An ASF+SDF Specification of a Query Optimizer for a RDBMS

Karin Zaadnoordijk

May 27, 1994

Contents

1 Introduction 5
- 1.1 Problem Description . . . 5
- 1.2 Outline . . . 5
- 1.3 Acknowledgements . . . 5

2 Relational Databases 6
- 2.1 Components of a DBMS . . . 6
- 2.2 Relational Model . . . 6
  - 2.2.1 Data Structure . . . 7
  - 2.2.2 Data Manipulation . . . 10
  - 2.2.3 Data Integrity . . . 12
- 2.3 Relational Database . . . 13

3 ASF+SDF 14
- 3.1 The ASF+SDF formalism . . . 14
  - 3.1.1 Syntax . . . 14
  - 3.1.2 Semantics . . . 16
  - 3.1.3 Modules . . . 17
- 3.2 Global Structure of the Meta-environment . . . 18

4 Query Optimization 21
- 4.1 Basic Types of Optimization . . . 21
- 4.2 Phases of Optimization . . . 22
- 4.3 Structure and Representation of our Specification . . . 23
  - 4.3.1 Structure . . . 23
  - 4.3.2 Representation . . . 24

5 Check 28
- 5.1 Undefined Relations . . . 29
- 5.2 Union-compatible Relations . . . 30
- 5.3 Use of Undefined Attributes . . . 31
  - 5.3.1 Undefined Attributes in Predicates . . . 31
  - 5.3.2 Undefined Attributes in Projectlists . . . 32
- 5.4 Normalized Relations . . . 33

6 **Equivalence Transformations** 34
6.1 Example . . . 34
6.2 Graphical Representation . . . 36
6.3 Processing of Selects . . . 36
6.3.1 Select over Union, Intersection, and Minus . . . 37
6.3.2 Select over Product . . . 38
6.3.3 Select over Join . . . 40
6.3.4 Select (cascade) . . . 40
6.3.5 Union, Intersection, and Minus over Select . . . 41
6.4 Processing of Projects . . . 42
6.4.1 Project over Union, Intersection, and Difference . . . 42
6.4.2 Project over Join . . . 43
6.4.3 Project (cascade) . . . 44
6.4.4 Project over Select . . . 45

7 **Access Strategies** 47
7.1 Physical Data Organization . . . 47
7.2 Rules for Factorisation . . . 48
7.3 Choice of Procedures for Select . . . 50
7.3.1 Used Procedures . . . 53
7.3.2 Examples . . . 53

8 **Extensions / Further work** 56

9 **Conclusions** 57

A 58
A.1 Access . . . 58
A.2 Check . . . 61
A.3 Eqtransforms . . . 75
A.4 Factorisation . . . 82
A.5 Operations . . . 83
A.6 Optimizer . . . 85
A.7 Relations . . . 88

List of Figures

3.1 Example of a module editor 18
3.2 Example of a term editor 19
3.3 The generated error message 19
3.4 A syntactically correct term (focussed) 20
3.5 Syntax-directed editing of a term 20
3.6 Term editor of the module Optimizer 20

4.1 Phases of Query Optimization 22
4.2 Import Graph of the Modules 24

6.1 Symbols used in Graphical Representation 36
6.2 The select over union example before and after transformation. 37
6.3 The select over product example before and after transformation. 39
6.4 The select over join example before and after transformation. 40
6.5 The select-cascade example before and after transformation 41
6.6 The union over select example before and after transformation. 42
6.7 The project over join example before and after transformation. 44
6.8 The project over select example before and after transformation. 46

7.1 Example 1, choice of procedures of select operation 54
7.2 Example 2, choice of procedures of select operation 55

List of Tables

2.1 STUDENT relation 8
2.2 COURSE relation with (columnar) repeating groups 8
2.3 COURSE relation without (columnar) repeating groups 9
2.4 The SELECTION operation 11
2.5 The PROJECTION operation 12
2.6 The JOIN operation 12

3.1 Keywords used in ASF+SDF Modules 17

Chapter 1

Introduction

1.1 Problem Description

1.2 Outline

Chapter 2 and 3 discuss the essentials of relational databases and the characteristics and usage of the ASF+SDF Meta-environment respectively. These chapters are included for readers that are not familiar with those subjects. A description of the general structure of a query optimizer is given in Chapter 4 by describing the three phases in query optimization. Also included in this chapter are the module structure of the specification and a description of the chosen representation of the queries. Chapters 5 through 7 give a more elaborate description of each of the phases of optimization seperately. Extensions, further work and conclusions can be read in Chapters 8 and 9. An appendix is included, which contains the complete specification.

1.3 Acknowledgements

Chapter 2

Relational Databases

2.1 Components of a DBMS

DataBase Management Systems (DBMSs) are highly complex, sophisticated pieces of software which aim at providing reliable management of shared data. It is not possible to generalize the component structure of a DBMS since this will vary considerably from system to system. However it is possible to identify a number of key functions which are common to all DBMSs and these will be discussed below.

At the heart of the DBMS lies the *system catalogue* or data dictionary, with which virtually all other components of the system interact. The catalogue contains all meta-data for the system, that is, a description of all data items (type, length, allowable values, etc), access permissions and so on.

The *query processor* is responsible for accepting commands from users to store, retrieve, and update data in the database. Using the information stored in the catalogue, it interprets these requests and translates them into physical data access requests which are passed to the operating system for processing. The efficiency of query processing is the subject of this thesis.

The process of retrieving and updating data by a user is called a transaction. A conventional DBMS is generally assumed to support many users "simultaneously" accessing the database (i.e. many transactions are active at the same time). It is the responsibility of the *transaction manager* of the DBMS to ensure that transactions do not interfere with each other and corrupt the database.

One of the most important aims of database technology was to provide better backup and recovery facilities than were provided by traditional file processing systems. Nothing is 100% reliable; systems must be designed to tolerate failures and recover from them correctly and reliably. It is the role of the *recovery manager* of the DBMS to minimize the effects of failures on the database. There are many different causes of failure in a database environment such as application program errors (e.g. trying to divide by zero), disk head crashes, power failures, and operator errors. The purpose of the recovery manager is mainly to restore the database into a state known to be consistent.

2.2 Relational Model

Classification of database management systems is generally based on how data is represented in the conceptual scheme (i.e. the conceptual data model). There are three main approaches:

1. relational;
2. network;
3. hierarchical.

The most important model is the relational one, and since it is used througout this thesis the discussion here will concentrate on such systems.

The relational model is a template for how data is presented to the user, how a user performs operations on data, and how data behave when they are manipulated. The relational model was formally introduced by Codd in 1970 (See [Fle89]).

The relational model has its roots in mathematical set theory. Furthermore, the relational model is a step towards simplicity. Simplicity implies that the model can be described using a few, familiar concepts. It is precisely this combination of simplicity and solid theoretical foundation that makes the relational model so valuable and long-lasting. The simplicity allows designers, developers, and users to communicate using terms and concepts understood by all. The solid theoretical foundation guarantees that results of relational requests are well-defined and, therefore, predictable.

The relational model consists of three components:

1. *data structure* — organization of the data, as users perceive them;
2. *data manipulation* — types of operations users perform on the relational data structure;
3. *data integrity* — set of rules that govern how relational data values behave when users perform relational operations.

2.2.1 Data Structure

Relational data are organized by relations. Data is viewed through two-dimensional structures known as *tables* or *relations*. Each table has a fixed number of columns called *attributes*, and a variable number of rows called *tuples*. Each named column is associated with a domain, where the domain (described in more detail in Section 2.2.3) is a specification of values that may appear in a column. Table 2.1 depicts a relation containing STUDENT-information. This relation is represented as having 2 columns (attributes) and 7 rows (tuples).

Relations have six special properties to distinguish them from non-relational (or partially relational) tables:

1. Entries in attributes are single-valued;
2. Entries in attributes are of the same type;
3. Each tuple is unique;
4. The order of attributes (left to right) is insignificant;
5. The order of tuples (top to bottom) in insignificant;
6. Each attribute has a unique name.

STUDENT

NUMBER	NAME
9012578	J. Murphy
8917788	A. Mahon
8991023	P. Farrell
9010110	P. White
9178912	M. Anderson
8811287	T. Robinson
8966453	M. Brennan

Table 2.1: STUDENT relation

COURSE

NUMBER	COURSE
9012578	database-management, algebra
8917788	database-management, expertsystems, algebra
8991023	algebra, software-engineering

Table 2.2: COURSE relation with (columnar) repeating groups

Property 1: Entries in attributes are single-valued. This property implies that attributes do not contain repeating groups. Often such tables are referred to as "normalized" or as being in "first normal form (1NF)". It is important to understand the significance and effects of this property because it is a cornerstone of the relational data structure.

Suppose we have a relation, COURSE, containing information about which students attend which courses. A student may attend more than one course. This information can be represented in several ways; however, not all representations are consistent with the single-valued property for relations.

One way is to require that each student is represented in the relation by one tuple. We can then define an attribute in which you place all courses of a student. Table 2.2 shows such a table. Note that multiple courses of a student appear as multiple values in the course attribute.

This approach is not consistent with the single-valued property of a relation, because the course attribute contains repeating groups, which is undesirable because it complicates data manipulation logic. Consider, for example, the complex logic of searching for student-names who attended the database-management course. The access language must offer the user possibilities to specify a search for a given character string (database-management) occurring anywhere within an attribute (course), or the DBMS must be able to detect automatically when such a search is required. Some relational operations (such as union, described later in this chapter) might not be possible using this table layout.

Another approach is to restrict the course attribute to a single course. In this case, multiple courses for a student are represented by multiple tuples in a relation. Table 2.3 illustrates this concept. This approach defines a relation that fulfills the single-valued property.

The basic difference between the first and the second approach is that the first represents repeating groups across attributes, whereas the second one represents repeating groups across

COURSE

NUMBER	COURSE
9012578	database-management
9012578	algebra
8917788	database-management
8917788	expertsystems
8917788	algebra
8991023	algebra
8991023	software-engineering

Table 2.3: COURSE relation without (columnar) repeating groups

tuples. On initial inspection, Table 2.3 may seem less intuitive. However, the absence of columnar repeating groups simplifies expression and evaluation of relational operations.

Property 2: Entries in attributes are of the same type. In relational terms, this property states that all values in a given attribute are taken from one domain. This property is significant and useful. First it simplifies data access because users can make assumptions about the type of data contained in a given attribute. Furthermore, this property simplifies data validation, because all values in any given attribute are of the same domain.

This property, in conjunction with the first property, gives a relation a very stable structure. It implies that every tuple in a relation has the same "shape": every tuple has the same number of attributes, and each attribute contains a value from the same domain.

Property 3: Each tuple is unique. This property ensures that no two tuples in a relation are identical; there is at least one attribute (or, set of attributes) which has values that uniquely identify specific tuples in the relation. As an example, the NUMBER attribute in the STUDENT relation of Table 2.1 distinguishes one student from another. Such attributes are called *primary keys*.

This property guarantees that every tuple in a relation is meaningful. Users can refer to a particular tuple by specifying the primary key value. The primary key in the relation of Table 2.3 consists of a set of two attributes (NUMBER and COURSE) because neither, by itself, identifies a unique tuple.

Property 4: The order of attributes (left to right) is insignificant. This property ensures that no hidden meaning is implied by the order in which attributes occur. Moreover, because there is no information implied by a particular sequence of attributes, a user can retrieve the attributes in any order. An obvious benefit is that the same relation can be shared by many users and can serve a multitude of access requirements. Also, designers are free to change the sequence in which attributes are physically stored (perhaps for performance reasons) without affecting the meaning of data or the formulation of queries by users.

Property 5: The order of tuples (top to bottom) is insignificant. This property is analogous to property 4, but it applies to tuples instead of attributes. The obvious benefit is the ability to retrieve tuples of a relation in any sequence. Thus, the same relation can be

shared among users, even when the users wish to view the tuples in different sequences. Furthermore, designers are free to modify the order in which tuples are physically stored (perhaps for performance reasons) without affecting the meaning or the users' perceptions of the data. Whenever a logical sequence of tuples is useful for a particular access requirement, we have to ensure that the values of one or more attributes can explicitly convey this sequencing.

Property 6: Each attribute has a unique name. Think of this property as an extension of property 4; that is, because the sequence of attributes is insignificant, attributes are referenced by name and not by position. In general, a attribute name need not be unique within an entire DBMS environment or even within a given database. However, typically the attribute name must be unique within the relation in which it appears.

2.2.2 Data Manipulation

The second component for the relational model is data manipulation, i.e. types of operations that users can perform on relations. There are two basic types of operations: assignment of relations to other relations (called relational assignment) and manipulation of relations using relational operators.

The concept of relational assignment is similar to the concept of assignment statements in a program: the result of an expression is assigned to a variable. The variable in a relational assignment is a relation, and the expression involves relations and relational operations. Thus, we can perform operations on relations and assign the result to another relation.

It is more interesting to look at the (seven) relational manipulation operators themselves. Before we discuss them, observe that they have two characteristics in common. First, relational operators are applied to and result in relations (i.e. sets of tuples and attributes). This is known as *set* processing, to be distinguished from *record-at-a-time* (or row-at-a-time) processing. A formal name for this set-to-set processing characteristic is *closure*, meaning that relational operations are closed within the universe of relations. In other words, a relational operation on one or more relations always produces another relation. Thus, result relations have the six properties mentioned above as well. Closure implies that relational operators can be applied to results of "previous" relational operations. Hence, we can nest series of relational operations.

The second common characteristic is that relational operators are unaffected by how the data are stored physically. Thus, the seven relational operators express functionality without concern for (or knowledge of) technical implementation. An obvious benefit is that we can apply relational operators without concern for storage and access techniques.

The relational operators can be divided into two groups: the traditional set operators union, intersection, difference, and product (all are somewhat modified to operate on relations rather than arbitrary sets); and the special relational operators selection, projection, and join.

Traditional Set Operations

The traditional set operations are union, intersection, difference, and product. For all, except product, the two operand relations must be *union-compatible*: the tuples in the operand relations must have the same number and type of attributes. To keep things simple we have chosen to insist that attributes that need to be unified, intersected, or differentiated must have the same name.

Union: The union of two (union-compatible) relations A and B is the set of all tuples t belonging to either A or B (or both).

SELECT FROM STUDENT WHERE NUMBER > 9000000

NUMBER	NAME
9012578	J. Murphy
9010110	P. White
9178912	M. Anderson

Table 2.4: The SELECTION operation

Intersection: The intersection of two (union-compatible) relations A and B is the set of all tuples t belonging to both A and B.

Difference: The difference between two (union-compatible) relations A and B (in that order) is the set of all tuples t belonging to A and not to B.

Product: The (Cartesian) product of two relations A and B is the set of all tuples t such that t is the concatenation of a tuple a belonging to A and a tuple b belonging to B. The concatenation of a tuple $a = (a_1,a_2,\ldots,a_m)$ and a tuple $b = (b_{m+1},b_{m+2},\ldots,b_{m+n})$ — in that order — is the tuple $t = (a_1,\ldots,a_m,b_{m+1},\ldots,b_{m+n})$.

The operations union, intersection, and difference are associative, (Cartesian) product is not.

Special Relational Operations

Selection: The algebraic selection operator (not to be confused with the SQL SELECT) yields a "horizontal" subset of a given relation. This subset of tuples within the given relation satisfies the specified predicate. The predicate is expressed as a boolean combination of terms, each term being a simple comparison that results in true or false for a given tuple by inspecting that tuple in isolation. (If a term involves a comparison between values of two attributes within the tuple, then those attributes must be defined within the same domain). Table 2.4 shows an example of the selection operation, the tuples from the STUDENT-relation (Table 2.1) of which the NUMBER-attribute is greater than 9000000 are selected.

Projection: The projection operator yields a "vertical" subset of a given relation, the subset obtained by selecting specified attributes, and then eliminating duplicate tuples within the attributes selected. Table 2.5 shows an example of the projection operation. In this example the COURSE-attribute of the COURSE-relation (Table 2.3) is projected.

Join: Two relations may be joined if each of them have an attribute drawn from a common domain. The resulting relation contains a set of tuples, each of which is formed by the concatenation of a tuple from one relation with a tuple from the other relation such that they have the same value in the common domain. Actually this definition corresponds to only one of many possible joins — namely the join in which the "joining condition" is based on equality between values in the common column. This join is known as *equi-join.* It is also possible to define, for example, a "greater-than" join, a "not-equal" join, and so on — though the equi-join is used most frequently. The result of the equi-join operation necessarily contains two identical

PROJECT COURSE OVER COURSENAME

COURSE
database-management
algebra
expertsystems
software-engineering

Table 2.5: The PROJECTION operation

JOIN STUDENT AND COURSE OVER NUMBER

STUDENT.NR	STUDENT.NAME	COURSE.NR	COURSE.COURSE
9012578	J. Murphy	9012578	database-management
9012578	J. Murphy	9012578	algebra
8917788	A. Mahon	8917788	database-management
8917788	A. Mahon	8917788	expertsystems
8917788	A. Mahon	8917788	algebra
8991023	P. Farrell	8991023	algebra
8991023	P. Farrell	8991023	software-engineering

Table 2.6: The JOIN operation

columns. Table 2.6 shows an example of the (equi-)join operation. It is of course possible to eliminate one of those two columns via the project operation; an equi-join with one of the two identical columns eliminated is called a *natural join.*

2.2.3 Data Integrity

The third component of the relational model is data integrity. Meaningful data within relations must comply with certain integrity rules. These integrity rules constrain permissible values within the attributes of relations. Data may easily contain incorrect, incomplete, or misleading values without such constraints. There are two general recognized rules for data integrity in the relational model: the entity integrity rule and the referential integrity rule. Furthermore, there are miscellaneous integrity rules. For ease of discussion, we classify these miscellaneous rules into a third general rule, domain integrity.

The *entity integrity rule* dictates that no component of the primary key (the attribute or set of attributes which values uniquely identify particular tuples) accepts null values. A null value implies that the value for a given attribute has not been supplied — it is either unknown or inappropriate. A primary key always identifies a unique tuple in a relation, thus its value must be appropriate and should never be unknown.

The *referential integrity rule* addresses integrity of foreign keys. A foreign key is an attribute or set of attributes in a relation that serves as a primary key in another relation. For example, the NUMBER-attribute in the COURSE-relation is a foreign key because its values refer to values in the NUMBER-attribute of the STUDENT-relation where the values function as a primary key. The referential integrity rule states that, if a relation has a foreign key, then every

value of that foreign key is either null or matches values in the relation for which that foreign key is a primary key.

We use the term *domain integrity* to describe the integrity rules for all attributes in a relation, including primary keys and foreign keys. Formally, a *domain* is a logical pool of values from which one or more attributes draw their values. We interpret domain rules to include rules for data type, length, range-checking, default values, uniqueness, nullability, and so on. Entity integrity and referential integrity are special cases of domain integrity. Yet, because they are so important, they are distinguished as separate rules.

2.3 Relational Database

Although the relational model is an intellectual concept, a relational database is an implementation of that concept using DBMS technology. As the words imply, a "relational database" inherits two sets of characteristics, one defining its *relational* aspect and another defining its *database* meaning.

We are now in a position to comprehend the relational aspect. A *relational* database comprises data that appear and behave to the user according to rules of the relational model. Users perceive the data as a set of relations that obey the six properties of relations, are manipulated via the seven relational operators or their equivalent, and are protected by the relational integrity rules. Because the relational model is well-defined, this relational aspect should be consistent across relational products.

The database aspect of relational databases is even more confusing. There is no universal definition of a relational *database* except that it is a computerized structure for storing data that the user regards as relations (tuples and attributes). Beyond that concept, every relational DBMS product relies on its own interpretation of what a database is.

Chapter 3

ASF+SDF

The ASF+SDF Meta-environment [Kli93] is an interactive system for manipulating programs, specifications, or other texts. The formalism used in the Meta-environment is ASF+SDF [HHKR89, Hen89]. It is a combination of the Algbraic Specification Formalism (ASF) [BHK87, Hen88, BHK89] and the Syntax Definition Formalism (SDF) [HHKR89]. ASF is a formalism that supports modularization and conditional equations. SDF has been developed to support the definition of lexical and context-free syntax.

The Meta-environment generates both scanners and parsers from ASF+SDF specifications as well as term rewriting systems, thus allowing the execution of ASF+SDF specifications. Moreover, it provides syntax-directed editors for modules in the specification, and generates language specific syntax directed editors. The system is able to perform several static semantic checks on the specifications, and supports the testing of specifications. It can be used to specify arbitrary (programming) languages. This means that both *syntax* and *semantics* of a language can be defined by this system.

3.1 The ASF+SDF formalism

3.1.1 Syntax

The syntax of a language is defined by the *lexical* and *context-free syntax*, which are sets of grammar rules. The lexical syntax describes the low level structure of sentences (sequences of characters) in terms of lexical tokens. A lexical token is a pair consisting of a *sort name* and a *lexeme*. The former is used to distinguish classes of lexical tokens, such as identifiers and numbers. The latter is the actual text of the token. The lexical syntax also defines which substrings of the sentence are layout symbols or comments. A typical layout definition is:

```
module Layout
exports
   lexical syntax
      "%%"~[\n]* [\n]  ->  LAYOUT
      [ \t\n]          ->  LAYOUT
end Layout
```

A space, tab or newline is a layout symbol, as well as everything between double percent signs and a new line. If tokens of sort LAYOUT are detected in a text, they are ignored.

The context-free syntax describes the concrete and abstract structure of sentences in a language. The definition of context-free syntax in an SDF definition consists of declarations of *context-free functions*. In their simplest form, context-free functions are declared by giving their syntax (a list consisting of zero or more literal symbols and/or names of sorts) and their result sort.

As an example we will have a closer look at an important feature of the ASF+SDF formalism: the list functions. Lists have a variable number of arguments. Suppose we would like to have a function `{ }` for the empty set, `{ E1 }` for a set with one element, `{ E1, E2 }` for a set with two elements, and so on. The way to define this in ASF+SDF is:

```
sorts ELEMENT SET
context-free functions
   "{" {ELEMENT ","}* "}"   ->  SET
```

The asterisk `*` indicates that the set may contain zero or more `ELEMENTS`, the comma is used as a separation between the `ELEMENTS`. Thus, a set consists of `ELEMENTS`, separated by commas and delimited by { and }.

This list notation is simply an abbreviation for the declaration of infinitely many functions { ... }, each with a different number of arguments. Likewise, the same (concrete) syntax could have been obtained without using lists by the following:

```
sorts ELEMENT ELEMS ELEMENTS SET
context-free functions
   "{" ELEMENTS "}"       -> SET
                          -> ELEMENTS
   ELEMS                  -> ELEMENTS
   ELEMENT                -> ELEMS
   ELEMENT "," ELEMS      -> ELEMS
```

In order to define equations over list functions, we need list variables:

```
variables
   Elts[123]     -> {ELEMENT ","}*
   i             -> ELEMENT
```

Here we defined `Elts1`, `Elts2` and `Elts3` to be list variables, ranging over zero or more `ELEMENT`s separated by commas. Variable `i` has been defined as a single `ELEMENT`.

To complete this example we will give an equation. Equations are described in more detail in the next section.

```
equations
   [eq1] {Elts1,i,Elts2,i,Elts3} = {Elts1,i,Elts2,Elts3}
```

Here we have specified in one single equation that elements of sets do not appear more than once. Any set containing element `i` at least two times is equal to the set containing one ocurrence less of `i`. This equation uses implicit backtracking, whereas in the alternative way of defining the `SET` syntax without using lists the backtracking is to be specified explicitly (we assume all variables have been defined correctly):

```
context-free functions
   remove ( SET )                 -> SET
```

```
    add ( ELEMENT , SET )          -> SET
    \\ Booleans will be examined later
    member-of ( ELEMENT , ELEMS ) -> BOOL

equations
    [r1] remove({}) = {}

    [r2] remove({element}) = {element}

    [r3] member-of(element,elems) = false
         ================================
         remove({element,elems}) = add(element,remove{elems})

    [r4] member-of(element,elems) = true
         ===============================
         remove({element,elems}) = remove({elems})


    [a1] add(element,{}) = {element}

    [a2] add(element,{elems}) = {element,elems}


    [m1] element1 != element2
         ====================
         member-of(element1,element2) = false

    [m2] member-of(element,element) = true

    [m3] element1 != element2
         ====================
         member-of(element1, element2,elems) = member-of(element1,elems)

    [m4] member-of(element, element,elems) = true
```

3.1.2 Semantics

ASF+SDF allows the use of rewrite rules, also called *equations*, to define the semantics. In the last section we already saw several examples of equations to remove multiple occurrences of elements in a set. Algebraic specifications become more powerful if *conditional equations* are used. We have already seen conditional equations in the specification of the last section. Equations [r3] and [r4] are examples of equations with *positive* conditions; equations [m1] and [m3] are examples of equations with *negative* conditions. The idea of conditions is that the consequence (below the bar) only holds if both sides of all conditions (above the bar) can be proved equal or unequal.

To facilitate the description of equations having an if-then-else like character, ASF+SDF supports the *default-equation* construct. Using this construct, the `member-of` function for the example using list functions can be defined as follows:

KEYWORD	FUNCTION
imports	defines the names of imported modules
exports	defines the names of exported items of this module
hiddens	defines the names of hidden items in this module
sorts	lists the sorts defined in this module
lexical syntax	defines the lexical syntax
context-free syntax	lists the function declarations
priorities	lists the priorities between function
variables	declaring the variables used in the equation
equations	listing the (conditional) equations of the module

Table 3.1: Keywords used in ASF+SDF Modules

```
[1] member-of(i,[Elts1,i,Elts2] = true

[default-2] member-of(i,[Elts1]) = false
```

The default-equation construct is merely an abbreviation, since it can be rewritten to a number of positive conditional equations.

3.1.3 Modules

A module in the ASF+SDF formalism may contain a number of keywords. They are listed in Table 3.1.

