2 in R_2 do
        output (t_1, t_2)
```

In the method above, $R_1$ is naturally called the *outer* relation and $R_2$ is called the *inner* relation. Main memory management can yield some savings, if we assume that disk access is relatively expensive. For instance, if $R_2$ fits in main memory, it makes sense to load it only once and use a hash table to store its contents.
into main memory once, and then process \( R_1 \) sequentially using the pre-loaded data. Of course, the system is free to reverse the inner/outer roles of the \( R_1, R_2 \) in the algorithm.

This method can easily be modified to process joins, by testing the join predicate on each pair of tuples \((t_1, t_2)\).

**Index and hash access.** For some match predicates, we can take advantage of an index or hash table kept on a file. If the match predicate is equality between fields \( f_1 \) and \( f_2 \) of tuples from \( R_1 \) and \( R_2 \), respectively, we can use the following procedure:

```plaintext
for \( t_1 \) in \( R_1 \) do
    get \( t_2 \) from \( R_2 \) such that \( t_1.f_1 = t_2.f_2 \)
    output \((t_1, t_2)\)
```

The *get* primitive above uses some structure for fast retrieval of a record from \( R_2 \) for a given field value. Depending on what structure is used, the algorithm is called *index-join* or *hash-join*. If neither structure is maintained by the file system for field \( f_2 \), we can scan \( R_2 \) sequentially to create a hash table, and then compute the join.
$e_0$: Edge connecting graph($T_l$) and graph($T_r$) in $G$.  
$\odot$: Operator for ($T_l \odot T_r$), assume that $R_i \in \text{leaves}(T_l)$.  
$E'$: Edges found in either graph($T_l$) or graph($T_r$).  

Figure 6.10: Operator selection for simple queries.

If $f_2$ is not a key for $R_2$, there may be several records with the same $f_2$ value, and we need to adjust the pseudo code above accordingly. The method can also be modified for the case where the equality is one of several conjuncts of the match predicate — retrieve tuples satisfying the equality using an index, and test the rest of the predicate before outputing the result.

In terms of cost, the method may read new pages from disk for each $t_2$ retrieved from $R_2$. For this reason, it works best when $R_1$ is significantly smaller than $R_2$.

To compute the outerjoin $R_1 \stackrel{p}{\bowtie} R_2$, we have to output each tuple $t_1$ for which no matching $t_2$ was found. The necessary logic can be added easily to the pseudocode above. $R_1 \triangleleft R_2$ and $R_1 \triangleright R_2$ can also be obtained by small modifications to this method.

Computing the two-sided outerjoin is more difficult. It requires keeping track of what tuples of $R_2$ have been output in the main loop, and then scanning $R_2$ at the end to output the remaining tuples — unmatched in $R_1$. A bit map or hash table can be used to keep track of $R_2$ tuples already present in the result. The same principle can be used to compute $R_2 \stackrel{p}{\bowtie} R_1$, $R_2 \stackrel{\triangleright}{\bowtie} R_1$ and $R_2 \triangleright R_1$.  

88


**Merge.** We can use a process similar to that of merging sorted files to compute join. If $R_1$ is sorted on $f_1$, $R_2$ is sorted on $f_2$, and the match predicate is equality on $f_1, f_2$, we can use *merge-join*:

\[
\begin{align*}
t_1 &= \text{first } R_1 \\
t_2 &= \text{first } R_2 \\
\text{while not EOF } R_1 \text{ and not EOF } R_2 \\
&\quad \text{if } t_1.f_1 = t_2.f_2 \\
&\quad \quad \text{output } (t_1, t_2) \\
&\quad \quad t_1 = \text{next } R_1 \\
&\quad \quad t_2 = \text{next } R_2 \\
&\quad \text{else if } t_1.f_1 < t_2.f_2 \\
&\quad \quad t_1 = \text{next } R_1 \\
&\quad \text{else} \\
&\quad \quad t_2 = \text{next } R_2 
\end{align*}
\]

This pseudocode assumes $f_1$ and $f_2$ are file keys. If they are not, it has to be modified to consider multiple tuples with the same value on $f_1$ or $f_2$.

Modifying the code to compute semijoin, antijoin, and all cases of outerjoin is easy, since we can determine immediately whether or not a particular tuple has a matching tuple in the other relation.

**Joining index files.** The use of an index to access a data file can be considered as an additional join. For instance, if we compute $R_1 \bowtie R_2$ using index $I_2$ whose records are of the form (field value, tuple address), we can view the computation essentially as $R_1 \bowtie I_2 \bowtie R_2$, for an appropriate predicate $\bowtie$ that represents a one-to-one mapping between $I_2$ and $R_2$. For predicates that reference only the indexed attribute, computing the semijoin or antijoin can be done without accessing $R_2$; that is, $R_1 \bowtie R_2 = R_1 \bowtie I_2$ and $R_1 \bowtie R_2 = R_1 \bowtie I_2$.

**Pipelining.** Our description of join methods above has implicitly assumed that relations are stored in system files, but join arguments can also be the result of other relational operators—e.g., $(R_1 \bowtie R_2) \bowtie R_3$. In this case it makes sense to use pipelining between the
two operators. If the “streams” of tuples are sorted on the appropriate fields, merge sort can be used directly.

For nested-loops, index-join, and hash-join, the outer relation — $R_1$ in our pseudocode — is scanned just once, sequentially, so it makes no difference if it is read from a file or if it is the output of another operator. No pipelining can be done on the inner relation because its records are accessed more than once, or non-sequentially. If relation $R_2$ in our pseudocode is the result of another operator, we will usually have to read it completely, perhaps creating a hash table [Kel89], before starting to execute the binary operator.
Chapter 7

Concluding remarks

Efficient processing of outerjoins has a significant impact on applications not currently served by query optimization technologies. In this thesis we have examined the algebraic tools and algorithms necessary to treat outerjoin in database optimizers.

