

EBERHARD KARLS UNIVERSITÄT TÜBINGEN
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl für Datenbanksysteme

SQL Code Generation for Non-Relational Source Languages

Diplomarbeit in Informatik

Thomas Ressel

Supervisor: Prof. Dr. Torsten Grust

Advisor: Dr. Jan Rittinger

April 30, 2011

Tübingen, den 30. April 2011

Ich versichere hiermit, diese Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Thomas Ressel

Abstract

Non-relational source languages are compiled into a relational query plan over the *Logical Algebra*. These plans are optimised by the relational query optimiser *Pathfinder* before being translated into SQL. So far, Pathfinder's SQL code generator translates the Logical Algebra plans directly into SQL. In this thesis we propose the *SQL Algebra*. The SQL Algebra is an intermediate algebra for the SQL code generation of Pathfinder and precisely reflects the structure of a SQL query. We describe the translation from the Logical Algebra into the SQL Algebra and the translation from the SQL Algebra into the target language SQL. Further, we present the SQL Algebra as a new level to perform optimisations on. These optimisations cope with artefacts that result from the transition from non-relational languages to relational languages and emphasise two aspects of SQL code: Readability and performance. We close the thesis with a discussion of the gains accomplished.

Acknowledgements

Above all, I want to thank my advisor Jan Rittinger for always having the time and the patience to answer my questions. I am also very thankful to my parents who supported me in so many ways during my studies. I express my gratitude to my girlfriend Susanna Hübschmann for always being there for me. Last but not least, I want to thank Sara for proofreading.

Contents

Abstract	5
Acknowledgements	7
1 Introduction	11
2 Logical Algebra	13
2.1 Operators	14
2.2 Number Generation	15
2.3 Example	16
3 SQL Algebra	19
3.1 Operators	19
3.2 Expressions	20
4 SQL	23
4.1 Common Table Expressions	23
4.2 Window Functions	25
5 Translation from Logical Algebra into SQL Algebra	27
5.1 Translation	28
5.1.1 Translating into SQL Algebra Operators	28
5.1.2 Translating into SQL Algebra Expressions	31
5.2 Binding	33
5.3 Translation Example	34
6 Optimisations on SQL Algebra	37
6.1 Building IN Lists	37
6.2 Merging of Adjacent Projections	38
6.3 Antijoin	42
7 Translation from SQL Algebra into SQL	45
7.1 Translation	46
7.1.1 Translating SQL Algebra Expressions	46
7.1.2 Translating SQL Algebra Operators	47
7.2 Binding	51

Contents

7.3	Aliases	53
7.4	Translation Example	54
8	Experiments	57
8.1	DB2	58
8.2	Postgres	60
9	Conclusions	63
9.1	Outlook	63
	List of Figures	65
	List of Tables	67
	Bibliography	69

1 Introduction

Database management systems are capable of querying large amounts of data in a very efficient manner. These capabilities which result from careful design performed over the past 30 years, are often left unused when processing data in computer programmes: The database management system, or DBMS, is used as pure data storage and to transfer data into the programming language’s runtime heap where operations are performed that also could have been performed in the DBMS.

The functional programming language *Ferry* [3] connects the two distant shores of non-relational programming languages and relational databases. The intention is to harness DBMSs as programming language co-processors: We avoid moving big amounts of data from the database into the programming language’s runtime heap. Instead, we strive to figure out which operations the programme performs on the data, outsourcing these operations to the database and hence, moving smaller amounts of data into the programming language’s runtime heap.

The central compilation strategy to gain a relational query plan of a non-relational language, called *loop lifting* [4], faces the challenge of mapping concepts of the non-relational world, such as ordered and potentially nested lists, into the relational world of unordered flat tables.

In the context of this thesis, relational query plans are processed by the relational query optimiser *Pathfinder*. Pathfinder expresses relational query plans over the *Logical Algebra*, an algebra with strong similarities to Relational Algebra. Pathfinder performs optimisations on the Logical Algebra plans and translates them into SQL code. The optimisations regard artefacts resulting from the transition from non-relational to relational languages and also aspects that are of more general nature.

So far, Pathfinder’s SQL code generator [6] translates directly from Logical Algebra into SQL code. In this thesis we propose the SQL code generation with an intermediate algebra, the *SQL Algebra*. In the compilation chain, it resides between Logical Algebra and SQL code. As the name implies, it exhibits a more SQL-like structure than the Logical Algebra. Further, we propose optimisations on SQL Algebra plans that strive to improve the performance and readability of the resulting SQL code.

Besides *Ferry*, other non-relational languages are compiled into a relational representation in the guise of plans over the Logical Algebra. These languages include *LINQ* [8], *Haskell* [2] and *Links* [9]. Pathfinder is also a relational XQuery compiler that translates a given XQuery into a Logical Algebra plan. The SQL query resulting from the Logical Algebra plan is then executed against the tabular representation of an XML document on a DBMS in order to compute the result of the XQuery. Target languages other than SQL are being addressed, one example being the X100 Algebra

[1].

This document starts with the introduction of the involved languages: Chapter 2 presents Pathfinder's internal representation language for relational query plans: The Logical Algebra. In Chapter 3 we propose the SQL Algebra, an intermediate algebra to generate SQL code of. Selected SQL language constructs are introduced in Chapter 4.

With the languages on hand, we begin the journey to SQL code starting from Logical Algebra, departing for SQL Algebra. This first translation is presented in Chapter 5. Chapter 6 discusses optimisations on SQL Algebra plans, having the target language SQL in mind. The translation introduced in Chapter 7 takes us from SQL Algebra to the target language SQL.

Testing and speed measurements will be discussed in Chapter 8. Chapter 9 provides a conclusion and outlook.

2 Logical Algebra

The *Logical Algebra* is the algebra Pathfinder uses to represent plans with and to perform optimisations on. Regarding the core set of operators, it shares strong similarities with Relational Algebra. Relational Algebra, known from database textbooks, is used by database management systems to represent queries internally and to perform optimisations on them. It is a query language for the Relational Model, and it operates over a set of relations in the mathematical sense. The core set of operators in both Relational Algebra and Logical Algebra consists of selection, projection, union, cross and difference operators. More complex operators such as the join operator can be expressed with this core set. Beside the core set, the Logical Algebra contains more complex operators that are used regularly, *e.g.* the join operator that is constituted by a cross operator and an select operator.

The Logical Algebra of Pathfinder consists of three types of operators: The before mentioned core set of operators, operators for table references (both references for database tables and references for literal tables) and operators that represent XQuery concepts such as an XML document encoded in the database. Operators representing XQuery concepts will be omitted in this introduction of the Logical Algebra since we focus on Logical Algebra plans derived from non-relational languages other than XQuery.

The differences between the Logical Algebra and Relational Algebra are: Relational Algebra operates on relations so duplicate tuples do not exist. This differs from Logical Algebra that operates on tables which can contain duplicate tuples. The duplicate tuples can be removed explicitly with a special operator, the so-called δ operator.

A key difference is the concept of number generation. In a nutshell, number generation produces numbers that satisfy certain predicates. In Relational Algebra, there are no numbers produced but only those present in the relations are used. In Chapter 2.2 we introduce the use and the semantics of number generation.

Another difference is that the Logical Algebra contains operators that expand the schema of a table. Such operators add one column to the schema containing the values of some operation¹. This differs from Relational Algebra where schema-expanding operators do not exist.

We now present the Logical Algebra operators. Chapter 2.2 introduces number generation in more detail and Chapter 2.3 gives an example of a query plan in Logical Algebra.

¹One such operation is *e.g.* number generation.

Operator	Semantics
$\circlearrowleft(v)$	Plan serialisation
$\pi_{col'_1:col_1,\dots,col'_n:col_n}(v)$	Project on columns col_i , rename col_i to col'_i
$\sigma_{col}(v)$	Select column col
$\cup(v, w), \setminus(v, w), \times(v, w)$	Union, difference and cross operator
$\bowtie_{pred_1,\dots,pred_n}(v, w)$	Join on predicates $pred_i$
$\delta(v)$	Removal of duplicate rows
$f_{res:\langle col_1,\dots,col_n \rangle}(v)$	Function f applied to columns col_1, \dots, col_n
$@_{col:value}(v)$	Attach column col containing constant value $value$
$\kappa_{col,type,res}(v)$	Values of column col cast to type $type$
$agg_{kind_1,res_1:\langle col_1 \rangle,\dots,kind_n,res_n:\langle col_n \rangle/p}(v)$	Generate $kind_i$ -aggregates of columns col_i , partitioned by column p
$\#_{kind,res:\langle col_1,\dots,col_n \rangle/p}(v)$	Generate numbers of kind $kind$ in col_i order, partitioned by column p (we assume that col_i contain ordering information $asc/desc$)
$\boxed{\begin{array}{l} name \\ col_1:cname_1 \dots col_n:cname_n \end{array}}$	Reference to table $name$ with original column names $cname_i$ and their respective algebra names col_i
$\boxed{\begin{array}{l} col_1 \dots col_n \\ tuple_1 \\ \vdots \\ tuple_m \end{array}}$	Literal table with columns col_i and tuples $tuple_j$

Table 2.1: Logical Algebra operators.

2.1 Operators

Table 2.1 gives an overview of the Logical Algebra operators and their semantics. We describe the meaning of the variables used in the table in the order of their respective appearance: The children of an operator are denoted with v and w . Column names are referred to as col and $cname$. The predicates $pred$ of the join operator are of the form $col_1 \otimes col_2$ with \otimes being a comparison function. Columns which contain the results of an operation are denoted with res . Such columns are added to the current schema. $value$ stands for an atomic value and $type$ stands for a data type. Both are general and not specific to the Logical Algebra. $kind$ denotes a Logical Algebra aggregation or number generation kind. Regarding the literal table operator, we always assume that tuples fit into the schema of their table, *i.e.* tuples $tuple_1, \dots, tuple_m$ are n -tuples if the schema is col_1, \dots, col_n . We call the information carried by an operator semantical content or semantical information.

<i>a</i>	<i>b</i>	<i>col·rowid</i>	<i>col·rank</i>	<i>col·rowrank</i>	<i>col·rownum</i>
3	1	0	1	2	3''
2	0	1	3	3	1
3	0	2	1	2	1
1	0	3	7	4	1'
3	1	4	1	2	2''
1	0	5	7	4	2'
2	1	6	3	3	2
4	2	7	0	1	1

Figure 2.1: Number generating operator example. The Logical Algebra operators for the col_{kind} columns are: $\#_{rowid, col\cdot rowid:\langle \rangle}$, $\#_{rank, col\cdot rank:\langle a[desc] \rangle}$, $\#_{rowrank, col\cdot rowrank:\langle a[desc] \rangle}$ and $\#_{rownum, col\cdot rownum:\langle b[asc] \rangle / a[asc]}$. Numbers with single (double) quotes can be used interchangeably.

2.2 Number Generation

In order to faithfully map information about ordering when translating ordered (potentially nested) lists from the non-relational world into unordered (flat) tables in the relational world, the Logical Algebra contains number generating operators. These operators are introduced by the loop lifting compilation and generate numbers that are attached to the schema. The generated numbers can be arbitrary numbers or numbers that obey a specified order. The numbers can further be generated regarding a certain partitioning. There are four kinds of number generation:

1. **rowid** Produces an unique number for every row.
2. **rank** Produces arbitrary numbers obeying a specified order. Identical rows regarding the order get the same number assigned.
3. **rowrank** Produces dense numbers (starting from 1) obeying a specified order. Similar to rank except that rows of neighbored groups (regarding the order) get numbers assigned that differ by exactly 1.
4. **rownum** Produces for each specified partition dense numbers (starting from 1) obeying a specified order. Within a partition, identical rows (regarding the order) do not get the same number assigned but numbers that differ by exactly 1. Distribution of numbers within a group of identical rows is non-deterministic.

Figure 2.1 provides an example for the generation of numbers performed by the number generating operator $\#$. Note that values marked with one single quote (two single quotes, respectively) could have also been assigned interchangeably because of the non-deterministic assignment of numbers within a group of identical columns (regarding the order).

<i>part</i>	<i>supplier</i>	<i>price</i>
wind screen	5	95
wind screen	1	90
tyre	3	30
steering wheel	4	33
steering wheel	1	32
steering wheel	2	32
ignition lock	1	20
ignition lock	2	15
ignition lock	3	19
ignition lock	4	17

Figure 2.2: Table *Parts*.

2.3 Example

As an example for a query plan in Logical Algebra, we try to find out which are the two cheapest suppliers per part in the table of spare parts *Parts* of Figure 2.2. The query plan is shown in Figure 2.3. The result as well as the two intermediate results marked ① and ② are presented in Figure 2.4. This little example that is supposed to demonstrate the behaviour of the number generating operator, also shows Logical Algebra peculiarities: In order to perform the computation we cannot use any values before they have been explicitly attached to the schema. Because of this we have a relatively high stack of nine operators for this rather simple computation.

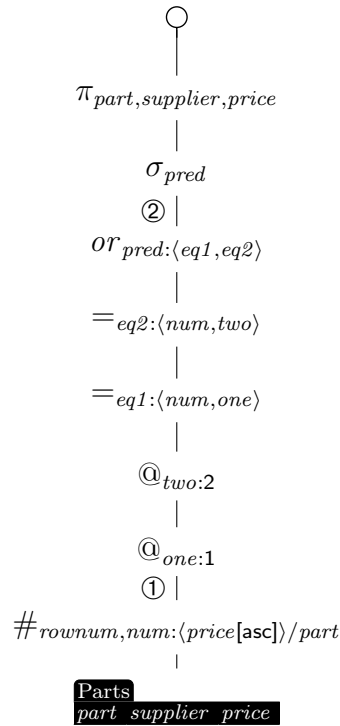


Figure 2.3: Example query in Logical Algebra. ① and ② denote intermediate results.

part	supplier	price	num	part	supplier	price	num	one	two	eq1	eq2	pred
wind screen	5	95	2	wind screen	5	95	2	1	2	false	true	true
wind screen	1	90	1	wind screen	1	90	1	1	2	true	false	true
tyre	3	30	1	tyre	3	30	1	1	2	true	false	true
steering wheel	4	33	3	steering wheel	4	33	3	1	2	false	false	false
steering wheel	1	32	1	steering wheel	1	32	1	1	2	true	false	true
steering wheel	2	32	2	steering wheel	2	32	2	1	2	false	true	true
ignition lock	1	20	4	ignition lock	1	20	4	1	2	false	false	false
ignition lock	2	15	1	ignition lock	2	15	1	1	2	true	false	true
ignition lock	3	19	3	ignition lock	3	19	3	1	2	false	false	false
ignition lock	4	17	2	ignition lock	4	17	2	1	2	false	true	true

(a) Intermediate result at ①

(b) Intermediate result at ②, not qualifying tuples regarding *pred* greyed out

part	supplier	price
wind screen	5	95
wind screen	1	90
tyre	3	30
steering wheel	1	32
steering wheel	2	32
ignition lock	2	15
ignition lock	4	17

(c) Result

Figure 2.4: Example query result.