The use of the import/export construct makes writing specifications more well-organized: large specifications can be split up in parts and the reusability of modules increases. An example of an ASF+SDF module is:

```
module Booleans
imports Layout
exports
   sorts BOOL
   context-free syntax
      true                    -> BOOL
      false                   -> BOOL
      BOOL "|" BOOL           -> BOOL {left}
      BOOL "&" BOOL           -> BOOL {left}
      not "(" BOOL ")"        -> BOOL
      "(" BOOL ")"            -> BOOL {bracket}
   variables
      Bool [0-9']*            -> BOOL
priorities
   "|" < "&"
equations
   [B1]    true | Bool  = true
   [B2]    false | Bool = Bool
```

```
    [B3]    true & Bool  = Bool
    [B4]    false & Bool = false

    [B5]    not(false)   = true
    [B6]    not(true)    = false
end Booleans
```

This module imports the Layout module. The Layout module defines layout-symbols (white space and comment recognition) using the designated sort LAYOUT.

The syntax of the Boolean language is defined by introducing a sort BOOL, which contains two constants `true` and `false`. Furthermore, the standard Boolean functions and (represented as &), or (represented as |) and not are defined as well as parentheses. The attribute `left` declares & and | as left-associative functions and the priorities section of the definition defines function grouping when no parentheses are present. Variable `Bool` has been defined of sort BOOL and is used in the equations part. The equations define equalities on Boolean terms.

3.2 Global Structure of the Meta-environment

The ASF+SDF Meta-environment is a collection of Generic Syntax-directed Editors (GSE's). For each module a combination of two GSE editors, one for the ASF-part and one for the SDF-part, can be invoked. We call such an editor a *module editor*, and is illustrated by Figure 3.1.

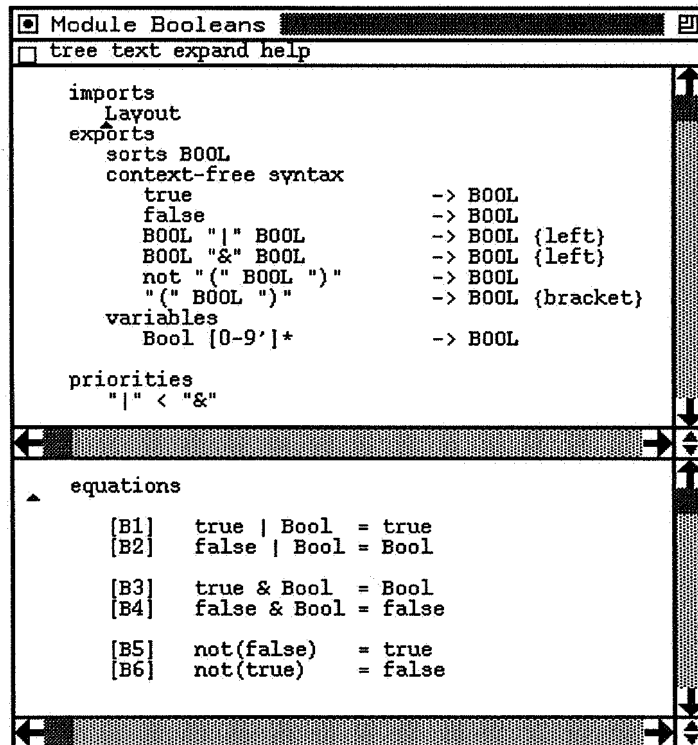

Figure 3.1: Example of a module editor

By editing one of the modules the complete specification may be changed. Changing the syntax part of a module has implications for all term editors (for an example of a term editor see Figure 3.2) depending on the module and also for all modules that import the current module. Lazy and incremental program generation techniques take care of automatic updating of scanner, parser, and rewriting system, if a module is modified.

The syntax and semantics of a module can be checked by invoking a *term editor* to create and evaluate terms over the language defined by the module. In a term editor ordinary textual operations can be performed, like cut and paste. As an illustration Figure 3.2 shows an arbitrary manually edited boolean expression.

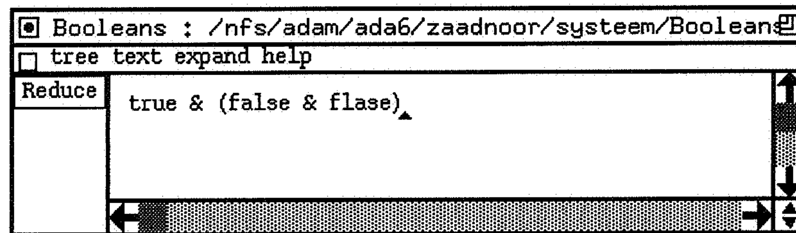

Figure 3.2: Example of a term editor

This term is syntactically checked. If the term is syntactically incorrect, the Meta-environment will generate an error-message, which will be the case for the boolean expression. The generated error-message is shown in Figure 3.3.

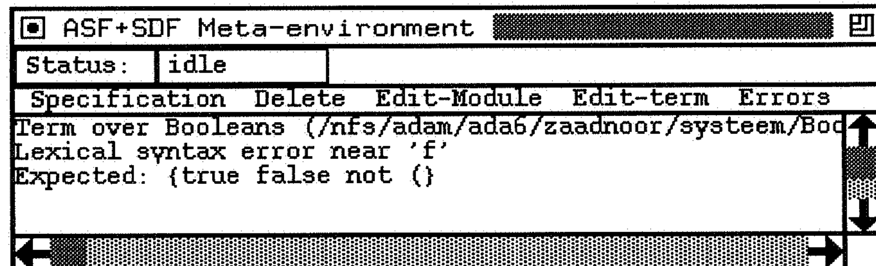

Figure 3.3: The generated error message

After correction of `flase` into `false`, the syntax check is performed again to see if the term is correct. This time the system generates no error message, but instead, to indicate the expression's correctness, a solid line is placed around the complete term. This is also called a *focus* on the text (See Figure 3.4).

Besides text editing facilities GSE offers also structure editing facilities, which are strongly based on the syntax of the programs being edited. This form of editing is also called
em syntax directed editing The previous boolean expression can then be constructed, as illustrated by Figure 3.5.

When the focus is positioned at a meta-variable, the **expand** menu of GSE contains all possible expansions of the meta-variable. These expansions may, again, contain meta-variables. As can be seen a meta-variable is written as the (in this case only possible) sort BOOL, surrounded by "<" and ">". Both meta-variables are replaced by the terminal `false`, chosen from the **expand** menu, and the original expression is obtained again.

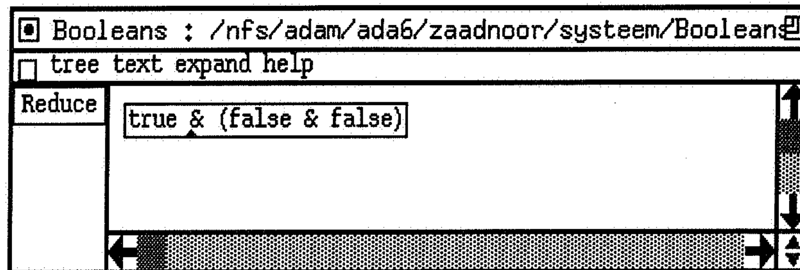

Figure 3.4: A syntactically correct term (focussed)

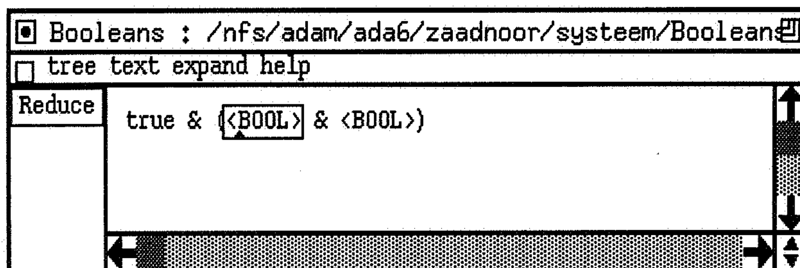

Figure 3.5: Syntax-directed editing of a term

Term editors can be customized by adding buttons, whose activation starts the evaluation of some specified function. In this way the applications' user-interface can be customized by adding, for instance, a type checking or evaluation button to a term editor. As an illustration Figure 3.6 shows a term editor of the module `Optimizer`, to which several user defined buttons are added.

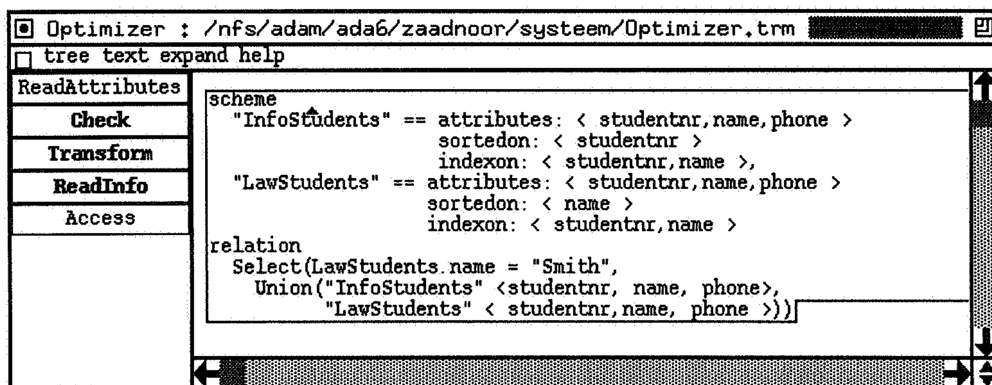

Figure 3.6: Term editor of the module Optimizer

Chapter 4

Query Optimization

Designing a query optimizer is a difficult and important task that has already received considerable attention ([Kim85, Garda89]). The efficiency of the query optimizer conditions the performance of the DBMS when performing high-level queries and thus its usability. The first Relational Database Management Systems (RDBMSs) were not so successful because they had a bad performance. The development of query optimizing methods have made RDBMSs popular.

This chapter describes the basic types of optimization, followed by a description of the phases of optimization. At the end of this chapter we show the structure of our specification.

4.1 Basic Types of Optimization

Query processing techniques are usually compared on the basis of the costs of query execution (execution cost) or on the basis of time used between the submission of a query and the receipt of the reply (response time).

Response times can be considered essential to an organisation, for example, missed sales or other similar opportunities due to poor response times are clearly undesirable. However, here we will only consider the response times of the system to the query.

Execution cost optimizers aim to minimize the use of the (total) system resources for a query and hence reduce its cost. The value of the objective function for the optimization process is therefore obtained from the sum of expressions representing the consumption of the individual system resources.

The most critical resources utilized in query evaluation in a centralized database are:

- CPU;
- I/O channels.

I/O channel usage contributes to the cost of loading data pages from secondary to primary storage and it is compounded by the cost of occupying secondary storage for temporary relations and buffers. Channels, intermediate storage, and buffers may each be a bottleneck in a system. Bottlenecks occur whenever there is more demand for resources than can be met at a particular time. CPU costs are often assumed to be low or even negligible in database systems, but they can be relatively high for complex queries.

Response time optimizers aim at minimizing the response time for a query by minimization of the time requirements of the query optimization process and the query execution. Heuristic

techniques are used to avoid excessive time consumption in the optimization process. The produced query programs based on these methods are often not optimal, whereas the total response times still may almost be optimal because heuristic optimization is fast. There exists no way to know whether the optimal query construction has been achieved, except for very simple queries.

It is easy to see that there could be a difference between the 'fastest' and the 'cheapest' execution strategies in particular cases.

4.2 Phases of Optimization

The problem of query optimization can be decomposed into several subproblems corresponding to various phases. These problems are not independent since decisions made for one subproblem may affect those made for another. However, to make this complex problem tractable, they are generally treated independently in a top-down fashion. In Figure 4.1, we give three phases for query optimizing; each of these phases solves a well-defined problem.

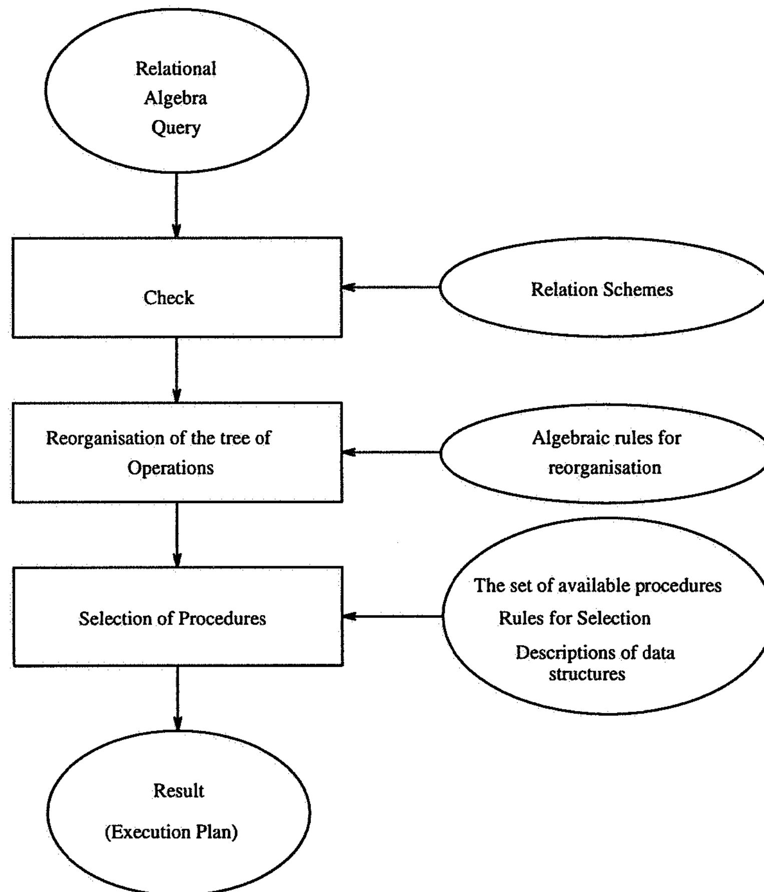

Figure 4.1: Phases of Query Optimization

In the first phase the structure of the query is checked. Tests that are performed here are:

- attributes used in the predicates of the selection operations must be element of the operand relation;
- attributes that are projected upon by the project operations must be element of the operand relation;
- join-attributes that are used for joining must be members of both operand relations.

The second phase consists of finding an optimal (or semi-optimal) ordering of the relational operations. A rule of thumb used in this optimization phase: *'projections and selections should be performed early, and joins (and other binary operations) late'*. This rule applies even more to distributed systems because large join sizes are undesirable for transfers between distributed sites. A detailed description of this phase can be found in Chapter 6.

The third phase consists of selecting the access strategies for the operations. This choice depends on the set of available procedures for the operations, the availability of indices, and the sort order of tuples. The output of the third phase is an execution plan, this is an optimized program of low-level (database) operations corresponding to the input-query. For a detailed description of the selection of access strategies see Chapter 7.

4.3 Structure and Representation of our Specification

In this section we describe the structure and representation of our specification. We have given the complete specification in the appendices, here we give a more abstract description of it.

4.3.1 Structure

The description of the structure of our specification is given to show the relation between the different phases of optimization given in Section 4.2 and the modules in the specification.

Our specification consists of several modules. The import graph representing the import of modules is given in Figure 4.2 in which M1 $\rightarrow$ M2 means: M1 is imported by M2 or, equivalently, M2 imports M1.

The modules Layout, Booleans, Integers, and Strings are standard modules and their functions are obvious (a detailed description of the Layout-module can be found in Section 3.1.1, the Boolean-module has been discussed in Section 3.1.3).

In the Relations-module we have defined the following:

```
[A-Za-z][A-Za-z0-9.]*                -> ATTRIBUTE
"<" {ATTRIBUTE ","}* ">"             -> ATTRIBUTELIST
STRING                               -> RELNAME
RELNAME ATTRIBUTELIST                -> REL
REL ATTRIBUTELIST ATTRIBUTELIST -> EXTREL
```

This definition tells us that an attribute is a string of letters digits and a point, beginning with a letter. An ATTRIBUTELIST is a list consisting of attributes separated by commas and delimited by < and >. A REL (relation) has been defined as something that consists of a RELNAME (a string) followed by an ATTRIBUTELIST. The last sort defined in the above definition is a EXTREL (an extended relation). This extended relation is a relation with extra information concerning the existence of sort orders of tuples and the availability of indices.

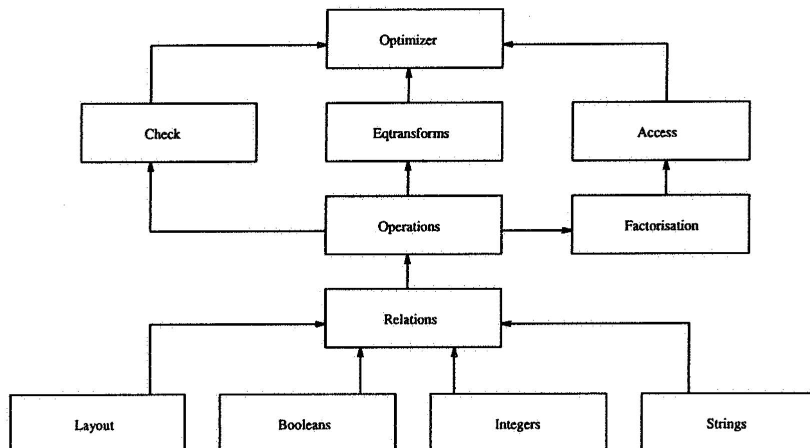

Figure 4.2: Import Graph of the Modules

We also defined a number of functions that operate on attributes and attributelists. These functions can be used by modules that (directly or indirectly) import the Relation-module.

In the Operations-module the traditional set operations (Union, Intersection, Difference, and Product) and the special relational operations (Selection, Projection, and Join) are defined. These definitions use the sorts defined in the Relations-module, that is imported by the Operations-module.

A detailed description of the Check-module, the Eqtransforms-module, and the Access-module can be found in subsequent chapters.

4.3.2 Representation

In this section we describe how queries are represented in our specification. In the following example we show how to define relations and how to structure a query:

```
scheme
   "InfoStudents" == attributes: <studentnr,name,phone>
                     sortedon: <name>
                     indexon: <studentnr>,
   "LawStudents" == attributes: <studentnr,name,phone>
                    sortedon: <name>
                    indexon: <studentnr>,
   "Members" == attributes: <studentnr,sport>
                sortedon: <>
                indexon: <studentnr>
query
   Select(studentnr > 9000000,
          Union("InfoStudents","LawStudents"))
```

The scheme-part of this example defines extended relations. It consists of a number of relation-definitions (RELDEF's). Each relation-definition consists of a relation-name (RELNAME), followed

by a double '==' sign, followed by the (extended) relation information. This information consists of three parts:

1. the string "attributes:" followed by an ATTRIBUTELIST;
2. the string "sortedon:" followed by an ATTRIBUTELIST;
3. the string "indexon:" followed by an ATTRIBUTELIST.

Relation definitions are separated by commas. The sorts RELNAME and ATTRIBUTELIST are defined in the Relations-module (see Section 4.3.1).

The query-part consists of a single query in which the relations defined in the scheme-part are used. A query is an expression over set operations (Union, Intersection, Minus, and Product) and/or relational operations (Select, Project, and Join) as defined in Section 2.2.2. The syntax definitions used are:

```
"scheme" SCHEME "query" QUERY                     -> REPRESENTATION1
"scheme" SCHEME "relation" REL                    -> REPRESENTATION2
"ReadAttributes" "(" REPRESENTATION1 ")"          -> REPRESENTATION2
"ReadInfo" "(" REPRESENTATION2 ")"                -> EXTREL
{RELDEF "," }+                                    -> SCHEME
RELNAME "==" "attributes:" ATTRIBUTELIST
             "sortedon:"   ATTRIBUTELIST
             "indexon:"    ATTRIBUTELIST          -> RELDEF

RELNAME                                           -> QUERY
"Union" "(" QUERY "," QUERY ")"                   -> QUERY
"Intersection" "(" QUERY "," QUERY ")"            -> QUERY
"Minus" "(" QUERY "," QUERY ")"                   -> QUERY
"Product" "(" QUERY "," QUERY ")"                 -> QUERY
"Select" "(" SELPRED "," QUERY ")"                -> QUERY
"Project" "(" ATTRIBUTELIST "," QUERY ")"         -> QUERY
"Join" "(" JOINPRED "," QUERY "," QUERY ")"       -> QUERY
```

Before the query can be transformed it is necessary to find the attributes of the relations that are used by the query. This is done by scanning the query and extending every relation name with the corresponding attributelist. The ReadAttributes-function accomplishes this. This function takes as its arguments the scheme-information and the original query. It then "strips" the query until only relation names remain. As an example of this "stripping" we show the function for the union-operation:

```
[RA1] scheme S relation Rel1 = ReadAttributes(scheme S query Q1),
      scheme S relation Rel2 = ReadAttributes(scheme S query Q2)
      ===========================================================
      ReadAttributes(scheme S query Union(Q1,Q2)) =
      scheme S relation Union(Rel1,Rel2)
```

The equations for the other operations look similar and can be found in the Optimizer-module (see appendix).

At one point the query consists just of a relation name; this is the moment the corresponding attributelist must be found. This is done using the following equations:

```
[RA8]  Attributelist = Lookup(Relname,S,1)
       ===================================
       ReadAttributes(scheme s query Relname) =
       scheme S relation Relname Attributelist

[L1]   Relname != Relname1
       ===================
       Lookup(Relname,Relname1 == attributes: Attributelist
                                  sortedon: Sortedon
                                  indexon: Indexon,Int) = <>

[L2]   Lookup(Relname,
          Relname == attributes: Attributelist
                     sortedon: Sortedon
                     indexon: Indexon, Reldefs,1) = Attributelist

[L3]   Lookup(Relname,
          Relname == attributes: Attributelist
                     sortedon: Sortedon
                     indexon: Indexon, Reldefs,2) = Sortedon

[L4]   Lookup(Relname,
          Relname == attributes: Attributelist
                     sortedon: Sortedon
                     indexon: Indexon, Reldefs,3) = Indexon

[L5]   Relname != Relname1
       ===================
       Lookup(Relname,Relname1 == attributes: Attributelist
              sortedon: Sortedon indexon: Indexon, Reldefs,Int) =
       Lookup(Relname,Reldefs,Int)
```

In the equations above attributes of a corresponding relation (name) are found using the Lookup-function. The Check-module (see Chapter 5) takes care of the situation that the relation-name does not occur in the scheme. Applying these functions (`ReadAttributes` and Lookup) to the example at the beginning of this section gives the following result:

```
scheme
   "Infostudents" ==
     attributes: <studentnr,name,phone>
     sortedon: <name>
     indexon: <studentnr>,
   "LawStudents" ==
     attributes: <studentnr,name,phone>
     sortedon: <name>
     indexon: <studentnr>
   "Members" ==
     attributes: <studentnr,sport>
     sortedon: <>
```

```
    indexon: <studentnr>
relation
   Select(studentnr > 9000000,
          Union("InfoStudents" <studentnr,name,phone>,
                "LawStudents" <studentnr,name,phone>))
```

If the query is in this form we can transform it using the Transform-function that is defined in module Eqtransforms (see Chapter 6). Before we can choose the access strategies for the query it is necessary to find the sort order- and index-information for the relations used by the query. The `ReadInfo`-function accomplishes this. This function is very similar to the `ReadAttributes`-function except that it takes a relation as second parameter (instead of a query). The result of this function is an extended relation. If we apply this `ReadInfo`-function (that uses the Lookup-function) to our example we get the following result:

```
Select(studentnr > 9000000,
       Union("InfoStudents" <studentnr,name,phone> <name> <studentnr>,
             "LawStudents" <studentnr,name,phone> <name> <studentnr>))
```

Chapter 5

Check

In Section 4.2 we already mentioned that checking of the structure of a query is the first phase in optimization. Here we will discuss this checking phase by explaining the specification of the Check-module, and by giving some (simple) examples.

The ASF+SDF system checks the syntax automatically. So we do not have to check for things like missing parenthesis. Also some kind of implicit type check is performed by the system because some type information is embodied in the context-free syntax definitions. An example of this implicit type checking is the syntax definitions of the (selection and join) predicates. Because selection predicates (`SELPRED`) and join predicates (`JOINPRED`) have been defined seperately, the systems checks automatically if the used predicates are correct according to the syntax definitions. Things we still have to check are:

- Undefined Relations. Undefined relations are relations that are used by a query and have not been defined in the scheme. An example of the use of undefined relations can be found in Section 5.1.
- Union-compatible Relations. This check applies to the union, intersection, and difference operations. An example can be found in Section 5.2.
- Use of Undefined Attributes. Undefined attributes can appear in selection predicates, join predicates and projectlists. For these three cases we have to check whether the attributes are properly defined. Examples can be found in Section 5.3.

Instead of a fancy yet complex error message mechanism, which is beyond the scope of this paper, we have chosen to present just simple and adequate error messages. The error messages consist of explanatory text mixed with relevant language clauses, as in:

```
"Relation" RELNAME ATTRIBUTELIST "and relation" RELNAME ATTRIBUTELIST
"of the union-operation are not union-compatible" -> MSG
```

When the `Check`-function finds two relations which are not union-compatible it generates this error message. Instead of printing `RELNAME ATTRIBUTELIST` the actual values are substituted. Examples using this concept can be found in this chapter.

These messages are defined by the type `MSG` (see module Check in the appendix). We used a binary operator `MSG && MSG -> MSG` to combine error messages. An example of the use of this operator can be found in Section 5.3.1 where multiple undefined attributes occur in a

selection predicate. Only errors that occur at the lowest level are returned because if one of the operands of a relation contains an error, it is very likely that relations at a higher level also contain errors because these relations are using this operand information.

The main function in the Check-module, the Check-function, takes a relation as its argument and returns the same relation if no errors can be found during the check-procedure. If an error did occur this function returns the relation followed by a message (type MSG) explaining the error(s).

If no errors are found with the Check-function a normalize function is applied to the original relation. The relation is normalized to ease further optimization. A description of this normalizing process can be found in Section 5.4.