Full reordering of outerjoins has been recognized as a complex problem, and few, fragmentary results had been published previous to our work. For select/project/join queries, the necessary properties of the basic relational operators do not require extensive algebraic analysis. Straightforward extensions of those properties and techniques to include outerjoins bring an overwhelmingly complex scenario —identify identities describing how join, semijoin, antijoin, and the various kinds of outerjoins behave with respect to each other. We have developed and formalized a set of structuring concepts that allow a systematic study of the problem. Some of those concepts were implicit in the reordering of join queries, but their formalization was unnecessary there.

Two major structures are developed in this thesis to deal with the problem of outerjoin processing. First, the abstraction of query reordering in terms of feasible associations and how it is tied to identities of particular operators, in chapter 5. Second, the definition of preserve DAGs as a conceptually manageable generalization of outerjoin expressions, in chapter 4. These two ideas provide the necessary support to prove and understand our results. They should also be useful to deal with other query processing problems.

Our formulation of relational algebra in chapter 2 simplifies the algebraic manipulation considerably. The conventional formalization requires an explicit padding with null values
in outerjoins, thus adding one more operator that needs to be handled.

The definition of a partial order of relation containment in chapter 4 serves to formulate concisely some general principles of query optimization. This partial order is useful in proving algebraic properties relevant to optimization.

Directions for future work

The work we have presented can be continued in several ways. Some important directions for future work are the following:

- Preserve DAGs constitute a more “declarative,” logic-like specification of join/outerjoin queries. Some of the language implications of freely-reorderable queries apply to preserve DAGs — no order of evaluation needs to be given. Applications and query languages may find it easier to use preserve DAGs directly, rather than fully-parenthesized join/outerjoin expressions.

- Further study of preserve DAGs may lead to more general results on join/outerjoin reordering. Our results of chapter 6 impose some restrictions on the format of outerjoin predicates. Those restrictions may be removed, at least partially.

- We need to explore the application of our general concepts of reordering for operators other than join and outerjoin. Candidate operators include semijoin and antijoin, as well as operators for other data types in which change in the order of processing may have an impact on performance — e.g. geometric operators.

- Our query optimization formalisms need to be extended to deal with the important issue of aggregate operators on nested relations — e.g. count, average. There must be simplification, reordering, and argument reduction strategies for join/outerjoin/aggregate queries. Natural joins and cartesian products should also be examined.
Appendix A

Proofs of algebraic identities

Identities (4.1) through (4.27) follow immediately from the definitions of operators, and we do not prove them here. We will first prove reduction identities, then identities for preserved relations, and finally association identities for join-like operators.

A.1 Reduction identities for joins and outerjoins

Identity (4.28)

If $A \subseteq \text{sch}(R_1)$, then

$$\pi_A(R_1 \overset{p}{\rightarrow} R_2) = \pi_A(R_1 \overset{p}{\rightarrow} R_2) = \pi_A R_1.$$  

Identity (4.29)

$$\sigma_{\text{def}_{\text{sch}(R_2)}}(R_1 \overset{p}{\rightarrow} R_2) = R_1 \bowtie R_2.$$
\[ \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\rightarrow} R_2) \cup \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\leftarrow} R_2), \text{ by (4.14)} \]
\[ = \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\rightarrow} R_2), \text{ by definition of \text{def 4}} \]
\[ = \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\times} R_2), \text{ by (2.1)} \]
\[ = \sigma_p(R_1 \times R_2), \text{ by (4.11)} \]
\[ = \sigma_p(R_1 \times R_2), \text{ by definition of \text{def 4}} \]
\[ = R_1 \overset{p}{\bowtie} R_2, \text{ by (2.1).} \]

\[ \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\leftarrow} R_2) = R_1 \overset{p}{\leftarrow} R_2. \]

Identity (4.30)

\[ \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\rightarrow} R_2) \]
\[ = \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\rightarrow} R_2) \cup R_1 \overset{p}{\leftarrow} R_2 \cup R_2 \overset{p}{\leftarrow} R_1, \text{ by (2.5) and (2.5)} \]
\[ = \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\rightarrow} R_2) \cup \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\rightarrow} R_2) \cup \sigma_{\text{def sch}(R_2)}(R_2 \overset{p}{\rightarrow} R_1), \text{ by (4.14)} \]
\[ = \sigma_{\text{def sch}(R_2)}(R_1 \overset{p}{\rightarrow} R_2) \cup \sigma_{\text{def sch}(R_2)}(R_2 \overset{p}{\rightarrow} R_1), \text{ by definition of \text{def 4}} \]
\[ = (R_1 \overset{p}{\bowtie} R_2) \cup (R_2 \overset{p}{\bowtie} R_1), \text{ by definition of \text{def 4}} \]
\[ = R_1 \overset{p}{\leftarrow} R_2, \text{ by (2.4).} \]

Identity (4.31)

First, note that, by (4.1),
\[ \sigma_p(R_1 \times R_2) \leq \tau_{\text{sch}(R_1)} \sigma_p(R_1 \times R_2) \times \tau_{\text{sch}(R_2)} \sigma_p(R_1 \times R_2) = (R_1 \overset{p}{\bowtie} R_2) \times (R_2 \overset{p}{\bowtie} R_1). \]

We know that \((R_i \overset{p}{\bowtie} R_j) \leq R_i\). Using the monotonicity of cartesian product (4.27), for any \(R_i^j\) such that \(R_1 \overset{p}{\bowtie} R_2 \leq R_i^j \leq R_1\)
\[ (R_1 \overset{p}{\bowtie} R_2) \times (R_2 \overset{p}{\bowtie} R_1) \leq R_i^j \times R_2 \leq R_1 \times R_2. \]

By monotonicity and idempotence of selection, (4.3) and (4.24),
\[ \sigma_p(R_1 \times R_2) \leq R_i^j \times R_2 \leq R_1 \times R_2 \]
\[ \sigma_p \sigma_p(R_1 \times R_2) \leq \sigma_p(R_i^j \times R_2) \leq \sigma_p(R_1 \times R_2) \]
\[ \sigma_p(R_1 \times R_2) \leq \sigma_p(R_i^j \times R_2) \leq \sigma_p(R_1 \times R_2) \]
\[ R_1 \overset{p}{\bowtie} R_2 \leq R_i^j \overset{p}{\bowtie} R_2 \leq R_1 \overset{p}{\bowtie} R_2. \]
So, $R_1 \bowtie_p R_2 = R'_1 \bowtie_p R_2$, for any $R'$ containing $R_1 \bowtie_p R_2$ and contained in $R_1$.