3 SQL Algebra

Like the Logical Algebra, the *SQL Algebra* is an algebra that operates over tables and that shares strong similarities with Relational Algebra. In contrast to the Logical Algebra it distinguishes between operators and expressions. Having in mind that SQL code is derived from the SQL Algebra, one can associate the SQL Algebra operators with SQL query clauses like **SELECT**, **FROM** or **WHERE**, whereas the SQL Algebra expressions are associated with SQL code that usually resides in lists tailing SQL query clauses, *e.g.* column names in a **SELECT** clause.

To get a better idea of how SQL Algebra steps towards SQL code coming from Logical Algebra we repeat the query from the previous chapter (Figure 2.3) in SQL Algebra: It is shown in Figure 3.1. The first thing that comes to attention is that only four SQL Algebra operators are consumed in comparison to nine Logical Algebra operators. This is because of the distinction between operators and expressions.

We stated that in SQL Algebra no values need to be attached to the schema before they can be used, which is in fact necessary in Logical Algebra. Taking a closer look to Figure 3.1, we notice that the lower SQL Algebra **PROJECT** operator adds a column *num* (containing numbers generated by a SQL Algebra number generating expression) to the schema that is then used in both equal comparison expressions in the SQL Algebra **SELECT** operator. This is not because we needed to attach the column *num* to the schema before we can use it but rather because of a property of the target language SQL: Expressions in the SQL Algebra **SELECT** operator will be compiled to SQL **WHERE** clauses and in such a **WHERE** clause, the SQL functions representing number generating expressions are not allowed¹. The projection takes place before the SQL Algebra **SELECT** operator in order to make the numbers generated by the number generating expression referable under column name *num*.

Both queries in Logical Algebra and SQL Algebra have a stacked shape but this is not necessarily true for all plans. Since we have operators with two children, plans can also come in the guise of trees and DAGs. In fact we generally assume that plans exhibit a DAG structure.

3.1 Operators

Table 3.1 introduces the SQL Algebra operators and their semantics. The meaning of the variables used in the table is the following: Children of an operator are denoted

¹see Chapter 4.2 for these SQL functions

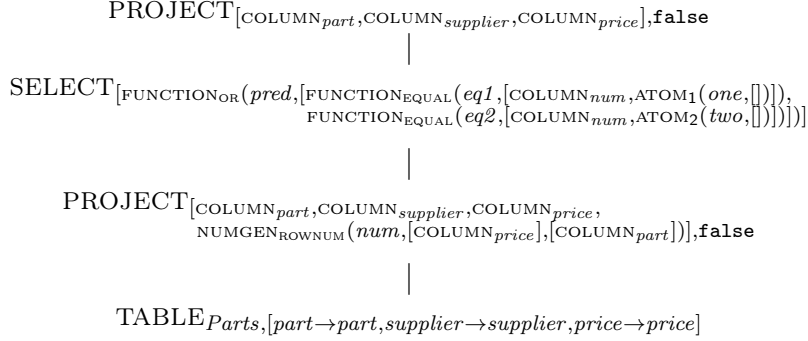


Figure 3.1: Example query in SQL Algebra. See Tables 3.1 and 3.2 for the semantics of the SQL Algebra operators and expressions.

with x and y . SQL Algebra expressions are referred to as e . The `distinct` flag of the project operator is a Boolean value. We further denote column names with col and $cname$. Regarding the `LITERAL TABLE` operator, we always assume that tuples fit into the schema of their table, *i.e.* tuples $tuple_1, \dots, tuple_m$ are n -tuples if the schema is col_1, \dots, col_n . Information contained in an operator is referred to as semantical content or semantical information.

3.2 Expressions

Table 3.2 shows the SQL Algebra expressions and their semantics. Every expression comes in the form `EXPR($res, [e^*]$)` with column res containing the result values and list $[e^*]$ of SQL Algebra expressions containing the input. The result values of the expression are either values produced by the expression, values from a table column that the expression represents or a constant value from an expression representing an atom. The variables used in the table have the following meaning: Column names are denoted as col . `value` stands for an atomic value and `type` stands for a data type. Both are general and not specific to the SQL Algebra. `FUNC` denotes the kind (or: name) of a SQL Algebra function and `KIND` denotes a SQL Algebra aggregation or number generation kind. We call the information carried by an expression semantical content or semantical information.

Operator	Semantics
SERIALIZE(x)	Plan serialisation
PROJECT _{[e_1, \dots, e_n], distinct} (x)	Project on expressions e_i , eliminate duplicates when flag <code>distinct</code> is <code>true</code>
SELECT _[e_1, \dots, e_n] (x)	Select expressions e_i
UNION(x, y), EXCEPT(x, y), CROSS(x, y)	Union, except and cross operator
JOIN _[e_1, \dots, e_n] (x, y)	Join on expressions e_i
GROUPBY _{[e_1, \dots, e_n], [$e_{p.1}, \dots, e_{p.m}$]} (x)	Project on expressions e_i , partition by expressions $e_{p.j}$
TABLE _{name, [$col_1 \rightarrow cname_1, \dots, col_n \rightarrow cname_n$]}	Reference to table $name$ with columns $cname_i$ and their respective algebra names col_i
LITERAL TABLE _{[col_1, \dots, col_n], [$tuple_1, \dots, tuple_m$]}	Literal table with columns col_i and tuples $tuple_j$

Table 3.1: SQL Algebra operators.

Expression	Semantics
COLUMN _{col} ($res, []$)	Reference to column col , renamed to res
COLUMN _{col}	Short for COLUMN _{col} ($col, []$)
ATOM _{value} ($res, []$)	Constant value <code>value</code> in column res
CONVERT _{type} ($res, [e]$)	Convert type of expression e to <code>type</code>
FUNCTION _{FUNC} ($res, [e_1, \dots, e_n]$)	Function <code>FUNC</code> applied to expressions e_i
AGGR _{KIND} ($res, [e]$)	KIND-Aggregate of expression e in res
NUMGEN _{KIND} ($res, [e_{s.1}, \dots, e_{s.n}], [e_{p.1}, \dots, e_{p.m}]$)	Generation of numbers of kind <code>KIND</code> in $e_{s.i}$ order, partitioned by $e_{p.j}$ (we assume that $e_{s.i}$ contain ordering information <code>asc/desc</code>)

Table 3.2: SQL Algebra expressions.

4 SQL

SQL is the universal query language for relational database systems. It is defined in an ANSI and ISO standard in different revisions. The code that is about to be derived from SQL Algebra plans follows the SQL:99 revision, the latest revision of the SQL standard is SQL:2008.

Since SQL is an extensive language, basic constructs will not be introduced here. We point to [7] for that. Instead, only concepts significant to our needs are introduced: Common table expressions and window functions.

4.1 Common Table Expressions

When formulating complex queries in SQL, it is useful to compose them of simpler subqueries. This helps keeping the queries readable and understandable. Assigning query blocks names that are referenced like table names in other query blocks further raises readability.

One way to accomplish this is using SQL's `VIEW` language construct. A view assigns a name to a query block and makes the name referable in the whole database. The downside of views is that, unless deleted, they remain in the database even when they are never referenced again.

Besides views, SQL offers the possibility to make query blocks referable under a name using *common table expressions* (CTE). They differ from views regarding the visibility: A common table expression is only visible in the context of a query and not database-wide. In particular, common table expressions are dropped when the query terminates.

Common table expressions are introduced by a SQL `WITH` clause and use the syntax shown in Figure 4.1. After the `WITH`, names are assigned to query blocks. The column list of a query can optionally be stated after the name that is assigned to the query block. This increases readability since the columns that are associated with a name are graspable at first sight. Providing a column list after the name is mandatory if the corresponding query block contains at least one duplicate name in the column list.

An example query using common table expressions is shown in Figure 4.2: It contains a subquery named `cte` that yields a table containing suppliers together with the number of parts priced below £ 30 they have in stock. From `cte`, the maximum number of parts available at one supplier is extracted. Then, again from `cte`, every supplier having the maximum number of affordable parts is selected.

```
WITH name1[(columnlist1)] AS  
    (query1)  
    , . . . ,  
    namen[(columnlistn)] AS  
    (queryn)  
query
```

Figure 4.1: SQL WITH clause syntax. *name* denotes a name, *columnlist* stands for a list of column names and *query* for an arbitrary SQL query block.

```
WITH cte (supplier, count) AS  
    (SELECT supplier,  
         COUNT(*) AS count  
     FROM Parts  
     WHERE price < 30  
     GROUP BY supplier)  
SELECT supplier, count  
FROM cte  
WHERE count = (SELECT MAX(count)  
              FROM cte)
```

Figure 4.2: Example SQL query with common table expressions. Using information from table *Parts* (Table 2.2), calculate which suppliers have the largest number of parts priced below £ 30 in stock.

Common table expressions help to represent the DAG shape of SQL Algebra plans. Plans in DAG shape imply that certain operators (which ultimately represent query blocks) can be referenced by more than one operator. In the context of a SQL query this means that the subquery has to be referable under a name and that is where common table expressions fit perfectly.

4.2 Window Functions

In the context of SQL code generation from SQL Algebra, we use window functions to model the number generating operators in SQL. Window functions are functions that compute the value for a given row using information from the row's window. Every row has a window, which is a set of rows, assigned. Windows are defined using the concepts of partitioning and ordering: Partitioning assigns every row the set of all rows of the partition the row is in. If no partitioning is used, every row gets the set of all rows of the table assigned as window. Ordering specifies the ordering of the rows in each window.

The window functions employed are `ROW_NUMBER` and `DENSE_RANK`, using the following syntax:

```
ROW_NUMBER|DENSE_RANK () OVER ([PARTITION BY partlist]  
                                [ORDER BY orderlist])
```

A window for a function is defined with an `OVER` clause. Note that both the partitioning and the ordering clauses are optional.

The different kinds of SQL Algebra number generating expressions are mapped to SQL functions `ROW_NUMBER` and `DENSE_RANK` as follows:

1. ROWID:
`ROW_NUMBER () OVER ()`
2. RANK:
`DENSE_RANK () OVER (ORDER BY orderlist)`
3. ROWRANK:
`DENSE_RANK () OVER (ORDER BY orderlist)`
4. ROWNUM:
`ROW_NUMBER () OVER (PARTITION BY partlist ORDER BY orderlist)`

Note the empty `OVER` clause for the SQL Algebra ROWID expression: The values do not have to obey an order, they only have to be unique. Having said this, it follows that partitioning must not be used in the `OVER` clause because values can occur repeatedly between partitions, offending the uniqueness demand.

SQL Algebra expressions of kind RANK and ROWRANK have an order list specified and the expression of kind ROWNUM additionally has a partitioning list as input. These lists are derived from the semantical content of the SQL Algebra number generating expressions.

The output of functions ROW_NUMBER and DENSE_RANK complies with the behaviour of the number generating operators as described in Chapter 2.2.

5 Translation from Logical Algebra into SQL Algebra

Plans in both Logical Algebra and SQL Algebra come in DAG shape. We perform the translation of a Logical Algebra plan into SQL Algebra by traversing the Logical Algebra plan bottom-up: Starting in the DAG's root we first descend to the leaves and then ascend the different paths back to the root, translating operator by operator. Because of the DAG shape, operators can be encountered twice or more times. Whenever this happens the operator will not be translated another time, but rather its translation result from the first encounter will be used. Further descending will not be performed at this point because all the operators below the re-encountered operator have already been translated. By translating we mean the application of an inference rule that specifies how the semantical information of the Logical Algebra operator is to be translated into SQL Algebra. We assume that the result of the translation is annotated to the Logical Algebra operator so that it can be found and used again when the operator is encountered another time.

With the distinction between operators and expressions in SQL Algebra in contrast to Logical Algebra, that consists only of operators, it is apparent that some Logical Algebra operators are translated into SQL Algebra expressions while others are translated into SQL Algebra operators. Generally, when a Logical Algebra operator has a corresponding operator in the SQL Algebra, it is translated to this correspondent. With two exceptions:

The first exception affects the translation of Logical Algebra select and project operators. We do not translate them directly into their correspondents in the SQL Algebra, but only when its necessary: While traversing the DAG plan bottom-up, we collect the semantical information of every Logical Algebra select and project operator in a selection and projection list. When at some point the construction of a SQL Algebra SELECT and PROJECT operator is unavoidable, a *bind* occurs. A bind marks the actual construction of a SQL Algebra SELECT and PROJECT operator on top of the current SQL Algebra operator using the semantical information collected in the selection and projection list. This way, the changes so far made in the schema and in the selection predicates are made visible. This compilation technique is called *lazy binding*. The construction of a SQL Algebra SELECT and PROJECT operator is considered unavoidable when we reach a set operator (as it demands identical schemas in its children) or when an operator is referred to more than once (as the following operators need to see the current schema).

The second exception regards the translation of the Logical Algebra distinct opera-

tor (δ). There is no dedicated distinct operator in the SQL Algebra like there is one in the Logical Algebra. Instead, the SQL Algebra PROJECT operator carries a Boolean flag **distinct** in accordance with the SQL language (**SELECT DISTINCT**). When translating a Logical Algebra distinct operator, we perform a bind and set the flag **distinct** to **true** in the SQL Algebra PROJECT operator. This complies with the semantics of the Logical Algebra distinct operator: Removal of duplicates at the current state of the plan, which means that we have to make all changes so far made visible.

We now introduce the inference rules of the translation from the Logical Algebra into SQL Algebra and their notation. We arranged the order of presentation in a way that the inference rules of those Logical Algebra operators that directly translate to a corresponding SQL Algebra operator appear first, followed by the rules that perform the translation into SQL Algebra expressions. After that we introduce the bind function in Chapter 5.2 to formalise the before mentioned concept of binding. For now it suffices to know that binding puts a SQL Algebra SELECT and PROJECT operator on top of a SQL Algebra operator using information from a selection and projection list that is filled while traversing the plan bottom-up. Chapter 5.3 provides an example translation.

5.1 Translation

The inference rules describing the translation from Logical Algebra operators into SQL Algebra operators and expressions use the following notation:

$$\overline{\circledast \Rightarrow (proj, sel, op)}$$

The rule reads as follows: Logical Algebra operator \circledast is translated (\Rightarrow) into a 3-tuple containing (1) a projection list *proj* of mappings $col \rightarrow e$ with *col* being a column name and *e* being a SQL Algebra expression, (2) a (semantically conjunctive) selection list *sel* of SQL Algebra expressions and (3) a SQL Algebra operator *op*.