We show the Check-function by giving several examples. All these examples use the following scheme (because we do not need the sortedon and indexon information the corresponding lists are empty):

```
scheme
   "InfoStudents" == attributes: < studentnr,name,phone >
                     sortedon: < >
                     indexon: < >,
   "LawStudents" == attributes: < studentnr,name,phone >
                    sortedon: < >
                    indexon: < >,
   "Members" == attributes: < studentnr,sport >
                sortedon: < >
                indexon: < >
```

5.1 Undefined Relations

First we consider a situation in which a query uses a relation which has not been defined in the scheme. If this is the case the Lookup-function of the Optimizer-module returns an empty attributelist. The Chck-function in the Check-module strips the query until a leaf relation, which consists of a relation name and an attributelist, has been found. If the attributelist is not empty this function returns the relation with the word good in front of it; otherwise an error message indicating the missing definition of a relation is returned. The used syntax definitions and equations are:

```
context-free syntax
   "Chck" "(" REL ")"                   -> REL
   good RELNAME ATTRIBUTELIST           -> REL
   MSG                                  -> REL
   "Relation" RELNAME "not defined" -> MSG

equations
   [3]  Attributelist != <>
        ===================
        Chck(Relname Attributelist) = good Relname Attributelist

   [4]  Chck(Relname <>) =
        Relation Relname not defined
```

Example

Suppose we have the following query:

```
query
   Select( studentnr > 9000000,
      Union("InfoStudents", "MathStudents"))
```

If we apply the ReadAttributes-function we get:

```
relation
   Select( studentnr > 9000000,
      Union("InfoStudents" < studentnr,name,phone >,"MathStudents" <  >))
```

Finally, the Check-function will result in:

```
Relation
  "MathStudents"
not defined
```

5.2 Union-compatible Relations

For the traditional set operations union, intersection, and difference it is necessary that the operand relations of these functions are union-compatible. As we have already stated in Chapter 2 we insist that besides the number of attributes, the names of the attributes must also be the same (the order of attributes is irrelevant). If the operand relations are not union-compatible, the Chck-function will return an error message indicating which operand relations of which function are not compatible.

Example

Suppose we have the following relation (the ReadAttributes-function has already been applied):

```
relation
   Intersection( "Members" < studentnr,sport >,
      Union( "InfoStudents" < studentnr,name,phone >,
             "LawStudents" < studentnr,name,phone >))
```

The Check-function will result in:

```
Relation
  "Members"
  < studentnr,sport >
and relation
  "InfoStudents@LawStudents"
  < studentnr,name,phone >
of the intersection-operation are not union-compatible
```

In this example the "InfoStudents" and the "LawStudents" relation can be unified without difficulties. The relation name of the result is "InfoStudents@LawStudents" (this concatenation of strings is done using constructor-functions). The operands of the intersection operation, "Members" and "InfoStudents@LawStudents", are not union-compatible because the relations have different sets of attributes.

5.3 Use of Undefined Attributes

Undefined attributes can appear in three places:

1. in a selection predicate;
2. in a join predicate; or
3. in a projectlist.

5.3.1 Undefined Attributes in Predicates

As stated above undefined attributes could appear in both selection predicates and join predicates. In this section we look at a situation in which multiple errors occur. This can only happen if several errors occur on the same level. In the following example we have a selection predicate in which two attributes have not been defined.

Example

Suppose we have the following relation:

```
relation
   Select(InfoStudents.studentnr > LawStudents.student and
                 LawStudents.nme = "Smith",
      Product( "InfoStudents" < studentnr,name,phone >,
               "LawStudents" < studentnr,name,phone >))
```

If we apply the Chck-function we get:

```
The attribute
  LawStudents.student
used in selection-predicate is not an attribute of relation
  "InfoStudents@LawStudents"
  < InfoStudents.studentnr,InfoStudents.name,InfoStudents.phone,
    LawStudents.studentnr,LawStudents.name,LawStudents.phone >
&&
The attribute
  LawStudents.nme
used in selection-predicate is not an attribute of relation
  "InfoStudents@LawStudents"
  < InfoStudents.studentnr,InfoStudents.name,InfoStudents.phone,
    LawStudents.studentnr,LawStudents.name,LawStudents.phone >
```

Here we see an example of the binary && operator. All different errors are separated by this double "&"-sign. In this example the two attributes are undefined because we have made some typing errors.

5.3.2 Undefined Attributes in Projectlists

In the following example we will find an undefined attribute in a projectlist. This attribute is actually not undefined but this attribute is not unique. To make this attribute unique (and legal) we have to place its relation name in front of it. The Chck-function does not mention the attributes name and sport as undefined attributes, because these attributes are unique, they only appear in one relation, so in this case it is allowed to skip the relation names.

The general rule for the naming of attributes is: if an attributename is unique, mentioning a relation name is optional, otherwise a relation name is required. In our specification "Relname..Attribute" represents an optional relation name. "Relname.Attribute" represents a necessary relation name. We construct these names using *constructor functions.* As an example we give the syntax and equations of the Single-function. This function places the relation name and a point in front of an attribute:

```
context-free syntax
   "Single" "(" RELNAME "," ATTRIBUTE ")" -> ATTRIBUTE
   RELNAME "." ATTRIBUTE                  -> ATTRIBUTE

equations
   [S1] str-con(""" Chars1 """) . attribute(Chars2) =
        attribute(Chars1 "." Chars2)

   [S2] Single(Relname,Attribute) = Relname . Attribute
```

The specification for the construction of "Relname..Attribute" is very similar to the above specification and can be found in the appendix. In the following error message string we find an attributelist. This list contains all possible attributenames. As we can see the phone, name, and sport attributes have optional relation names. For the studentnr attribute mentioning the relation name is necessary.

Example

Suppose we have the following relation:

```
relation
   Select(LawStudents.name = "Smith",
          Project( <studentnr,name,sport>,
                   Product( "LawStudents" < studentnr,name,phone >,
                            "Members" < studentnr,sport >)))
```

If we apply the Check-function we get:

```
The attribute
  studentnr
of the projectlist is not defined in relation
  "LawStudents@Members"
  < LawStudents..phone,LawStudents..name,LawStudents.studentnr,
    Members.studentnr,Members..sport >
```

5.4 Normalized Relations

If no errors were found during the check phase (the function `Chck` has returned a relation of the form good `Relname Attributelist` a normalize function is applied to the original relation.

First the whole query is processed to build an attributelist with all attributes in their normalized forms. So for each attribute that appears in the query we determine whether a relationname is required or not. Using this list of normalized attribute names we process the query again and for each attribute we check whether the attribute name is in its normalized form. This means we have to check attributes in leaf relations, selection predicates, projectlists and join predicates. The equation that performs this check for the project operation is given below.

```
[CN7] Projectlist1 = CheckAttributes(Projectlist, Attributelist)
      ==========================================================
      CheckName(Project(Projectlist,Rel),Attributelist) =
      Project(Projectlist1, CheckName(Rel, Attributelist))
```

The function `CheckName` processes the whole relation and for each appearance of attributes it checks whether the attribute names are in normalized form. In the above equation the `Projectlist` contains the attributes to be checked and the `Attributelist` variable contains the list of normalized attribute names. `Projectlist1` will contain the result of function `CheckAttributes` which checks and changes the attribute names if they are not in their normalized forms. Finally we give an example in which we see an relation before and after the normalization.

Example

The relation before the `Normalize`-function:

```
relation
   Project(< LawStudents.name, LawStudents.studentnr >,
      Product("LawStudents" < studentnr, name, phone >,
         Select(sport = "Tennis", "Members" < studentnr, sport >)))
```

The relation after the `Normalize`-function has been applied:

```
relation
   Project(< name,LawStudents.studentnr >,
      Product("LawStudents" < LawStudents.studentnr, name, phone >,
         Select(sport = "Tennis", "Members" < Members.studentnr, name >)))
```

Chapter 6

Equivalence Transformations

In this chapter we describe how we have specified query transformations. We start with giving an example to illustrate the amount of actual savings that can be obtained by query transformations. In Section 6.2 we give a graphically representation of the queries.

The rule of thumb used in query transformation is: *'projections and selections should be performed early, and joins (and other binary operations) late'*. In Section 6.3 we give a detailed description of the transformations in which the selection-operation is involved. In Section 6.4 we describe the equivalence transformations which are based on the projection-operation.

Most examples in this chapter use the following scheme (because we don't need the sortedon and indexon information in this phase, the corresponding attributelists are empty):

```
scheme
   "InfoStudents" == attributes: < studentnr,name,phone >
                     sortedon: < >
                     indexon: < >,
   "LawStudents" == attributes: < studentnr,name,phone >
                    sortedon: < >
                    indexon: < >,
   "Members" == attributes: < studentnr,sport >
                sortedon: < >
                indexon: < >
```

6.1 Example

We will illustrate the degree of actual savings that can be obtained by query transformations with an example that has been taken from [Graef87].

Consider the relational schema of a database that describes employees offering computer lectures to departments of a geographically distributed organization:

```
departments(dname, city, street address)
courses(cnr, cname, abstract)
lectures(dname, cnr, daytime)
```

Key attributes are given in italics. Assume that a user is interested in:

"The names of departments located in New York offering courses on database management."

There are 100 departments, 5 of which are located in New York. A physical block contains 5 department records. There are 500 courses, 20 of which are on database management. The physical block size is 10 records. There are 2000 lectures, 300 are on database management, 100 are held in New York, and 20 (from 3 departments) satisfy both conditions. The physical block size is 10 records.

Assume that the sorting time is $N * log_2 N$, where N is the file size in blocks, and that there is a buffer of one block for each relation. All relations are sorted on ascending key values.

There are three example strategies given in [Graef87]. Only the I/O cost is considered here. For each step the number of (r)ead and (w)rite disk accesses are given.

- Strategy 1
  1. Form the Cartesian product of the `courses`, `lectures`, and `department` relations. (r: 200,000)
  2. Retain the *dname* column of those department records, for which the *cnr* of `courses` and `lectures` match, the *dname* of `lectures` and `departments` match, `cname` = "database management", and `city` = New York (w:1)

  Total: approximately 200,000 accesses.

- Strategy 2
  1. Merge `courses` and `lectures` (r: 50+200, w: 400)
  2. Sort the results by *dname* (r+w: 400 $\log_2 400$)
  3. Merge the result with `departments` (r: 400+20, w: 400+400)
  4. Select the combinations with `city` = "New York" and `cname` = "database management" (r:800)
  5. Keep only the dname column (w:1)

  Total: approximately 6000 accesses.

- Strategy 3
  1. Merge `courses` with `lectures` (r: 50+200)
  2. Keep only the *dnames* of combinations with `cname` = "database management" (w: 2)
  3. Sort the *dname* list generated (r+w: 2)
  4. Merge the result with the `departments` relation (r:2+20)
  5. Keep only those with `city` = "New York" (w: 1)

  Total: 277 accesses.

If we compare strategies 1 and 3 we can conclude: "A reduction by a factor of approximately 700 has been achieved. For larger databases and more complex queries, more sophisticated techniques may even result in even higher reductions."

6.2 Graphical Representation

A general query can be converted into an equivalent set of relational algebra expressions. The reciprocal conversion is also possible. Therefore, we have chosen to work with relational algebra expressions.

A graphical representation of a relational algebra query is useful for manipulation. A relational query can be described by a *relational algebra tree/tree of operations.* A relational algebra tree is a tree describing a query. A leaf in this relational algebra tree represents a base relation, an internal node represents an intermediate relation obtained by applying a relational operation, and the root represents the result of the query. The data flow is directed from the leaves to the root. Figure 6.1 shows the used symbols.

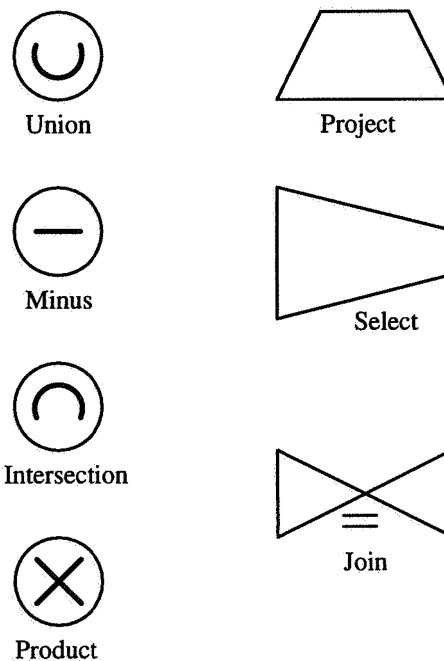

Figure 6.1: Symbols used in Graphical Representation

Different queries may be equivalent. Different relational algebra trees may be equivalent as well. Since relational operations have different complexity, some trees may provide a much better performance than others. Important factors for optimization are the ordering of relational operations and the sizes of intermediate relations generated. Relational algebra trees can be restructured using transformation rules. Figure 6.2 is given as an example; it is a graphical representation of the example used in Section 4.3.2.

6.3 Processing of Selects

In the following, we use the heuristic that the select-operation reduces the size of the operand relation and thus should be applied first.

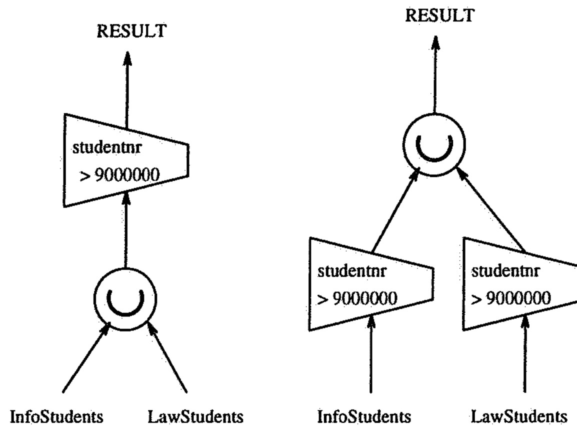

Figure 6.2: The select over union example before and after transformation.

We begin the reorganization of the tree of operations by distributing select operations down the tree, and by combining the consecutive ones and those having the same operands into one select operation. This is done to reduce the cardinalities (number of tuples) of processed relations as early as possible and to avoid unnecessary pipelining of tuples between operations. By performing selections as early as possible we can save orders of magnitude in execution time, since it tends to make the intermediate results of multistep evaluations small.

6.3.1 Select over Union, Intersection, and Minus

The first set of equivalence transforms we describe apply to queries that contain a select-operation over an union-, intersection- or minus-operation. The underlying idea of these three rules is the same, therefore we only present the select-over-union rule here. The rules for select over minus (set difference) and for select over intersection can be found in the appendix. The transformation for the select-over-union can be written down in the following way:

Expression: Select(Selpred,Union(Rel1,Rel2))
Result : Union(Select(Selpred,Rel1),Select(selpred,Rel2))

By first applying the select-operation to the two operand relations we reduce the cardinality of the intermediate results. This means that less tuples need to be unified compared to the case where the select-operation is applied after the union-operation.

Note that distribution into one operand would be enough for the intersection- and the minus-operation. For those two operations we could replace the second selection by the original relation. However, it is usually at least as efficient to perform the selection as it is to work with the original relation, because the former is a smaller set than the latter.

Next, we give an example of this select-over-union rule; see Figure 6.2 for a graphical representation of this rule.

Example: Select over Union

```
query
```

```
Select(studentnr > 9000000,
   Union("InfoStudents","LawStudents"))
```

gives as a result:

```
Union(
   Select(studentnr > 9000000,
          "InfoStudents"<studentnr,name,phone>),
   Select(studentnr > 9000000,
          "LawStudents"<studentnr,name,phone>))
```

6.3.2 Select over Product

In [Ull82] the principle of this transformation rule is described by the following three rules:

1. *Select(selpred,Product(Rel1,Rel2)) = Product(Select(Selpred,Rel1),Rel2)*
   This rule can be applied if all attributes used in the selection predicate are attributes of the first operand relation (Rel1).
2. *Select(Selpred,Product(Rel1,Rel2)) =*
   Product(Select(Selpred1,Rel1), Select(Selpred2,Rel2))
   This rule can be applied if Selpred1 involves only attributes of the first operand relation (Rel1), and Selpred2 involves only attributes of the second operand relation (Rel2).
3. *Select(Selpred,Product(Rel1,Rel2)) =*
   Select(Selpred2,Product(Select(Selpred1,Rel1),Rel2))
   This rule can be applied if Selpred1 involves only attributes of the first operand relation (Rel1), but Selpred2 involves attributes of both Rel1 and Rel2.

We used these rules from [Ull82] to specify the more general rule:

Expression: Select(Selpred,Product(Rel1,Rel2))
Result : Select(Selpred3,Product(Select(Selpred1,Rel1), Select(Selpred2,Rel2))

where Selpred1 involves only attributes of Rel1, Selpred2 involves only attributes of Rel2, and Selpred3 involves attributes of both Rel1 and Rel2.

The specification of this equivalence transform is rather complicated, therefore we give a more elaborated explanation of it. The equation we are going to examine:

```
[TRANS7]  Selpred1 = Conjunctive(Selpred),
          Selpred2 = Distribute(Selpred1,Rel1),
          Selpred3 = Distribute(Selpred1,Rel2),
          Selpred4 = Collect(Selpred1,Selpred2,Selpred3),
          Selpred1 != Selpred4
          ==============================================
          Trans(Select(Selpred,Product(Rel1,Rel2))) =
          Trans(Select(Selpred4,
                Product(Select(Selpred2,Rel1),Select(Selpred3,Rel2))))
```

Predicate F (in the specification we used selpred) must be in the conjunctive normalform, in order to be able to divide it in the tree parts as mentioned above. In the above equation the first condition (`selpred1 = Conjunctive(selpred)`) assigns the conjunctive normalform of predicate F (selpred) to the predicate-variable selpred1. The second condition (`selpred2 = Distribute(selpred1,rel1)`) selects those parts of predicate F (which is now in conjunctive normalform) that only use attributes of the first operand of the product-operation (rel1). The third condition does the same for the second operand of the product-operation. The fourth condition (`selpred4 = Collect(selpred1,selpred2,selpred3)`) retrieves those subpredicates that only involves attributes of both relations. The last condition is added to prevent an endless loop. By adding `selpred1 != selpred4` we make sure this equation is applied only when some parts of the original selection predicate can be moved down the tree (past the product-operation).

Next, we give an example of this select-over-product rule; see also Figure 6.3 for a graphical representation of this rule.

Example: Select over Product

```
query
   Select(InfoStudents.studentnr = Members.studentnr and sport = "Tennis",
      Product("InfoStudents","Members"))
```

gives as a result:

```
Select( InfoStudents.studentnr = Members.studentnr,
   Product("InfoStudents" < studentnr,name,phone >,
           Select(sport = "Tennis","Members" < studentnr,sport >)))
```

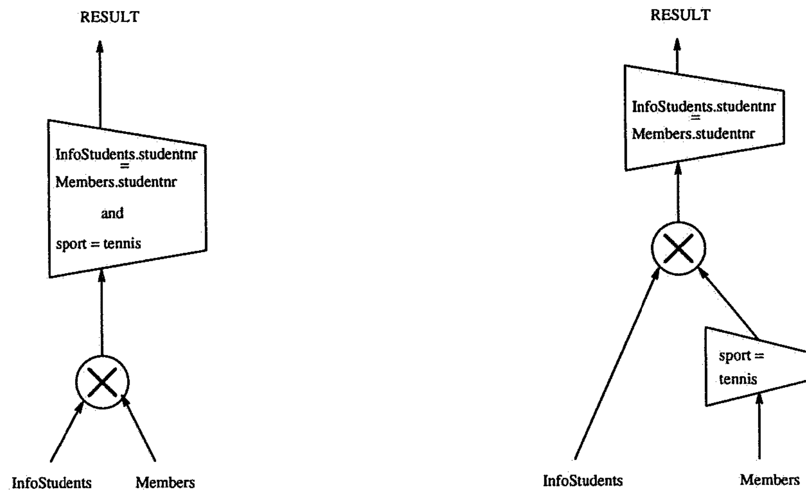

Figure 6.3: The select over product example before and after transformation.

6.3.3 Select over Join

The underlying idea of the select-over-join transformation is very similar to the idea of the select-over-product transformation. Therefore we will only give an example of this transformation. See Figure 6.4 for a graphical representation of this rule.

Example: Select over Join

```
query
   Select(InfoStudents.studentnr < 9000000 and sport = "Tennis",
      Join(InfoStudents.studentnr = Members.studentnr,
           "InfoStudents","Members")))
```

gives as a result:

```
Join(InfoStudents.studentnr = Members.studentnr,
   Select(InfoStudents.studentnr < 9000000,
          "InfoStudents" < studentnr,name,phone >),
   Select(sport = "Tennis",
          "Members" < studentnr,sport >))
```

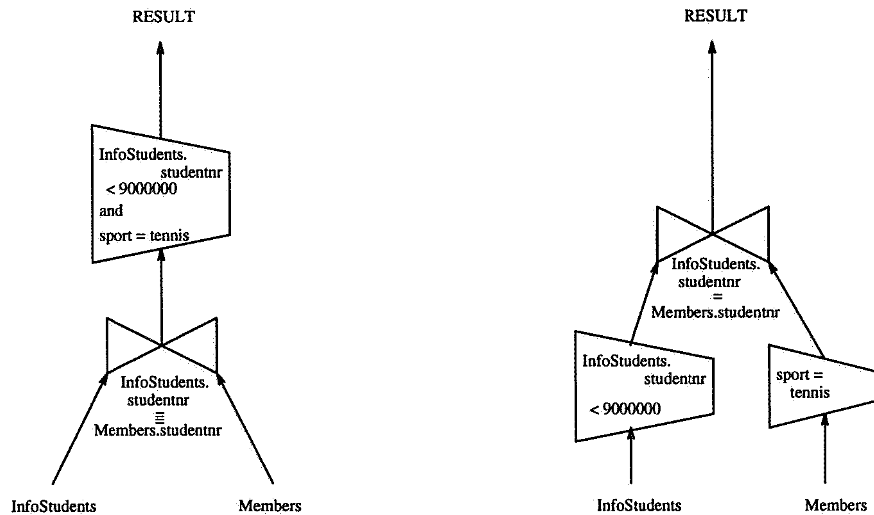

Figure 6.4: The select over join example before and after transformation.

6.3.4 Select (cascade)

This rule is used to combine consecutive select-operations into one by combining the predicates. The select-predicate is defined as a boolean combination of terms, therefore it is sufficient to take the conjunction of the predicates.

Expression: Select(selpred2,Select(selpred1,r))

Result : Select((selpred1 and selpred2),r)

An example to illustrate this rule (see also Figure 6.5 for the graphical representation):

Example: Select (cascade)

```
query
   Select(sport = "tennis",
      Select(studentnr > 9000000,"Members"))
```

gives as a result:

```
Select((sport = "tennis" and studentnr > 9000000,
   Members < studentnr,sport >)
```

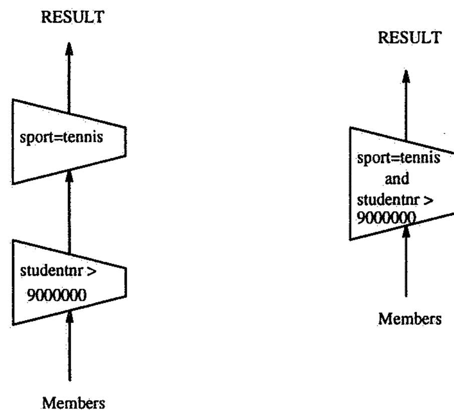

Figure 6.5: The select-cascade example before and after transformation

6.3.5 Union, Intersection, and Minus over Select

The next set of transformations we discuss are transformations in which several selects have the same operands. In these cases it is possible to combine these selects into one operation. This combining is done using the following rules:

Expression: Union(Select(Selpred1,Rel),Select(Selpred2,Rel))
Result : Select((Selpred1 or Selpred2),Rel)

Expression: Intersection(Select(Selpred1,Rel),Select(Selpred2,Rel)
Result : Select((Selpred1 and Selpred2,Rel)

Expression: Minus(Select(Selpred1,Rel),Select(Selpred2,Rel))
Result : Select((Selpred1 and not Selpred2),Rel)

The following example shows such a transformation (see also Figure 6.6).

Example: Union over Select

```
query
   Union(Select(studentnr > 9000000,"InfoStudents"),
         Select(name < "Smith","InfoStudents")))
```

gives as a result:

```
Select(studentnr > 9000000 or name < "Smith",
       "InfoStudents" < studentnr,name,phone >)
```

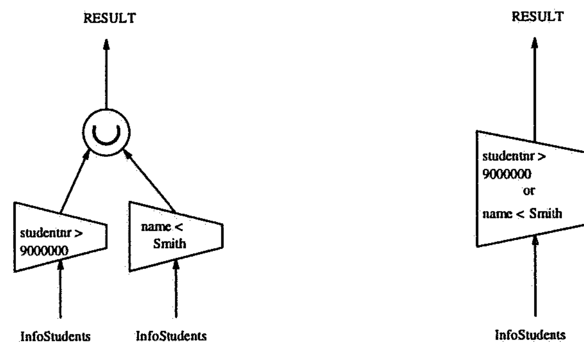

Figure 6.6: The union over select example before and after transformation.

6.4 Processing of Projects

In the following we use the heuristic that the project-operation reduces the size of the operand relation and thus should be applied first.

6.4.1 Project over Union, Intersection, and Difference

The project-over-union rule is very similar to the select-over-union rule (see Section 6.3.5) and therefore we will not discuss this rule any further.

It is not possible to push the project-operation over the intersection- or difference-operation. This is not possible because tuples that are the same after a project-operation could have been different tuples before the project-operation. An example to clarify this:

Example: Project over Intersection

In this example we use the following relations:

```
scheme
   "InfoStudents" == <studentnr,name>,
   "LawStudents"  == <studentnr,name>
```

We are going to compare the following two queries:

```
query1:
   Project(<name>,Intersection("InfoStudents","LawStudents"))

query2:
   Intersection(Project(<name>,"InfoStudents"),
                Project(<name>,"LawStudents"))
```

Suppose we have the following tuples:

```
"InfoStudents"
   9012578, J. Murphy
   8917788, A. Mahon
   8991023, P. Farrell

"LawStudents"
   9178912, M. Anderson
   8917788, A. Mahon
   8976523, P. Farrell
```

If we apply query1 to our tuples the answer would consist of one tuple (A. Mahon) but if we apply query2 to the same tuples the answer would consist of two tuples (A. Mahon, P. Farrell). This happens because we have two tuples in which the first attribute (studentnr) is different, but the second attribute (name) is the same. Using query1 the two tuples are different when the intersection-operation has to be applied, but if we use query2 the tuples only consist of the name-attribute (due to the project-operation) and the two tuples are the same.