To see that $R_1 \bowtie_p R_2$ is the minimal argument to compute the join, consider $R'_1$ such that $R''_1 < R_1 \bowtie_p R_2$.

$$\pi_{sch(R_1)}(R'_1 \bowtie_p R_2) = (R''_1 \bowtie_p R_2) \leq R''_1 < \pi_{sch(R_1)}(R_1 \bowtie_p R_2),$$

and therefore $(R''_1 \bowtie_p R_2) \neq (R_1 \bowtie_p R_2)$.

---

Identity (4.32)

$$R_1 \overset{p}{\rightarrow} R_2 = R_1 \overset{p}{\rightarrow} (R_2 \bowtie_p R_1).$$

---

Identity (4.33)

$$\pi_{sch(R_1)}(R_1 \overset{p}{\rightarrow} R_2) = R_1.$$

---

Identity (4.34)

$$\pi_{sch(R_1)}(R_1 \overset{p}{\leftrightarrow} R_2) = R_1.$$
A.2 Identities for preserved relations

Identity (4.35)

\[ R_1 \xrightarrow{p} R_2 = \text{PresR}(R_1 \bowtie R_2, R_1). \]

\[ \text{PresR}(R_1 \bowtie R_2, R_1) = (R_1 \bowtie R_2 - \pi_{\text{sch}(R_1)} \cup \pi_{\text{sch}(R_2)} R_1) \cup (R_1 - \pi_{\text{sch}(R_1)}(R_1 \bowtie R_2)), \]

by definition of PresR()

\[ = R_1 \bowtie R_2 \cup (R_1 - \pi_{\text{sch}(R_1)}(R_1 \bowtie R_2)), \text{ difference of disjoint sets} \]

\[ = R_1 \bowtie R_2 \cup R_1 \triangleright R_2, \text{ by (2.2) and (2.3)} \]

\[ = R_1 \xrightarrow{p} R_2, \text{ by (2.4)}. \]

Identity (4.36)

\[ R_1 \xleftarrow{p} R_2 = \text{PresR}(R_1 \bowtie R_2, R_1, R_2). \]

\[ \text{PresR}(R_1 \bowtie R_2, R_1, R_2) = (R_1 \bowtie R_2 - \pi_{\text{sch}(R_1)} \cup \pi_{\text{sch}(R_2)}(R_1 \cup R_2)) \cup (R_1 - \pi_{\text{sch}(R_1)}(R_1 \bowtie R_2 \cup R_2)) \]

\[ \cup (R_2 - \pi_{\text{sch}(R_2)}(R_1 \bowtie R_2 \cup R_1)), \text{ by definition of PresR()} \]

\[ = R_1 \bowtie R_2 \cup (R_1 - \pi_{\text{sch}(R_1)}(R_1 \bowtie R_2)) \cup (R_2 - \pi_{\text{sch}(R_2)}(R_1 \bowtie R_2)), \]

\[ \text{difference of disjoint sets} \]

\[ = R_1 \bowtie R_2 \cup R_1 \triangleright R_2 \cup R_2 \triangleright R_1, \text{ by (2.2) and (2.3)} \]

\[ = R_1 \xleftarrow{p} R_2 \cup R_2 \xleftarrow{p} R_1, \text{ by (2.4)} \]

\[ = R_1 \xleftarrow{p} R_2, \text{ by (2.5)}. \]

Identity (4.37)

If \( R_1, \ldots, R_n \) define a preserve DAG,

\[ R_i \subseteq \pi_{\text{sch}(R_i)} \text{PresR}(R_1, \ldots, R_n). \]

Recall that

\[ \text{PresR}(R_1, \ldots, R_n) = R'_1 \cup \cdots \cup R'_n, \]

where

\[ R'_j = R_j - \pi_{\text{sch}(R_j)} \left( \bigcup_{k \neq j} R_k \right). \]
Now, call
\[ R''_i = \pi_{\text{sch}(R_i)} \text{PresR}(R_1, \ldots, R_n). \]

By the definition of preserve DAG, \( \pi_{\text{sch}(R_i)} R_j \) is disjoint with \( R_i \), if \( j \neq i \) and \( R_j \) is not an ancestor of \( R_i \). The proof that \( R_i \subseteq R''_i \) is by induction on the number of ancestors of \( R_i \) in the preserve DAG.

**Base.** Assume \( R_i \) is the root of the DAG. Then
\[
R'_i = R_i - \pi_{\text{sch}(R_i)} \left( \bigcup_{k \neq j} R_k \right) = R_i.
\]
\[
R''_i = \pi_{\text{sch}(R_i)} \left( R'_1 \cup \cdots \cup R'_n \right)
= \pi_{\text{sch}(R_i)} R'_i \cup \pi_{\text{sch}(R_i)} \left( R'_1 \cup \cdots \cup R'_n \right)
= R_i \cup \pi_{\text{sch}(R_i)} (R'_1 \cup \cdots \cup R'_n).
\]

So, \( R_i \subseteq R''_i \).

**Induction.** Assume that \( R_1, \ldots, R_{i-1} \) are the ancestors of \( R_i \). Then
\[
R'_i = R_i - \pi_{\text{sch}(R_i)} \left( \bigcup_{k \neq i} R_k \right)
= R_i - \pi_{\text{sch}(R_i)} (R_1 \cup \cdots \cup R_{i-1}) - \pi_{\text{sch}(R_i)} (R_{i+1} \cup \cdots \cup R_n)
= R_i - \pi_{\text{sch}(R_i)} (R_1 \cup \cdots \cup R_{i-1}).