In the inference rules, we use *v* and *w* to denote Logical Algebra operators. *col* and *res* denote column names in the Logical Algebra as well as in the SQL Algebra. We further use *e* to denote SQL Algebra expressions. The symbol ++ stands for the concatenation of lists. Whenever a value does not matter we use – as a placeholder.

5.1.1 Translating into SQL Algebra Operators

We start with the inference rules of the table operators, which reside in the plan leaves and are therefore translated first. They either reference a table in a database or provide a literal table with actual tuples. The translation builds a SQL Algebra TABLE operator and a projection list containing information about the columns of the table.

REFTBL

$$\begin{array}{c}
e_i \equiv \text{COLUMN}_{col_i} \mid_{i=1, \dots, n} \\
cols \equiv [col_1 \rightarrow cname_1, \dots, col_n \rightarrow cname_n] \\
\hline
\boxed{\begin{array}{l} name \\ col_1:cname_1 \dots col_n:cname_n \end{array}} \Rightarrow ([col_1 \rightarrow e_1, \dots, col_n \rightarrow e_n], [], \text{TABLE}_{name, cols})
\end{array}$$

LITTLBL

$$\begin{array}{c}
e_i \equiv \text{COLUMN}_{col_i} \mid_{i=1, \dots, n} \\
cols \equiv [col_1, \dots, col_n] \\
\hline
\boxed{\begin{array}{l} col_1 \dots col_n \\ tuple_1 \\ \vdots \\ tuple_m \end{array}} \Rightarrow ([col_1 \rightarrow e_1, \dots, col_n \rightarrow e_n], [], \text{LITERAL TABLE}_{cols, [tuple_1, \dots, tuple_m]})
\end{array}$$

Translating the Logical Algebra set operators difference (\setminus) and union (\cup), we encounter the first situation where binding is necessary. The set operators demand that the schemas of their children are identical regarding the column names as well as regarding the order in which the columns appear.¹ That means we have to make the information which has so far been collected about the schema and the selection predicates visible. In order to do so, we perform a bind that uses the before mentioned selection and projection lists to construct a SQL Algebra SELECT and PROJECT operator. The bind function² \Rightarrow further consumes a SQL Algebra operator and a Boolean flag **distinct**. The constructed SQL Algebra SELECT and PROJECT operators are then put on top of the SQL Algebra operator and the SQL Algebra PROJECT operator gets the flag **distinct** set to **false**.

DIFFERENCE, UNION

$$\begin{array}{c}
(\odot, \oslash) \in \{(\setminus, \text{EXCEPT}), (\cup, \text{UNION})\} \\
v \Rightarrow (proj_v, sel_v, op_v) \\
(proj_v, sel_v, op_v, \mathbf{false}) \Rightarrow (proj'_v, [], op'_v) \\
w \Rightarrow (proj_w, sel_w, op_w) \\
(proj_w, sel_w, op_w, \mathbf{false}) \Rightarrow (proj'_w, [], op'_w) \\
\hline
\odot(v, w) \Rightarrow (proj'_v, [], \oslash(op'_v, op'_w))
\end{array}$$

Translating the cross operator there is no need to bind the children except there are name conflicts present in the projection lists of the children: This means that *e.g.* in both projection lists of the children operators v and w , a column col is referenced. This occurs only if v and w origin from the same operator, *i.e.* the cross operator is an 'self-cross' (following the naming style of self-join).³

CROSS

$$\begin{array}{c}
v \Rightarrow (proj_v, sel_v, op_v) \\
w \Rightarrow (proj_w, sel_w, op_w) \\
\hline
\times(v, w) \Rightarrow (proj_v ++ proj_w, sel_v ++ sel_w, \text{CROSS}(op_v, op_w))
\end{array}$$

¹We call schemas and column lists with this property aligned.

²The bind function is introduced formally in Chapter 5.2.

³Coping with name conflicts is not reflected in the rules. The implementation however does perform a bind when name conflicts are present.

Translating the thetajoins operator builds SQL Algebra function expressions that represent the comparison predicates specified in the Logical Algebra thetajoins operator and constructs a SQL Algebra JOIN operator using these predicate expressions. If name conflicts are present (self-join), the children of the thetajoins have to be bound.³

THETAJOIN

$$\begin{array}{c}
 v \Rightarrow (proj_v, sel_v, op_v) \\
 w \Rightarrow (proj_w, sel_w, op_w) \\
 \left. \begin{array}{l}
 (col_{l.i} \rightarrow e_{l.i}) \in proj'_v \\
 (col_{r.i} \rightarrow e_{r.i}) \in proj'_w \\
 (\otimes_i, FUNC_i) \in \{ (=, EQUAL), (<, LT), \dots \} \\
 e_i \equiv FUNCTION_{FUNC_i}(-, [e_{l.i}, e_{r.i}]) \Big|_{i=1, \dots, n}
 \end{array} \right\} \\
 \hline
 \bowtie_{col_{l.1} \otimes_1 col_{r.1}, \dots, col_{l.n} \otimes_n col_{r.n}} (v, w) \Rightarrow (proj_v ++ proj_w, sel_v ++ sel_w, JOIN_{[e_1, \dots, e_n]}(op_v, op_w))
 \end{array}$$

Translating the distinct operator causes a bind with the Boolean flag `distinct` set to `true` in the SQL Algebra PROJECT operator: Making all changes visible and removing duplicates reflects the semantics of the Logical Algebra distinct operator.

DISTINCT

$$\begin{array}{c}
 v \Rightarrow (proj, sel, op) \quad (proj, sel, op, \mathbf{true}) \Rightarrow (proj', [], op') \\
 \hline
 \delta(v) \Rightarrow (proj', [], op')
 \end{array}$$

Translating an aggregation operator causes the construction of SQL Algebra aggregation expressions. Then, a SQL Algebra GROUPBY operator is built using the SQL Algebra aggregation expressions. To preserve semantics, a bind has to be performed before aggregating information.

AGG1

$$\begin{array}{c}
 v \Rightarrow (proj, sel, op) \\
 (proj, sel, op, \mathbf{false}) \Rightarrow (proj', [], op') \\
 \left. \begin{array}{l}
 (col_i \rightarrow e_{a.i}) \in proj' \\
 (kind_i, KIND_i) \in \{ (sum, SUM), (min, MIN), \dots \} \\
 e_{c.i} \equiv COLUMN_{res_i} \\
 e_i \equiv AGGR_{KIND_i}(res_i, [e_{a.i}]) \Big|_{i=1, \dots, n}
 \end{array} \right\} \\
 \hline
 (agg_{kind_1, res_1: \langle col_1 \rangle, \dots, kind_n, res_n: \langle col_n \rangle}(v) \Rightarrow \\
 ([res_1 \rightarrow e_{c.1}, \dots, res_n \rightarrow e_{c.n}], [], GROUPBY_{[e_1, \dots, e_n], []}(op'))
 \end{array}$$

Unlike the preceding translation, the following translation of the aggregate operator also takes a partitioning column into account: Information is aggregated after the input has been partitioned according to the values contained in the partitioning column.

AGG2

$$\begin{array}{c}
v \Rightarrow (proj, sel, op) \\
(proj, sel, op, \mathbf{false}) \Rightarrow (proj', [], op') \\
\left. \begin{array}{l}
(col_i \rightarrow e_{a_i}) \in proj' \\
(kind_i, KIND_i) \in \{(sum, SUM), (min, MIN), \dots\} \\
e_{c_i} \equiv COLUMN_{res_i} \\
e_i \equiv AGGR_{KIND_i}(res_i, [e_{a_i}]) \\
(p \rightarrow e_p) \in proj' \\
e_{c_p} \equiv COLUMN_p
\end{array} \right|_{i=1, \dots, n} \\
\hline
agg_{kind_1, res_1: \langle col_1 \rangle, \dots, kind_n, res_n: \langle col_n \rangle / p}(v) \Rightarrow \\
([p \rightarrow e_{c_p}, res_1 \rightarrow e_{c_1}, \dots, res_n \rightarrow e_{c_n}], [], GROUPBY_{[e_p, e_1, \dots, e_n], [e_p]}(op'))
\end{array}$$

The serialisation operator marks the plan root. It completes the translation of the plan by binding its upmost operator and performing one last projection onto the result columns. Because of the bottom-up traversal of the DAG shaped Logical Algebra plan, it is the very last operator that is translated.

SER-REL

$$\begin{array}{c}
v \Rightarrow (proj, sel, op) \\
(proj, sel, op, \mathbf{false}) \Rightarrow (proj', [], op') \\
(iter \rightarrow e_{iter}) \in proj' \\
(pos \rightarrow e_{pos}) \in proj' \\
(item_i \rightarrow e_{item_i})|_{i=1, \dots, n} \in proj' \\
\hline
\bigcirc_{iter, pos, [item_1, \dots, item_n]}(v) \Rightarrow (-, -, SERIALIZE_{e_{iter}, [e_{pos}], [e_{item_1}, \dots, e_{item_n}]}(op'))
\end{array}$$

5.1.2 Translating into SQL Algebra Expressions

Translating the select operator does not lead to the construction of a SQL Algebra SELECT operator. Instead, the SQL Algebra expression e that corresponds to the selected column col is added to the selection list. This way, various Logical Algebra select operators can be merged until a bind actually builds a SQL Algebra SELECT operator.

$$\begin{array}{c}
\text{SELECT} \\
v \Rightarrow (proj, sel, op) \quad (col \rightarrow e) \in proj \\
\hline
\sigma_{col}(v) \Rightarrow (proj, sel++[e], op)
\end{array}$$

Translating the project operator does not provoke the construction of a SQL Algebra PROJECT operator. Instead, the SQL Algebra expressions of the columns which it projects on are constituting a new projection list. At this point, a possible column renaming has to be taken into account: Auxiliary function $copy(e, col)$ returns a physical copy of SQL Algebra expression $e \equiv \text{EXPR}(res, [-])$ with its res field replaced by col : $\text{EXPR}(col, [-])$

PROJECT

$$\frac{\begin{array}{l} v \Rightarrow (proj, sel, op) \\ (col_i \rightarrow e_i) \in proj \\ e'_i \equiv copy(e_i, col'_i) \Big|_{i=1, \dots, n} \end{array}}{\pi_{col'_1:col_1, \dots, col'_n:col_n}(v) \Rightarrow ([col'_1 \rightarrow e'_1, \dots, col'_n \rightarrow e'_n], sel, op)}$$

Translating the Logical Algebra operators function, attach and cast operators are straightforward. With the semantical contents we build SQL Algebra expressions that are added to the projection list.

FUNC

$$\frac{\begin{array}{l} v \Rightarrow (proj, sel, op) \\ (col_i \rightarrow e_i) \in proj \Big|_{i=1, \dots, n} \\ (f, FUNC) \in \{ (=, EQUAL), (<, LT), (+, ADD), (and, AND), \dots \} \\ e \equiv FUNCTION_{FUNC}(res, [e_1, \dots, e_n]) \end{array}}{f_{res:\langle col_1, \dots, col_n \rangle}(v) \Rightarrow (proj++[res \rightarrow e], sel, op)}$$

ATTACH

$$\frac{\begin{array}{l} v \Rightarrow (proj, sel, op) \quad e \equiv ATOM_{value}(col, []) \end{array}}{\textcircled{a}_{col:value}(v) \Rightarrow (proj++[col \rightarrow e], sel, op)}$$

CAST

$$\frac{\begin{array}{l} v \Rightarrow (proj, sel, op) \\ (col \rightarrow e_{col}) \in proj \\ e \equiv CONVERT_{type}(res, [e_{col}]) \end{array}}{\kappa_{col,type,res}(v) \Rightarrow (proj++[res \rightarrow e], sel, op)}$$

Translating the number generation operator involves constructing a SQL Algebra number generating expression and adding it to the projection list. The translation additionally entails a binding. This is semantically only strictly necessary for Logical Algebra number generating operators of kind *rowid* but does on the other hand introduce no overhead since an optimisation will merge the possibly resulting adjacent project operators later. Regard the Logical Algebra plan in Figure 2.3 for an example: When the Logical Algebra number generating operator of kind *rownum* is bound, a SQL Algebra PROJECT operator is built. A bind at the Logical Algebra function operator of kind *or* would result in another SQL Algebra PROJECT operator directly above the first one, containing all columns that are added by Logical Algebra operators between markings ① and ②. This adjacent SQL Algebra PROJECT operators will be merged when performing the optimisation described in Chapter 6.2.

NUMGEN1

$$\begin{array}{c}
v \Rightarrow (proj, sel, op) \\
(col_i \rightarrow e_i) \in proj|_{i=1,\dots,n} \\
(kind, KIND) \in \left\{ \begin{array}{l} (rowid, ROWID), (rank, RANK), \\ (rowrank, ROWRANK), (rownum, ROWNUM) \end{array} \right\} \\
e \equiv \text{NUMGEN}_{KIND}(res, [e_1, \dots, e_n], []) \\
(proj++[res \rightarrow e], sel, op) \Rightarrow (proj', [], op') \\
\hline
\#_{kind, res: \langle col_1, \dots, col_n \rangle}(v) \Rightarrow (proj', [], op')
\end{array}$$

Number generating operators can also come with a partitioning column. In this case, numbers are generated for each partition that is constituted by the values in the partitioning column.

NUMGEN2

$$\begin{array}{c}
v \Rightarrow (proj, sel, op) \\
(col_i \rightarrow e_i) \in proj|_{i=1,\dots,n} \\
(p \rightarrow e_p) \in proj \\
(kind, KIND) \in \left\{ \begin{array}{l} (rowid, ROWID), (rank, RANK), \\ (rowrank, ROWRANK), (rownum, ROWNUM) \end{array} \right\} \\
e \equiv \text{NUMGEN}_{KIND}(res, [e_1, \dots, e_n], [e_p]) \\
(proj++[res \rightarrow e], sel, op) \Rightarrow (proj', [], op') \\
\hline
\#_{kind, res: \langle col_1, \dots, col_n \rangle/p}(v) \Rightarrow (proj', [], op')
\end{array}$$

5.2 Binding

We now present the inference rules for the technique of binding which was introduced at the beginning of this chapter. The inference rules use the notation

$$\overline{(proj, sel, op, distinct)} \Rightarrow (proj', [], op')$$

and read as follows: A list of (1) a projection list $proj$, (2) a selection list sel , (3) a SQL Algebra operator op and (4) a Boolean flag **distinct** are translated (\Rightarrow) into a 3-tuple of (1) an altered projection list $proj'$, (2) an empty selection list $[]$ and (3) a SQL Algebra PROJECT operator op' . This means setting a SQL Algebra PROJECT operator⁴ on top of the operator op to make the in the projection list $proj$ collected information about the schema⁵ visible to all operators above op . Binding is triggered whenever we have at least two references to a Logical Algebra operator or when it is necessary to make the changes visible because of the semantics of an operator. This is the case for set operators union (\cup) or difference (\setminus) that demand their children to have aligned schemas. It is further necessary for the translation of the aggregate operator and the number generating operators.