6.4.2 Project over Join

This rule is similar to the project-over-union rule. Only one extra operation has to be performed. The original attributelist has to be split into two attributelists.

Expression: Project(Attributelist,Join(Joinpred,Rel1,Rel2))
Result : Join(Joinpred,Project(Attributelist1,Rel1), Project(Attributelist2,Rel2))

In this result Attributelist1 consists of attributes that are element of both the original attributelist and the attributelist of relation Rel1. Attributelist2 consists of attributes that are element of both the original attributelist and the attributelist of relation Rel2. In our specification this splitting is done using the MakeSub-function:

```
context-free syntax
   "MakeSub" "(" ATTRIBUTELIST "," REL ")"     -> ATTRIBUTELIST

equations
   [M1] MakeSub(<>,Rel) = <>

   [M2] ElemOf(Attribute,Attributelist) = false
        ==========================================
        MakeSub(<Attribute,Attributes>,Relname Attributelist) =
        MakeSub(<Attributes>,Relname Attributelist)
```

```
[M3] ElemOf(Attribute,Attributelist) = true
     ===========================================
     MakeSub(<Attribute,Attributes>,Relname Attributelist) =
     Add(Attribute, MakeSub(<Attributes>, Relname Attributelist))
```

This MakeSub-function checks for every attribute in the attributelist (the first parameter) whether this attribute is also an element of the attributelist of the operand relation (the second parameter). If attributes meet this requirement, they are added to the resulting attributelist.

The following example is used as an illustration of the project over join rule (see also Figure 6.7). Notice that both the attributes of the join predicate must be an element of the projectlist for the transformation to be possible.

Example: Project over Join

```
query
   Project(< InfoStudents.studentnr, sport, Members.studentnr >,
           Join(InfoStudents.studentnr = Members.studentnr,
               "InfoStudents","Members"))
```

gives as a result:

```
Join(InfoStudents.studentnr = Members.studentnr,
   Project(<InfoStudents.studentnr>, "InfoStudents"<studentnr,name,phone>),
   Project(<Members.studentnr,sport>,"Members"<studentnr,sport>))
```

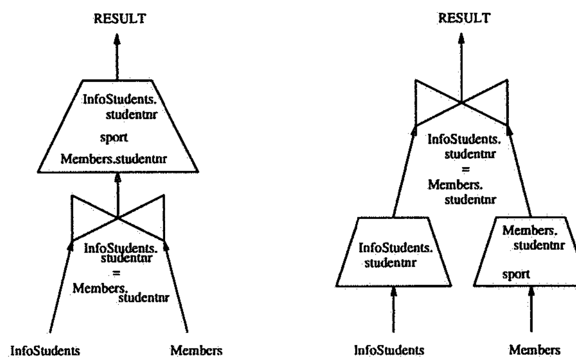

Figure 6.7: The project over join example before and after transformation.

6.4.3 Project (cascade)

The project-cascade rule combines all consecutive projects into one operation. The transformation rule is of the form:

Expression: Project(attributelist1,Project(attributelist2,r))
Result : Project(attributelist1,r)

The condition involved in this transformation rule is that the attributes of attributelist1 must be a subset of the attributes of attributelist2. But if we look at the equation that takes care of this transformation we can see that this rule has no condition at all. This condition has been omitted because the Check-module (see Chapter 5) already checks whether the operations are legal.

Example: Project (cascade)

```
query
   Project(< studentnr >,
      Project(< studentnr, phone >,"InfoStudents"))
```

gives as a result:

```
Project(< studentnr >,
   "InfoStudents" < studentnr,name,phone >)
```

6.4.4 Project over Select

This last transformation rule we discuss is the project-over-select rule. The purpose of this rule is to reduce the degree (the number of attributes) of the operand relation as soon as possible. This rule is of the form:

Expression: Project(attributelist1,Select(selpred,rel))
Result : Project(attributelist1,Select(selpred,Project(attributelist2,rel))

Attributelist2 is attributelist1 extended with attributes that appear in the selection predicate but were not a member of attributelist1. If all attributes used in the selection predicate were already a member of attributelist1, the last project-operation is not necessary . The result becomes then:

Result : Select(selpred,Project(attributelist2,rel))

In the next example we show the transformation rule for both cases (see also Figure 6.8 for the second example).

Example: Project over Select

Example 1:

```
query
   Project(< studentnr, name >,
      Select(studentnr > 9000000,"InfoStudents"))
```

gives as a result:

```
Select(studentnr > 9000000,
   Project(< studentnr, name >, "InfoStudents" < studentnr, name, phone>)
```

Example 2:

```
query
   Project(< name >, Select(studentnr > 9000000,"InfoStudents"))
```

gives as a result:

```
Project(< name >, Select(studentnr > 9000000,
   Project(< studentnr, name >,"InfoStudents" < studentnr, name, phone >))
```

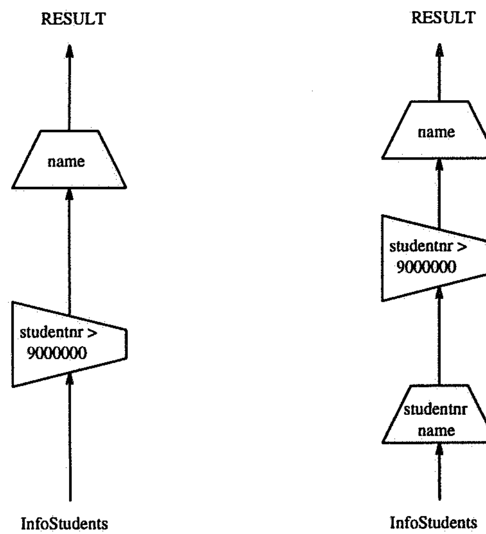

Figure 6.8: The project over select example before and after transformation.

Chapter 7

Access Strategies

The selection of access strategies for operations is a critical phase in query optimization. In this phase important decisions affecting data transfer are made. Access strategies for the operations participating in the query are selected by selecting one of the alternative procedures for these operations.

In this chapter we give a description of how we have specified the access strategies for the select operation. This description uses the algorithm given in [Jar81]. Before we give this description we first discuss the physical data organization (Section 7.1). In Section 7.2 we describe a function called factorisation, this is a technique to determine which part of a (selection) predicate can be evaluated using indices.

In Section 7.3 we discuss the selection of access strategies for the select operation. We have chosen to restrict our description to the select operation only because this is the most interesting operation of our set of relational algebra operations. It is an interesting operation because there are many different possibilities to apply selection predicates on operand relations. We have not described (or specified) the processing of the other relational algebra operations to limit the size of the paper.

The algorithm used to determine the access strategies processes the tree of operations twice. The first pass (up-pass) proceeds from the leaves to the root and for each operation different possible sort orders of records (which represent tuples) are inferred from different possible sort orders of input records. During the second pass (down-pass), proceeding from the root down to the leaves, sort orders and index usage are fixed by selecting procedures, and adding auxiliary procedures (for sorting and index generation) to the tree.

7.1 Physical Data Organization

The basic problem in physical database representation is how to store a *file* consisting of *records*, where each record has an identical format. A *record format* consists of a list of field names, where each field consists of a fixed number of bytes and has a fixed data type. A *record* contains (specific) values for each field. The typical operations we desire to perform on a file are:

1. insert a record,

2. delete a record,

3. modify a record, and

4. find a record with a particular value in a particular field, or a combination of values in a combination of fields.

The complexity of organizing a file for storage depends on the combination of these operations that we intend to perform on the file. If operation 4 (find a record) is permitted we also need to consider whether desired records are specified by value, by location, or a combination of both, and whether only one or several different fields may be involved in different lookups.

A file system using disk storage usually divides the disk into equal sized ***physical blocks.*** Each physical block (or just ***block***) has an address, which is an absolute address on the disk.

A file is stored in one or more blocks, where one or more records are stored in each block. We assume there is a file system that translates file names into the absolute addresses of the blocks occupied by the file. A block may contain bytes that are not used by any record. Some of these are really unused. Others are devoted to the block *header*, a collection of bytes at the beginning of the block used for special purposes. For example, the header may contain information about the connection of this block to other blocks used to store the same file, or information about how the individual bytes of the block represent the file in question.

Records have an address, which is either the absolute address of the first byte of the record on the disk, or a combination of the address of the block and an *offset*, the number of bytes in the block preceding the beginning of the record.

In our specification we allow indices for some particular set of fields, that do not necessarily form the key of the file in question. This kind of indices are called *secondary indices.* A secondary index for field F is a relationship between domain D and a set of records of the file in question. We can represent a secondary index as a logical file with format:

VALUE (RECORD)*

An instance of VALUE is a value from D (in our specification this is an integer or a string). An instance of RECORD could be either

1. A pointer to a record with the associated value in field F, or
2. A key value for a record with the desired value in field F.

If option 1 is used, the pointer could point to the subblock containing the record, or it could point to the block containing the record, in which case a search of the block would be necessary to find the desired record or records. In either case, the records of the file are consequently pinned, at least within the block.

With option 2 records of the main file are not pinned by pointers from the secondary index. However, compared with option 1, option 2 will required several additional block accesses to perform a lookup of a record given its key value, while option 1 goes directly to that record, or at least to its block.

7.2 Rules for Factorisation

Factorisation is a technique to find out which part of a (selection) predicate can be evaluated using indices. By factorisation we convert the function F into the form:

F = factor & residue,

where the *factor* contains the part of the selection predicate that can be evaluated by using indices and the *residue* contains the part of the selection predicate that can not be evaluated by using indices. The rules in this section define the function

$fac(F,Q) = (F_1,F_2)$,

where F is the original predicate, Q is a set of indices, predicate F_1 contains sub-predicates of F that only use attributes that are also element of set Q. Predicate F_2 is the residue.

Rule 1: F and $\emptyset$ = F and F or $\emptyset$ = $\emptyset$, where $\emptyset$ is the empty predicate.

Rule 2: **if** F is unary (contains only one f)
then if $f \in Q$,
then fac(F,Q) = (F,$\emptyset$).
else fac(F,Q) = ($\emptyset$,F).

Rule 3: fac((F),Q) = fac(F,Q).

Rule 4: **if** F = G and H, where fac(G,Q) = (G_1,G_2) and fac(H,Q) = (H_1,H_2),
then fac(F,Q) = (G_1 and H_1, G_2 and H_2).

Rule 5: **if** F = G or H, where fac(G,Q) = (G_1,G_2) and fac(H,Q) = (H_1,H_2),
then fac(F,Q) = (G_1 | H_1, (G_1 or H_2) and (G_2 or H_1) and (G_2 or H_2)).

Looking at the above rules we see two unexpected things. The first strangeness is found in Rule 1 where we see that the and of some sub-predicate F with the empty predicate $\emptyset$ gives as a result sub-predicate F, while following the normal definitions of the and function we should expect the empty predicate as a result. But if we look at what the term *F* and $\emptyset$ stands for we see that applying some sub-predicate F and the empty predicate (this means no condition) to some relation is the same as applying only sub-predicate F to that relation. The same kind of explanation can be given for the term *F* or $\emptyset = \emptyset$. If records have to meet some condition F or this record has to meet no condition at all is the same as meeting no condition at all.

The second strangeness is found in Rule 5. We see that sub-predicates G_1 and H_1, which can be evaluated using indices occur in the residue. It may be possible that either subpredicate G or subpredicate H can be evaluated using indices but not both. Using the or operator this means that we can not evaluate *G* or *H* using indices.

The `fac`-function has been specified in module Factorisation (see Appendix)

Example

Suppose we have the following scheme and query:

```
scheme
   "SportStudent" == attributes: < studentnr, name, sport1, sport2 >
                     sortedon: < >
                     indexon: < studentnrm, name, sport1 >
query
```

```
Select( studentnr > 9000000 and (name > "Smith" and
        (sport1 = "Tennis" or sport2 = "Tennis")), "SportStudent")
```

We can rewrite the selection predicate into the form:

F= f1 and (f2 and (f3 or f4))

and indices exist on f_1, f_2 and f_3. Hence, let Q = { f1, f2, f3 }. Some factorisations:

```
fac(f3 or f4,{f1,f2,f3}) =
(f3 or 0,(f3 or f4) and (0 or 0) and (0 or f4)) =
(0,(f3 or f4))

fac((f3 or f4),{f1,f2,f3}) =
fac(f3 or f4,{f1,f2,f3}) =
(0,(f3 or f4))

fac(f2 and (f3 or f4),{f1,f2,f3}) =
(f2 and 0,0 and (f3 or f4)) =
(f2,(f3 or f4))

fac((f2 and (f3 or f4)),{f1,f2,f3}) =
fac(f2 and (f3 or f4),{f1,f2,f3}) =
(f2,(f3 or f4))

fac(f1 and (f2 and (f3 or f4)),{f1,f2,f3}) =
(f1 and f2, 0 and (f3 or f4)) =
(f1 and f2, (f3 or f4))
```

Using the above factorisations, it is easy to see that the result is:

fac(F,Q) = (f1 and f2, (f3 or f4)),

If we rewrite this predicate using the original values we can conclude that `studentnr > 9000000 and name > "Smith"` can be evaluated using indices, while this is not possible for `(sport1 = "Tennis" or sport2 = "Tennis")`.

7.3 Choice of Procedures for Select

A select operation often radically reduces the cardinality of its operand relation. If the blocks on which the records corresponding to the (few) tuples of the result are located could be determined using an index, considerable reductions in data transfer can be achieved. If this is not possible, the complete file corresponding to the operand relation must be read.

First we consider the possible sort orders of stored files. Let G_R (where R refers to the operand file) be a set of attributes corresponding to the fields of R. If R is a stored file, G_R contains one attribute if access in sorted order is possible; otherwise G_R is empty. This information has to be specified in the scheme part that belongs to the query that has to be optimized (see module Optimizer in the Appendix). If R is an intermediate result, G_R may contain several attributes according to alternative access strategies for operations which produce R.

Up-pass

For select procedures we know that the order of records in the result will be the same as the order of the input records. Hence, in the up-pass we write for selects of the tree:

$G_T = G_R$,

where T refers to the output relation. In our specification we use the following equation to specify this rule:

```
[ACCESS4]  Extrel1 = Up(Extrel)
           ====================
           Up(Select(Selpred,Extrel,Sortedon,Indexon)) =
           Select(Selpred, Extrel1, GetSorted(Extrel1),Indexon)
```

This rule specifies that the Sortedon attributelist after the select operation contains the same attributes as the Sortedon attributelist of the operand of the select operation. The Sortedon attributelist of the operand is found using the GetSorted-function (which is a simple lookup procedure).

We need to pass this information to "higher" operations to be able to determine the access strategies. For example, suppose we have a join with one of its arguments an select operation. If we know that the record of the operands of the join operation (in our case a select operation) are sorted on certain fields (attributes), we can use this information to choose appropriate procedures for the join operation. An example of how the sort order information can be used in the up-pass of the join operation (which has not been specified):

If both operands of the join being examined are leaves of the tree, we use:
$G_T =$ **if** degree(R) $= 1$ **and** $(z_S \in I_S$ **or** $\neg\ (z_S \in G_S))$
then G_S
else if degree(S) $= 1$ **and** $(z_R \in I_R$ **or** $\neg\ (z_R \in G_R))$
then G_R
else $\{ z_R, z_S \}$.

where:

- z_R, z_S are the attributes used in the join predicate,
- G_R, G_S are the sortedon attributelists of the operands,
- I_R, I_S are the indexon attributelists of the operands,
- degree(R) $= 1$, tests whether R is unary or not. A relation is unary when it contains only one attribute.

Down-pass

During the down-pass of the select operation, the attribute(s) of G_T specify the preferred sort order(s) of records. If the records of the operand file cannot be accessed in one of the required orders (eg. $G_T \bigcap G_R = \emptyset$), the output of the select procedure have to be sorted by an added sort procedure. If the required order can be obtained without sorting, we write, for operations not at a leaf of the tree:

$G_R = G_R \bigcap G_T$

In this way the operations towards the leaves are informed of the required sort order. In our specification equations [ACCESS5e] and [ACCESS5f] specify this rule.

Normally, if indices exist (this can only be the case if the operand of the select operation is a leaf relation), only a part of the select operation can be evaluated using an indices. The rest of the index must be evaluated after records located via indices are moved into main storage. To find the maximal subsets of the predicate that may be evaluated using indices, the rules for factorisation must be applied (see Section 7.2).

If an elementary predicate to be evaluated via index is of the form *$z\ \theta$ constant*, procedure H1 will be chosen. If the elementary function is of the form $z_i\ \theta\ z_j$ (i = 1,2,...,n and i $\neq$ j), procedure H2 will be chosen. Both produce lists of record identifiers (pointers to records or key values of records) by processing the indices.

Procedures H1 and H2 operate on elementary predicates and the factor to be evaluated via indices may contain several elementary predicates connected with and and or operators. To overcome this problem each conjunction is substituted by an intersection (procedure I1), and each disjunction is substituted by an union (procedure U1). Both may be implemented by efficient merge procedures if the record identifiers produced by H1 and H2 are sorted. The whole factor may be evaluated in this way. Given the resulting list of record identifiers, procedure REAL fetches the corresponding relations.

To evaluate the residual predicate yielded by factorisation, procedure RAJ is used. This procedure processes its operand once and applies predicate F on each record to produce the result. If no indices exist in the operand file or none of them is of use, only one RAJ procedure is required for evaluation. With the procedures H1, H2, REAL and RAJ access strategies for all selects can be determined in the down-pass.

In our specification we first check if the operand of the selection predicate is a leaf relation using condition *IsLeave(Extrel) = true.* This check is performed because normally indices only exist for stored files (and not for intermediate results). If this condition is met, the existence of indices in the operand file is checked (*GetIndex(Extrel)* $\neq$ <>. If this is the case the factorisation function is applied:

```
(Selpred1, Selpred2) = Fac(Selpred, Indexon)
```

where Selpred1 contains the part of the selection-predicate that can be evaluated using indices and Selpred2 contains the part of the selection-predicate that can not be evaluated using indices.

If the factorisation function results in *(NULL, Selpred2)* where *Selpred2* $\neq$ *NULL*, this means that none of the existing indices are of use for the (original) selection predicate Selpred. In this case procedure RAJ is selected to perform the select operation (equation [ACCESS5b]).

If the factorisation function results in *(Selpred1, NULL)* where *Selpred1* $\neq$ *NULL*, this means that the whole predicate Selpred can be evaluated using indices. Here we choose procedure REAL to perform the select operation. If Selpred1 consists of several elementary predicates, function `RewritePred` will find the right procedures to perform the selections.

If both Selpred1 and Selpred2 contain parts of the original predicate, we first have to apply procedure REAL to Selpred1, followed by procedure RAJ to perform the selections that can not be done using indices. If Selpred1 consists of several elementary predicates function `RewritePred` will find the right procedures to perform the selections.

It must be noted that factors found to be evaluated via indices do not always lead to reductions

in data transfer. If the factor is of the form $z < constant$, and the identifiers of the records meeting this criterion are evenly distributed over the blocks of the file, we must in the worst case read the whole file plus the index. Generally, factors of the form:

$F = f_1$ and f_2 and ... and f_n and (F'),

where f1,f2,...,fn are all of the form $z = constant$ and F' is the residue (with possible or connectors), will lead to considerable reductions in data transfer when evaluated using indices.

7.3.1 Used Procedures

procedure H1(F,R)
(assumption: F is of the form "$z\ \theta$ constant" and there exists an index on field z of file R)
Read the index on z and apply function F for each value. If the criterion is met, write the record identifiers (RIDs) of the corresponding records as its result.

procedure H2(F,R)
(assumption: F is of the form "$z_i\ \theta\ z_j$" and there exists indices on both fields z_i & z_j of file R)
Read the index on z_i and examine for each value, whether there are values in the other index (on z_j) which meet the criterion. If this is the case, intersect the corresponding sets of RIDs and return the result.

procedure REAL(RIDs,R)
Fetch the blocks of R which contain records identified in the set of RIDs.

procedure RAJ(F,R)
Process file R in the order it is accessed, apply function F on each record and write the records that meet the criterion as its the result.

procedure SORT(z,R)
This procedure sorts file R on field z using the methods supported by the operating system.

procedure I1(RIDs1,RIDs2)
Efficient merge procedure for the intersection operation supported by the operating system.

procedure U1(RIDs1,RIDs2)
Efficient merge procedure for the union operation supported by the operating system.

7.3.2 Examples

Example1

Suppose we have the `LawStudents` relation with indices on the `studentnr` and `name` attributes. The file in which the `LawStudents` records are stored is sorted on the `studentnr` attribute. Assume that a user is interested in all `LawStudents` with `name` is "Smith" and `phone` greater than 123456. We can find these using the following scheme and query:

```
scheme
   "LawStudents" == attributes: < studentnr,name,phone >
```

```
                    sortedon: < studentnr >
                    indexon: < studentnr,name >
query
  Select(name = "Smith" and phone > 123456, "LawStudents")
```

After applying the Check, Transform and ReadInfo functions to this query we get the following result:

```
Select( name = "Smith" and phone > 123456,
   "LawStudents" < studentnr,name,phone > < studentnr > < studentnr,name >,
   < >, < >)
```

In the above temporary result we see two empty attributelists. These two lists serve to store information about possible sort orders and index usage during the up- and down-pass. If we apply the Access function we get the following result:

```
Procedure "RAJ" ( phone > 123456 ,
   Procedure "REAL" ( [ Procedure "H1" ( name = "Smith" ,
                              [ Index on < name > of "LawStudents" ] ) ] ,
                      [ "LawStudents" ] ) ) ]
```

We can use the index on the name attribute of the LawStudents relation using procedure H1. There exists no index on the phone attribute, so we have to apply procedure RAJ to test the records of the LawStudents file on this subpredicate. See Figure 7.1 for a graphical representation of this example.

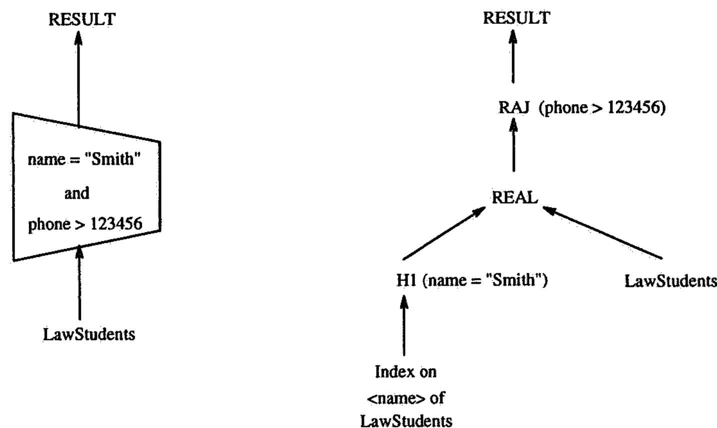

Figure 7.1: Example 1, choice of procedures of select operation

Example2

Using the scheme information of example 1 we want to evaluate the following query:

```
query
   Select(name = "Smith" or
          name = "Jones" and phone > 123456, "LawStudents")
```

The result of the function Fac will be:

```
(name = "Smith" or name = "Jones" , name = "Smith" or phone > 123456 )
```

Here we see that subpredicate name = "Smith" appears in the residue of the factorisation function though an index exists for the name attribute. The reason to do this is to restrict the number of records as early as possible by selecting records with value "Smith" or value "Jones" in the name-field using an index. Then among the those records another selection is made; if the value in the phone-field is greater than 123456 or the value in the name-field is "Smith" the records are selected. This last selection has to be done without the use of indices. The chosen procedures are (see Figure 7.2 for a graphical representation of this example):

```
[ Procedure "RAJ" ( name = "Smith" or phone > 123456,
  Procedure "REAL" ( [ Procedure "U1" (
     [ Procedure "H1" ( name = "Smith" ,
       [ Index on < name > of "LawStudents" ] ) ] ,
     [ Procedure "H1" ( name = "Jones" ,
       [ Index on < name > of "LawStudents" ] ) ] ) ],
  [ "LawStudents" ] ) ) ]
```