\]
\[
R''_i = \pi_{\text{sch}(R_i)} \left( R'_1 \cup \cdots \cup R'_n \right)
= \pi_{\text{sch}(R_i)} (R'_1 \cup \cdots \cup R'_n) \cup \pi_{\text{sch}(R_i)} R'_i \cup \pi_{\text{sch}(R_i)} (R'_{i+1} \cup \cdots \cup R'_n)
= \pi_{\text{sch}(R_i)} (R'_1 \cup \cdots \cup R'_n) \cup \cdots \cup \pi_{\text{sch}(R_i)} (R'_1 \cup \cdots \cup R'_n)
\quad \text{\( i-1 \) times}
\cup \pi_{\text{sch}(R_i)} R'_i \cup \pi_{\text{sch}(R_i)} (R'_{i+1} \cup \cdots \cup R'_n)
= \pi_{\text{sch}(R_i)} \pi_{\text{sch}(R_i)} (R'_1 \cup \cdots \cup R'_n) \cup \cdots \cup \pi_{\text{sch}(R_i)} \pi_{\text{sch}(R_{i+1})} (R'_1 \cup \cdots \cup R'_n)
\quad \cup \pi_{\text{sch}(R_i)} R'_i \cup \pi_{\text{sch}(R_i)} (R'_{i+1} \cup \cdots \cup R'_n)
= \pi_{\text{sch}(R_i)} R''_1 \cup \cdots \cup \pi_{\text{sch}(R_i)} R''_{i-1} \cup R'_i \cup \pi_{\text{sch}(R_i)} (R'_{i+1} \cup \cdots \cup R'_n).
\]

Then, substituting \( R'_i \),
\[
R_i - \pi_{\text{sch}(R_i)} (R_1 \cup \cdots \cup R_{i-1}) \cup \pi_{\text{sch}(R_i)} (R''_1 \cup \cdots \cup R''_{i-1}) \subseteq R''_i. \quad (A.1)
\]

97
But, by induction hypothesis \( R_k \subseteq R''_i \), for \( k = 1, \ldots, i - 1 \). Then, by monotonicity of union and projection

\[
\pi_{\text{sch}(R_i)}(R_1 \cup \cdots \cup R_{i-1}) \subseteq \pi_{\text{sch}(R_i)}(R''_1 \cup \cdots \cup R''_{i-1}).
\]

Since \( A \subseteq A - B \cup C \), for \( B \subseteq C \),

\[
R_i \subseteq R_i - \pi_{\text{sch}(R_i)}(R_1 \cup \cdots \cup R_{i-1}) \cup \pi_{\text{sch}(R_i)}(R''_1 \cup \cdots \cup R''_{i-1}). \tag{A.2}
\]

The inequalities (A.1) and (A.2) imply \( R_i \subseteq R''_i \).

Identity (4.38) If \( R_1, \ldots, R_n \) define a preserve DAG, and \( R_1, \ldots, R_{n-1} \) are all ancestors of \( R_n \), then

\[
R_n = \pi_{\text{sch}(R_n)} \text{PresR}(R_1, \ldots, R_n).
\]

Recall that

\[
\text{PresR}(R_1, \ldots, R_n) = R'_1 \cup \cdots \cup R'_n,
\]

where

\[
R'_j = R_j - \pi_{\text{sch}(R_j)} \left( \bigcup_{k \neq j} R_k \right).
\]

For \( j = 1, \ldots, n \), \( R'_j \subseteq R_j \). By definition of preserve DAG, since all other relations are ancestors of \( R_n \), \( \pi_{\text{sch}(R_n)} R_j \subseteq R_n \). By monotonicity of projection,

\[
\pi_{\text{sch}(R_n)} R'_j \subseteq \pi_{\text{sch}(R_n)} R_j \subseteq R_n, \text{ for } j = 1, \ldots, n.
\]

Then we have

\[
\pi_{\text{sch}(R_n)} \text{PresR}(R_1, \ldots, R_n)
\]

\[
= \pi_{\text{sch}(R_n)}(R'_1 \cup \cdots \cup R'_n) = \pi_{\text{sch}(R_n)} R'_1 \cup \cdots \cup \pi_{\text{sch}(R_n)} R'_n \subseteq R_n.
\]

But by (4.37), \( R_n \subseteq \pi_{\text{sch}(R_n)} \text{PresR}(R_1, \ldots, R_n) \). Therefore

\[
R_n = \pi_{\text{sch}(R_n)} \text{PresR}(R_1, \ldots, R_n).
\]
If $R_1, \ldots, R_n$ define a preserve DAG; $A \cap \text{sch}(R_i) \neq \emptyset$, for $i = 1, \ldots, k$; and $A \cap \text{sch}(R_i) = \emptyset$, for $i = k + 1, \ldots, n$, then

$$\pi_A \text{PresR}(R_1, \ldots, R_n) = \pi_A \text{PresR}(R_1, \ldots, R_k).$$

Using the definition,

$$R'_j = R_j - \pi_{\text{sch}(R_j)} \left( \bigcup_{l \neq j} R_l \right),$$

$$\pi_A \text{PresR}(R_1, \ldots, R_n) = \pi_A (R'_1 \cup \cdots \cup R'_n)$$

$$= \pi_A (R'_1 \cup \cdots \cup R'_k) \cup \pi_A (R'_{k+1} \cup \cdots \cup R'_n)$$

$$= \pi_A (R'_1 \cup \cdots \cup R'_k), \quad (A.3)$$

because $\text{sch}(R'_i) = \text{sch}(R_i)$ and $A \cap \text{sch}(R_i) = \emptyset$, for $i = k + 1, \ldots, n$.