⁴and a SQL Algebra SELECT operator, if selection list sel was not empty

⁵and about the selection predicates, if selection list sel was not empty

The first bind function puts a SQL Algebra PROJECT operator on top of the given operator. No SQL Algebra SELECT operator is built because the selection list is empty. The altered projection list is filled with SQL Algebra column expressions. These are the SQL Algebra COLUMN expressions the rest of the plan works with.

$$\begin{array}{c}
 \text{BIND1} \\
 e_{c.i} \equiv \text{COLUMN}_{col_i} \mid_{i=1,\dots,n} \\
 proj' \equiv [col_1 \rightarrow e_{c.1}, \dots, col_n \rightarrow e_{c.n}] \\
 op' \equiv \text{PROJECT}_{[e_1,\dots,e_n],\text{distinct}}(op) \\
 \hline
 ([col_1 \rightarrow e_1, \dots, col_n \rightarrow e_n], [], op, \text{distinct}) \Rightarrow (proj', [], op')
 \end{array}$$

The second bind function performs the same task as the first one except that it additionally builds a SQL Algebra SELECT operator with the information contained in the non-empty selection list.

$$\begin{array}{c}
 \text{BIND2} \\
 e_{c.i} \equiv \text{COLUMN}_{col_i} \mid_{i=1,\dots,n} \\
 proj' \equiv [col_1 \rightarrow e_{c.1}, \dots, col_n \rightarrow e_{c.n}] \\
 op' \equiv \text{PROJECT}_{[e_1,\dots,e_n],\text{distinct}}(\text{SELECT}_{[e_{sel.1},\dots,e_{sel.m}]}(op)) \\
 \hline
 ([col_1 \rightarrow e_1, \dots, col_n \rightarrow e_n], [e_{sel.1}, \dots, e_{sel.m}], op, \text{distinct}) \Rightarrow (proj', [], op')
 \end{array}$$

The following meta rule expresses that a bind is triggered automatically if a Logical Algebra operator is referenced more than once. $d(v)$ denotes the number of references to Logical Algebra operator v .

$$\begin{array}{c}
 \text{TRIGGER-BIND} \\
 d(v) > 1 \\
 v \Rightarrow (proj, sel, op) \quad (proj, sel, op, \text{false}) \Rightarrow (proj', [], op') \\
 \hline
 v \Rightarrow (proj', [], op')
 \end{array}$$

5.3 Translation Example

In Figure 5.1 we present the translation of the Logical Algebra plan shown in Figure 2.3 into SQL Algebra: The Logical Algebra operators are annotated with the information collected while traversing the plan bottom-up. This information are the projection list $proj$, the selection list sel and a SQL Algebra operator op as known from the inference rules presented in Chapter 5.1. In order to save space, we omitted the annotations for some Logical Algebra operators. The items contained in the projection lists $proj$ are also abbreviated: Instead of $col \rightarrow e$ we write e . Further, we abbreviate COLUMN to COL, FUNCTION to FUN and EQUAL to EQ.

The number generating operator ($\#$) in the Logical Algebra plan has two annotations. The lower annotation results from the translation of the actual number generating operator. This annotation is replaced by the upper annotation because of the bind that takes place when translating number generating operators. The Logical

Algebra project operator is also bound because it marks the plan root. We ignore the serialisation operator in this example translation as well as information about ordering.

The root of the resulting SQL Algebra plan is the SQL Algebra operator op (marked by ③) in the annotation of the root of the Logical Algebra plan.

6 Optimisations on SQL Algebra

SQL Algebra plans that result from the 'greedy' translation described in the previous chapter still contain artefacts from Logical Algebra plans. These artefacts result from the compilation of non-relational languages to a relational representation in the form of plans over the Logical Algebra and lead to non-optimal SQL code.

In this chapter we discuss some of these artefacts in order to optimise the SQL code derived from SQL Algebra plans regarding readability and performance, taking into account properties of the target language SQL. The three proposed optimisations named IN lists optimisation, merge projections and anti-join optimisation are introduced in Chapters 6.1, 6.2 and 6.3, respectively.

6.1 Building IN Lists

One problem regarding the readability of SQL code arises because common predicates that compare one column to a list of values have the shape of trees in SQL Algebra. Such trees result in 'nested' SQL code such as the following:

```
SELECT *
  FROM Parts
 WHERE ((price = 95) OR ((price = 90) OR
    ((price = 33) OR ((price = 32) OR
      ((price = 30) OR ((price = 20) OR
        ((price = 19) OR (price = 15))))))))))
```

We propose a simple optimisation that leads to more readable SQL code:

```
SELECT *
  FROM Parts
 WHERE price IN (95, 90, 33, 32, 30, 20, 19, 15)
```

For this optimisation we introduce an IN-ATOMS expression to the SQL Algebra. This expression expands Table 3.2 by adding

$$\text{IN-ATOMS}(res, [e], [e_1, \dots, e_n])$$

to express the condition that the list of SQL Algebra expressions $[e_1, \dots, e_n]$ contains the SQL Algebra expression e . The result of the check is attached in column res .

The inference rules for the IN lists optimisation we introduce now are applied when traversing the expressions of a SQL Algebra SELECT, PROJECT or JOIN operator.

Rule START-IN-LIST determines if SQL Algebra expression e represents a disjunction of two equality comparisons of a column with constant values and builds a SQL Algebra IN-ATOMS expression which replaces SQL Algebra expression e .

$$\begin{array}{c}
 \text{START-IN-LIST} \\
 e_{col} \equiv \text{COLUMN}_{col} \\
 e_{atom.i} \equiv \text{ATOM}_{-}(-, [])|_{i=1,2} \\
 e \equiv \text{FUNCTION}_{\text{OR}}(res, [\text{FUNCTION}_{\text{EQUAL}}(-, [e_{col}, e_{atom.1}]), \\
 \text{FUNCTION}_{\text{EQUAL}}(-, [e_{col}, e_{atom.2}])]) \\
 \hline
 e \xrightarrow{*} \text{IN-ATOMS}(res, [e_{col}], [e_{atom.1}, e_{atom.2}])
 \end{array}$$

Rule EXPAND-IN-LIST expands an existing SQL Algebra IN-LIST expression by an atomic value if there is a disjunction of the IN-LIST expression and an equality comparison that compares the same column col as the IN-LIST expression to an atomic value. SQL Algebra expression e is replaced by this expanded IN-LIST expression.

$$\begin{array}{c}
 \text{EXPAND-IN-LIST} \\
 e_{col} \equiv \text{COLUMN}_{col} \\
 e_{atom} \equiv \text{ATOM}_{-}(-, []) \\
 e \equiv \text{FUNCTION}_{\text{OR}}(res, [\text{IN-ATOMS}(-, [e_{col}], [e_1, \dots, e_n]), \\
 \text{FUNCTION}_{\text{EQUAL}}(-, [e_{col}, e_{atom}])]) \\
 \hline
 e \rightarrow \text{IN-ATOMS}(res, [e_{col}], [e_1, \dots, e_n, e_{atom}])
 \end{array}$$

6.2 Merging of Adjacent Projections

In SQL Algebra plans, various projections can follow each other. Such adjacent projections are often unnecessary, *e.g.* when they do not change the schema at all. Adjacent projections result for instance from the restrictive translation of the Logical Algebra number generating operators: In the translation from Logical Algebra into SQL Algebra, every time a Logical Algebra number generating operator is encountered, a bind takes place, introducing a SQL Algebra PROJECT operator. This is only strictly necessary when the number generating operator is of kind *rowid* since this operator contains no semantical information that would enable us to restore its result values later in the plan.

We introduce an optimisation to merge such adjacent projections in order to avoid unreadable SQL Algebra plans as well as unnecessary **SELECT FROM WHERE** blocks in the resulting SQL code. Assume that we have a stack of three projections over a table in a SQL Algebra plan. The situation is depicted in Figure 6.1. We observe that instead of introducing the SQL Algebra expression $\text{FUNCTION}_{\text{ADD}}(d, [\text{COLUMN}_b, \text{ATOM}_5(-, [])])$ in the lowest SQL Algebra PROJECT operator of the plan and the SQL Algebra expression $\text{NUMGEN}_{\text{RANK}}(e, [\text{COLUMN}_a, \text{COLUMN}_c], [])$ in the middle PROJECT operator,

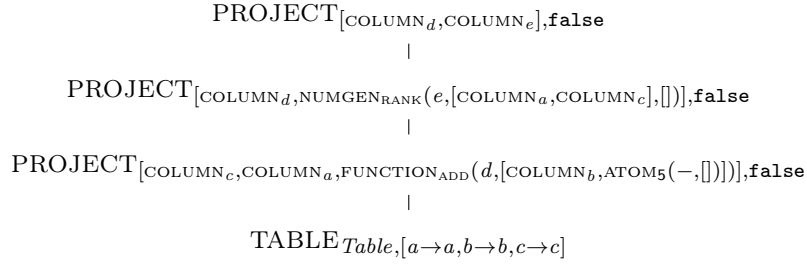


Figure 6.1: Merge projections example.

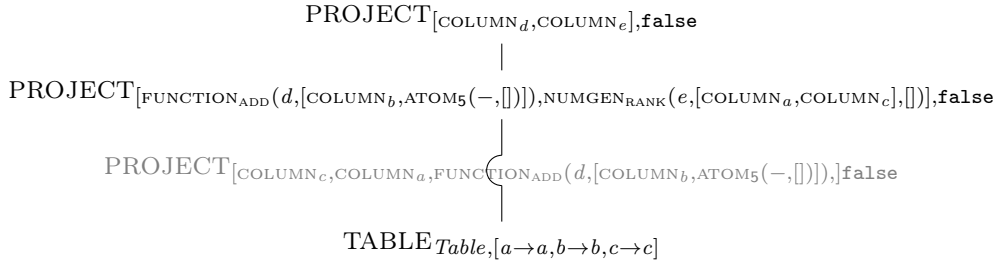


Figure 6.2: Merge projections example. Situation after first merge.

we could introduce both in a single PROJECT operator. Further, the upmost project operator does not change the schema at all.

The approach to merge projections is to traverse the plan bottom-up looking for adjacent SQL Algebra PROJECT operators. For any such pair we merge the PROJECT operators by replacing every SQL Algebra COLUMN expression in the expressions of the upper PROJECT operator with a physical copy of the corresponding SQL Algebra expression (regarding the column name) that appears in the expressions of the lower PROJECT operator. Then, the lower PROJECT operator is bypassed and not visible anymore in the plan.

Figure 6.2 shows the situation after the first merge. The lower PROJECT operator is bypassed. In addition, a physical copy of the SQL Algebra expression $\text{FUNCTION}_{\text{ADD}}(d, [\text{COLUMN}_b, \text{ATOM}_5(-, [])])$ is put in the place of the SQL Algebra expression COLUMN_d in the upper PROJECT operator.

The situation after the second merge is shown in Figure 6.3. The lower PROJECT operator is bypassed. In the upper PROJECT operator, the SQL Algebra expressions COLUMN_d and COLUMN_e are replaced by physical copies of the corresponding expressions (regarding the column name) that appear in the expressions of the lower PROJECT operator: SQL Alg. expression $\text{FUNCTION}_{\text{ADD}}(d, [\text{COLUMN}_b, \text{ATOM}_5(-, [])])$ and SQL Algebra expression $\text{NUMGEN}_{\text{RANK}}(e, [\text{COLUMN}_a, \text{COLUMN}_c], [])$.

When merging adjacent projections we have to take special care when SQL Algebra NUMGEN expressions of kind ROWID appear in the expressions of a project operator that is referenced more than once. Since the values produced by expressions

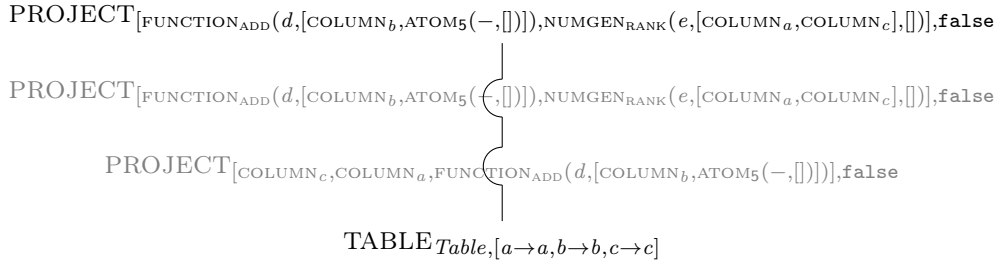


Figure 6.3: Merge projections example. Situation after second merge.

of kind ROWID directly depend on the current schema, copying them into another PROJECT operator with a different schema results in inconsistencies. Figure 6.4a depicts the situation using an simplified SQL Algebra operator notation: Assume the upper PROJECT operator contains a reference to the column of the ROWID expression that resides in the lower PROJECT operator. If the PROJECT operators are merged, a physical copy of the ROWID expression is copied into the upper PROJECT operator. The situation after the merge is shown in Figure 6.4b: Due to its semantics, the ROWID expression in the upper PROJECT operator produces other values than the ROWID expression in the lower PROJECT operator. Assuming further that the following PROJECT operators references the column of the ROWID expression, the plan can not be any longer expected to produce a correct result.

We avoid this erroneous behaviour by simply not merging PROJECT operators that (1) contain SQL Algebra expressions of kind ROWID and (2) that are referenced more than once and (3) if at least one expression of kind ROWID is referenced by more than one following operator. Note that if all SQL Algebra expressions of kind ROWID are referenced by only one following operator, merging poses no problem.

To preserve plan semantics, we have also to be careful when adjacent PROJECT operators have the Boolean flag `distinct` set to `true`. If only the flag `distinct` of the upper PROJECT operator is `true`, merging poses no problem. On the other hand, if only the `distinct` flag of the lower PROJECT operator is `true`, no binding is performed. If the `distinct` flags of both PROJECT operators are `true`, binding is only allowed if both PROJECT operators have the same schema.

We present the inference rules for the merge projections optimisations: The rule REPLACE-COL-REFS is supposed to be applied when traversing a tree of SQL Algebra expressions in a SQL Algebra PROJECT operator. Consuming a SQL Algebra COLUMN expression and a list of SQL Algebra expressions, it looks up the corresponding SQL Algebra expression for the SQL Algebra COLUMN expression (regarding column name *col*) and returns a physical copy of it, whose result column was changed to *res* in order to get hold of a possible column renaming: $\text{copy}(\text{EXPR}(\text{col}, [-]), \text{res}) \equiv \text{EXPR}(\text{res}, [-])$.