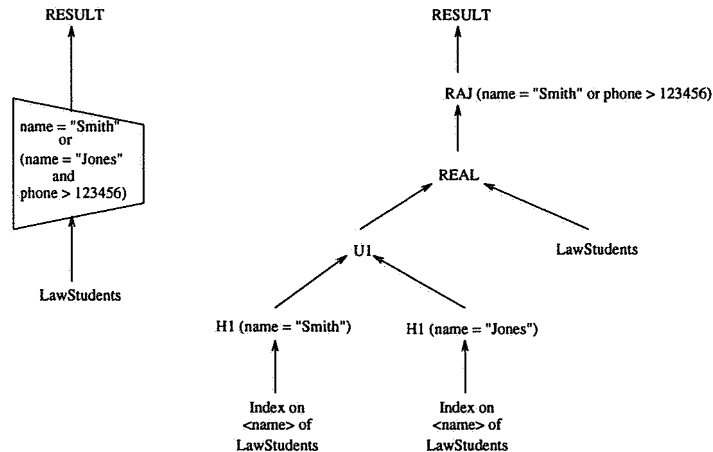

Figure 7.2: Example 2, choice of procedures of select operation

Chapter 8

Extensions / Further work

Chapter 9

Conclusions

Appendix A

A.1 Access

imports Factorisation$^{(A.4)}$
exports
sorts EXTREL PLAN
context-free syntax

"*Process*" "(" EXTREL ")"	$\rightarrow$ PLAN
"*Up*" "(" EXTREL ")"	$\rightarrow$ EXTREL
"*Down*" "(" EXTREL "," ATTRIBUTELIST "," ATTRIBUTELIST ")"	$\rightarrow$ PLAN
REL ATTRIBUTELIST ATTRIBUTELIST	$\rightarrow$ EXTREL
"*Union*" "(" EXTREL "," EXTREL "," ATTRIBUTELIST "," ATTRIBUTELIST ")"	$\rightarrow$ EXTREL
"*Intersection*" "(" EXTREL "," EXTREL "," ATTRIBUTELIST "," ATTRIBUTELIST ")"	$\rightarrow$ EXTREL
"*Minus*" "(" EXTREL "," EXTREL "," ATTRIBUTELIST "," ATTRIBUTELIST ")"	$\rightarrow$ EXTREL
"*Product*" "(" EXTREL "," EXTREL "," ATTRIBUTELIST "," ATTRIBUTELIST ")"	$\rightarrow$ EXTREL
"*Select*" "(" SELPRED "," EXTREL "," ATTRIBUTELIST "," ATTRIBUTELIST ")"	$\rightarrow$ EXTREL
"*Project*" "(" ATTRIBUTELIST "," EXTREL "," ATTRIBUTELIST "," ATTRIBUTELIST ")"	$\rightarrow$ EXTREL
"*Join*" "(" JOINPRED "," EXTREL "," EXTREL "," ATTRIBUTELIST "," ATTRIBUTELIST ")"	$\rightarrow$ EXTREL
"[" {PLAN ","}* "]"	$\rightarrow$ PLAN
RELNAME	$\rightarrow$ PLAN
"*Index␣on*" ATTRIBUTELIST "*of*" RELNAME	$\rightarrow$ PLAN
"*Procedure*" STRING	$\rightarrow$ PLAN
"*Procedure*" STRING "(" SELPRED "," PLAN ")"	$\rightarrow$ PLAN
"*Procedure*" STRING "(" PLAN "," PLAN ")"	$\rightarrow$ PLAN
"*Procedure*" STRING "(" ATTRIBUTE "," PLAN ")"	$\rightarrow$ PLAN
"*RewritePred*" "(" SELPRED "," EXTREL ")"	$\rightarrow$ PLAN

"*GetSorted*" "(" EXTREL ")" → ATTRIBUTELIST
"*GetIndex*" "(" EXTREL ")" → ATTRIBUTELIST
"*IsLeave*" "(" EXTREL ")" → BOOL

variables

Extrel [*0-9*]* → EXTREL
Sortedon [*0-9*]* → ATTRIBUTELIST
Indexon [*0-9*]* → ATTRIBUTELIST

equations

main function:

[ACCESS1] $Process(Extrel) \ = \ Down(\,Up(Extrel), <\ >, <\ >)$

Up-pass for leave-relation:

[ACCESS2] $Up(Relname\ Attributelist\ Sortedon\ Indexon) \ =$
$Relname\ Attributelist\ Sortedon\ Indexon$

Down-pass for leave-relation:

$$\text{[ACCESS3a]}\ \frac{Indexon_1 \neq <\ >}{\begin{array}{l} Down(Relname\ Attributelist\ Sortedon\ Indexon,\ Sortedon_1,\ Indexon_1) \ = \\ [Index_{\sqcup}on\ Indexon_1\ \ of\ Relname] \end{array}}$$

[ACCESS3b] $Down(Relname\ Attributelist\ Sortedon\ Indexon,\ Sortedon_1,\ <\ >) \ =$
$[Relname]$

Up-pass for select-operation:

$$\text{[ACCESS4]}\ \frac{Extrel_1 \ = \ Up(Extrel)}{\begin{array}{l} Up(\,Select(\,Selpred,\ Extrel,\ Sortedon,\ Indexon)) \ = \\ Select(\,Selpred,\ Extrel_1,\ GetSorted(Extrel_1),\ Indexon) \end{array}}$$

Down-pass for select-operation: no indices exist:

$$\text{[ACCESS5a]}\ \frac{\begin{array}{c} IsLeave(Extrel) = true, \\ Indexon_2 = GetIndex(Extrel), \\ Indexon_2 = <\ > \end{array}}{\begin{array}{l} Down(\,Select(\,Selpred,\ Extrel,\ Sortedon,\ Indexon),\ Sortedon_1,\ Indexon_1) \ = \\ [Procedure\ \text{"RAJ"}(\,Selpred,\ Down(Extrel,\ Sortedon,\ Indexon))] \end{array}}$$

no indices are of use:

$$\text{[ACCESS5b]}\ \frac{\begin{array}{c} IsLeave(Extrel) = true, \\ Indexon_2 = GetIndex(Extrel), \\ Indexon_2 \neq <\ >, \\ (Selpred_1,\ Selpred_2) = Fac(Selpred,\ Indexon_2), \\ Selpred_1 \ = NULL \end{array}}{\begin{array}{l} Down(\,Select(\,Selpred,\ Extrel,\ Sortedon,\ Indexon),\ Sortedon_1,\ Indexon_1) \ = \\ [Procedure\ \text{"RAJ"}(\,Selpred,\ Down(Extrel,\ Sortedon,\ Indexon))] \end{array}}$$

part of predicate can be evaluated using indices:

$$[\text{ACCESS5c}]\ \frac{\begin{array}{c} IsLeave(Extrel) = true, \\ Indexon_2 = GetIndex(Extrel), \\ Indexon_2 \neq < \ >, \\ (Selpred_1, Selpred_2) = Fac(Selpred, Indexon_2), \\ Selpred_1 \neq NULL, \\ Selpred_2 \neq NULL \end{array}}{\begin{array}{l} Down(Select(Selpred, Extrel, Sortedon, Indexon), Sortedon_1, Indexon_1) = \\ [Procedure \text{ "RAJ"} (Selpred_2, Procedure \text{ "REAL"} \\ \qquad (RewritePred(Selpred_1, Extrel), \\ \qquad Down(Extrel, Sortedon, < \ >)))] \end{array}}$$

whole predicate can be evaluated using indices:

$$[\text{ACCESS5d}]\ \frac{\begin{array}{c} IsLeave(Extrel) = true, \\ Indexon_2 = GetIndex(Extrel), \\ Indexon_2 \neq < \ >, \\ (Selpred_1, Selpred_2) = Fac(Selpred, Indexon_2), \\ Selpred_1 \neq NULL, \\ Selpred_2 = NULL \end{array}}{\begin{array}{l} Down(Select(Selpred, Extrel, Sortedon, Indexon), Sortedon_1, Indexon_1) = \\ [Procedure \text{ "REAL"} \\ \qquad (RewritePred(Selpred_1, Extrel), \\ \qquad Down(Extrel, Sortedon, < \ >))] \end{array}}$$

no indices are of use:

$$[\text{ACCESS5e}]\ \frac{\begin{array}{c} IsLeave(Extrel) = false, \\ Sortedon_2 = Intersect(Sortedon, < Attribute, Attributes >), \\ Sortedon_2 = < \ > \end{array}}{\begin{array}{l} Down(Select(Selpred, Extrel, Sortedon, Indexon), \\ \qquad < Attribute, Attributes > , Indexon_1) \quad = \\ [Procedure \text{ "SORT"} (Attribute, \\ \qquad [Procedure \text{ "RAJ"} (Selpred, Down(Extrel, Sortedon, Indexon))])] \end{array}}$$

no indices are of use:

$$[\text{ACCESS5f}]\ \frac{\begin{array}{c} IsLeave(Extrel) = false, \\ Sortedon_2 = Intersect(Sortedon, Sortedon_1), \\ Sortedon_2 \neq < \ > \end{array}}{\begin{array}{l} Down(Select(Selpred, Extrel, Sortedon, Indexon), Sortedon_1, Indexon_1) = \\ [Procedure \text{ "RAJ"} (Selpred, Down(Extrel, Sortedon_2, Indexon))] \end{array}}$$

Up-pass for other operations [ACCESS6] nog uitwerken Down-pass for other operations [ACCESS7] nog uitwerken Function RewritePred. Determine which select-procedure we should choose:

$$[\text{R1}]\ RewritePred(Attribute\ Op\ Str, Extrel) =$$
$$[Procedure \text{ "H1"} (Attribute\ Op\ Str, Down(Extrel, < \ > , < Attribute >))]$$

[R2] *RewritePred*(*Attribute Op Int*, *Extrel*) =
[*Procedure* "*H1*"(*Attribute Op Int*, *Down*(*Extrel*, < > , < *Attribute* >))]

[R3] *RewritePred*($Attribute_1$ *Op* $Attribute_2$, *Extrel*) =
[*Procedure* "*H2*"($Attribute_1$ *Op* $Attribute_2$, *Down*(*Extrel*, < > , < $Attribute_1$, $Attribute_2$ >))]

[R4] *RewritePred*($Selpred_1$ *and* $Selpred_2$, *Extrel*) =
[*Procedure* "*I1*"(*RewritePred*($Selpred_1$, *Extrel*), *RewritePred*($Selpred_2$, *Extrel*))]

[R5] *RewritePred*($Selpred_1$ *or* $Selpred_2$, *Extrel*) =
[*Procedure* "*U1*"(*RewritePred*($Selpred_1$, *Extrel*), *RewritePred*($Selpred_2$, *Extrel*))]

Function GetSorted: get Sortedon attributelist from an EXTREL

[G1] *GetSorted*(*Relname Attributelist Sortedon Indexon*) = *Sortedon*
[G2] *GetSorted*(*Select*(*Selpred*, *Extrel*, *Sortedon*, *Indexon*)) = *Sortedon*

Function GetIndex: get Indexon attributelist from an EXTREL

[G1] *GetIndex*(*Relname Attributelist Sortedon Indexon*) = *Indexon*
[G2] *GetIndex*(*Extrel*) = < > **otherwise**

IsLeave function for EXTREL

[I1] *IsLeave*(*Relname Attributelist Sortedon Indexon*) = *true*
[I2] *IsLeave*(*Extrel*) = *false* **otherwise**

A.2 Check

imports Operations(A.5) Relations(A.7)
exports
sorts MSG
context-free syntax

"*Check*" "(" REL ")"	→ REL
"*Chck*" "(" REL ")"	→ REL
good RELNAME ATTRIBUTELIST	→ REL
REL "*error(s)*" MSG	→ REL
MSG	→ REL
MSG "*&&*" MSG	→ MSG
"*The␣attribute*" ATTRIBUTE "*of␣the␣projectlist␣is␣not␣defined␣in␣relation*" RELNAME ATTRIBUTELIST	→ MSG

"*The␣attributes*" ATTRIBUTELIST
"*of␣the␣projectlist␣are␣not␣defined␣in␣relation*"
RELNAME ATTRIBUTELIST → MSG

"*The␣attribute*" ATTRIBUTE
"*used␣in␣selection-predicate␣is␣not␣an␣attribute␣of␣relation*"
RELNAME ATTRIBUTELIST → MSG

"*Relation*" RELNAME "*not␣defined*" → MSG

"*The␣attribute*" ATTRIBUTE
"*used␣in␣join-predicate␣is␣not␣an␣attribute␣of␣relation*"
RELNAME ATTRIBUTELIST → MSG

"*Relation*" RELNAME ATTRIBUTELIST "*and␣relation*"
RELNAME ATTRIBUTELIST
"*of␣the␣union-operation␣are␣not␣union-compatible*" → MSG

"*Relation*" RELNAME ATTRIBUTELIST "*and␣relation*"
RELNAME ATTRIBUTELIST
"*of␣the␣intersection-operation␣are␣not␣union-compatible*" → MSG

"*Relation*" RELNAME ATTRIBUTELIST "*and␣relation*"
RELNAME ATTRIBUTELIST
"*of␣the␣minus-operation␣are␣not␣union-compatible*" → MSG

"*ConcatName*" "(" RELNAME "," RELNAME ")" → RELNAME
RELNAME "@" RELNAME → RELNAME

"*NecName*" "(" RELNAME ATTRIBUTELIST ","
RELNAME ATTRIBUTELIST ")" → ATTRIBUTELIST
"*NewList*" "(" RELNAME ATTRIBUTELIST "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"*OptName*" "(" RELNAME "," ATTRIBUTELIST ")" → ATTRIBUTELIST

"*Single*" "(" RELNAME "," ATTRIBUTE ")" → ATTRIBUTE
RELNAME "." ATTRIBUTE → ATTRIBUTE
"*Double*" "(" RELNAME "," ATTRIBUTE ")" → ATTRIBUTE
RELNAME ".." ATTRIBUTE → ATTRIBUTE

"*HasName*" "(" ATTRIBUTE ")" → BOOL
"*HasSingleName*" "(" ATTRIBUTE ")" → BOOL
"*HasDoubleName*" "(" ATTRIBUTE ")" → BOOL

"*CheckElemOf*" "(" ATTRIBUTE "," ATTRIBUTELIST ")" → BOOL
"*CheckList*" "(" ATTRIBUTELIST "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"*SubList*" "(" ATTRIBUTELIST "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"*Sub*" "(" ATTRIBUTE "," ATTRIBUTELIST ")" → ATTRIBUTELIST

"*RName*" "(" ATTRIBUTE ")" → ATTRIBUTE
"*RemName*" "(" ATTRIBUTELIST ")" → ATTRIBUTELIST

"*SingleToDouble*" "(" ATTRIBUTE ")" → ATTRIBUTE

"*RDouble*" "(" ATTRIBUTE ")" → ATTRIBUTE
"*RemDouble*" "(" ATTRIBUTELIST ")" → ATTRIBUTELIST

"*RDoubleName*" "(" ATTRIBUTE ")" → ATTRIBUTE
"*RemDoubleName*" "(" ATTRIBUTELIST ")" → ATTRIBUTELIST

"*Normalize*" "(" REL ")" → REL
"*NormList*" "(" REL ")" → ATTRIBUTELIST
"*NewName*" "(" ATTRIBUTELIST "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"*CheckName*" "(" REL "," ATTRIBUTELIST ")" → REL
"*CheckAttribute*" "(" ATTRIBUTE "," ATTRIBUTELIST ")" → ATTRIBUTE
"*CheckAttributes*" "(" ATTRIBUTELIST "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"*CheckPred*" "(" SELPRED "," ATTRIBUTELIST ")" → SELPRED

variables

Chars [*123*]→ CHAR∗
Msg [*0-9*]∗ → MSG

equations

Check function: if the query/relation is correct the normalized query/relation is returned, otherwise a message explaining the error is returned.

$$[1]\ \frac{Chck(Rel) = good\ Relname\ Attributelist}{Check(Rel) = Normalize(Rel)}$$

$$[2]\ \frac{Chck(Rel) = Msg}{Check(Rel) = Msg}$$

Check(Rel) = Rel error(s) Msg

$$[3]\ \frac{Attributelist \neq <\ >}{Chck(Relname\ Attributelist) = good\ Relname\ Attributelist}$$

$$[4]\ Chck(Relname <\ >) = Relation\ Relname\ not{\sqcup}defined$$

$$
\begin{array}{lll}
[5] & Chck(Union(Rel_1, Rel_2)) & = Union(Chck(Rel_1), Chck(Rel_2)) \\
[6] & Chck(Intersection(Rel_1, Rel_2)) & = Intersection(Chck(Rel_1), Chck(Rel_2)) \\
[7] & Chck(Minus(Rel_1, Rel_2)) & = Minus(Chck(Rel_1), Chck(Rel_2)) \\
[8] & Chck(Product(Rel_1, Rel_2)) & = Product(Chck(Rel_1), Chck(Rel_2)) \\
[9] & Chck(Select(Selpred, Rel)) & = Select(Selpred, Chck(Rel)) \\
[10] & Chck(Project(Projectlist, Rel)) & = Project(Projectlist, Chck(Rel)) \\
[11] & Chck(Join(Joinpred, Rel_1, Rel_2)) & = Join(Joinpred, Chck(Rel_1), Chck(Rel_2))
\end{array}
$$

Check on Union/Intersection/Minus: Check if the used relations are Union-compatible.

$$[UIM1]\ \frac{\begin{array}{r}Equal(RemDoubleName(Attributelist_1),\ RemDoubleName(Attributelist_2)) = true,\\ Relname = ConcatName(Relname_1, Relname_2),\\ Attributelist_3 = OptName(Relname_1, Attributelist_1),\\ Attributelist_4 = OptName(Relname_2, Attributelist_2),\\ Attributelist_5 = Add(Attributelist_3, Attributelist_4)\end{array}}{\begin{array}{l}Union(good\ Relname_1\ Attributelist_1,\ good\ Relname_2\ Attributelist_2) =\\ good\ Relname\ Attributelist_5\end{array}}$$

$$[UIM2]\ \frac{Equal(RemDoubleName(Attributelist_1),\ RemDoubleName(Attributelist_2)) = false}{\begin{array}{l}Union(good\ Relname_1\ Attributelist_1,\ good\ Relname_2\ Attributelist_2) =\\ Relation\ Relname_1\ Attributelist_1\\ and_relation\ Relname_2\ Attributelist_2\ of_the_union\text{-}operation_are_not_union\text{-}compatible\end{array}}$$

$$[UIM3]\ \frac{\begin{array}{r}Equal(RemDoubleName(Attributelist_1),\ RemDoubleName(Attributelist_2)) = true,\\ Relname = ConcatName(Relname_1, Relname_2),\\ Attributelist_3 = OptName(Relname_1, Attributelist_1),\\ Attributelist_4 = OptName(Relname_2, Attributelist_2),\\ Attributelist_5 = Add(Attributelist_3, Attributelist_4)\end{array}}{\begin{array}{l}Intersection(good\ Relname_1\ Attributelist_1,\ good\ Relname_2\ Attributelist_2) =\\ good\ Relname\ Attributelist_5\end{array}}$$

$$[UIM4]\ \frac{Equal(RemDoubleName(Attributelist_1),\ RemDoubleName(Attributelist_2)) = false}{\begin{array}{l}Intersection(good\ Relname_1\ Attributelist_1,\ good\ Relname_2\ Attributelist_2) =\\ Relation\ Relname_1\ Attributelist_1\\ and_relation\ Relname_2\ Attributelist_2\ of_the_intersection\text{-}operation_are_not_union\text{-}compati\end{array}}$$

$$[UIM5]\ \frac{\begin{array}{r}Equal(RemDoubleName(Attributelist_1),\ RemDoubleName(Attributelist_2)) = true,\\ Relname = ConcatName(Relname_1, Relname_2),\\ Attributelist_3 = OptName(Relname_1, Attributelist_1),\\ Attributelist_4 = OptName(Relname_2, Attributelist_2),\\ Attributelist_5 = Add(Attributelist_3, Attributelist_4)\end{array}}{\begin{array}{l}Minus(good\ Relname_1\ Attributelist_1,\ good\ Relname_2\ Attributelist_2) =\\ good\ Relname\ Attributelist_5\end{array}}$$

$$[UIM6]\ \frac{Equal(RemDoubleName(Attributelist_1),\ RemDoubleName(Attributelist_2)) = false}{\begin{array}{l}Minus(good\ Relname_1\ Attributelist_1,\ good\ Relname_2\ Attributelist_2) =\\ Relation\ Relname_1\ Attributelist_1\\ and_relation\ Relname_2\ Attributelist_2\ of_the_minus\text{-}operation_are_not_union\text{-}compatible\end{array}}$$

$$
[\mathrm{P1}]\ \frac{\begin{array}{c}\mathit{Relname} = \mathit{ConcatName}(\mathit{Relname}_1, \mathit{Relname}_2),\\ \mathit{Attributelist}_3 = \mathit{NecName}(\mathit{Relname}_1\ \mathit{Attributelist}_1, \mathit{Relname}_2\ \mathit{Attributelist}_2)\end{array}}{\begin{array}{l}\mathit{Product}(\mathit{good}\ \mathit{Relname}_1\ \mathit{Attributelist}_1, \mathit{good}\ \mathit{Relname}_2\ \mathit{Attributelist}_2) =\\ \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}_3\end{array}}
$$

Check on Select: check if the attributes used in the selection-predicate are either attributes of the attributelist of the operand relation or attributes of the extended operand relation.

$$
[\mathrm{SEL1}]\ \frac{\mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{true}}{\begin{array}{l}\mathit{Select}(\mathit{Attribute}\ \mathit{Op}\ \mathit{Str}, \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}) =\\ \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}\end{array}}
$$

$$
[\mathrm{SEL2}]\ \frac{\mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{false}}{\begin{array}{l}\mathit{Select}(\mathit{Attribute}\ \mathit{Op}\ \mathit{Str}, \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}) =\\ \mathit{The{\sqcup}attribute}\ \mathit{Attribute}\\ \mathit{used{\sqcup}in{\sqcup}selection\text{-}predicate{\sqcup}is{\sqcup}not{\sqcup}an{\sqcup}attribute{\sqcup}of{\sqcup}relation}\ \mathit{Relname}\ \mathit{Attributelist}\end{array}}
$$

$$
[\mathrm{SEL3}]\ \frac{\mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{true}}{\mathit{Select}(\mathit{Attribute}\ \mathit{Op}\ \mathit{Int}, \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}) = \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}}
$$

$$
[\mathrm{SEL4}]\ \frac{\mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{false}}{\begin{array}{l}\mathit{Select}(\mathit{Attribute}\ \mathit{Op}\ \mathit{Int}, \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}) =\\ \mathit{The{\sqcup}attribute}\ \mathit{Attribute}\\ \mathit{used{\sqcup}in{\sqcup}selection\text{-}predicate{\sqcup}is{\sqcup}not{\sqcup}an{\sqcup}attribute{\sqcup}of{\sqcup}relation}\ \mathit{Relname}\ \mathit{Attributelist}\end{array}}
$$

$$
[\mathrm{SEL5}]\ \frac{\begin{array}{c}\mathit{CheckElemOf}(\mathit{Attribute}_1, \mathit{Attributelist}) = \mathit{true},\\ \mathit{CheckElemOf}(\mathit{Attribute}_2, \mathit{Attributelist}) = \mathit{true}\end{array}}{\begin{array}{l}\mathit{Select}(\mathit{Attribute}_1\ \mathit{Op}\ \mathit{Attribute}_2, \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}) =\\ \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}\end{array}}
$$

$$
[\mathrm{SEL6}]\ \frac{\begin{array}{c}\mathit{CheckElemOf}(\mathit{Attribute}_1, \mathit{Attributelist}) = \mathit{false},\\ \mathit{CheckElemOf}(\mathit{Attribute}_2, \mathit{Attributelist}) = \mathit{true}\end{array}}{\begin{array}{l}\mathit{Select}(\mathit{Attribute}_1\ \mathit{Op}\ \mathit{Attribute}_2, \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}) =\\ \mathit{The{\sqcup}attribute}\ \mathit{Attribute}_1\\ \mathit{used{\sqcup}in{\sqcup}selection\text{-}predicate{\sqcup}is{\sqcup}not{\sqcup}an{\sqcup}attribute{\sqcup}of{\sqcup}relation}\ \mathit{Relname}\ \mathit{Attributelist}\end{array}}
$$

$$
[\mathrm{SEL7}]\ \frac{\begin{array}{c}\mathit{CheckElemOf}(\mathit{Attribute}_1, \mathit{Attributelist}) = \mathit{true},\\ \mathit{CheckElemOf}(\mathit{Attribute}_2, \mathit{Attributelist}) = \mathit{false}\end{array}}{\begin{array}{l}\mathit{Select}(\mathit{Attribute}_1\ \mathit{Op}\ \mathit{Attribute}_2, \mathit{good}\ \mathit{Relname}\ \mathit{Attributelist}) =\\ \mathit{The{\sqcup}attribute}\ \mathit{Attribute}_2\\ \mathit{used{\sqcup}in{\sqcup}selection\text{-}predicate{\sqcup}is{\sqcup}not{\sqcup}an{\sqcup}attribute{\sqcup}of{\sqcup}relation}\ \mathit{Relname}\ \mathit{Attributelist}\end{array}}
$$

$$[\mathrm{SEL8}]\ \frac{\begin{array}{c}CheckElemOf(Attribute_1,\ Attributelist) = false,\\ CheckElemOf(Attribute_2,\ Attributelist) = false\end{array}}{\begin{array}{l}Select(Attribute_1\ Op\ Attribute_2,\ good\ Relname\ Attributelist)\ =\\ The\textvisiblespace attribute\ Attribute_1\\ used\textvisiblespace in\textvisiblespace selection\text{-}predicate\textvisiblespace is\textvisiblespace not\textvisiblespace an\textvisiblespace attribute\textvisiblespace of\textvisiblespace relation\ Relname\ Attributelist\\ \&\&\ The\textvisiblespace attribute\ Attribute_2\\ \quad used\textvisiblespace in\textvisiblespace selection\text{-}predicate\textvisiblespace is\textvisiblespace not\textvisiblespace an\textvisiblespace attribute\textvisiblespace of\textvisiblespace relation\ Relname\ Attributelist\end{array}}$$

$$[\mathrm{SEL9}]\ \frac{\begin{array}{l}Select(Selpred_1,\ good\ Relname\ Attributelist) = good\ Relname\ Attributelist,\\ Select(Selpred_2,\ good\ Relname\ Attributelist) = good\ Relname\ Attributelist\end{array}}{\begin{array}{l}Select(Selpred_1\ and\ Selpred_2,\ good\ Relname\ Attributelist)\ =\\ good\ Relname\ Attributelist\end{array}}$$

$$[\mathrm{SEL10}]\ \frac{\begin{array}{r}Msg = Select(Selpred_1,\ good\ Relname\ Attributelist),\\ Select(Selpred_2,\ good\ Relname\ Attributelist) = good\ Relname\ Attributelist\end{array}}{Select(Selpred_1\ and\ Selpred_2,\ good\ Relname\ Attributelist) = Msg}$$

$$[\mathrm{SEL11}]\ \frac{\begin{array}{r}Msg = Select(Selpred_2,\ good\ Relname\ Attributelist),\\ Select(Selpred_1,\ good\ Relname\ Attributelist) = good\ Relname\ Attributelist\end{array}}{Select(Selpred_1\ and\ Selpred_2,\ good\ Relname\ Attributelist) = Msg}$$

$$[\mathrm{SEL12}]\ \frac{\begin{array}{l}Msg_1 = Select(Selpred_1,\ good\ Relname\ Attributelist),\\ Msg_2 = Select(Selpred_2,\ good\ Relname\ Attributelist)\end{array}}{\begin{array}{l}Select(Selpred_1\ and\ Selpred_2,\ good\ Relname\ Attributelist)\ =\\ Msg_1\ \&\&\ Msg_2\end{array}}$$

$$[\mathrm{SEL13}]\ \frac{\begin{array}{l}Select(Selpred_1,\ good\ Relname\ Attributelist) = good\ Relname\ Attributelist,\\ Select(Selpred_2,\ good\ Relname\ Attributelist) = good\ Relname\ Attributelist\end{array}}{\begin{array}{l}Select(Selpred_1\ or\ Selpred_2,\ good\ Relname\ Attributelist)\ =\\ good\ Relname\ Attributelist\end{array}}$$

$$[\mathrm{SEL14}]\ \frac{\begin{array}{r}Msg = Select(Selpred_1,\ good\ Relname\ Attributelist),\\ Select(Selpred_2,\ good\ Relname\ Attributelist) = good\ Relname\ Attributelist\end{array}}{Select(Selpred_1\ or\ Selpred_2,\ good\ Relname\ Attributelist) = Msg}$$

$$[\mathrm{SEL15}]\ \frac{\begin{array}{r}Msg = Select(Selpred_2,\ good\ Relname\ Attributelist),\\ Select(Selpred_1,\ good\ Relname\ Attributelist) = good\ Relname\ Attributelist\end{array}}{Select(Selpred_1\ or\ Selpred_2,\ good\ Relname\ Attributelist) = Msg}$$

$$[\mathrm{SEL16}]\ \frac{\begin{array}{l}Msg_1 = Select(Selpred_1,\ good\ Relname\ Attributelist),\\ Msg_2 = Select(Selpred_2,\ good\ Relname\ Attributelist)\end{array}}{\begin{array}{l}Select(Selpred_1\ or\ Selpred_2,\ good\ Relname\ Attributelist)\ =\\ Msg_1\ \&\&\ Msg_2\end{array}}$$