Now, for the computation of each $R'_i$, for $i = 1, \ldots, k$,

$$R'_i = R_i - \pi_{\text{sch}(R_i)} (R_1 \cup \cdots \cup R_{i-1} \cup R_{i+1} \cup \cdots \cup R_k \cup R_{k+1} \cup \cdots \cup R_n)$$

$$= R_i - \pi_{\text{sch}(R_i)} (R_1 \cup \cdots \cup R_{i-1} \cup R_{i+1} \cup \cdots \cup R_k)$$

$$- \pi_{\text{sch}(R_i)} R_{k+1} - \cdots - \pi_{\text{sch}(R_i)} R_n$$

$$= R_i - \pi_{\text{sch}(R_i)} (R_1 \cup \cdots \cup R_{i-1} \cup R_{i+1} \cup \cdots \cup R_k), \quad (A.4)$$

because there are attributes in $R_i$ that are not in $R_j$, for $j = k + 1, \ldots, n$, so $R_j$ cannot be an ancestor of $R_i$, and their projection on $\text{sch}(R_i)$ does not intersect $R_i$. So, by (A.3) and (A.4),

$$\pi_A \text{PresR}(R_1, \ldots, R_n)$$

$$= (R_1 - \pi_{\text{sch}(R_1)} (R_2 \cup \cdots \cup R_k)) \cup \cdots \cup (R_k - \pi_{\text{sch}(R_k)} (R_1 \cup \cdots \cup R_{k-1}))$$

$$= \pi_A \text{PresR}(R_1, \ldots, R_k) \quad \text{Identity (4.40)}$$

If the following properties hold

- $R_1, \ldots, R_n$ define a preserve DAG, with root $R_1$;
- $S_1, \ldots, S_m$ define a preserve DAG, with root $S_1$;
- $R_1, \ldots, R_n, S_1, \ldots, S_m$ define a preserve DAG;
there is no \( i, j \) such that \( S_i \) is ancestor of \( R_j \); and

if for some \( j \) \( \text{sch}(S_1) \cap \text{sch}(R_j) = A \neq \emptyset \), then there is an \( S_k \) such that \( \text{sch}(S_k) = A \); then

\[
\text{PresR}(\text{PresR}(R_1, \ldots, R_n), \text{PresR}(S_1, \ldots, S_m)) = \text{PresR}(R_1, \ldots, R_n, S_1, \ldots, S_m).
\]

The expression is expanded by definition as

\[
\text{PresR}(\text{PresR}(R_1, \ldots, R_n), \text{PresR}(S_1, \ldots, S_m)) = \underbrace{\text{PresR}(R_1, \ldots, R_n) - \pi_{\text{sch}(R_1)} \text{PresR}(S_1, \ldots, S_m)}_{\alpha} \cup \underbrace{\text{PresR}(S_1, \ldots, S_m) - \pi_{\text{sch}(S_1)} \text{PresR}(R_1, \ldots, R_n)}_{\beta}.
\]

We will first work on part \( \alpha \) and then on \( \beta \). The inner \( \text{PresR}() \) are expanded as follows:

\[
\text{PresR}(R_1, \ldots, R_n) = R'_1 \cup \cdots \cup R'_n,
\]

\[
\text{PresR}(S_1, \ldots, S_m) = S'_1 \cup \cdots \cup S'_m,
\]

where

\[
R'_j = R_j - \pi_{\text{sch}(R_j)} \left( \bigcup_{k \neq j} R_k \right)
\]

\[
S'_j = S_j - \pi_{\text{sch}(S_j)} \left( \bigcup_{k \neq j} S_k \right)
\]

We can do the following with the \( \alpha \) part:

\[
\alpha = \text{PresR}(R_1, \ldots, R_n) - \pi_{\text{sch}(R_1)} \text{PresR}(S_1, \ldots, S_m)
\]

\[
= \text{PresR}(R_1, \ldots, R_n) - \text{PresR}(S_1, \ldots, S_m), \text{ because } \text{sch}(S_1) \subseteq \text{sch}(R_1),
\]

\[
= (R'_1 \cup \cdots \cup R'_n) - (S'_1 \cup \cdots \cup S'_n)
\]

\[
= R'_1 \cup \cdots \cup R'_n, \text{ because relations defining a preserve DAG are disjoint.}
\]

To compute each \( R'_j \) in \( \alpha \),

\[
R'_j = R_j - \pi_{\text{sch}(R_j)} \left( \bigcup_{k \neq j} R_k \right)
\]

\[
= R_j - \pi_{\text{sch}(R_j)} \left( \bigcup_{k \neq j} R_k \right) - \pi_{\text{sch}(R_j)} S_1 \cdots - \pi_{\text{sch}(R_j)} S_m, \text{ since no } S_i \text{ is ancestor of } R_j.
\]

So,

\[
\alpha = R_1 - \pi_{\text{sch}(R_1)} (R_2 \cup \cdots R_n \cup S_1 \cup \cdots S_m) \cup \cdots \cup R_n - \pi_{\text{sch}(R_n)} (R_1 \cup \cdots R_{n-1} \cup S_1 \cup \cdots S_m).
\]

100
Now, for the $\beta$ part, we proceed as follows:

$$\beta = \text{PresR}(S_1, \ldots, S_m) - \pi_{\text{sch}(S_1)} \text{PresR}(R_1, \ldots, R_n)$$

$$= S'_1 \cup \cdots \cup S'_m - \pi_{\text{sch}(S_1)}(R'_1 \cup \cdots \cup R'_n)$$

$$= S'_1 - \pi_{\text{sch}(S_1)}(R'_1 \cup \cdots \cup R'_n) \cup \cdots \cup S'_m - \pi_{\text{sch}(S_1)}(R'_1 \cup \cdots \cup R'_n) \cup \cdots$$

$$\cup S'_n - \pi_{\text{sch}(S_1)}(R'_1 \cup \cdots \cup R'_n).$$

First, expand $\gamma_i$:

$$\gamma_i = S'_i - \pi_{\text{sch}(S_1)} \text{PresR}(R_1, \cdots, R_n)$$

$$= S_i - \pi_{\text{sch}(S_1)}(S_1 \cup \cdots \cup S_{i-1} \cup S_{i+1} \cup \cdots \cup S_m)$$

$$- \pi_{\text{sch}(S_1)} R'_1 - \cdots - \pi_{\text{sch}(S_1)} R'_n.$$

We will now replace each $\pi_{\text{sch}(S_1)} R'_j$ by $\pi_{\text{sch}(S_i)} R'_j$ in $\gamma_i$. We have the following cases:

- If $\text{sch}(S_1) \cap \text{sch}(R_j) = \emptyset$, then $\pi_{\text{sch}(S_1)} R'_j = \emptyset = \pi_{\text{sch}(S_i)} R'_j$.

- If $\text{sch}(S_1) \cap \text{sch}(R_j) = \text{sch}(R_i)$, then $\pi_{\text{sch}(S_1)} R'_j = \pi_{\text{sch}(S_i)} R'_j$.