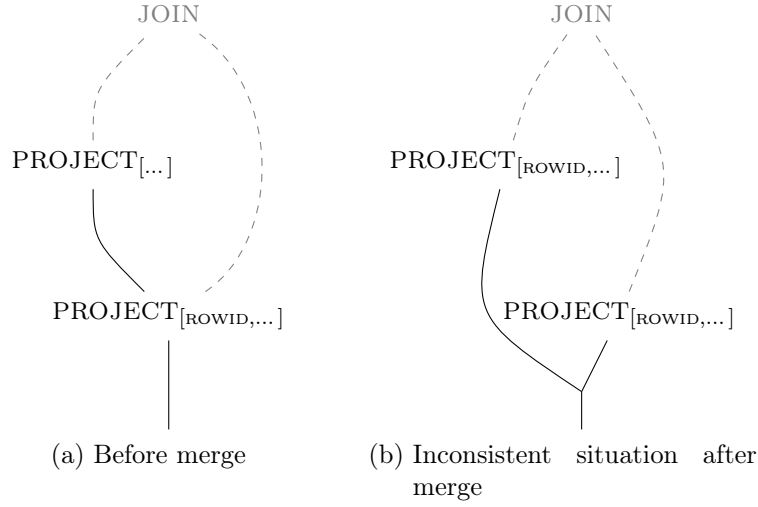


Figure 6.4: Merge projections: Problematic ROWID expressions. An simplified SQL Algebra operator notation was used.

$$\frac{\text{REPLACE-COL-REFS} \quad e \equiv \text{EXPR}(\text{col}, [-])}{\text{COLUMN}_{\text{col}}(\text{res}, []), (\dots, e, \dots) \xrightarrow{*} \text{copy}(e, \text{res})}$$

The rule MERGE-PROJECTIONS performs the merging and is applied during a traversal of the SQL Algebra plan: The SQL Algebra COLUMN expressions in the upper SQL Algebra PROJECT operator are replaced by the corresponding SQL Algebra expressions (regarding column name) of the lower SQL Algebra PROJECT operator.¹ This replacement is performed by rule REPLACE-COL-REFS.

$$\frac{\text{MERGE-PROJECTIONS1} \quad e_i, (e'_1, \dots, e'_m) \xrightarrow{*} e''_{i=1, \dots, n}}{\text{PROJECT}_{[e_1, \dots, e_n], \text{distinct}}(\text{PROJECT}_{[e'_1, \dots, e'_m], \text{false}}(x)) \rightarrow \text{PROJECT}_{[e''_1, \dots, e''_n], \text{distinct}}(x)}$$

The following rule covers the case when both adjacent SQL Algebra PROJECT operators have the Boolean flag **distinct** set to **true**. In this situation, merging is only allowed if the operators have the same schema.

$$\frac{\text{MERGE-PROJECTIONS2} \quad \left. \begin{array}{l} e_i \equiv \text{EXPR}(\text{col}_i, [-]) \\ e'_i \equiv \text{EXPR}(\text{col}_i, [-]) \\ e_i, (e'_1, \dots, e'_n) \xrightarrow{*} e''_{i=1, \dots, n} \end{array} \right|}{\text{PROJECT}_{[e_1, \dots, e_n], \text{true}}(\text{PROJECT}_{[e'_1, \dots, e'_n], \text{true}}(x)) \rightarrow \text{PROJECT}_{[e''_1, \dots, e''_n], \text{true}}(x)}$$

¹The rules do not reflect the check for presence of problematic number generating expressions of kind ROWID.

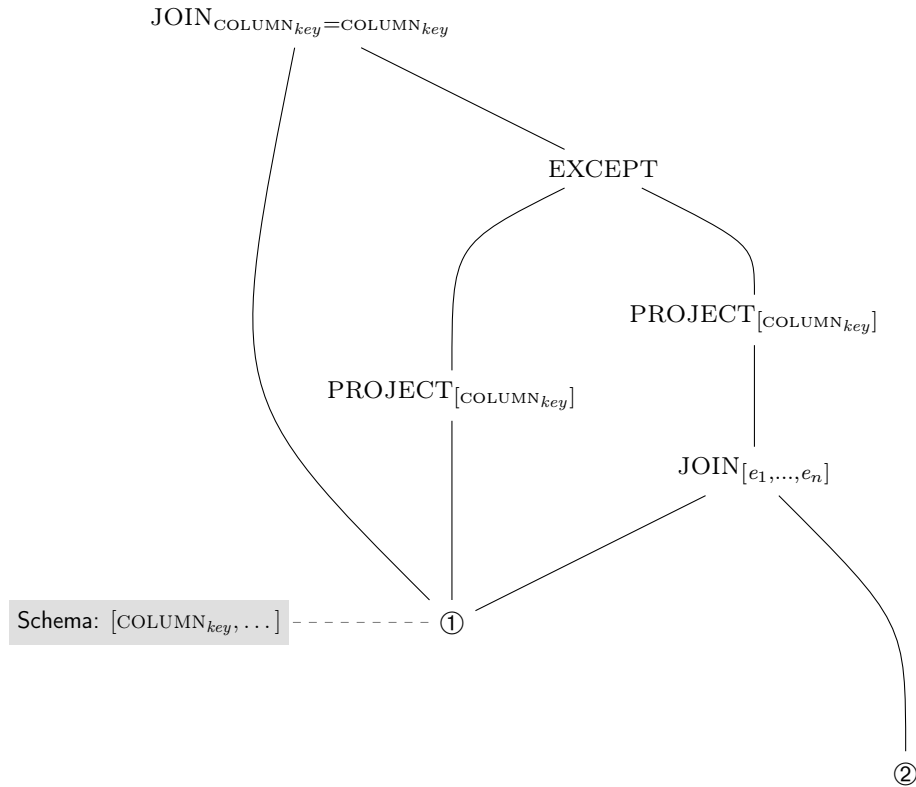


Figure 6.5: Pattern of SQL Algebra operators resulting from compilation of SQL statements `NOT IN` and `NOT EXISTS`. *key* denotes a column with key property. ① and ② mark arbitrary SQL Algebra operators. An simplified SQL Algebra operator notation was used to keep the pattern readable.

6.3 Antijoin

So far, Pathfinder's SQL code generator does not know of the SQL statements `NOT IN` and `NOT EXISTS`. This means that compiling a SQL query into a SQL Algebra plan results the `NOT IN` and `NOT EXISTS` statements of the SQL query to be expressed with the Logical Algebra operators known from Table 2.1.

Compiling SQL statements `NOT IN` and `NOT EXISTS` into SQL Algebra (via Logical Algebra) results in a pattern of SQL Algebra operators around an `EXCEPT` operator. The SQL equivalent `EXCEPT` of this operator may turn out to be very expensive to evaluate and thus causes Pathfinder to produce inefficient SQL code. Figure 6.5 shows the pattern using a simplified SQL Algebra operator notation. The pattern is similar for both statements, showing the only difference in the join predicates e_1, \dots, e_n . We defer the explanation of this difference until we explained the pattern: The `EXCEPT` operator subtracts from all rows that are present in the SQL Algebra operator marked ① those rows that have an join partner in SQL Algebra operator marked ② regarding

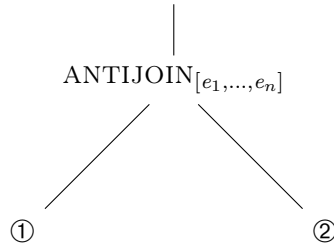


Figure 6.6: Pattern replaced by an anti-join operator.

the theta join predicates e_1, \dots, e_n . This means that above the EXCEPT operator, only those rows are present, that do not have an join partner. Since the following equal join on key columns key does simply re-expand the schema, the whole pattern can be substituted with an anti-join: All rows except those that do have a join partner equal the rows that do not have an join partner. This matches the semantics of the anti-join. We introduce an anti-join operator to the set of SQL Algebra operators presented in Table 3.1:

$$\text{ANTIJOIN}_{[e_1, \dots, e_n]}(x, y)$$

SQL Algebra expressions $[e_1, \dots, e_n]$ denote the anti-join predicates and SQL Algebra operators x and y denote the children of the anti-join. This SQL Algebra variant of the anti-join behaves like an anti-semijoin: Rows of the right child y are only considered in an existential manner. We can therefore ignore duplicate removal in the right child, if present.

The situation after the replacement of the pattern with an anti-join is shown in Figure 6.6: The children of the anti-join are the operators marked ① and ② in Figure 6.5. The anti-join predicates are the same as the predicates of the theta join: $[e_1, \dots, e_n]$.

An anti-join can be expressed in SQL by NOT IN and NOT EXISTS statements. The decision whether to build a NOT IN or a NOT EXISTS statement depends on the anti-join predicates: If the predicates consist solely of equality comparisons, the anti-join can be translated into an uncorrelated NOT IN statement. Otherwise, a correlated NOT EXISTS statement is built. NOT IN statements are easier to process for DBMSs as the correlation is already resolved. If a DBMS fails to resolve the correlation of a NOT EXISTS statement, the correlated subquery has to be computed for each row of the outer query, which results in unacceptable running time.

Instead of an inference rule we list the conditions that must hold to safely replace the pattern shown in Figure 6.5 by a SQL Algebra ANTIJOIN operator. The conditions are: (1) SQL Algebra operator ① contains a column with key property, denoted COLUMN_{key} . (2) SQL Algebra operator ① is the child of a SQL Algebra EXCEPT operator with schema key . (3) ① is also the second child of the SQL Algebra EXCEPT operator, after being joined with SQL Algebra operator ② regarding the arbitrary join

predicates $[e_1, \dots, e_n]$. (4) The except operator and operator $\textcircled{1}$ are natural-joined in order to re-expand the schema of the EXCEPT operator. (5) Selection of rows is performed exclusively in the theta join. If all these conditions are satisfied, a SQL Algebra ANTIJOIN operator can replace the pattern while preserving plan semantics.

7 Translation from SQL Algebra into SQL

We present the translation from SQL Algebra into SQL. In contrast to the translation presented in Chapter 5 we do not translate from algebra to algebra, but we aim to construct executable SQL code that reflects the semantics of SQL Algebra plans. We picked the SQL Algebra having this translation into SQL code in mind. Therefore, SQL Algebra shows strong similarities to SQL code.

At the beginning of Chapter 5 we introduced the principle of binding. We do the same here and immediately point out that here binding means something fundamentally different: By binding we mean the construction of meaningful SQL code. In order to do so, while traversing the DAG shaped plan bottom-up, information is collected in a column list, a from list and a where list until we can or must build a **SELECT FROMWHERE** block. This block can then be used in two manners: Firstly, we can make the block referable under a certain name in the query. This is realised by the means of *common table expressions* if there is more than one reference to the SQL Algebra operator which compiles to the block. The second option is to put the block into the from list, an action that results in nested subqueries. This happens if the SQL Algebra operator is referred to only once.

Similar to the translation presented in Chapter 5, the DAG shaped SQL Algebra plans are translated applying inference rules during a bottom-up traversal. We suppose the translation results to be annotated to the SQL Algebra operators in order to reuse them when the operator is re-encountered.

We now introduce the inference rules for the translation from SQL Algebra into SQL code and their notation. First, the inference rules for the SQL Algebra expressions are shown, followed by the inference rules for the SQL Algebra operators. We chose this order because the translation of operators often embodies the translation of expressions. After that we introduce the bind function formally in Chapter 7.2. For now it suffices to know that binding means the construction of a **SELECT FROMWHERE** block. In Chapter 7.3 we introduce rules that cope with a peculiarity that needs to be considered when translating SQL Algebra plans into SQL, namely, table aliases. Chapter 7.4 gives an example of a translation.

7.1 Translation

7.1.1 Translating SQL Algebra Expressions

The inference rules describing the translation from SQL Algebra expressions into SQL code use the following notation:

$$\overline{C, \otimes \mapsto^* (res, s)}$$

The rule is to be read as follows: A column list C (containing mappings $col \rightarrow s$ with col being a column name and s being a SQL expression) and a SQL Algebra expression \otimes are translated (\mapsto^*) into a pair of (1) a column name res and (2) a SQL expression s .

In the rules, col and res denote column names in both SQL Algebra and SQL code. We further use e to denote SQL Algebra expressions and s to denote SQL expressions *e.g.* $AVG(price)$.

Associating a column name res with a SQL expression s in pair (res, s) is needed to guarantee consistent connection between **SELECT FROM WHERE** blocks regarding column names.

We start with the inference rules of the SQL Algebra **COLUMN** and **ATOM** expressions since they constitute the leaves of every SQL Algebra expression tree.

Translating a **COLUMN** expression means looking up its corresponding SQL expression in the column list where it is filed under a certain name.

COLUMN

$$\overline{[\dots, (col \rightarrow s), \dots], \text{COLUMN}_{col}(res, []) \mapsto^* (res, s)}$$

Translating an **ATOM** expression simply returns a pair of the atom's column name and its value.

ATOM

$$\overline{C, \text{ATOM}_{value}(col, []) \mapsto^* (col, value)}$$

Translating the residuary expressions involves the translation of the expression's arguments to SQL expressions and applying functions to these SQL expressions. The result column name and the constructed SQL expression are returned.

FUNCTION

$$\overline{C, e_i \mapsto^* (col_i, s_i) |_{i=1, \dots, n} \\ (\text{FUNC}, f) \in \{(\text{EQUAL}, =), (\text{LT}, <), (\text{ADD}, +), (\text{AND}, \text{AND}), \dots\}} \\ C, \text{FUNCTION}_{\text{FUNC}}(res, [e_1, \dots, e_n]) \mapsto^* (res, f(s_1, \dots, s_n))$$

$$\text{AGGR} \quad \frac{C, e \mapsto^* (col, s) \quad (\text{KIND}, f) \in \{(\text{SUM}, \text{SUM}), (\text{MIN}, \text{MIN}), \dots\}}{C, \text{AGGR}_{\text{KIND}}(res, [e]) \mapsto^* (res, f(s))}$$

$$\text{CONVERT} \quad \frac{C, e \mapsto^* (col, s)}{C, \text{CONVERT}_{\text{type}}(res, [e]) \mapsto^* (res, \text{CAST}(s \text{ AS type}))}$$

For the following inference rule, we assume that the ordering information (`ASC`, `DESC`) for the `ORDER BY` clause is contained in SQL expressions $s_{s.i}$ implicitly.

$$\text{NUMGEN} \quad \frac{C, e_{s.i} \mapsto^* (col_{s.i}, s_{s.i})|_{i=1, \dots, n} \quad C, e_{p.i} \mapsto^* (col_{p.i}, s_{p.i})|_{i=1, \dots, m} \quad (\text{KIND}, f) \in \left\{ \begin{array}{l} (\text{ROWID}|\text{ROWNUM}, \text{ROW_NUMBER}), \\ (\text{RANK}|\text{ROWRANK}, \text{DENSE_RANK}) \end{array} \right\}}{C, \text{NUMGEN}_{\text{KIND}}(res, [e_{s.1}, \dots, e_{s.n}], [e_{p.1}, \dots, e_{p.m}]) \mapsto^* (res, f() \text{ OVER } (\text{PARTITION BY } s_{p.1}, \dots, s_{p.m} \text{ ORDER BY } s_{s.1}, \dots, s_{s.n}))}$$

$$\text{IN-ATOMS} \quad \frac{C, e \mapsto^* (col, s) \quad C, e_i \mapsto^* (col_i, s_i)|_{i=1, \dots, n}}{C, \text{IN-ATOMS}(res, [e], [e_1, \dots, e_n]) \mapsto^* (res, s \text{ IN } (s_1, \dots, s_n))}$$

7.1.2 Translating SQL Algebra Operators

The inference rules describing the translation from SQL Algebra operators into SQL code use the following notation:

$$\overline{CTE, \otimes \mapsto (C, F, W, CTE')}$$

The rule reads as follows: A list of common table expressions CTE (containing mappings $name \rightarrow \text{SFW}^1$ of table name $name$ to a `SELECT FROM WHERE` block SFW^1) and a SQL Algebra operator \otimes are translated (\mapsto) into a 4-tuple containing (1) a column list C (containing mappings $col \rightarrow s$ with col being a column name and s being a SQL expression), (2) a from list F (containing mappings $alias \rightarrow name|\text{VALUES}|\text{SFW}^1$ of a table name alias $alias$ to a table name $name$ or a SQL `VALUES` expression or

¹can also be a `UNION` or `EXCEPT` block: $\text{SFW}_1 \text{ UNION|EXCEPT } \text{SFW}_2$

an **SELECT FROM WHERE** block SFW^1), (3) a (semantically conjunctive) where list W containing SQL expressions and (4) a potentially expanded list CTE' .