Check on Project: check if the attributes that are projected upon are attributes of the attributelist of the operand relation.

$$[\text{PRO1}]\ \frac{\begin{array}{l} \mathit{CheckList}(\mathit{Projectlist},\ \mathit{Attributelist}) = <\ >, \\ \qquad \mathit{Projectlist}_1 = \mathit{SubList}(\mathit{Projectlist},\ \mathit{Attributelist}) \end{array}}{\begin{array}{l} \mathit{Project}(\mathit{Projectlist},\ \textit{good}\ \mathrm{Relname}\ \mathit{Attributelist}) = \\ \textit{good}\ \mathrm{Relname}\ \mathit{Projectlist}_1 \end{array}}$$

$$[\text{PRO2}]\ \frac{\begin{array}{l} \mathit{Projectlist}_1 = \mathit{CheckList}(\mathit{Projectlist},\ \mathit{Attributelist}), \\ \mathit{Projectlist}_1 = <\mathit{Attribute}> \end{array}}{\begin{array}{l} \mathit{Project}(\mathit{Projectlist},\ \textit{good}\ \mathrm{Relname}\ \mathit{Attributelist}) = \\ \textit{The\textvisiblespace attribute}\ \mathit{Attribute} \\ \textit{of\textvisiblespace the\textvisiblespace projectlist\textvisiblespace is\textvisiblespace not\textvisiblespace defined\textvisiblespace in\textvisiblespace relation}\ \mathrm{Relname}\ \mathit{Attributelist} \end{array}}$$

$$[\text{PRO3}]\ \frac{\begin{array}{l} \mathit{Projectlist}_1 = \mathit{CheckList}(\mathit{Projectlist},\ \mathit{Attributelist}), \\ \mathit{Projectlist}_1 = <\mathit{Attribute}_1,\ \mathit{Attribute}_2,\ \mathit{Attributes}> \end{array}}{\begin{array}{l} \mathit{Project}(\mathit{Projectlist},\ \textit{good}\ \mathrm{Relname}\ \mathit{Attributelist}) = \\ \textit{The\textvisiblespace attributes}\ \mathit{Projectlist}_1 \\ \textit{of\textvisiblespace the\textvisiblespace projectlist\textvisiblespace are\textvisiblespace not\textvisiblespace defined\textvisiblespace in\textvisiblespace relation}\ \mathrm{Relname}\ \mathit{Attributelist} \end{array}}$$

Check on Join: check if the left attribute is an attribute of the first relation and the right one an attribute of the second relation.

$$[\text{JOI1}]\ \frac{\begin{array}{r} \mathit{Attributelist}_3 = \mathit{NewList}(\mathrm{Relname}_1\ \mathit{Attributelist}_1,\ \mathit{Attributelist}_2 \\ \mathit{Attributelist}_4 = \mathit{NewList}(\mathrm{Relname}_2\ \mathit{Attributelist}_2,\ \mathit{Attributelist}_1 \\ \mathit{CheckElemOf}(\mathit{Attribute}_1,\ \mathit{Attributelist}_3) = \textit{true}, \\ \mathit{CheckElemOf}(\mathit{Attribute}_2,\ \mathit{Attributelist}_4) = \textit{true}, \\ \mathrm{Relname} = \mathit{ConcatName}(\mathrm{Relname}_1,\ \mathrm{Relname}_2), \\ \mathit{Attributelist}_5 = \mathit{Add}(\mathit{Attributelist}_3,\ \mathit{Attributelist}_4) \end{array}}{\begin{array}{l} \mathit{Join}(\mathit{Attribute}_1 = \mathit{Attribute}_2,\ \textit{good}\ \mathrm{Relname}_1\ \mathit{Attributelist}_1,\ \textit{good}\ \mathrm{Relname}_2\ \mathit{Attributelist}_2) \\ \textit{good}\ \mathrm{Relname}\ \mathit{Attributelist}_5 \end{array}}$$

$$[\text{JOI2}]\ \frac{\begin{array}{r} \mathit{Attributelist}_3 = \mathit{NewList}(\mathrm{Relname}_1\ \mathit{Attributelist}_1,\ \mathit{Attributelist}_2 \\ \mathit{Attributelist}_4 = \mathit{NewList}(\mathrm{Relname}_2\ \mathit{Attributelist}_2,\ \mathit{Attributelist}_1 \\ \mathit{CheckElemOf}(\mathit{Attribute}_1,\ \mathit{Attributelist}_3) = \textit{false}, \\ \mathit{CheckElemOf}(\mathit{Attribute}_2,\ \mathit{Attributelist}_4) = \textit{true} \end{array}}{\begin{array}{l} \mathit{Join}(\mathit{Attribute}_1 = \mathit{Attribute}_2,\ \textit{good}\ \mathrm{Relname}_1\ \mathit{Attributelist}_1,\ \textit{good}\ \mathrm{Relname}_2\ \mathit{Attributelist}_2) \\ \textit{The\textvisiblespace attribute}\ \mathit{Attribute}_1 \\ \textit{used\textvisiblespace in\textvisiblespace join-predicate\textvisiblespace is\textvisiblespace not\textvisiblespace an\textvisiblespace attribute\textvisiblespace of\textvisiblespace relation}\ \mathrm{Relname}_1\ \mathit{Attributelist}_3 \end{array}}$$

$$[\text{JOI3}]\ \frac{\begin{array}{r} \mathit{Attributelist}_3 = \mathit{NewList}(\mathrm{Relname}_1\ \mathit{Attributelist}_1,\ \mathit{Attributelist}_2 \\ \mathit{Attributelist}_4 = \mathit{NewList}(\mathrm{Relname}_2\ \mathit{Attributelist}_2,\ \mathit{Attributelist}_1 \\ \mathit{CheckElemOf}(\mathit{Attribute}_1,\ \mathit{Attributelist}_3) = \textit{true}, \\ \mathit{CheckElemOf}(\mathit{Attribute}_2,\ \mathit{Attributelist}_4) = \textit{false} \end{array}}{\begin{array}{l} \mathit{Join}(\mathit{Attribute}_1 = \mathit{Attribute}_2,\ \textit{good}\ \mathrm{Relname}_1\ \mathit{Attributelist}_1,\ \textit{good}\ \mathrm{Relname}_2\ \mathit{Attributelist}_2) \\ \textit{The\textvisiblespace attribute}\ \mathit{Attribute}_2 \\ \textit{used\textvisiblespace in\textvisiblespace join-predicate\textvisiblespace is\textvisiblespace not\textvisiblespace an\textvisiblespace attribute\textvisiblespace of\textvisiblespace relation}\ \mathrm{Relname}_2\ \mathit{Attributelist}_4 \end{array}}$$

$$\frac{\begin{array}{r} Attributelist_3 = NewList(Relname_1\ Attributelist_1, Attributelist_2) \\ Attributelist_4 = NewList(Relname_2\ Attributelist_2, Attributelist_1) \\ CheckElemOf(Attribute_1, Attributelist_3) = false, \\ CheckElemOf(Attribute_2, Attributelist_4) = false \end{array}}{\begin{array}{l} Join(Attribute_1 = Attribute_2, good\ Relname_1\ Attributelist_1, good\ Relname_2\ Attributelist_2) \\ The\sqcup attribute\ Attribute_1 \\ used\sqcup in\sqcup join\text{-}predicate\sqcup is\sqcup not\sqcup an\sqcup attribute\sqcup of\sqcup relation\ Relname_1\ Attributelist_3 \\ \&\&\ The\sqcup attribute\ Attribute_2 \\ \quad used\sqcup in\sqcup join\text{-}predicate\sqcup is\sqcup not\sqcup an\sqcup attribute\sqcup of\sqcup relation\ Relname_2\ Attributelist_4 \end{array}} \text{[JOI4]}$$

If an error is found return this error (Get the error at the lowest level)

[ERR1] $Union(Msg, Rel) = Msg$
[ERR2] $Union(Rel, Msg) = Msg$
[ERR3] $Intersection(Msg, Rel) = Msg$
[ERR4] $Intersection(Rel, Msg) = Msg$
[ERR5] $Minus(Msg, Rel) = Msg$
[ERR6] $Minus(Msg, Rel) = Msg$
[ERR7] $Product(Msg, Rel) = Msg$
[ERR8] $Product(Rel, Msg) = Msg$
[ERR9] $Select(Selpred, Msg) = Msg$
[ERR10] $Project(Attributelist, Msg) = Msg$
[ERR11] $Join(Joinpred, Msg, Rel) = Msg$
[ERR12] $Join(Joinpred, Rel, Msg) = Msg$

NecName/NewList function: these functions check if relationnames belonging to an attribute are optional or not.

[NEC1] $NecName(Relname_1\ Attributelist_1, Relname_2\ Attributelist_2) = Add(NewList(Relname_1\ Attributelist_1, Attributelist_2), NewList(Relname_2\ Attributelist_2, Attributelist_1))$

[NEW1] $NewList(Relname < > , Attributelist) = < >$

$$\frac{\begin{array}{r} HasName(Attribute) = false, \\ ElemOf(Attribute, RemName(Attributelist)) = false, \\ Attribute_1 = Double(Relname, Attribute) \end{array}}{\begin{array}{l} NewList(Relname < Attribute, Attributes > , Attributelist) = \\ Add(Attribute_1, NewList(Relname < Attributes > , Attributelist)) \end{array}} \text{[NEW2]}$$

$$\frac{\begin{array}{r} HasName(Attribute) = false, \\ ElemOf(Attribute, RemName(Attributelist)) = true, \\ Attribute_1 = Single(Relname, Attribute) \end{array}}{\begin{array}{l} NewList(Relname < Attribute, Attributes > , Attributelist) = \\ Add(Attribute_1, NewList(Relname < Attributes > , Attributelist)) \end{array}} \text{[NEW3]}$$

$$
[\mathrm{NEW4}]\ \frac{\begin{array}{r} \mathit{HasDoubleName}(\mathit{Attribute}) = \mathit{true}, \\ \mathit{ElemOf}(\mathit{RName}(\mathit{Attribute}), \mathit{RemName}(\mathit{Attributelist})) = \mathit{false} \end{array}}{\begin{array}{l} \mathit{NewList}(\mathit{Relname} < \mathit{Attribute}, \mathit{Attributes} > , \mathit{Attributelist}) = \\ \mathit{Add}(\mathit{Attribute}, \mathit{NewList}(\mathit{Relname} < \mathit{Attributes} > , \mathit{Attributelist})) \end{array}}
$$

$$
[\mathrm{NEW5}]\ \frac{\begin{array}{r} \mathit{HasDoubleName}(\mathit{Attribute}) = \mathit{true}, \\ \mathit{ElemOf}(\mathit{RName}(\mathit{Attribute}), \mathit{RemName}(\mathit{Attributelist})) = \mathit{true}, \\ \mathit{Attribute}_1 = \mathit{RDouble}(\mathit{Attribute}) \end{array}}{\begin{array}{l} \mathit{NewList}(\mathit{Relname} < \mathit{Attribute}, \mathit{Attributes} > , \mathit{Attributelist}) = \\ \mathit{Add}(\mathit{Attribute}_1, \mathit{NewList}(\mathit{Relname} < \mathit{Attributes} > , \mathit{Attributelist})) \end{array}}
$$

$$
[\mathrm{NEW6}]\ \frac{\mathit{HasSingleName}(\mathit{Attribute}) = \mathit{true}}{\begin{array}{l} \mathit{NewList}(\mathit{Relname} < \mathit{Attribute}, \mathit{Attributes} > , \mathit{Attributelist}) = \\ \mathit{Add}(\mathit{Attribute}, \mathit{NewList}(\mathit{Relname} < \mathit{Attributes} > , \mathit{Attributelist})) \end{array}}
$$

OptName function: this function places the "Relname.." in front of attributes if attributes has no relname yet (the relation name is optional).

$[\mathrm{OPT1}]\ \mathit{OptName}(\mathit{Relname}, < \ >) = < \ >$

$$
[\mathrm{OPT2}]\ \frac{\begin{array}{l} \mathit{HasName}(\mathit{Attribute}) = \mathit{false}, \\ \qquad \mathit{Attribute}_1 = \mathit{Double}(\mathit{Relname}, \mathit{Attribute}) \end{array}}{\begin{array}{l} \mathit{OptName}(\mathit{Relname}, < \mathit{Attribute}, \mathit{Attributes} >) = \\ \mathit{Add}(\mathit{Attribute}_1, \mathit{OptName}(\mathit{Relname}, < \mathit{Attributes} >)) \end{array}}
$$

$$
[\mathrm{OPT3}]\ \frac{\mathit{HasName}(\mathit{Attribute}) = \mathit{true}}{\begin{array}{l} \mathit{OptName}(\mathit{Relname}, < \mathit{Attribute}, \mathit{Attributes} >) = \\ \mathit{Add}(\mathit{Attribute}, \mathit{OptName}(\mathit{Relname}, < \mathit{Attributes} >)) \end{array}}
$$

Single function: this function places "Relname." in front of the attribute.

$[\mathrm{S1}]\ \mathit{str\text{-}con}(\text{“"”}\ \mathit{Chars}_1\ \text{“"”})\ .\ \mathit{attribute}(\mathit{Chars}_2) =$
$\mathit{attribute}(\mathit{Chars}_1\ \text{“.”}\ \mathit{Chars}_2)$

$[\mathrm{S2}]\ \mathit{Single}(\mathit{Relname}, \mathit{Attribute}) = \mathit{Relname}\ .\ \mathit{Attribute}$

Double function: this function places "Relname.." in front of the attribute.

$[\mathrm{D1}]\ \mathit{str\text{-}con}(\text{“"”}\ \mathit{Chars}_1\ \text{“"”})\ ..\ \mathit{attribute}(\mathit{Chars}_2) =$
$\mathit{attribute}(\mathit{Chars}_1\ \text{“.”}\ \text{“.”}\ \mathit{Chars}_2)$

$[\mathrm{D2}]\ \mathit{Double}(\mathit{Relname}, \mathit{Attribute}) = \mathit{Relname}\ ..\ \mathit{Attribute}$

HasName function, this function checks if an attribute has a relationname in front of it (either "Relname.." or "Relname.").

$$
[\mathrm{HN1}]\ \frac{\mathit{HasSingleName}(\mathit{Attribute}) = \mathit{true}}{\mathit{HasName}(\mathit{Attribute}) = \mathit{true}}
$$

$$[\text{HN2}]\ \frac{\mathit{HasDoubleName}(\mathit{Attribute}) = \mathit{true}}{\mathit{HasName}(\mathit{Attribute}) = \mathit{true}}$$

$[\text{HN3}]\ \mathit{HasName}(\mathit{Attribute}) \ = \ \mathit{false}$ **otherwise**

HasSingleName function, this function checks if an attribute has "Relname." in front of it.

$$[\text{H1}]\ \frac{\begin{array}{c}\mathit{Attribute} = \mathit{attribute}(\mathit{Chars}_1 \text{ "." } \mathit{Chars}_2),\\ \mathit{HasSingleName}(\mathit{attribute}(\mathit{Chars}_1)) = \mathit{false},\\ \mathit{HasSingleName}(\mathit{attribute}(\mathit{Chars}_2)) = \mathit{false}\end{array}}{\mathit{HasSingleName}(\mathit{Attribute}) = \mathit{true}}$$

$[\text{H2}]\ \mathit{HasSingleName}(\mathit{Attribute}) \ = \ \mathit{false}$ **otherwise**

HasDoubleName function, this function checks if an attribute has "Relname.." in front of it.

$$[\text{H1}]\ \frac{\mathit{Attribute} = \mathit{attribute}(\mathit{Chars}_1 \text{ "." "." } \mathit{Chars}_2)}{\mathit{HasDoubleName}(\mathit{Attribute}) = \mathit{true}}$$

$[\text{H2}]\ \mathit{HasDoubleName}(\mathit{Attribute}) \ = \ \mathit{false}$ **otherwise**

ConcatName function: this function concatenates two relation names and places a @-character between them.

$[\text{CN0}]\ \mathit{str\text{-}con}(\text{"""}\ \mathit{Chars}_1\ \text{"""})\ @\ \mathit{str\text{-}con}(\text{"""}\ \mathit{Chars}_2\ \text{"""}) \ =$
$\mathit{str\text{-}con}(\text{"""}\ \mathit{Chars}_1\ \text{"@"}\ \mathit{Chars}_2\ \text{"""})$

$[\text{CN1}]\ \mathit{ConcatName}(\mathit{Relname}_1, \mathit{Relname}_2) \ = \ \mathit{Relname}_1\ @\ \mathit{Relname}_2$

CheckElemOf function: this function looks if the attribute from the selection predicate is an attribute from the extended attributelist or from the attributelist in which the optional relationnames are removed.

$$[\text{CEO1}]\ \frac{\mathit{ElemOf}(\mathit{Attribute}, \mathit{RemDouble}(\mathit{Attributelist})) = \mathit{true}}{\mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{true}}$$

$$[\text{CEO2}]\ \frac{\mathit{ElemOf}(\mathit{Attribute}, \mathit{RemDoubleName}(\mathit{Attributelist})) = \mathit{true}}{\mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{true}}$$

$[\text{CEO3}]\ \mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) \ = \ \mathit{false}$ **otherwise**

CheckList function: this function looks which attributes from the first parameter aren't attributes from the extended attributelist (2nd parameter) or from the second parameter in which the optional relationnames are removed

$[\text{CL1}]\ \mathit{CheckList}(< \ >, \mathit{Attributelist}) \ = \ < \ >$

$$[\text{CL2}]\ \frac{\mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{true}}{\begin{array}{l}\mathit{CheckList}(< \mathit{Attribute}, \mathit{Attributes} >, \mathit{Attributelist}) = \\ \mathit{CheckList}(< \mathit{Attributes} >, \mathit{Attributelist})\end{array}}$$

$$[\text{CL3}]\ \frac{\mathit{CheckElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{false}}{\begin{array}{l}\mathit{CheckList}(< \mathit{Attribute}, \mathit{Attributes} >, \mathit{Attributelist}) = \\ \mathit{Add}(\mathit{Attribute}, \mathit{CheckList}(< \mathit{Attributes} >, \mathit{Attributelist}))\end{array}}$$

SubList function: this function selects attributes that are element of the project-list (with or without (optional) relname).

$$[\text{SL1}]\ \mathit{SubList}(< >, \mathit{Attributelist}) = < >$$

$$[\text{SL2}]\ \frac{\begin{array}{l}\mathit{HasName}(\mathit{Attribute}) = \mathit{false}, \\ \mathit{Attributelist}_1 = \mathit{Sub}(\mathit{Attribute}, \mathit{Attributelist})\end{array}}{\begin{array}{l}\mathit{SubList}(< \mathit{Attribute}, \mathit{Attributes} >, \mathit{Attributelist}) = \\ \mathit{Add}(\mathit{Attributelist}_1, \mathit{SubList}(< \mathit{Attributes} >, \mathit{Attributelist}))\end{array}}$$

$$[\text{SL3}]\ \frac{\begin{array}{l}\mathit{HasName}(\mathit{Attribute}) = \mathit{true}, \\ \mathit{ElemOf}(\mathit{Attribute}, \mathit{Attributelist}) = \mathit{true}\end{array}}{\begin{array}{l}\mathit{SubList}(< \mathit{Attribute}, \mathit{Attributes} >, \mathit{Attributelist}) = \\ \mathit{Add}(\mathit{Attribute}, \mathit{SubList}(< \mathit{Attributes} >, \mathit{Attributelist}))\end{array}}$$

$$[\text{SL4}]\ \frac{\begin{array}{l}\mathit{HasName}(\mathit{Attribute}) = \mathit{true}, \\ \mathit{Attribute}_1 = \mathit{SingleToDouble}(\mathit{Attribute}), \\ \mathit{ElemOf}(\mathit{Attribute}_1, \mathit{Attributelist}) = \mathit{true}, \\ \mathit{Attributelist}_1 = \mathit{Sub}(\mathit{RName}(\mathit{Attribute}), \mathit{Attributelist})\end{array}}{\begin{array}{l}\mathit{SubList}(< \mathit{Attribute}, \mathit{Attributes} >, \mathit{Attributelist}) = \\ \mathit{Add}(\mathit{Attributelist}_1, \mathit{SubList}(< \mathit{Attributes} >, \mathit{Attributelist}))\end{array}}$$

Sub function.

$$[\text{S1}]\ \mathit{Sub}(\mathit{Attribute}, < >) = < >$$

$$[\text{S2}]\ \frac{\mathit{Attribute} = \mathit{RDoubleName}(\mathit{Attribute}_1)}{\begin{array}{l}\mathit{Sub}(\mathit{Attribute}, < \mathit{Attribute}_1, \mathit{Attributes} >) = \\ \mathit{Add}(\mathit{Attribute}_1, \mathit{Sub}(\mathit{Attribute}, < \mathit{Attributes} >))\end{array}}$$

$$[\text{S3}]\ \frac{\mathit{Attribute} \neq \mathit{RDoubleName}(\mathit{Attribute}_1)}{\begin{array}{l}\mathit{Sub}(\mathit{Attribute}, < \mathit{Attribute}_1, \mathit{Attributes} >) = \\ \mathit{Sub}(\mathit{Attribute}, < \mathit{Attributes} >)\end{array}}$$

RName function: remove all relname information of an attribute.

$$[\text{RN1}]\ \begin{array}{l}\mathit{RName}(\mathit{attribute}(\mathit{Chars}_1\ \text{``.''}\ \mathit{Chars}_2)) = \\ \mathit{RName}(\mathit{attribute}(\mathit{Chars}_2))\end{array}$$

[RN2] $RName(Attribute) = Attribute$ **otherwise**

RemName function: remove all relname information of all attributes in the attributelist.

[RNS1] $RemName(< >) = < >$

[RNS2] $RemName(< Attribute, Attributes >) =$
$Add(RName(Attribute), RemName(< Attributes >))$

RDoubleName function: if the attribute is of the form "Relname..attribute" change it to "attribute.

[RDN1] $RDoubleName(attribute(Chars_1$ "." "." $Chars_2)) =$
$attribute(Chars_2)$

[RDN2] $RDoubleName(Attribute) = Attribute$ **otherwise**

RemDoubleName function: if there are attributes of the form "Relname..attribute" in the attributelist change them to "attribute".

[RDNS1] $RemDoubleName(< >) = < >$

[RDNS2] $RemDoubleName(< Attribute, Attributes >) =$
$Add(RDoubleName(Attribute), RemDoubleName(< Attributes >))$

SingleToDouble function: this function changes an attribute of the form "Relname.attribute" to "Relname..attribute.

[STD1] $SingleToDouble(attribute(Chars_1$ "." $Chars_2)) =$
$attribute(Chars_1$ "." "." $Chars_2)$

RDouble function: if the attribute is of the form "Relname..attribute" change it to "Relname.attribute.

[RD1] $RDouble(attribute(Chars_1$ "." "." $Chars_2)) =$
$attribute(Chars_1$ "." $Chars_2)$

[RD2] $RDouble(Attribute) = Attribute$ **otherwise**

RemDouble function: if there are attributes of the form "Relname..attribute" in the attributelist change them to "Relname.attribute".