- Otherwise, there must be an $S_k$ such that $\text{sch}(S_1) \cap \text{sch}(R_j) = \text{sch}(S_k)$, so $\pi_{\text{sch}(S_1)} R'_j = \pi_{\text{sch}(S_k)} R'_j$.

Since the relations define a preserve DAG and $R_j$ is an ancestor of $S_k$, $\pi_{\text{sch}(S_k)} R'_j \subseteq \pi_{\text{sch}(S_k)} R_j \subseteq S_k$. Relations $R_i, R_k$ are disjoint, so

$$R_i - \pi_{\text{sch}(S_k)} R'_j = R_i.$$

Since $\pi_{\text{sch}(S_1)} R_j = \pi_{\text{sch}(S_k)} R_j \subseteq S_k$, by monotonicity of projection, $\pi_{\text{sch}(S_i)} R_j \subseteq \pi_{\text{sch}(S_i)} S_k$. Since $A - B = A - B - C$ if $C \subseteq B$,

$$S_i - \pi_{\text{sch}(S_i)} S_k = S_i - \pi_{\text{sch}(S_i)} S_k - \pi_{\text{sch}(S_i)} R_j.$$

Since we have managed to replace each $\pi_{\text{sch}(S_1)} R'_j$ by $\pi_{\text{sch}(S_i)} R'_j$, $\gamma_i$ takes the form

$$\gamma_i = S_i - \pi_{\text{sch}(R_i)} \left( \bigcup_{k \neq j} R_k \right) - \pi_{\text{sch}(S_i)} R_1 - \cdots - \pi_{\text{sch}(S_i)} R_n,$$

101
and now $\beta$ can now be rewritten as follows:

$$\beta = S_1 - \pi_{\text{sch}(S_1)}(S_2 \cup \cdots S_m \cup R_1 \cup \cdots R_n) \cup \cdots \cup S_m - \pi_{\text{sch}(S_m)}(S_1 \cup \cdots S_{m-1} \cup R_1 \cup \cdots R_n).$$

Finally,

$$\alpha \cup \beta = \text{PresR}(R_1, \ldots, R_n, S_1, \ldots, S_m).$$

### A.3 Associativity of join-like operators

In identities 4.44–4.56 and their proofs, we call $p^{ij}$ a predicate $p$ such that $\text{sch}(p) \subseteq (\text{sch}(R_i) \cup \text{sch}(R_j))$. The strategy to prove associativity of join like operators is to construct the same $\text{PresR}()$ expression for both the left and the right hand side of identities.

The expansion of join or outerjoin applied on a $\text{PresR}()$ are used frequently in the proofs. We show them once now to avoid unnecessary repetition.

Assume $p$ is a strong predicate with respect to $A$; $A \subseteq \text{sch}(R_i)$, for $i = 1, \ldots, k$; and $A \cap \text{sch}(R_i) = \emptyset$, for $i = k + 1, \ldots, n$. Then, by the distributivity properties of cartesian product and selection with respect to union, we have:

$$\text{PresR}(R_1, \ldots, R_n) \times R = \text{PresR}(R_1 \times R, \ldots, R_n \times R).$$

$$\sigma_p \text{PresR}(R_1, \ldots, R_n) = \text{PresR}(\sigma_p R_1, \ldots, \sigma_p R_k, \ldots, \sigma_p R_n)$$

$$= \text{PresR}(\sigma_p R_1, \ldots, \sigma_p R_k).$$

$$\text{PresR}(R_1, \ldots, R_n) \bowtie_p R = \text{PresR}(R_1 \bowtie_p R, \ldots, R_k \bowtie_p R).$$ (A.5)

$$\text{PresR}(R_1, \ldots, R_n) \overset{p}{\rightarrow} S = \text{PresR}(\text{PresR}(R_1, \ldots, R_n), \text{PresR}(R_1 \bowtie_p S, \ldots, R_k \bowtie_p S))$$

$$= \text{PresR}(\text{PresR}(R_1, \ldots, R_n), \text{PresR}(R_1 \bowtie_p S, \ldots, R_k \bowtie_p S))$$

$$= \text{PresR}(R_1, \ldots, R_n, R_1 \bowtie_p S, \ldots, R_k \bowtie_p S).$$ (A.6)

$$S \overset{p}{\rightarrow} \text{PresR}(R_1, \ldots, R_n) = \text{PresR}(S, R_1 \bowtie_p S, \ldots, R_k \bowtie_p S).$$ (A.7)

$$\text{PresR}(R_1, \ldots, R_n) \overset{p}{\rightarrow} S = \text{PresR}(R_1, \ldots, R_n, R_1 \bowtie_p S, \ldots, R_k \bowtie_p S, S).$$ (A.8)
If \( \text{sch}(p_1) \subseteq \text{sch}(R_1) \),

\[
\sigma_{p_1}(R_1 \overset{p_2}{\bowtie} R_2) = R_1 \overset{p_1 \wedge p_2}{\bowtie} (\sigma_{p_1}(R_1) \overset{p_2}{\bowtie} R_2).
\]

\[
\sigma_{p_1}(R_1 \overset{p_2}{\bowtie} R_2)
= \sigma_{p_1}(\sigma_{p_2}(R_1 \times R_2)), \text{ by (2.1)}
= \sigma_{p_2}(\sigma_{p_1}(R_1 \times R_2)), \text{ by (4.11)}
= \sigma_{p_2}((\sigma_{p_1}R_1) \times R_2), \text{ by (4.16)}
= (\sigma_{p_1}R_1) \overset{p_2}{\bowtie} R_2, \text{ by (2.1)}.
\]

Identity (4.42)

If \( \text{sch}(p_1) \subseteq \text{sch}(R_1) \),

\[
\sigma_{p_1}(R_1 \overset{p_2}{\bowtie} R_2) = (\sigma_{p_1}R_1) \overset{p_2}{\bowtie} R_2.