Generally, the list CTE can be thought of as a global list which is passed through the plan in order to collect all common table expressions that are built.

In the inference rules now being introduced, x and y denote SQL Algebra operators and col and $cname$ denote column names in both SQL Algebra and SQL. We further use e to denote SQL Algebra expressions and s to denote SQL expressions. We use symbol $++$ to denote the concatenation of lists.

Translating the **TABLE** operator, which references a table in a database, means filling the column list with the columns of the table and assigning the table name an alias.

$$\begin{array}{c}
 \text{TABLE} \\
 \\
 \begin{array}{c}
 alias \equiv newAlias() \\
 s_i \equiv alias.cname_i|_{i=1,\dots,n}
 \end{array} \\
 \hline
 CTE, \text{TABLE}_{name,[col_1 \rightarrow cname_1, \dots, col_n \rightarrow cname_n]} \mapsto \\
 ([col_1 \rightarrow s_1, \dots, col_n \rightarrow s_n], [alias \rightarrow name], [], CTE)
 \end{array}$$

Translating the **LITERAL TABLE** operator involves filling the column list with the columns of the literal table and constructing a SQL **VALUES** statement, which is added to the from list under an alias. Tuples $tuple_i$ are assumed to be expanded in the SQL **VALUES** statement.

$$\begin{array}{c}
 \text{LITERAL TABLE} \\
 \\
 \begin{array}{c}
 alias \equiv newAlias() \\
 s_i \equiv alias.col_i|_{i=1,\dots,n} \\
 \text{VALUES} ((tuple_1) \\
 values \equiv \quad \quad \quad , \dots , \\
 \quad \quad \quad (tuple_m))
 \end{array} \\
 \hline
 CTE, \text{LITERAL TABLE}_{[col_1, \dots, col_n][tuple_1, \dots, tuple_m]} \mapsto \\
 ([col_1 \rightarrow s_1, \dots, col_n \rightarrow s_n], [alias \rightarrow values], [], CTE)
 \end{array}$$

Translating the **SELECT** operator, we translate its SQL Algebra expressions to SQL expressions and add them to the where list.

$$\begin{array}{c}
 \text{SELECT} \\
 \\
 CTE, x \mapsto (C, F, W, CTE') \\
 C, e_i \xrightarrow{*} (col_i, s_i)|_{i=1,\dots,n} \\
 \hline
 CTE, \text{SELECT}_{e_1, \dots, e_n}(x) \mapsto (C, F, W ++ [s_1, \dots, s_n], CTE')
 \end{array}$$

Translating a **PROJECT** operator with Boolean flag **distinct** we first translate the SQL Algebra expressions in its semantical content into SQL expressions. These constitute a new column list while the remaining lists stay unchanged. Then, a bind is performed. We observe that bind function $\mapsto \circ$ consumes a column list, a from list and a where list and generally builds a **SELECT FROM WHERE** block using these lists. The binding function is presented in more detail in Chapter 7.2.

PROJECT

$$\begin{array}{c}
CTE, x \mapsto (C, F, W, CTE') \\
C, e_i \xrightarrow{*} (col_i, s_i)|_{i=1, \dots, n} \\
\hline
([col_1 \rightarrow s_1, \dots, col_n \rightarrow s_n], F, W, CTE', \mathbf{distinct}) \mapsto (C', F', [], CTE'') \\
\hline
CTE, \mathbf{PROJECT}_{e_1, \dots, e_n, \mathbf{distinct}}(x) \mapsto (C', F', [], CTE'')
\end{array}$$

Translating the **GROUPBY** operator makes binding its child operator necessary. Here, the binding is not performed by the bind function (\mapsto) but by the rule itself, which is because the bind rule does not take SQL **GROUP BY** clauses into account. With the semantical content of the **GROUPBY** operator, a new column list as well as a groupby list are built. The from list and the where list stay unchanged. All these lists build an **SELECT FROM WHERE GROUP BY** block that is added to the list of common table expressions. Note that the implementation not always puts the block into the list of common table expressions: It also considers putting it into the from list, if appropriate.

GROUPBY

$$\begin{array}{c}
CTE, x \mapsto \left(C, [(alias_1 \rightarrow name_1), \dots, (alias_k \rightarrow name_k)], \right. \\
\left. [s_{w-1}, \dots, s_{w-l}], CTE' \right) \\
C, e_{c-i} \xrightarrow{*} (col_{c-i}, s_{c-i})|_{i=1, \dots, n} \\
C, e_{p-i} \xrightarrow{*} (col_{p-i}, s_{p-i})|_{i=1, \dots, m} \\
\mathbf{SELECT} s_{c-1} \mathbf{AS} col_{c-1}, \dots, s_{c-n} \mathbf{AS} col_{c-n} \\
\mathbf{FROM} name_1 \mathbf{AS} alias_1, \dots, name_k \mathbf{AS} alias_k \\
q \equiv \mathbf{WHERE} s_{w-1} \mathbf{AND}, \dots, \mathbf{AND} s_{w-l} \\
\mathbf{GROUP BY} s_{p-1}, \dots, s_{p-m} \\
alias \equiv newAlias() \\
name \equiv newName() \\
\hline
CTE, \mathbf{GROUPBY}_{[e_{c-1}, \dots, e_{c-n}], [e_{p-1}, \dots, e_{p-m}]}(x) \mapsto \\
([col_{c-1} \rightarrow alias.col_{c-1}, \dots, col_{c-n} \rightarrow alias.col_{c-n}], [alias \rightarrow name], \\
[], CTE' ++ [name \rightarrow q])
\end{array}$$

Set operators **UNION** and **EXCEPT** demand that both input children have aligned schemas. Therefore, both children are bound and their resulting **SELECT FROM WHERE** blocks are used in a **UNION** or **EXCEPT** statement. This statement is then added to the from list under a new alias. Note that the implementation not always adds the statement to the from list: It also considers adding it to the list of common table expressions, if appropriate. The remaining rules will again consistently add newly built **SELECT FROM WHERE** blocks in the list of common table expressions, this being the only rule showing what it formally looks like to add a **SELECT FROM WHERE** block¹ to the from list.

¹in this rule, actually an **SFW₁ UNION|EXCEPT SFW₂** block.

$$\begin{array}{l}
 \text{UNION, EXCEPT} \\
 (\odot, \oslash) \in \{(\text{EXCEPT}, \text{EXCEPT}), (\text{UNION}, \text{UNION})\} \\
 CTE, x \mapsto (C_x, F_x, W_x, CTE') \\
 (C_x, F_x, W_x, CTE', \text{false}) \mapsto \left(\begin{array}{l} [col_1 \rightarrow -, \dots, col_n \rightarrow -], \\ [alias_x \rightarrow name_x], [], CTE'' \end{array} \right) \\
 CTE'', y \mapsto (C_y, F_y, W_y, CTE''') \\
 (C_y, F_y, W_y, CTE''', \text{false}) \mapsto \left(\begin{array}{l} [col_1 \rightarrow -, \dots, col_n \rightarrow -], \\ [alias_y \rightarrow name_y], [], CTE'''' \end{array} \right) \\
 alias \equiv newAlias() \\
 \hline
 CTE, \odot(x, y) \mapsto ([col_1 \rightarrow alias.col_1, \dots, col_n, alias.col_n], \\
 [alias \rightarrow name_x \oslash name_y], [], CTE''''')
 \end{array}$$

Translating the CROSS operator, involves simply appending the column lists, the from lists and the where lists of the child operators. See Chapter 7.3 for a peculiarity of SQL code that we have to respect when translating CROSS operators this way.

$$\begin{array}{l}
 \text{CROSS} \\
 CTE, x \mapsto (C_x, F_x, W_x, CTE') \\
 CTE', y \mapsto (C_y, F_y, W_y, CTE'') \\
 \hline
 CTE, \text{CROSS}(x, y) \mapsto \\
 (C_x ++ C_y, F_x ++ F_y, W_x ++ W_y, CTE'')
 \end{array}$$

Translating the JOIN operator is similar to translating the CROSS operator, *i.e.* appending all lists. Additionally the SQL Algebra expressions contained in the SQL Algebra JOIN operator have to be translated into SQL expressions and added to the where list.

$$\begin{array}{l}
 \text{JOIN} \\
 CTE, x \mapsto (C_x, F_x, W_x, CTE') \\
 CTE', y \mapsto (C_y, F_y, W_y, CTE'') \\
 C_x ++ C_y, e_i \xrightarrow{*} (col_i, s_i) |_{i=1, \dots, n} \\
 \hline
 CTE, \text{JOIN}_{[e_1, \dots, e_n]}(x, y) \mapsto \\
 (C_x ++ C_y, F_x ++ F_y, W_x ++ W_y ++ [s_1, \dots, s_n], CTE'')
 \end{array}$$

Translating the ANTIJOIN operator into a NOT IN statement takes place when the anti-join predicates consist exclusively of equality comparison functions. From these comparisons, two column lists are derived for the NOT IN.

$$\begin{array}{l}
 \text{ANTIJOIN1} \\
 CTE, x \mapsto (C_x, F_x, W_x, CTE') \\
 CTE', y \mapsto (C_y, [(alias_1 \rightarrow name_1), \dots, (alias_m \rightarrow name_m)], [s_{w.1}, \dots, s_{w.k}], CTE'') \\
 e_i \equiv \text{FUNCTION}_{\text{EQUAL}}(-, [e_{l.i}, e_{r.i}]) \Big| \\
 C_x, e_{l.i} \xrightarrow{*} (col_{l.i}, s_{l.i}) \\
 C_y, e_{r.i} \xrightarrow{*} (col_{r.i}, s_{r.i}) \Big|_{i=1, \dots, n} \\
 s \equiv s_{l.1}, \dots, s_{l.n} \text{ NOT IN (SELECT } s_{r.1} \text{ AS } col_{r.1}, \dots, s_{r.n} \text{ AS } col_{r.n} \\
 \text{FROM } name_1 \text{ AS } alias_1, \dots, name_m \text{ AS } alias_m \\
 \text{WHERE } s_{w.1} \text{ AND, } \dots, \text{ AND } s_{w.k}) \\
 \hline
 CTE, \text{ANTIJOIN}_{[e_1, \dots, e_n]}(x, y) \mapsto (C_x, F_x, W_x ++ [s], CTE'')
 \end{array}$$

Translating the ANTIJOIN operator into a NOT EXISTS statement adds the anti-join predicates into the where list of the subquery. At this point, correlation is introduced. Because of the existential semantics of NOT EXISTS we can select arbitrary columns in the subquery.

ANTIJOIN2

$$\begin{array}{l}
 CTE, x \mapsto (C_x, F_x, W_x, CTE') \\
 CTE', y \mapsto (C_y, [(alias_1 \rightarrow name_1), \dots, (alias_m \rightarrow name_m)], [s_{w.1}, \dots, s_{w.k}], CTE'') \\
 C_x ++ C_y, e_i \xrightarrow{*} (col_i, s_i) |_{i=1, \dots, n} \\
 s \equiv \text{NOT EXISTS (SELECT *} \\
 \quad \text{FROM } name_1 \text{ AS } alias_1, \dots, name_m \text{ AS } alias_m \\
 \quad \text{WHERE } s_{w.1} \text{ AND, } \dots, \text{ AND } s_{w.k} \\
 \quad \text{AND } s_1 \text{ AND, } \dots, \text{ AND } s_n) \\
 \hline
 CTE, \text{ANTIJOIN}_{[e_1, \dots, e_n]}(x, y) \mapsto (C_x, F_x, W_x ++ [s], CTE'')
 \end{array}$$

Translating the SQL Algebra SERIALIZE operator builds a SELECT FROM WHERE ORDER BY block after printing all common table expressions. The ordering information (ASC, DESC) for the ORDER BY clause is assumed to be contained in SQL expressions s_{iter} , $s_{order.i}$ implicitly. For simplicity reasons this information is not reflected in the rule.