[RDS1] $RemDouble(< >) = < >$

[RDS2] $RemDouble(< Attribute, Attributes >) =$
$Add(RDouble(Attribute), RemDouble(< Attributes >))$

Normalize function: this function normalizes the query. Only necessary relationnames are put in front of the attribute. This is done to simplify (further) optimizing.

$$[\text{NORM1}]\frac{Attributelist = RemDoubleName(NormList(Rel))}{Normalize(Rel) = CheckName(Rel,\ Attributelist)}$$

$$\begin{aligned}
&[\text{NL1}]\ NormList(Relname\ Attributelist) &&= OptName(Relname,\ Attributelist)\\
&[\text{NL2}]\ NormList(Union(Rel_1,\ Rel_2)) &&= NormList(Rel_1)\\
&[\text{NL3}]\ NormList(Intersection(Rel_1,\ Rel_2)) &&= NormList(Rel_1)\\
&[\text{NL4}]\ NormList(Minus(Rel_1,\ Rel_2)) &&= NormList(Rel_1)
\end{aligned}$$

$$[\text{NL5}]\frac{\begin{array}{l} Attributelist_1 = NormList(Rel_1),\\ Attributelist_2 = NormList(Rel_2),\\ Attributelist_3 = NewName(Attributelist_1,\ Attributelist_2),\\ Attributelist_4 = NewName(Attributelist_2,\ Attributelist_1)\end{array}}{NormList(Product(Rel_1,\ Rel_2)) = Add(Attributelist_3,\ Attributelist_4)}$$

$$\begin{aligned}
&[\text{NL6}]\ NormList(Select(Selpred,\ Rel)) &&= NormList(Rel)\\
&[\text{NL7}]\ NormList(Project(Attributelist,\ Rel)) &&= NormList(Rel)
\end{aligned}$$

$$[\text{NL8}]\frac{\begin{array}{l} Attributelist_1 = NormList(Rel_1),\\ Attributelist_2 = NormList(Rel_2),\\ Attributelist_3 = NewName(Attributelist_1,\ Attributelist_2),\\ Attributelist_4 = NewName(Attributelist_2,\ Attributelist_1)\end{array}}{NormList(Join(Joinpred,\ Rel_1,\ Rel_2)) = Add(Attributelist_3,\ Attributelist_4)}$$

NewName function

$$[\text{NN1}]\ NewName(<\ >,\ Attributelist) = <\ >$$

$$[\text{NN2}]\frac{ElemOf(RDoubleName(Attribute),\ RemDoubleName(Attributelist)) = false}{\begin{array}{l} NewName(< Attribute,\ Attributes >,\ Attributelist) =\\ Add(Attribute,\ NewName(< Attributes >,\ Attributelist))\end{array}}$$

$$[\text{NN3}]\frac{\begin{array}{r} ElemOf(RDoubleName(Attribute),\ RemDoubleName(Attributelist)) = true,\\ Attribute_1 = RDouble(Attribute)\end{array}}{\begin{array}{l} NewName(< Attribute,\ Attributes >,\ Attributelist) =\\ Add(Attribute_1,\ NewName(< Attributes >,\ Attributelist))\end{array}}$$

CheckName function

$$[\text{CN1}]\frac{\begin{array}{l} Attributelist_2 = OptName(Relname,\ Attributelist),\\ Attributelist_3 = CheckAttributes(Attributelist_2,\ Attributelist_1)\end{array}}{\begin{array}{l} CheckName(Relname\ Attributelist,\ Attributelist_1) =\\ Relname\ Attributelist_3\end{array}}$$

[CN2] $CheckName(Union(Rel_1, Rel_2), Attributelist) =$
$Union(CheckName(Rel_1, Attributelist), CheckName(Rel_2, Attributelist))$

[CN3] $CheckName(Intersection(Rel_1, Rel_2), Attributelist) =$
$Intersection(CheckName(Rel_1, Attributelist), CheckName(Rel_2, Attributelist))$

[CN4] $CheckName(Minus(Rel_1, Rel_2), Attributelist) =$
$Minus(CheckName(Rel_1, Attributelist), CheckName(Rel_2, Attributelist))$

[CN5] $CheckName(Product(Rel_1, Rel_2), Attributelist) =$
$Product(CheckName(Rel_1, Attributelist), CheckName(Rel_2, Attributelist))$

$$[CN6]\frac{Selpred_1 = CheckPred(Selpred, Attributelist)}{\begin{array}{l}CheckName(Select(Selpred, Rel), Attributelist) = \\ Select(Selpred_1, CheckName(Rel, Attributelist))\end{array}}$$

$$[CN7]\frac{Projectlist_1 = CheckAttributes(Projectlist, Attributelist)}{\begin{array}{l}CheckName(Project(Projectlist, Rel), Attributelist) = \\ Project(Projectlist_1, CheckName(Rel, Attributelist))\end{array}}$$

$$[CN8]\frac{\begin{array}{l}Attribute_{11} = CheckAttribute(Attribute_1, Attributelist), \\ Attribute_{22} = CheckAttribute(Attribute_2, Attributelist)\end{array}}{\begin{array}{l}CheckName(Join(Attribute_1 = Attribute_2, Rel_1, Rel_2), Attributelist) = \\ Join(Attribute_{11} = Attribute_{22}, \\ \quad CheckName(Rel_1, Attributelist), CheckName(Rel_1, Attributelist))\end{array}}$$

CheckPred function

$$[CP1]\frac{Attribute_1 = CheckAttribute(Attribute, Attributelist)}{\begin{array}{l}CheckPred(Attribute\ Op\ Str, Attributelist) = \\ Attribute_1\ Op\ Str\end{array}}$$

$$[CP2]\frac{Attribute_1 = CheckAttribute(Attribute, Attributelist)}{\begin{array}{l}CheckPred(Attribute\ Op\ Int, Attributelist) = \\ Attribute_1\ Op\ Int\end{array}}$$

$$[CP3]\frac{\begin{array}{l}Attribute_{11} = CheckAttribute(Attribute_1, Attributelist), \\ Attribute_{22} = CheckAttribute(Attribute_2, Attributelist)\end{array}}{\begin{array}{l}CheckPred(Attribute_1\ Op\ Attribute_2, Attributelist) = \\ Attribute_{11}\ Op\ Attribute_{22}\end{array}}$$

[CP4] $CheckPred(Selpred_1\ and\ Selpred_2, Attributelist) =$
$CheckPred(Selpred_1, Attributelist)\ and\ CheckPred(Selpred_2, Attributelist)$

[CP5] $CheckPred(Selpred_1 \text{ or } Selpred_2, Attributelist) =$
$CheckPred(Selpred_1, Attributelist) \text{ or } CheckPred(Selpred_2, Attributelist)$

CheckAttribute function

[CA1] $CheckAttribute(Attribute, < Attribute, Attributes >) =$
$Attribute$

$$[CA2]\ \frac{Attribute_1 = RDoubleName(Attribute)}{CheckAttribute(Attribute, < Attribute_1, Attributes >) = Attribute_1}$$

$$[CA3]\ \frac{Attribute_1 = RDouble(Attribute)}{CheckAttribute(Attribute, < Attribute_1, Attributes >) = Attribute_1}$$

$$[CA4]\ \frac{Attribute_1 = RName(Attribute)}{CheckAttribute(Attribute, < Attribute_1, Attributes >) = Attribute_1}$$

[CA4] $CheckAttribute(Attribute, < Attribute_1, Attributes >) =$
$CheckAttribute(Attribute, < Attributes >)$ **otherwise**

CheckAttributes function

[CAS1] $CheckAttributes(< >, Attributelist) = < >$

[CAS2] $CheckAttributes(< Attribute, Attributes >, Attributelist) =$
$Add(CheckAttribute(Attribute, Attributelist),$
$CheckAttributes(< Attributes >, Attributelist))$

A.3 Eqtransforms

imports Operations[(A.5)]
exports
context-free syntax
"*Transform*" "(" REL ")" → REL
"*Detrans*" "(" REL ")" → REL
hiddens
context-free syntax
"*Trans*" "(" REL ")" → REL
"*Uses*" "(" SELPRED "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"NULL" → SELPRED

"*Conjunctive*" "(" SELPRED ")" → SELPRED
"*Distribute*" "(" SELPRED "," REL ")" → SELPRED
"*Distribute1*" "(" SELPRED "," REL ")" → SELPRED
"*Collect*" "(" SELPRED "," SELPRED "," SELPRED ")" → SELPRED
"*Collect1*" "(" SELPRED "," SELPRED "," SELPRED ")" → SELPRED
"*ElemOfSel*" "(" SELPRED "," SELPRED ")" → BOOL
"*GetAtList*" "(" REL ")" → ATTRIBUTELIST

equations

[TF1] $Transform(Relname\ Attributelist) = Trans(Relname\ Attributelist)$

[TF2] $Transform(Union(Rel_1, Rel_2)) = Trans(Union(Transform(Rel_1), Transform(Rel_2)))$

[TF3] $Transform(Intersection(Rel_1, Rel_2)) = Trans(Intersection(Transform(Rel_1), Transform(Rel_2)))$

[TF4] $Transform(Minus(Rel_1, Rel_2)) = Trans(Minus(Transform(Rel_1), Transform(Rel_2)))$

[TF5] $Transform(Product(Rel_1, Rel_2)) = Trans(Product(Transform(Rel_1), Transform(Rel_2)))$

[TF6] $Transform(Select(Selpred, Rel)) = Trans(Select(Selpred, Transform(Rel)))$

[TF7] $Transform(Project(Projectlist, Rel)) = Trans(Project(Projectlist, Transform(Rel)))$

[TF8] $Transform(Join(Joinpred, Rel_1, Rel_2)) = Trans(Join(Joinpred, Transform(Rel_1), Transform(Rel_2$

Project over Select:

$$\text{[TRANS1a]}\ \frac{Projectlist_1 = Uses(Selpred, Projectlist),\quad Projectlist = Projectlist_1}{Trans(Project(Projectlist, Trans(Select(Selpred, Rel)))) = Trans(Select(Selpred, Trans(Project(Projectlist_1, Rel))))}$$

$$\text{[TRANS1b]}\ \frac{Projectlist_1 = Uses(Selpred, Projectlist),\quad Projectlist \neq Projectlist_1,\quad Projectlist_1 \neq GetAtList(Detrans(Rel))}{Trans(Project(Projectlist, Trans(Select(Selpred, Rel)))) = Trans(Project(Projectlist, Trans(Select(Selpred, Trans(Project(Projectlist_1, Rel))))))}$$

Select (cascade):

[TRANS2] $Trans(Select(Selpred_2, Trans(Select(Selpred_1, Rel)))) = Trans(Select(Selpred_1\ and\ Selpred_2, Rel))$

Project (cascade):

[TRANS3] $Trans(Project(Attributelist_1, Trans(Project(Attributelist_2, Rel)))) = Trans(Project(Attributelist_1, Rel))$

Select over Union:

[TRANS4] $Trans(Select(Selpred, Trans(Union(Rel_1, Rel_2)))) = Trans(Union(Trans(Select(Selpred, Rel_1)), Trans(Select(Selpred, Rel_2))))$

Select over Set Difference (Minus):

$$\text{[TRANS5]}\ \mathit{Trans}(\mathit{Select}(\mathit{Selpred},\ \mathit{Trans}(\mathit{Minus}(\mathit{Rel}_1,\ \mathit{Rel}_2)))) = \mathit{Trans}(\mathit{Minus}(\mathit{Trans}(\mathit{Select}(\mathit{Selpred},\ \mathit{Rel}_1)),\ \mathit{Trans}(\mathit{Select}(\mathit{Selpred},\ \mathit{Rel}_2))))$$

Select over Intersection:

$$\text{[TRANS6]}\ \mathit{Trans}(\mathit{Select}(\mathit{Selpred},\ \mathit{Trans}(\mathit{Intersection}(\mathit{Rel}_1,\ \mathit{Rel}_2)))) = \mathit{Trans}(\mathit{Intersection}(\mathit{Trans}(\mathit{Select}(\mathit{Selpred},\ \mathit{Rel}_1)),\ \mathit{Trans}(\mathit{Select}(\mathit{Selpred},\ \mathit{Rel}_2))))$$

Select over Product:

$$\text{[TRANS7]}\ \frac{\begin{array}{l}\mathit{Selpred}_1 = \mathit{Conjunctive}(\mathit{Selpred}),\\ \mathit{Selpred}_2 = \mathit{Distribute}(\mathit{Selpred}_1,\ \mathit{Rel}_1),\\ \mathit{Selpred}_3 = \mathit{Distribute}(\mathit{Selpred}_1,\ \mathit{Rel}_2),\\ \mathit{Selpred}_4 = \mathit{Collect}(\mathit{Selpred}_1,\ \mathit{Selpred}_2,\ \mathit{Selpred}_3),\\ \mathit{Selpred}_1 \neq \mathit{Selpred}_4\end{array}}{\begin{array}{l}\mathit{Trans}(\mathit{Select}(\mathit{Selpred},\ \mathit{Trans}(\mathit{Product}(\mathit{Rel}_1,\ \mathit{Rel}_2)))) =\\ \mathit{Trans}(\mathit{Select}(\mathit{Selpred}_4,\ \mathit{Trans}(\mathit{Product}(\mathit{Trans}(\mathit{Select}(\mathit{Selpred}_2,\ \mathit{Rel}_1)),\\ \qquad \mathit{Trans}(\mathit{Select}(\mathit{Selpred}_3,\ \mathit{Rel}_2))))))\end{array}}$$

Select over Join:

$$\text{[TRANS8]}\ \frac{\begin{array}{l}\mathit{Selpred}_1 = \mathit{Conjunctive}(\mathit{Selpred}),\\ \mathit{Selpred}_2 = \mathit{Distribute}(\mathit{Selpred}_1,\ \mathit{Rel}_1),\\ \mathit{Selpred}_3 = \mathit{Distribute}(\mathit{Selpred}_1,\ \mathit{Rel}_2),\\ \mathit{Selpred}_4 = \mathit{Collect}(\mathit{Selpred}_1,\ \mathit{Selpred}_2,\ \mathit{Selpred}_3),\\ \mathit{Selpred}_1 \neq \mathit{Selpred}_4\end{array}}{\begin{array}{l}\mathit{Trans}(\mathit{Select}(\mathit{Selpred},\ \mathit{Trans}(\mathit{Join}(\mathit{Joinpred},\ \mathit{Rel}_1,\ \mathit{Rel}_2)))) =\\ \mathit{Trans}(\mathit{Select}(\mathit{Selpred}_4,\\ \qquad \mathit{Trans}(\mathit{Join}(\mathit{Joinpred},\ \mathit{Trans}(\mathit{Select}(\mathit{Selpred}_2,\ \mathit{Rel}_1)),\\ \qquad\qquad \mathit{Trans}(\mathit{Select}(\mathit{Selpred}_3,\ \mathit{Rel}_2))))))\end{array}}$$

Project over Product:

$$\text{[TRANS9]}\ \frac{\begin{array}{l}\mathit{Projectlist}_1 = \mathit{MakeSub}(\mathit{Projectlist},\ \mathit{Detrans}(\mathit{Rel}_1)),\\ \mathit{Projectlist}_2 = \mathit{MakeSub}(\mathit{Projectlist},\ \mathit{Detrans}(\mathit{Rel}_2))\end{array}}{\begin{array}{l}\mathit{Trans}(\mathit{Project}(\mathit{Projectlist},\ \mathit{Trans}(\mathit{Product}(\mathit{Rel}_1,\ \mathit{Rel}_2)))) =\\ \mathit{Trans}(\mathit{Product}(\mathit{Trans}(\mathit{Project}(\mathit{Projectlist}_1,\ \mathit{Rel}_1)),\\ \qquad \mathit{Trans}(\mathit{Project}(\mathit{Projectlist}_2,\ \mathit{Rel}_2))))\end{array}}$$

Project over Join:

$$\text{[TRANS10]}\ \frac{\begin{array}{r}\mathit{Projectlist}_1 = \mathit{MakeSub}(\mathit{Projectlist},\ \mathit{Detrans}(\mathit{Rel}_1)),\\ \mathit{Projectlist}_2 = \mathit{MakeSub}(\mathit{Projectlist},\ \mathit{Detrans}(\mathit{Rel}_2)),\\ \mathit{ElemOf}(\mathit{Attribute}_1,\ \mathit{Attributelist}) = \mathit{true},\\ \mathit{ElemOf}(\mathit{Attribute}_2,\ \mathit{Attributelist}) = \mathit{true}\end{array}}{\begin{array}{l}\mathit{Trans}(\mathit{Project}(\mathit{Projectlist},\ \mathit{Trans}(\mathit{Join}(\mathit{Attribute}_1 = \mathit{Attribute}_2,\ \mathit{Rel}_1,\ \mathit{Rel}_2)))) =\\ \mathit{Trans}(\mathit{Join}(\mathit{Attribute}_1 = \mathit{Attribute}_2,\ \mathit{Trans}(\mathit{Project}(\mathit{Projectlist}_1,\ \mathit{Rel}_1)),\\ \qquad \mathit{Trans}(\mathit{Project}(\mathit{Projectlist}_2,\ \mathit{Rel}_2))))\end{array}}$$

Project over Union:

[TRANS11] *Trans(Project(Projectlist, Union(*Rel_1*,* Rel_2*)))* =
Trans(Union(Trans(Project(Projectlist, Rel_1*)),*
Trans(Project(Projectlist, Rel_2*))))*

Union over Select:

[TRANS12] *Trans(Union(Trans(Select(* $Selpred_1$*, Rel)), Trans(Select(* $Selpred_2$*, Rel))))* =
Trans(Select($Selpred_1$ *or* $Selpred_2$*, Rel))*

Intersection over Select:

[TRANS13] *Trans(Intersection(Trans(Select(* $Selpred_1$*, Rel)), Trans(Select(* $Selpred_2$*, Rel))))* =
Trans(Select($Selpred_1$ *and* $Selpred_2$*, Rel))*

Minus over Select:

[TRANS14] *Trans(Minus(Trans(Select(* $Selpred_1$*, Rel)), Trans(Select(* $Selpred_2$*, Rel))))* =
Trans(Select($Selpred_1$ *and Neg(* $Selpred_2$*), Rel))*

[TRANS15] *Trans(Union(Rel, Rel))* = *Trans(Rel)*

[TRANS16] *Trans(Intersection(Rel, Rel))* = *Trans(Rel)*

[DTF1] *Detrans(Trans(Relname Attributelist))* = *Relname Attributelist*
[DTF2] *Detrans(Trans(Union(*Rel_1*,* Rel_2*)))* = *Union(Detrans(*Rel_1*), Detrans(*Rel_2*))*
[DTF3] *Detrans(Trans(Intersection(*Rel_1*,* Rel_2*)))* = *Intersection(Detrans(*Rel_1*), Detrans(*Rel_2*))*
[DTF4] *Detrans(Trans(Minus(*Rel_1*,* Rel_2*)))* = *Minus(Detrans(*Rel_1*), Detrans(*Rel_2*))*
[DTF5] *Detrans(Trans(Product(*Rel_1*,* Rel_2*)))* = *Product(Detrans(*Rel_1*), Detrans(*Rel_2*))*
[DTF6] *Detrans(Trans(Select(Selpred, Rel)))* = *Select(Selpred, Detrans(Rel))*
[DTF7] *Detrans(Trans(Project(Attributelist, Rel)))* = *Project(Attributelist, Detrans(Rel))*
[DTF8] *Detrans(Trans(Join(Joinpred,* Rel_1*,* Rel_2*)))* = *Join(Joinpred, Detrans(*Rel_1*), Detrans(*Rel_2*))*
[DTF9] *Detrans(Rel)* = *Rel* **otherwise**

Uses function. This function checks if all attributes used in the selection-predicate are attributes from the attributelist (the second parameter). If it finds an attribute that is not an element of the attributelist, it adds this attribute to the list

[U1a]
ElemOf(Attribute, Attributelist) = false

Uses(Attribute Op Int, Attributelist) =
Add(Attribute, Attributelist)

[U1b]
ElemOf(Attribute, Attributelist) = true

Uses(Attribute Op Int, Attributelist) =
Attributelist

$$[U2a]\ \frac{ElemOf(Attribute,\ Attributelist) = false}{Uses(Attribute\ Op\ Str,\ Attributelist) = Add(Attribute,\ Attributelist)}$$

$$[U2b]\ \frac{ElemOf(Attribute,\ Attributelist) = true}{Uses(Attribute\ Op\ Str,\ Attributelist) = Attributelist}$$

$$[U3a]\ \frac{ElemOf(Attribute_1,\ Attributelist) = true,\quad ElemOf(Attribute_2,\ Attributelist) = true}{Uses(Attribute_1\ Op\ Attribute_2,\ Attributelist) = Attributelist}$$

$$[U3b]\ \frac{ElemOf(Attribute_1,\ Attributelist) = true,\quad ElemOf(Attribute_2,\ Attributelist) = false}{Uses(Attribute_1\ Op\ Attribute_2,\ Attributelist) = Add(Attribute_2,\ Attributelist)}$$

$$[U3c]\ \frac{ElemOf(Attribute_1,\ Attributelist) = false,\quad ElemOf(Attribute_2,\ Attributelist) = true}{Uses(Attribute_1\ Op\ Attribute_2,\ Attributelist) = Add(Attribute_1,\ Attributelist)}$$

$$[U3d]\ \frac{ElemOf(Attribute_1,\ Attributelist) = false,\quad ElemOf(Attribute_2,\ Attributelist) = false}{Uses(Attribute_1\ Op\ Attribute_2,\ Attributelist) = Add(Attribute_1,\ Add(Attribute_2,\ Attributelist))}$$

[U4] $Uses(Selpred_1\ and\ Selpred_2,\ Attributelist) = Uses(Selpred_1,\ Uses(Selpred_2,\ Attributelist))$

[U5] $Uses(Selpred_1\ or\ Selpred_2,\ Attributelist) = Uses(Selpred_1,\ Uses(Selpred_2,\ Attributelist))$

Conjunctive. This function rewrites a selection predicate into conjunctive normalform

[C1] $Conjunctive(Selpred_1\ or\ Selpred_2\ and\ Selpred_3) = Conjunctive((Selpred_1\ or\ Selpred_2)\ and\ (Selpred_1\ or\ Selpred_3))$

[C2] $Conjunctive(Selpred_1\ and\ Selpred_2\ or\ Selpred_3) = Conjunctive((Selpred_1\ or\ Selpred_3)\ and\ (Selpred_2\ or\ Selpred_3))$

$$[C3]\ \frac{Selpred_3 = Conjunctive(Selpred_1),\quad Selpred_3 \neq Selpred_1}{Conjunctive(Selpred_1 \ or \ Selpred_2) = Conjunctive(Selpred_3 \ or \ Selpred_2)}$$

$$[C4]\ \frac{Selpred_3 = Conjunctive(Selpred_2),\quad Selpred_3 \neq Selpred_2}{Conjunctive(Selpred_1 \ or \ Selpred_2) = Conjunctive(Selpred_1 \ or \ Selpred_3)}$$

$$[C5]\ \frac{Selpred_3 = Conjunctive(Selpred_1),\quad Selpred_3 \neq Selpred_1}{Conjunctive(Selpred_1 \ and \ Selpred_2) = Conjunctive(Selpred_3 \ and \ Selpred_2)}$$

$$[C6]\ \frac{Selpred_3 = Conjunctive(Selpred_2),\quad Selpred_3 \neq Selpred_2}{Conjunctive(Selpred_1 \ and \ Selpred_2) = Conjunctive(Selpred_1 \ and \ Selpred_3)}$$

$$[C7]\ Conjunctive(Selpred) = Selpred \qquad \textbf{otherwise}$$

Distribute-function. This function takes as arguments a selection-predicate and a relation. This function selects those subpredicates that only have attributes that belong to the relation.

$$[Dis1]\ \frac{Rel_1 = Detrans(Rel)}{Distribute(Selpred_1 \ and \ Selpred_2, Rel) = Distribute(Selpred_1, Rel_1) \ and \ Distribute(Selpred_2, Rel_1)}$$

$$[Dis2]\ \frac{Rel_1 = Detrans(Rel)}{Distribute(Selpred, Rel) = Distribute1(Selpred, Rel_1)} \qquad \textbf{otherwise}$$

$$[D1]\ \frac{Attributelist = GetAtList(Rel),\quad ElemOf(Attribute, Attributelist) = true}{Distribute1(Attribute \ Op \ Int, Rel) = Attribute \ Op \ Int}$$

$$[D2]\ \frac{Attributelist = GetAtList(Rel),\quad ElemOf(Attribute, Attributelist) = false}{Distribute1(Attribute \ Op \ Int, Rel) = NULL}$$

$$[D3]\ \frac{Attributelist = GetAtList(Rel),\quad ElemOf(Attribute, Attributelist) = true}{Distribute1(Attribute \ Op \ Str, Rel) = Attribute \ Op \ Str}$$

$$[D4]\ \frac{Attributelist = GetAtList(Rel),\quad ElemOf(Attribute, Attributelist) = false}{Distribute1(Attribute\ Op\ Str, Rel) = NULL}$$

$$[D5]\ \frac{Attributelist = GetAtList(Rel),\quad ElemOf(Attribute_1, Attributelist) = true,\quad ElemOf(Attribute_2, Attributelist) = true}{Distribute1(Attribute_1\ Op\ Attribute_2, Rel) = Attribute_1\ Op\ Attribute_2}$$

[D6] $Distribute1(Attribute_1\ Op\ Attribute_2, Rel) = NULL$ **otherwise**

$$[D7]\ \frac{Distribute1(Selpred_1, Rel) \neq NULL,\quad Distribute1(Selpred_2, Rel) \neq NULL}{Distribute1(Selpred_1\ or\ Selpred_2, Rel) = Selpred_1\ or\ Selpred_2}$$

[D8] $Distribute1(Selpred_1\ or\ Selpred_2, Rel) = NULL$ **otherwise**

[N1] $NULL\ and\ Selpred = Selpred$

[N2] $Selpred\ and\ NULL = Selpred$

[N3] $Trans(Select(NULL, Rel)) = Rel$

Collect-function. This function selects those subpredicates that are a part of the first predicate and not a member of the second or third predicate

[Col1] $Collect(Selpred_1\ and\ Selpred_2, Selpred_3, Selpred_4) = Collect(Selpred_1, Selpred_3, Selpred_4)\ and\ Collect(Selpred_2, Selpred_3, Selpred_4)$

[Col2] $Collect(Selpred_1, Selpred_2, Selpred_3) = Collect1(Selpred_1, Selpred_2, Selpred_3)$ **otherwise**

$$[C1]\ \frac{ElemOfSel(Selpred_1, Selpred_2) = false,\quad ElemOfSel(Selpred_1, Selpred_3) = false}{Collect1(Selpred_1, Selpred_2, Selpred_3) = Selpred_1}$$

[C2] $Collect1(Selpred_1, Selpred_2, Selpred_3) = NULL$ **otherwise**

ElemOfSel-function. This function checks if the first predicate is a subpredicate of the second predicate.