\]

\[
\sigma_{p_1}(R_1 \overset{p_2}{\bowtie} R_2)
= \sigma_{p_1}(R_1 \overset{p_2}{\bowtie} R_2 \cup R_1 \overset{p_2}{\bowdownarrow} R_2), \text{ by (2.4)}
= \sigma_{p_1}(R_1 \overset{p_2}{\bowtie} R_2) \cup \sigma_{p_1}(R_1 \overset{p_2}{\bowdownarrow} R_2), \text{ by (4.14)}
= (\sigma_{p_1}R_1) \overset{p_2}{\bowtie} R_2 \cup (\sigma_{p_1}R_1) \overset{p_2}{\bowdownarrow} R_2, \text{ by (4.41) and (4.11)}
= (\sigma_{p_1}R_1) \overset{p_2}{\bowtie} R_2, \text{ by (2.4)}.
\]

Identity (4.43)

If \( \text{sch}(p_1) \subseteq \text{sch}(R_1) \),

\[
\sigma_{p_1}(R_1 \overset{p_1 \wedge p_2}{\bowtie} R_2) = (\sigma_{p_1}R_1) \overset{p_2}{\bowtie} R_2.
\]

\[
\sigma_{p_1}(R_1 \overset{p_1 \wedge p_2}{\bowtie} R_2)
= R_1 \overset{p_1 \wedge p_2}{\bowtie} R_2 \cup R_2 \overset{p_1 \wedge p_2}{\bowdownarrow} R_1, \text{ by (2.4)}
= R_1 \overset{p_1 \wedge p_2}{\bowtie} R_2 \cup R_2 - \pi_{\text{sch}(R_2)}(R_1 \overset{p_1 \wedge p_2}{\bowtie} R_2), \text{ by (2.3) and (2.2)}
= (\sigma_{p_1}R_1) \overset{p_1}{\bowtie} R_2 \cup R_2 - \pi_{\text{sch}(R_2)}((\sigma_{p_1}R_1) \overset{p_1}{\bowtie} R_2), \text{ by (4.41)}
= (\sigma_{p_1}R_1) \overset{p_2}{\bowtie} R_2, \text{ by (2.2), (2.3), and (2.4)}.
\]

Identity (4.44)

\[
(R_1 \overset{p_1^1}{\bowtie} R_2) \overset{p_1^2 \wedge p_2^3}{\bowtie} R_3 = R_1 \overset{p_1^2}{\bowtie} \overset{p_1^1 \wedge p_2^3}{\bowtie} (R_2 \overset{p_2^3}{\bowtie} R_3).
\]

\[
(R_1 \overset{p_1^1}{\bowtie} R_2) \overset{p_1^2 \wedge p_2^3}{\bowtie} R_3
= \sigma_{p_1^2 \sigma_{p_1^3 \wedge p_2^3}}((R_1 \times R_2) \times R_3), \text{ by (2.1) and (4.16)}
\]

103
\[ \sigma_{p^{12} \wedge p^{13}}(R_1 \times (R_2 \times R_3)), \text{ by (4.22)} \]
\[ R_1 \overset{p^{12} \wedge p^{13}}{\triangleright} (R_2 \triangleright R_3), \text{ by (4.16) and (2.1).} \]

Identity (4.45)

\[ (R_1 \overset{p^{12}}{\triangleright} R_2) \overset{p^{23}}{\rightarrow} R_3 = R_1 \overset{p^{12}}{\triangleright} (R_2 \overset{p^{23}}{\rightarrow} R_3). \]

If \( p^{23} \) is strong with respect to \( \text{sch}(R_2) \),

Identity (4.46)

\[ (R_1 \overset{p^{12}}{\rightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3 = R_1 \overset{p^{12}}{\rightarrow} (R_2 \overset{p^{23}}{\rightarrow} R_3). \]

Identity (4.47)

\[ (R_1 \overset{p^{12}}{\leftarrow} R_2) \overset{p^{23}}{\rightarrow} R_3 = R_1 \overset{p^{12}}{\leftarrow} (R_2 \overset{p^{23}}{\rightarrow} R_3). \]
If \( p^{23} \) is strong with respect to \( \text{sch}(R_2) \),

\[
(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_2 \xrightarrow{p^{12}} R_1 \xRightarrow{R_2} \xrightarrow{p^{23}} R_3.
\]

\[(R_1 \xrightarrow{p^{12}} R_2) \xRightarrow{p^{23}} R_3
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2, R_1) \xRightarrow{R_2} R_3, \text{ by (4.35)}
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2 \xRightarrow{p^{23}} R_3), \text{ by (A.5)}
\]

\[
= R_1 \xrightarrow{p^{12}} R_2 \xRightarrow{p^{23}} R_3.
\]

Identity (4.49)

If \( p^{23} \) is strong with respect to \( \text{sch}(R_2) \),

\[
(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3 = (R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3.
\]

\[(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2, R_1) \xRightarrow{R_2} R_3, \text{ by (4.35)}
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2 \xRightarrow{p^{23}} R_3, R_3), \text{ by (A.7)}.
\]

\[
(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2 \xRightarrow{p^{23}} R_3, R_3), \text{ by (4.35)}.
\]

Identity (4.50)

If \( p^{23} \) and \( p^{12} \) are strong with respect to \( \text{sch}(R_2) \),

\[
(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3 = (R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3.
\]

\[(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2, R_1, R_2) \xRightarrow{R_2} R_3, \text{ by (4.36)}
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2 \xRightarrow{p^{23}} R_3, R_2 \xRightarrow{p^{12}} R_3, R_1 \xRightarrow{p^{12}} R_2, R_1, R_2, R_3), \text{ by (A.8)}.
\]

\[
R_1 \xrightarrow{p^{12}} (R_2 \xrightarrow{p^{23}} R_3)
\]

\[
= R_1 \xrightarrow{p^{12}} \text{PresR}(R_2 \xRightarrow{p^{23}} R_3, R_2, R_3), \text{ by (4.36)}
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2 \xRightarrow{p^{23}} R_3, R_1 \xRightarrow{p^{12}} R_2, R_2 \xRightarrow{p^{12}} R_3, R_1, R_2, R_3), \text{ by (A.8)}.
\]

Identity (4.51)

If \( p^{23} \) is strong with respect to \( \text{sch}(R_2) \),

\[
(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3 = (R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3.