SERIALIZE

$$\begin{array}{l}
 CTE, x \mapsto (C, F, W, CTE') \\
 C, iter \xrightarrow{*} (col_{iter}, s_{iter}) \\
 C, order_i \xrightarrow{*} (col_{order.i}, s_{order.i}) |_{i=1, \dots, n} \\
 C, item_i \xrightarrow{*} (col_{item.i}, s_{item.i}) |_{i=1, \dots, m} \\
 F \equiv [(alias_1 \rightarrow name_{f.1}), \dots, (alias_k \rightarrow name_{f.k})] \\
 W \equiv [s_{w.1}, \dots, s_{w.l}] \\
 CTE' \equiv [name_{c.1} \rightarrow entry_1, \dots, name_{c.r} \rightarrow entry_r] \\
 \hline
 CTE, \text{SERIALIZE}_{iter, [order_1, \dots, order_n], [item_1, \dots, item_m]}(x) \xrightarrow{SER} \\
 \text{WITH } name_{c.1} \text{ AS } (entry_1), \dots, name_{c.r} \text{ AS } (entry_r) \\
 \text{SELECT } s_{iter} \text{ AS } col_{iter}, s_{order.1} \text{ AS } col_{order.1}, \dots, s_{order.n} \text{ AS } col_{order.n}, \\
 \quad s_{item.1} \text{ AS } col_{item.1}, \dots, s_{item.m} \text{ AS } col_{item.m} \\
 \text{FROM } name_{f.1} \text{ AS } alias_1, \dots, name_{f.k} \text{ AS } alias_k \\
 \text{WHERE } s_{w.1} \text{ AND, } \dots, \text{ AND } s_{w.l} \\
 \text{ORDER BY } s_{iter}, s_{order.1}, \dots, s_{order.m}
 \end{array}$$

7.2 Binding

We now introduce the bind function formally. Note that it adds the constructed SELECT FROM WHERE block to the common table expressions list. The implementation of this function additionally considers adding the SELECT FROM WHERE block to the from list if the operator was referred to only once. The bind function is of the form

$$\overline{(C, F, W, CTE, \text{distinct}) \mapsto (C', F', [], CTE')}$$

and reads as follows: A list of (1) a column list C (containing mappings $col \rightarrow s$ with col being a column name and s being a SQL expression), (2) a from list F (containing mappings $alias \rightarrow name$ |VALUES|SFW¹ of a table name $alias$ to a table name $name$ or a SQL VALUES expression or a SELECT FROM WHERE block¹), (3) a (semantically conjunctive) where list W containing SQL expressions, (4) a list of common table expressions and (5) a Boolean flag **distinct** are translated (\mapsto) into a 4-tuple of (1) an altered column list C' , (2) a from list F' (containing a single mapping from a new table name $alias$ to a new table name $name$), (3) an empty where list $[]$ and (4) a list CTE' (which is list CTE expanded by a new mapping from the new table name $name$ to the built SELECT FROM WHERE block).

BIND

$$\begin{array}{l} C \equiv [col_1 \rightarrow s_{c.1}, \dots, col_n \rightarrow s_{c.n}] \\ F \equiv [alias_1 \rightarrow name_1, \dots, alias_k \rightarrow name_k] \\ W \equiv [s_{w.1}, \dots, s_{w.l}] \\ (\text{distinct}, \delta) \in \{(\text{true}, \text{DISTINCT}), (\text{false}, \epsilon)\} \\ \text{SELECT } \delta \text{ } s_{c.1} \text{ AS } col_1, \dots, s_{c.n} \text{ AS } col_n \\ q \equiv \text{FROM } name_1 \text{ AS } alias_1, \dots, name_k \text{ AS } alias_k \\ \text{WHERE } s_{w.1} \text{ AND } \dots \text{ AND } s_{w.l} \\ \quad alias \equiv \text{newAlias}() \\ \quad name \equiv \text{newName}() \\ \hline (C, F, W, CTE, \text{distinct}) \mapsto \\ ([col_1 \rightarrow alias.col_1, \dots, col_n \rightarrow alias.col_n], [alias \rightarrow name], [], \\ CTE++[name \rightarrow q]) \end{array}$$

The following meta rule performs a binding when an operator is referred to more than once. The binding function \mapsto adds a newly constructed SELECT FROM WHERE block to the common table expressions list, which is the only proper action to take when encountering operators with more than one reference. We use $d(x)$ to denote the number of references to SQL Algebra operator x .

TRIGGER-BIND

$$\begin{array}{l} d(x) > 1 \\ CTE, x \mapsto (C, F, W, CTE') \\ (C, F, W, CTE', \text{false}) \mapsto (C', F', W', CTE'') \\ \hline x \mapsto (C', F', W', CTE'') \end{array}$$

¹can also be a UNION or EXCEPT block: SFW₁ UNION|EXCEPT SFW₂

7.3 Aliases

Although translation has so far been straightforward and the SQL Algebra is already close to SQL code, we have to consider that it is not translatable in an one to one fashion. As an example, an one to one translation of a self-cross would lead to the following SQL code:

```
SELECT *
FROM Parts AS alias1, Parts AS alias1
```

This situation arises because SQL Algebra operators get an alias assigned when being translated the first time. When encountered the second time, the result of the first translation and with it the first alias are used again. The solution is to change the alias of an operator already translated every time it is re-encountered. This procedure is again presented in the form of inference rules that are applied when traversing the DAG bottom-up.

We first introduce the RENAME rule that simply replaces the alias of a SQL expression $alias.col$ with a new alias from a mapping list containing pairs of new and old aliases.

RENAME

$$\frac{}{[\dots, (alias, newalias), \dots], alias.col \xrightarrow{R} newalias.col}$$

Rule SUBSTITUTE-ALIASES performs the substitution of aliases. It consumes a 3-tuple containing a column list, a from list and a where list and first constructs a list of mappings from old to new aliases. This list is then used to substitute all aliases contained in SQL column expressions. Aliases in the from list are also replaced.

SUBSTITUTE-ALIASES

$$\frac{\begin{array}{l} newalias_i = newAlias()|_{i=1,\dots,m} \\ map \equiv [(alias_1, newalias_1), \dots, (alias_m, newalias_m)] \\ map, s_i \xrightarrow{R} s'_i|_{i=1,\dots,n} \\ map, s_{wh.i} \xrightarrow{R} s'_{wh.i}|_{i=1,\dots,k} \end{array}}{\left(\begin{array}{l} [(col_1 \rightarrow s_1), \dots, (col_n \rightarrow s_n)], \\ [alias_1 \rightarrow name_1, \dots, alias_m \rightarrow name_m], \\ [s_{wh.1}, \dots, s_{wh.k}] \end{array} \right) \xrightarrow{S} \left(\begin{array}{l} [(col_1 \rightarrow s'_1), \dots, (col_n \rightarrow s'_n)], \\ [newalias_1 \rightarrow name_1, \dots, newalias_m \rightarrow name_m], \\ [s'_{wh.1}, \dots, s'_{wh.k}] \end{array} \right)}$$

Meta rule TRIGGER-SUBST intends to trigger the substitution of aliases whenever necessary, *i.e.* when a SQL Algebra operator x is already translated and then re-encountered in the bottom-up traversal of the DAG shaped plan. $ann(x) \equiv$

(C, F, W, CTE) means that SQL Algebra operator x was previously translated into (C, F, W, CTE) and that this translation result is annotated to it, which is expressed by $ann(x)$.

$$\begin{array}{c}
 \text{TRIGGER-SUBST} \\
 ann(x) \equiv (C, F, W, CTE) \\
 (C, F, W) \xrightarrow{s} (C', F', W') \\
 ann'(x) \equiv (C', F', W', CTE) \\
 \hline
 ann(x) \mapsto ann'(x)
 \end{array}$$

7.4 Translation Example

In Figure 7.1 we present an example translation of the Logical Algebra plan in Figure 3.1: The SQL Algebra operators have portions of SQL code annotated in a column list C , a from list F , a where list W and a list of common table expressions CTE as known from the inference rules in Chapter 7.1. In order to save space, we abbreviate COLUMN to COL, FUNCTION to FUN and EQUAL to EQ. Further, we ignore ordering information in the SQL Algebra COLUMN expressions COL_{price} .

The PROJECT operators have two annotations, the lower one resulting from the actual translation and the upper one resulting from the bind that is performed when encountering a SQL Algebra PROJECT operator. We ignored the SQL Algebra SERIALIZE operator and only state the resulting query (marked by ③) produced by it.

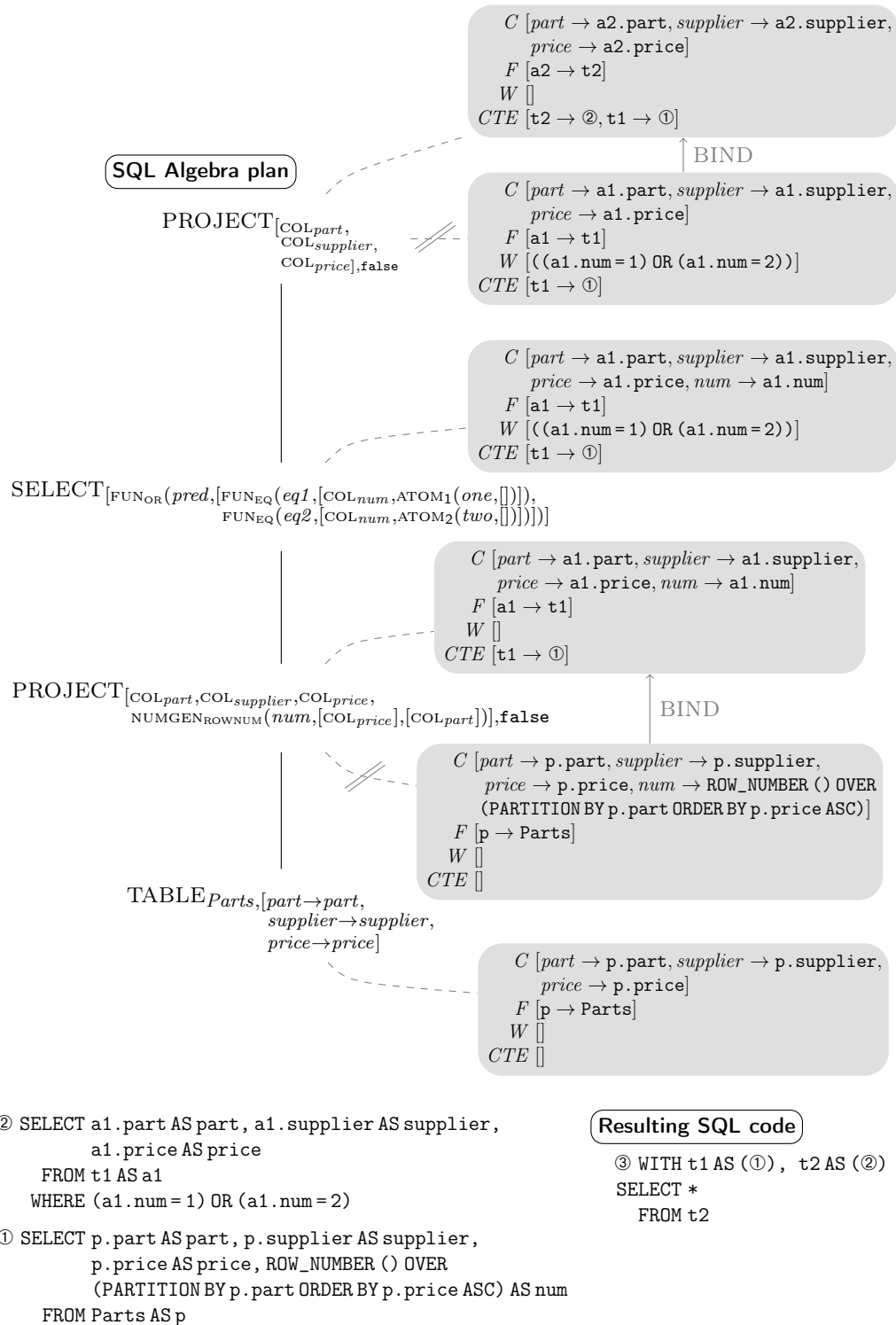


Figure 7.1: SQL Algebra to SQL translation example.

8 Experiments

Logical Algebra plans have so far been supposed to be derived from non-relational source languages. With the SQL to Logical Algebra compiler presented in [5], we are able to compile a SQL query to a Logical Algebra plan and from there to SQL again, using Pathfinder’s SQL code generator. Comparing the SQL query emitted from Pathfinder with the original version of the SQL query, we can assess what impact the Pathfinder compilation chain has on the running time of the queries. The compilation chain is depicted in Figure 8.1: First, the original SQL query SQL_{org} is compiled to Logical Algebra. Then, Pathfinder performs optimisation on the Logical Algebra plan and translates it into SQL_{old} . We call this path in the compilation chain the old SQL code generator.

In this thesis we introduced the SQL Algebra as an intermediate algebra. We compile the optimised Logical Algebra plan to SQL Algebra, perform further optimisations and translate it into SQL_{new} . We call this path in the compilation chain the new SQL code generator.

To examine the queries emitted from the new SQL code generator we use the following experimental setup: From SQL queries we compile SQL queries with (1) the old SQL code generator and (2) the new SQL code generator. The running times of the queries emitted from the old SQL code generator and the new SQL code generator are compared to each other.

As in [5] we use the SQL queries of the TPC benchmark in order to get comparable results. TPC-H is a widely used decision support benchmark that claims to have a broad practical relevance regarding the data it contains and also regarding the queries it runs on this data. TPC-H database instances contain – briefly summarised – data of

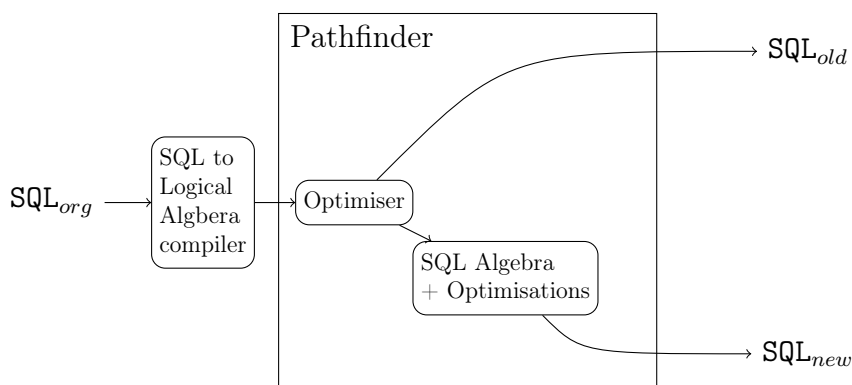


Figure 8.1: Compilation chain. SQL denotes SQL query code.

an scenario of customers that place orders for parts that come from different suppliers. The 22 SQL queries that TPC-H provides are of high complexity and examine large amounts of data.

The experiments focus on the comparison of the running times of the SQL_{old} and SQL_{new} queries, the running time of the SQL_{old} queries being the reference point. We do not aim to compare different DBMSs to each other but to compare the running times of the two versions of emitted SQL code. Additionally we measured the running time of the original SQL queries as provided by TPC-H in order to asses the impact of the different compilations performed by Pathfinder on the running times of the queries.

We used standard TPC-H instances on IBM's DB2 V9 and Postgres 8.4 employing the schema and data generation facilities provided from TPC. The scale factors of the database instances were 0.01, 0.1 and 1, equalling database sizes of 10MB, 100MB and 1GB, respectively. Regarding the indexes we followed the advice from DB2's `db2advise` given the workload of all 22 SQL queries in the versions SQL_{org} , SQL_{old} and SQL_{new} . The advised indexes were created in the database instances on both DB2 and Postgres.

All queries were executed once in order to warm the cache before measuring the average time of 10 runs. To measure time we used the built-in timer mechanisms of the DBMSs. All experiments were performed on a Sun Fire X4275 server with 16 2.93GHz Intel XeonTM CPUs and 70GB of main memory, SCSI disc memory, running a Linux 2.6 kernel. In what follows, we present the results of DB2 and Postgres.