[E1] $ElemOfSel(Selpred, Selpred) = true$

[E2] $ElemOfSel(Selpred,\ Selpred_1\ and\ Selpred_2)\ =$
$ElemOfSel(Selpred,\ Selpred_1)\ |\ ElemOfSel(Selpred,\ Selpred_2)$

[E3] $ElemOfSel(Selpred_1,\ Selpred_2)\ =\ false$ **otherwise**

GetAtList function. This functions finds the attributelist of the relation

[G1] $GetAtList(Relname\ Attributelist) = Attributelist$
[G2] $GetAtList(Union(Rel_1, Rel_2)) = GetAtList(Rel_1)$
[G3] $GetAtList(Intersection(Rel_1, Rel_2)) = GetAtList(Rel_1)$
[G4] $GetAtList(Minus(Rel_1, Rel_2)) = GetAtList(Rel_1)$
[G5] $GetAtList(Product(Rel_1, Rel_2)) = Add(GetAtList(Rel_1), GetAtList(Rel_2))$
[G6] $GetAtList(Select(Selpred, Rel)) = GetAtList(Rel)$
[G7] $GetAtList(Project(Projectlist, Rel)) = Projectlist$
[G8] $GetAtList(Join(Joinpred, Rel_1, Rel_2)) = Add(GetAtList(Rel_1), GetAtList(Rel_2))$

A.4 Factorisation

imports Operations$^{(A.5)}$
exports
sorts RESULT
context-free syntax
"(" SELPRED "," SELPRED ")" → RESULT
"*Fac*" "(" SELPRED "," ATTRIBUTELIST ")" → RESULT
"*Unary*" "(" SELPRED ")" → BOOL
"*ElemOf*" "(" SELPRED "," ATTRIBUTELIST ")" → BOOL
"NULL" → SELPRED
variables
Faclist [*0-9*]*→ ATTRIBUTELIST
equations

[F1a] $Selpred\ and\ NULL\ =\ Selpred$
[F1b] $NULL\ and\ Selpred\ =\ Selpred$

[F1c] $Selpred\ or\ NULL\ =\ NULL$
[F1d] $NULL\ or\ Selpred\ =\ NULL$

$$\text{[F2a]}\ \frac{Unary(Selpred) = true,\quad ElemOf(Selpred, Faclist) = true}{Fac(Selpred, Faclist) = (Selpred, NULL)}$$

$$[F2b]\ \frac{Unary(Selpred) = true,\quad ElemOf(Selpred, Faclist) = false}{Fac(Selpred, Faclist) = (NULL, Selpred)}$$

$$[F4]\ \frac{Fac(Selpred_1, Faclist) = (Selpred_{11}, Selpred_{12}),\quad Fac(Selpred_2, Faclist) = (Selpred_{21}, Selpred_{22})}{Fac(Selpred_1\ and\ Selpred_2, Faclist) = (Selpred_{11}\ and\ Selpred_{21}, Selpred_{12}\ and\ Selpred_{22})}$$

$$[F5]\ \frac{Fac(Selpred_1, Faclist) = (Selpred_{11}, Selpred_{12}),\quad Fac(Selpred_2, Faclist) = (Selpred_{21}, Selpred_{22})}{Fac(Selpred_1\ or\ Selpred_2, Faclist) = (Selpred_{11}\ or\ Selpred_{21}, (Selpred_{11}\ or\ Selpred_{22})\ and\ (Selpred_{12}\ or\ Selpred_{21})\ and\ (Selpred_{12}\ or\ Selpred_{22}))}$$

Function Unary. Check if the predicate (parameter) is unary.

[U1] $Unary(Attribute\ Op\ Int) = true$

[U2] $Unary(Attribute\ Op\ Str) = true$

[U3] $Unary(Attribute_1\ Op\ Attribute_2) = true$

[U4] $Unary(Selpred) = false$ **otherwise**

Function ElemOf. Check if the first parameter (SELPRED) is an element of the second parameter (ATTRIBUTELIST). This function works only for unary predicates.

[E1] $ElemOf(Attribute\ Op\ Int, Faclist) = ElemOf(Attribute, Faclist)$

[E2] $ElemOf(Attribute\ Op\ Str, Faclist) = ElemOf(Attribute, Faclist)$

[E3] $ElemOf(Attribute_1\ Op\ Attribute_2, Faclist) = ElemOf(Attribute_1, Faclist)\ \&\ ElemOf(Attribute_2, Faclist)$

[E4] $ElemOf(Selpred, Faclist) = false$ **otherwise**

A.5 Operations

imports Relations(A.7)
exports
sorts SELPRED JOINPRED OP
context-free syntax

ATTRIBUTE OP INT → SELPRED
ATTRIBUTE OP STRING → SELPRED
ATTRIBUTE OP ATTRIBUTE → SELPRED
SELPRED “*and*” SELPRED → SELPRED **{left}**
SELPRED “*or*” SELPRED → SELPRED **{left}**
“(” SELPRED “)” → SELPRED **{bracket}**

ATTRIBUTE “=” ATTRIBUTE → JOINPRED

“=” → OP
“$<$” → OP
“$\leq$” → OP
“$>$” → OP
“$\geq$” → OP
“$\neq$” → OP

“*Union*” “(” REL “,” REL “)” → REL
“*Intersection*” “(” REL “,” REL “)” → REL
“*Minus*” “(” REL “,” REL “)” → REL
“*Product*” “(” REL “,” REL “)” → REL

“*Select*” “(” SELPRED “,” REL “)” → REL
“*Project*” “(” ATTRIBUTELIST “,” REL “)” → REL
“*Join*” “(” JOINPRED “,” REL “,” REL “)” → REL

“*Neg*” “(” SELPRED “)” → SELPRED

priorities

SELPRED “*or*”SELPRED → SELPRED $<$ SELPRED “*and*”SELPRED → SELPRED

variables

Selpred [*0-9*]* → SELPRED
Joinpred [*0-9*]* → JOINPRED
Projectlist [*0-9*]* → ATTRIBUTELIST
Op [*0-9*]* → OP

equations

[N1] $Neg(Attribute = Str) \; = \; Attribute \neq Str$

[N2] $Neg(Attribute < Str) \; = \; Attribute \geq Str$

[N3] $Neg(Attribute \leq Str) \; = \; Attribute > Str$

[N4] $Neg(Attribute > Str) \; = \; Attribute \leq Str$

[N5] $Neg(Attribute \geq Str) \; = \; Attribute < Str$

[N6] $Neg(Attribute \neq Str) \; = \; Attribute = Str$

[N7] $Neg(Attribute = Int) \; = \; Attribute \neq Int$

[N8] $Neg(Attribute < Int) \; = \; Attribute \geq Int$

[N9] $Neg(Attribute \leq Int) \; = \; Attribute > Int$

[N10] $Neg(Attribute > Int) \; = \; Attribute \leq Int$

[N11] $Neg(Attribute \geq Int) \; = \; Attribute < Int$

[N12] $Neg(Attribute \neq Int) \ = \ Attribute = Int$

[N13] $Neg(Attribute = Attribute) \ = \ Attribute \neq Attribute$
[N14] $Neg(Attribute < Attribute) \ = \ Attribute \geq Attribute$
[N15] $Neg(Attribute \leq Attribute) \ = \ Attribute > Attribute$
[N16] $Neg(Attribute > Attribute) \ = \ Attribute \leq Attribute$
[N17] $Neg(Attribute \geq Attribute) \ = \ Attribute < Attribute$
[N18] $Neg(Attribute \neq Attribute) \ = \ Attribute = Attribute$

[N19] $Neg(Selpred_1 \ and \ Selpred_2) \ = \ Neg(Selpred_1) \ or \ Neg(Selpred_2)$
[N20] $Neg(Selpred_1 \ or \ Selpred_2) \ = \ Neg(Selpred_1) \ and \ Neg(Selpred_2)$

A.6 Optimizer

imports Check$^{(A.2)}$ Eqtransforms$^{(A.3)}$ Access$^{(A.1)}$
exports
sorts SCHEME RELDEF QUERY REPRESENTATION1 REPRESENTATION2
context-free syntax

"*scheme*" SCHEME "*query*" QUERY	→ REPRESENTATION1
"*scheme*" SCHEME "*relation*" REL	→ REPRESENTATION2
"*ReadAttributes*" "(" REPRESENTATION1 ")"	→ REPRESENTATION2
"*ReadInfo*" "(" REPRESENTATION2 ")"	→ EXTREL
{RELDEF ","}+	→ SCHEME
RELNAME "≡" "*attributes:*" ATTRIBUTELIST "*sortedon:*" ATTRIBUTELIST "*indexon:*" ATTRIBUTELIST	→ RELDEF
RELNAME	→ QUERY
"*Union*" "(" QUERY "," QUERY ")"	→ QUERY
"*Intersection*" "(" QUERY "," QUERY ")"	→ QUERY
"*Minus*" "(" QUERY "," QUERY ")"	→ QUERY
"*Product*" "(" QUERY "," QUERY ")"	→ QUERY
"*Select*" "(" SELPRED "," QUERY ")"	→ QUERY
"*Project*" "(" ATTRIBUTELIST "," QUERY ")"	→ QUERY
"*Join*" "(" JOINPRED "," QUERY "," QUERY ")"	→ QUERY
"*Lookup*" "(" RELNAME "," SCHEME "," INT ")"	→ ATTRIBUTELIST
"*Check*" "(" REPRESENTATION2 ")"	→ REPRESENTATION2
MSG	→ REPRESENTATION2
"*Transform*" "(" REPRESENTATION2 ")"	→ REPRESENTATION2

variables
S [0-9]* → SCHEME
Q [0-9]*→ QUERY

Reldefs → {RELDEF ","}*

equations

ReadAttributes. This function searches the attributes for a relation.

$$\text{[RA1]}\ \frac{\begin{array}{l} \textit{scheme } S_1 \textit{ relation } Rel_1 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_1), \\ \textit{scheme } S_2 \textit{ relation } Rel_2 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_2) \end{array}}{\begin{array}{l} \textit{ReadAttributes}(\textit{scheme } S \textit{ query } \textit{Union}(Q_1, Q_2)) = \\ \textit{scheme } S \textit{ relation } \textit{Union}(Rel_1, Rel_2) \end{array}}$$

$$\text{[RA2]}\ \frac{\begin{array}{l} \textit{scheme } S_1 \textit{ relation } Rel_1 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_1), \\ \textit{scheme } S_2 \textit{ relation } Rel_2 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_2) \end{array}}{\begin{array}{l} \textit{ReadAttributes}(\textit{scheme } S \textit{ query } \textit{Intersection}(Q_1, Q_2)) = \\ \textit{scheme } S \textit{ relation } \textit{Intersection}(Rel_1, Rel_2) \end{array}}$$

$$\text{[RA3]}\ \frac{\begin{array}{l} \textit{scheme } S_1 \textit{ relation } Rel_1 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_1), \\ \textit{scheme } S_2 \textit{ relation } Rel_2 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_2) \end{array}}{\begin{array}{l} \textit{ReadAttributes}(\textit{scheme } S \textit{ query } \textit{Minus}(Q_1, Q_2)) = \\ \textit{scheme } S \textit{ relation } \textit{Minus}(Rel_1, Rel_2) \end{array}}$$

$$\text{[RA4]}\ \frac{\begin{array}{l} \textit{scheme } S_1 \textit{ relation } Rel_1 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_1), \\ \textit{scheme } S_2 \textit{ relation } Rel_2 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_2) \end{array}}{\begin{array}{l} \textit{ReadAttributes}(\textit{scheme } S \textit{ query } \textit{Product}(Q_1, Q_2)) = \\ \textit{scheme } S \textit{ relation } \textit{Product}(Rel_1, Rel_2) \end{array}}$$

$$\text{[RA5]}\ \frac{\textit{scheme } S_1 \textit{ relation } Rel = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q)}{\begin{array}{l} \textit{ReadAttributes}(\textit{scheme } S \textit{ query } \textit{Select}(\text{Selpred}, Q)) = \\ \textit{scheme } S \textit{ relation } \textit{Select}(\text{Selpred}, Rel) \end{array}}$$

$$\text{[RA6]}\ \frac{\textit{scheme } S_1 \textit{ relation } Rel = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q)}{\begin{array}{l} \textit{ReadAttributes}(\textit{scheme } S \textit{ query } \textit{Project}(\text{Projectlist}, Q)) = \\ \textit{scheme } S \textit{ relation } \textit{Project}(\text{Projectlist}, Rel) \end{array}}$$

$$\text{[RA7]}\ \frac{\begin{array}{l} \textit{scheme } S_1 \textit{ relation } Rel_1 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_1), \\ \textit{scheme } S_2 \textit{ relation } Rel_2 = \textit{ReadAttributes}(\textit{scheme } S \textit{ query } Q_2) \end{array}}{\begin{array}{l} \textit{ReadAttributes}(\textit{scheme } S \textit{ query } \textit{Join}(\text{Joinpred}, Q_1, Q_2)) = \\ \textit{scheme } S \textit{ relation } \textit{Join}(\text{Joinpred}, Rel_1, Rel_2) \end{array}}$$

$$\text{[RA8]}\ \frac{\text{Attributelist} = \textit{Lookup}(\text{Relname}, S, 1)}{\begin{array}{l} \textit{ReadAttributes}(\textit{scheme } S \textit{ query } \text{Relname}) = \\ \textit{scheme } S \textit{ relation } \text{Relname Attributelist} \end{array}}$$

ReadInfo. This function searches sortedon- and indexon-information for a relation.

$$\text{[RI1]}\ \frac{\begin{array}{l} Extrel_1 = \textit{ReadInfo}(\textit{scheme } S \textit{ relation } Rel_1), \\ Extrel_2 = \textit{ReadInfo}(\textit{scheme } S \textit{ relation } Rel_2) \end{array}}{\begin{array}{l} \textit{ReadInfo}(\textit{scheme } S \textit{ relation } \textit{Union}(Rel_1, Rel_2)) = \\ \textit{Union}(Extrel_1, Extrel_2, < >, < >) \end{array}}$$

$$[RI2]\ \frac{Extrel_1 = ReadInfo(scheme\ S\ relation\ Rel_1),\quad Extrel_2 = ReadInfo(scheme\ S\ relation\ Rel_2)}{ReadInfo(scheme\ S\ relation\ Intersection(Rel_1,\ Rel_2)) = Intersection(Extrel_1,\ Extrel_2,\ <\ >,\ <\ >)}$$

$$[RI3]\ \frac{Extrel_1 = ReadInfo(scheme\ S\ relation\ Rel_1),\quad Extrel_2 = ReadInfo(scheme\ S\ relation\ Rel_2)}{ReadInfo(scheme\ S\ relation\ Minus(Rel_1,\ Rel_2)) = Minus(Extrel_1,\ Extrel_2,\ <\ >,\ <\ >)}$$

$$[RI4]\ \frac{Extrel_1 = ReadInfo(scheme\ S\ relation\ Rel_1),\quad Extrel_2 = ReadInfo(scheme\ S\ relation\ Rel_2)}{ReadInfo(scheme\ S\ relation\ Product(Rel_1,\ Rel_2)) = Product(Extrel_1,\ Extrel_2,\ <\ >,\ <\ >)}$$

$$[RI5]\ \frac{Extrel = ReadInfo(scheme\ S\ relation\ Rel)}{ReadInfo(scheme\ S\ relation\ Select(Selpred,\ Rel)) = Select(Selpred,\ Extrel,\ <\ >,\ <\ >)}$$

$$[RI6]\ \frac{Extrel = ReadInfo(scheme\ S\ relation\ Rel)}{ReadInfo(scheme\ S\ relation\ Project(Projectlist,\ Rel)) = Project(Projectlist,\ Extrel,\ <\ >,\ <\ >)}$$

$$[RI7]\ \frac{Extrel_1 = ReadInfo(scheme\ S\ relation\ Rel_1),\quad Extrel_2 = ReadInfo(scheme\ S\ relation\ Rel_2)}{ReadInfo(scheme\ S\ relation\ Join(Joinpred,\ Rel_1,\ Rel_2)) = Join(Joinpred,\ Extrel_1,\ Extrel_2,\ <\ >,\ <\ >)}$$

$$[RI8]\ \frac{Sortedon = Lookup(Relname,\ S,\ 2),\quad Indexon = Lookup(Relname,\ S,\ 3)}{ReadInfo(scheme\ S\ relation\ Relname\ Attributelist) = Relname\ Attributelist\ Sortedon\ Indexon}$$

Lookup function. This function searches for the relationame and returns the wanted information (attributelist, sortedon, indexon).

$$[L1]\ \frac{Relname \neq Relname_1}{Lookup(Relname,\ Relname_1 \equiv attributes\colon Attributelist\ \ sortedon\colon Sortedon\ \ indexon\colon Indexon,\ Int) = <\ >}$$

$$[L2]\ Lookup(Relname,\ Relname \equiv attributes\colon Attributelist\ \ sortedon\colon Sortedon\ \ indexon\colon Indexon,\ Reldefs,\ 1) = Attributelist$$

[L3] *Lookup*(*Relname*, *Relname* ≡ ***attributes***: *Attributelist*
sortedon: *Sortedon*
indexon: *Indexon*, *Reldefs*, 2) = *Sortedon*

[L4] *Lookup*(*Relname*, *Relname* ≡ ***attributes***: *Attributelist*
sortedon: *Sortedon*
indexon: *Indexon*, *Reldefs*, 3) = *Indexon*

[L5]
$$\frac{Relname \neq Relname_1}{\begin{array}{l} Lookup(Relname,\ Relname_1 \equiv \mathbf{attributes}\text{: } Attributelist \\ \quad sortedon\text{: } Sortedon\ indexon\text{: } Indexon,\ Reldefs,\ Int) = \\ Lookup(Relname,\ Reldefs,\ Int) \end{array}}$$

The functions in the Eqtransforms-module does not need the scheme information. The following equation takes care of this.

[TRANS]
$$\frac{Rel_1 = Detrans(Transform(Rel))}{\begin{array}{l} Transform(scheme\ S\ relation\ Rel) = \\ scheme\ S\ relation\ Rel_1 \end{array}}$$

The Check-function in the Check-module only needs rel-information. The following equations take care of this. If an error was found during the check-procedure only this error is returned and further processing (equivalence transforms and determining access strategies) is not possible.

[CHECK1]
$$\frac{Rel_1 = Check(Rel)}{\begin{array}{l} Check(scheme\ S\ relation\ Rel) = \\ scheme\ S\ relation\ Rel_1 \end{array}} \quad \textbf{otherwise}$$

[CHECK2]
$$\frac{Msg = Check(Rel)}{\begin{array}{l} Check(scheme\ S\ relation\ Rel) = \\ scheme\ S\ relation\ Rel\ error(s)\ Msg \end{array}}$$

A.7 Relations

imports Layout(??) Booleans(??) Integers(??) Strings(??)
exports
  sorts ATTRIBUTE ATTRIBUTELIST RELNAME REL
  lexical syntax
    *[A-Za-z][A-Za-z0-9.]** → ATTRIBUTE
  context-free syntax
    “<” {ATTRIBUTE “,”}* “>” → ATTRIBUTELIST
    STRING → RELNAME
    RELNAME ATTRIBUTELIST → REL

    “*Includes*” “(” ATTRIBUTELIST “,” ATTRIBUTELIST “)” → BOOL

"*MakeSub*" "(" ATTRIBUTELIST "," REL ")" → ATTRIBUTELIST
"*ElemOf*" "(" ATTRIBUTE "," ATTRIBUTELIST ")" → BOOL
"*Add*" "(" ATTRIBUTE "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"*Add*" "(" ATTRIBUTELIST "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"*Equal*" "(" ATTRIBUTELIST "," ATTRIBUTELIST ")" → BOOL
"*NumberOfElems*" "(" ATTRIBUTELIST ")" → INT

"*Un*" "(" ATTRIBUTELIST "," ATTRIBUTELIST ")" → ATTRIBUTELIST
"*Intersect*" "(" ATTRIBUTELIST "," ATTRIBUTELIST ")" → ATTRIBUTELIST

variables

Relname [*0-9*]* → RELNAME
Rel [*0-9*]* → REL
Attribute [*0-9*]* → ATTRIBUTE
Attributes [*0-9*]* → {ATTRIBUTE ","}*
Attributelist [*0-9*]* → ATTRIBUTELIST
Chars [*12*]* → CHAR*

equations

Includes function:

[I1] $Includes(Attributelist, <\ >) \ = \ true$

$$\text{[I2]}\ \frac{ElemOf(Attribute, Attributelist) = false}{Includes(Attributelist, < Attribute, Attributes >) = false}$$

$$\text{[I3]}\ \frac{ElemOf(Attribute, Attributelist) = true}{\begin{array}{l} Includes(Attributelist, < Attribute, Attributes >) \ = \\ Includes(Attributelist, < Attributes >) \end{array}}$$

MakeSub function:

[M1] $MakeSub(<\ >, Rel) \ = \ <\ >$

$$\text{[M2]}\ \frac{ElemOf(Attribute, Attributelist) = false}{\begin{array}{l} MakeSub(< Attribute, Attributes >, Relname\ Attributelist) \ = \\ MakeSub(< Attributes >, Relname\ Attributelist) \end{array}}$$

$$\text{[M3]}\ \frac{ElemOf(Attribute, Attributelist) = true}{\begin{array}{l} MakeSub(< Attribute, Attributes >, Relname\ Attributelist) \ = \\ Add(Attribute, MakeSub(< Attributes >, Relname\ Attributelist)) \end{array}}$$

ElemOf function:

[E1] $ElemOf(Attribute, <\ >) \ = \ false$

$$\text{[E2]}\ \frac{Attribute \neq Attribute_1}{\begin{array}{l} ElemOf(Attribute, < Attribute_1, Attributes >) \ = \\ ElemOf(Attribute, < Attributes >) \end{array}}$$

$$[E3]\ \frac{Attribute = Attribute_1}{ElemOf(Attribute, < Attribute_1, Attributes >) = true}$$

Add function: add an attribute to an attributelist.

$$[A1]\ \frac{ElemOf(Attribute, < Attributes >) = false}{Add(Attribute, < Attributes >) = < Attribute, Attributes >}$$

$$[A2]\ \frac{ElemOf(Attribute, Attributelist) = true}{Add(Attribute, Attributelist) = Attributelist}$$

Add function: add one attributelist to another.

$$[Add1]\ Add(< \ >, Attributelist) = Attributelist$$

$$[Add2]\ Add(< Attribute, Attributes >, Attributelist) = Add(< Attributes >, Add(Attribute, Attributelist))$$

The Equal-function. This boolean-function returns true if the two lists contain the same attributes (sequence of attributes does not have to be the same). The result will be false if the list does not contain the same attributes.

$$[Eq1]\ Equal(< \ >, < \ >) = true$$

$$[Eq2]\ Equal(< Attributes_1, Attribute, Attributes_2 >, < Attributes_3, Attribute, Attributes_4 >) = Equal(< Attributes_1, Attributes_2 >, < Attributes_3, Attributes_4 >)$$

$$[Eq3]\ Equal(Attributelist_1, Attributelist_2) = false \qquad \textbf{otherwise}$$

Function Intersect:

$$[I1]\ Intersect(Attributelist, < \ >) = < \ >$$

$$[I2]\ \frac{ElemOf(Attribute, Attributelist) = true}{Intersect(Attributelist, < Attribute, Attributes >) = Add(Attribute, Intersect(Attributelist, < Attributes >))}$$

$$[I3]\ \frac{ElemOf(Attribute, Attributelist) = false}{Intersect(Attributelist, < Attribute, Attributes >) = Intersect(Attributelist, < Attributes >)}$$

Function Un:

$$[U1]\ Un(Attributelist, < \ >) = Attributelist$$

$$[U2]\ \frac{ElemOf(Attribute, Attributelist) = true}{Un(Attributelist, < Attribute, Attributes >) = Un(Attributelist, < Attributes >)}$$

$$[U3]\ \frac{ElemOf(Attribute, Attributelist) = false}{Un(Attributelist, < Attribute, Attributes >) = Add(Attribute, Un(Attributelist, < Attributes >))}$$

NumberOfElems function:

[N1] $NumberOfElems(< >) = 0$

[N2] $NumberOfElems(< Attribute, Attributes >) = 1 + NumberOfElems(< Attributes >)$

Bibliography

[Bel92] D. Bell, and J. Grimson, *Distributed Database Systems*, Addison-Wesley Publishing Company, Inc, 1992.

[BHK87] J.A. Bergstra, J. Heering, and P. Klint, *ASF — and algebraic specification formalism*, Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint (eds.), *Algebraic Specification*, ACM Press in co-operation with Addison-Wesley, 1989.

[Fle89] Fleming and von Halle, *Handbook of Relational Database Design*, Addison-Wesley Publishing Company, Inc, 1989.

[Garda89] G. Gardarin, P. Valduriez, *Relational Databases and Knowledge Bases*, Addison-Wesley Publishing Company, Inc, 1989.

[Graef87] G. Graefe, *Rule-Based Query Optimization in Extensible Database Systems*, Computer Sciences Technical Report nr. 724, University of Wisconsin — Madison, 1987.

[Hen88] P.R.H. Hendriks, *ASF system user's guide*, Report CS-R8823, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[Hen89] P.R.H. Hendriks, *Lists and associative functions in algebraic specifications — semantics and implementation*, Report CS-R8908, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[HHKR89] J. Heering, P.R.H. Hendriks. P. Klint, and J. Rekers, *The syntax definition formalism SDF — reference manual*, SIGPLAN Notices, Vol 14, pp: 43-75, 1989.

[Jar81] Kalervo Järvelin, *A query optimizer for a relational database system*, Department of Mathematical Sciences, University of Tampere, Finland, 1981.

[Kim85] W. Kim, D. Reiner, and D. Batory, *Query Processing in Database Systems*, Springer-Verlag, Berlin Heidelberg, 1985.

[Kli93] P. Klint, editor, *The ASF+SDF Meta-environment User's Guide*, 1993.

[Pro93] Vakgroep Programmatuur, *Syllabus van het college Programmeeromgevingen II*, Centre for Mathematics and Computer Science, Amsterdam, 1993.

[Ull82] J.D. Ullman, *Principles of Database Systems, second edition*, Computer Science Press, Inc, 1982.