\]

\[(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3
\]

\[
= \text{PresR}(R_1 \xrightarrow{p^{12}} R_2, R_1, R_2) \xRightarrow{R_2} R_3
\]

105
by (4.36)

\[ R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) \]

= PresR(R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3, R_2 \overset{p^{12}}{\bowtie} R_3, R_1 \overset{p^{12}}{\bowtie} R_2, R_1, R_2), \text{ by (A.6).}

\[ R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) \]

= R_1 \overset{p^{12}}{\leftrightarrow} \text{PresR}(R_2 \overset{p^{23}}{\bowtie} R_3, R_2), \text{ by (4.35)}

= \text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3, R_1 \overset{p^{12}}{\bowtie} R_2, R_2 \overset{p^{12}}{\bowtie} R_3, R_1, R_2), \text{ by (A.8).}

Identity (4.53)

If \( p^{23} \) is strong with respect to \( \text{sch}(R_2) \),

\[ R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) = (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \text{ GOJ}[p^{23}, \text{sch}(R_1)] \text{ R}_3. \]

\[ R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) \]

= \text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3, R_1), \text{ by (4.35).}

\[ (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \text{ GOJ}[p^{23}, \text{sch}(R_1)] \text{ R}_3 \]

= \text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2, R_1) \text{ GOJ}[p^{23}, \text{sch}(R_1)] \text{ R}_3, \text{ by (4.35)}

= \text{PresR}(\text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2, R_1) \overset{p^{23}}{\bowtie} R_3, \pi_{\text{sch}(R_1)} \text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2, R_1)), \text{ by (4.52)}

= \text{PresR}(\text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2, R_1) \overset{p^{23}}{\bowtie} R_3, R_1), \text{ by (4.38)}

= \text{PresR}(\text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3, R_1), \text{ by (A.5)}

= \text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3, R_1), \text{ by (4.40).}

Identity (4.54)

If \( p^{23} \) is strong with respect to \( \text{sch}(R_2) \),

\[ R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) = (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \text{ GOJ}[p^{23}, \text{sch}(R_1)] \text{ R}_3. \]

\[ R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) \]

= \text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3, R_2 \overset{p^{23}}{\bowtie} R_3, R_1), \text{ by (4.36)}

\[ (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \text{ GOJ}[p^{23}, \text{sch}(R_1)] \text{ R}_3 \]

= \text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2, R_1, R_2) \text{ GOJ}[p^{23}, \text{sch}(R_1)] \text{ R}_3, \text{ by (4.36)}

= \text{PresR}(R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3, R_2 \overset{p^{23}}{\bowtie} R_3, R_1), \text{ by (4.52), (4.38), (A.5), and (4.40).}

Identity (4.55)

If \( p^{23} \) and \( p^{12} \) are strong with respect to \( \text{sch}(R_2) \),

\[ R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) = (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \text{ GOJ}[p^{23}, \text{sch}(R_1), \text{sch}(R_3)] \text{ R}_3. \]
\( R_1^{p^{12}} ( R_2^{p^{23}} R_3 ) = R_1^{p^{12}} \text{PresR}( R_2^{p^{23}} R_3, R_3 ), \text{ by (4.35)} \)
\( = \text{PresR}( R_1^{p^{12}} R_2^{p^{23}}, R_2^{p^{23}} R_3, R_1, R_3 ), \text{ by (A.8)} \).

\( (R_1^{p^{12}} R_2) \text{GOJ}[p^{23}, \text{sch}(R_1), \text{sch}(R_3)] R_3 = \text{PresR}( R_1^{p^{12}} R_2, R_1, R_2 ) \text{GOJ}[p^{23}, \text{sch}(R_1), \text{sch}(R_3)] R_3, \text{ by (4.36)} \)
\( = \text{PresR}( R_1^{p^{12}} R_2^{p^{23}} R_3, R_2^{p^{23}} R_3, R_1, R_3 ), \text{ by (4.52), (4.38), (A.5), and (4.40)} \).

Identity (4.56)

If \( p^{12} \) is strong with respect to a set of attributes \( A_s \), and \( A_s \) is disjoint with \( A_{21}, ..., A_{2n} \),

\( R_1^{p^{12}} ( R_2 \text{GOJ}[p^{23}, A_{21}, ..., A_{2n}, A_{31}, ..., A_{3m}] R_3 ) = ( R_1^{p^{12}} R_2 ) \text{GOJ}[p^{23}, \text{sch}(R_1), A_{21}, ..., A_{2n}, A_{31}, ..., A_{3m}] R_3 \).

\( R_1^{p^{12}} ( R_2 \text{GOJ}[p^{23}, A_{21}, ..., A_{2n}, A_{31}, ..., A_{3m}] R_3 ) = R_1^{p^{12}} \text{PresR}( R_2^{p^{23}} R_3 \pi A_{21} R_1, ..., \pi A_{2n} R_2, \pi A_{31} R_2, ..., \pi A_{3m} R_3 ), \text{ by (4.52)} \)
\( = \text{PresR}( R_1^{p^{12}} R_2^{p^{23}} R_3, \pi A_{21} R_1, ..., \pi A_{2n} R_2, \pi A_{31} R_2, ..., \pi A_{3m} R_3, R_1 ), \text{ by (A.8)} \).

\( (R_1^{p^{12}} R_2) \text{GOJ}[p^{23}, \text{sch}(R_1), A_{21}, ..., A_{2n}, A_{31}, ..., A_{3m}] R_3 \)
\( = \text{PresR}( R_1^{p^{12}} R_2, R_1, R_2 ) \text{GOJ}[p^{23}, \text{sch}(R_1), A_{21}, ..., A_{2n}, A_{31}, ..., A_{3m}] R_3, \text{ by (4.36)} \)
\( = \text{PresR}( R_1^{p^{12}} R_2^{p^{23}} R_3, \pi A_{21} R_1, ..., \pi A_{2n} R_2, \pi A_{31} R_2, ..., \pi A_{3m} R_3, R_1 ), \text{ by (4.52), (4.38), (A.5), and (4.40)} \).

107
Bibliography