8.1 DB2

Table 8.1 shows the execution times for the TPC-H queries on different TPC-H instances running on IBM's DB2 V9.

For most of the queries, the SQL_{old} and SQL_{new} versions showed equal running times, *i.e.* running times that differ by not more than 10%.

Queries whose execution times differ considerably are *e.g.* Q_{16} and Q_{21} , showing a vast performance improvement in the SQL_{new} version. These queries contain `NOT IN` and `NOT EXISTS` statements that result basically in an `EXCEPT` operator in SQL Algebra. The anti-join optimisation described in Chapter 6.3 replaces this `EXCEPT` operator with a SQL Algebra `ANTIJOIN` operator that is translated into a `NOT IN` or `NOT EXISTS` statement. The performance improvement is visible on all database instances.

The SQL_{new} version of Q_{15} shows a considerably worse execution time than in the SQL_{old} version on the scale factor 0.01 database instance. The reason for this is the `CREATEVIEW` statement that the query contains and whose execution time is also measured. On scale factor 0.1 instances, both versions exhibit equal execution times.

Q_{22} runs faster in the SQL_{new} version on the scale factor 0.01 and scale factor 0.1 instances but experiences significant performance losses compared to the SQL_{old} ver-

	SF 0.01			SF 0.1			SF 1		
	SQL _{org}	SQL _{old}	SQL _{new}	SQL _{org}	SQL _{old}	SQL _{new}	SQL _{org}	SQL _{old}	SQL _{new}
Q_1	0.318	0.249	0.274	2.58	2.56	2.57	40.1	39.9	39.4
Q_2	0.006	DNF	DNF	0.15	DNF	DNF	1.9	DNF	DNF
Q_3	0.049	0.046	0.046	2.45	2.22	2.27	43.0	43.3	44.1
Q_4	0.021	0.028	0.028	0.14	0.14	0.13	16.1	16.3	16.4
Q_5	0.039	0.040	0.039	10.97	10.97	10.96	3.5	3.5	3.5
Q_6	0.013	0.010	0.010	0.12	0.12	0.13	1.0	1.0	1.0
Q_7	0.038	0.037	0.035	1.09	1.07	1.06	11.7	11.8	11.8
Q_8	0.069	0.111	0.108	18.64	19.98	20.01	84.2	88.0	88.1
Q_9	0.117	0.254	0.228	2.75	2.83	2.85	33.4	33.2	33.5
Q_{10}	0.071	0.033	0.034	0.48	0.47	0.46	6.7	6.7	6.7
Q_{11}	0.003	0.229	0.241	0.03	38.52	39.21	0.5	DNF	DNF
Q_{12}	0.023	DNF	DNF	0.18	DNF	DNF	4.5	DNF	DNF
Q_{13}	0.029	0.015	0.015	0.29	0.29	0.30	3.5	3.4	3.5
Q_{14}	0.006	0.010	0.006	0.03	0.07	0.07	0.4	0.7	0.8
Q_{15}	0.075	0.062	0.143	0.11	5.55	5.85	2.0	DNF	DNF
Q_{16}	0.011	57.361	0.013	0.17	DNF	0.10	0.6	DNF	0.6
Q_{17}	0.000	DNF	DNF	0.03	DNF	DNF	1.2	DNF	DNF
Q_{18}	0.032	DNF	DNF	0.36	DNF	DNF	4.4	DNF	DNF
Q_{19}	0.010	DNF	DNF	1.12	DNF	DNF	2.9	DNF	DNF
Q_{20}	0.003	DNF	DNF	0.04	DNF	DNF	4.0	DNF	DNF
Q_{21}	0.095	DNF	0.077	3.91	DNF	3.63	31.9	DNF	32.1
Q_{22}	0.006	0.097	0.061	0.02	0.83	0.66	0.2	6.9	9.2

Table 8.1: IBM DB2 V9 experiments. Average running times in seconds of 10 runs after one cache-warming run. DNF means that the query did not finish within 5 minutes. SF indicates the scale factor of the database instance. Time pairs of interest are marked bold.

sion on the scale factor 1 instance. This query also contains a `NOT EXISTS` statement that leads to an `EXCEPT` operator in the SQL Algebra plan. This `EXCEPT` operator cannot be replaced by a SQL Algebra `ANTIJOIN` operator using the anti-join optimisation described in Chapter 6.3 because a necessary condition is not satisfied (key column condition). Refining the compilation from SQL to Logical Algebra can enable the new SQL code generator to apply the anti-join optimisation and give the `SQLnew` version of the query a performance advantage.

Comparing the execution times of original SQL code against the execution times of SQL code from Pathfinder's SQL code generators, we generally observe performance losses. Here, further research on optimisations at the SQL Algebra level can help to ameliorate the running times towards the running times of the original queries. We sketch possible future optimisations in Chapter 9.

8.2 Postgres

Table 8.2 presents the execution times of the TPC-H queries on TPC-H instances running on Postgres 8.4.

On Postgres, a comparatively large part of the queries ran faster in the `SQLnew` version. These queries are Q_4 , Q_6 , Q_9 , Q_{14} and Q_{22} , showing improvements mainly because the new SQL code generator produces fewer common table expressions: An `SELECT FROM WHERE` block is only bound to a common table expression if it is referenced more than once. Otherwise it is used as a nested subquery. The old SQL code generator, on the other hand, binds every `SELECT FROM WHERE` block to a common table expression. Postgres optimises per common table expression and thus finds available more information to base optimisation decisions on in the `SQLnew` version. All `SQLnew` queries that showed performance gains contain on average 40% fewer common table expression bindings than the corresponding `SQLold` query. Query Q_{16} benefits from the anti-join optimisation described in Chapter 6.3.

Q_8 exhibits a significant improvement in the `SQLnew` version, which results from the fact that the old SQL code generator binds a join of all 8 TPC-H tables in three different common table expressions, whereas the new SQL code generator binds the join only in one common table expression. The Postgres optimiser seems incapable of detecting the equivalence of the common table expressions, whereas the DB2 optimiser seems to detect the equivalence, showing equal running times for the `SQLold` and `SQLnew` version of the query.

Q_7 in the `SQLnew` version shows a performance loss against the `SQLold` version of the query. This is because the joins of the `SQLorg` version are not translated into SQL Algebra `JOIN` operators but instead to combinations of `CROSS` operators and `SELECT` operators. The translation from SQL Algebra into SQL results a join of 6 tables to be separated from its predicates in an subquery. Here, the Postgres optimiser fails to perform a selection pushdown of the join predicates into the subquery: Inserting the selection pushdown manually in the `SQLnew` version of the query results in equal

execution times of SQL_{old} and SQL_{new} . The problem arises because of the restrictive binding behaviour regarding the children of SQL Algebra CROSS operators of the translation from Logical Algebra into SQL Algebra. A possible optimisation for this situation is described in Chapter 9.

The remaining queries show running times that differ by not more than 10%.

Again, Pathfinder's emitted SQL code runs slower than the original SQL code. See Chapter 9 for possible optimisations to improve the running time of Pathfinder's SQL code.

	SF 0.01			SF 0.1			SF 1		
	SQL _{org}	SQL _{old}	SQL _{new}	SQL _{org}	SQL _{old}	SQL _{new}	SQL _{org}	SQL _{old}	SQL _{new}
Q ₁	0.288	0.320	0.304	2.47	2.50	2.47	24.2	24.4	24.3
Q ₂	0.010	DNF	DNF	0.06	DNF	DNF	0.6	DNF	DNF
Q ₃	0.040	0.042	0.040	0.27	0.32	0.29	2.9	2.9	2.9
Q ₄	0.056	0.123	0.054	0.32	1.44	0.36	2.9	17.5	4.9
Q ₅	0.020	0.021	0.021	0.18	0.18	0.19	2.5	2.5	2.5
Q ₆	0.016	0.030	0.022	0.15	0.21	0.16	2.5	3.1	2.6
Q ₇	0.018	0.017	2.125	0.16	0.21	DNF	3.5	3.5	DNF
Q ₈	0.024	0.052	0.028	0.22	0.57	0.23	1.0	2.9	1.5
Q ₉	0.089	0.117	0.072	0.80	0.79	0.47	12.4	10.6	7.0
Q ₁₀	0.032	0.033	0.031	0.39	0.39	0.38	5.7	5.8	5.7
Q ₁₁	0.008	0.170	0.292	0.04	11.13	6.76	0.4	DNF	DNF
Q ₁₂	0.027	DNF	DNF	0.17	DNF	DNF	2.9	DNF	DNF
Q ₁₃	0.027	0.027	0.025	0.21	0.22	0.20	2.7	2.7	2.7
Q ₁₄	0.010	0.024	0.022	0.09	0.23	0.19	0.9	2.9	1.9
Q ₁₅	0.021	DNF	DNF	0.16	DNF	DNF	3.0	DNF	DNF
Q ₁₆	0.013	111.741	18.022	0.07	DNF	DNF	0.8	DNF	DNF
Q ₁₇	0.004	DNF	DNF	0.01	DNF	DNF	0.1	DNF	DNF
Q ₁₈	0.070	DNF	DNF	0.43	DNF	DNF	7.0	DNF	DNF
Q ₁₉	0.009	DNF	DNF	0.02	DNF	DNF	0.2	DNF	DNF
Q ₂₀	0.008	DNF	237.262	0.03	DNF	DNF	0.2	DNF	DNF
Q ₂₁	0.012	DNF	DNF	0.54	DNF	DNF	5.1	DNF	DNF
Q ₂₂	0.008	0.064	0.047	0.05	0.40	0.34	0.4	6.2	4.3

Table 8.2: Postgres 8.4 experiments. Average running times in seconds of 10 runs after one cache-warming run. DNF means that the query did not finish within 5 minutes. SF indicates the scale factor of the database instance. Time pairs of interest are marked bold.

9 Conclusions

Pathfinder so far derived SQL code directly from Logical Algebra plans. In this thesis we proposed an intermediate language to Pathfinder’s SQL generation: The SQL Algebra. The translation from the Logical Algebra into the SQL Algebra as well as the translation from the SQL Algebra into SQL was described. Additionally we presented the SQL Algebra as a new level to perform optimisations on. These optimisations take into account properties of SQL.

With the SQL Algebra we took a step towards a SQL-like algebra: The distinction between operators and expression reflects the structure of a SQL query more precisely than the Logical Algebra which only consists of operators. Therefore, SQL Algebra plans indicate more clearly what the corresponding SQL query looks like.

Additionally, the SQL Algebra provides a basis to build further algebras to represent queries of languages that also consist of operators and expressions. One example is the X100 Algebra [1].

The performance improvements observed for Q_{16} and Q_{21} in the SQL_{new} versions provide evidence that expressing SQL language constructs with semantically equivalent but yet different algebra operators introduces overhead to the SQL code derived from plans over this algebra. Injecting such overhead can in some cases be avoided by aligning the internal representation algebra with the source language: The SQL code gained from the SQL Algebra plans for Q_{16} and Q_{21} contains `NOT IN` and `NOT EXISTS` statements like the original queries.

In other cases, it is exactly the approach to replace operators with equivalents that leads to performance improvements: In Logical Algebra, the left child of a difference operator does never contain duplicate tuples. Therefore, it can be expressed in SQL as `EXCEPT ALL`, telling the optimiser of the DBMS to not care about duplicate removal.

In summary, the presented approach of SQL code generation achieves the emphasised aspects, but also leaves space for further optimisations. We present possible future optimisations in the following.

9.1 Outlook

A portion of the queries emitted from Pathfinder did not finish within 5 minutes in both the SQL_{old} and the SQL_{new} version¹. We examined some of these queries and discovered optimisation potential.

¹See Tables 8.1 and 8.2

In the SQL Algebra plan for Q_{12} , a CROSS operator is directly followed by a PROJECT operator containing a SQL Algebra number generating expression of kind ROWID. This prevents the optimiser of the DBMS to push down selection predicates in order to downsize the result of the CROSS operator before generating ROWID numbers. A possible optimisation to resolve such situations is to switch a SQL Algebra PROJECT operator that is referenced only once and that contains a SQL Algebra number generating expression of kind ROWID with a SQL Algebra SELECT operator that follows.

Another aspect of the SQL code emitted from the new SQL code generator that harms performance is the number of bindings. The new SQL code generator does produce fewer common table expression than the old SQL code generator, but generally performs twice as much bindings, *i.e.* SELECT FROM WHERE blocks. This not only affects the readability but also introduces obstacles to optimisers that need to analyse which projections are in fact needed. Restricting the number of bindings to a minimum should lead to performance improvements. To start with the reduction of the number of bindings, the binding behaviour regarding the children of CROSS and JOIN operators of the translation from Logical Algebra into SQL Algebra needs to be changed: Being restrictive because of possible name conflicts, we can ignore name conflicts on algebra level since they are resolved when the different references to tables are given different aliases in SQL.

List of Figures

2.1	Number generation example	15
2.2	Table <i>Parts</i>	16
2.3	Example query in Logical Algebra	17
2.4	Example query result	17
3.1	Example query in SQL Algebra	20
4.1	SQL WITH clause syntax	24
4.2	Example SQL query using common table expressions	24
5.1	Logical Algebra to SQL Algebra translation example.	36
6.1	Merge projections example	39
6.4	Merge projections: Problematic ROWID expressions	41
6.5	Anti-join pattern	42
6.6	Anti-join pattern replaced by anti-join	43
7.1	SQL Algebra to SQL translation example.	55
8.1	SQL code compilation chain	57

List of Tables

2.1	Logical Algebra operators	14
3.1	SQL Algebra operators	21
3.2	SQL Algebra expressions	21
8.1	IBM DB2 V9 experiments	59
8.2	Postgres 8.4 experiments	62

Bibliography

- [1] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the 2005 CIDR Conference*, 2005.
- [2] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell Boards the Ferry. In *Proceedings of the 22nd Symposium on Implementation and Application of Functional Languages*, 2010.
- [3] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-Supported Program Execution. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, 2009.
- [4] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL hosts. In *Proceedings of the VLDB Endowment*, 2004.
- [5] Fabian Kliebhan. A Truly Compositional SQL Debugger. Diploma thesis, Universität Tübingen, Germany, 2010.
- [6] Manuel Mayr. A SQL:1999 Code Generator for Pathfinder (in German). Diploma thesis, Technische Universität München, Germany, 2007.
- [7] Jim Melton and Alan R. Simon. *SQL:1999: Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, 2002.
- [8] Tom Schreiber, Simone Bonetti, Torsten Grust, Manuel Mayr, and Jan Rittinger. Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider. In *Proceedings of the VLDB Endowment*, 2010.
- [9] Alexander Ulrich. A Ferry-Based Query Backend for the Links Programming Language. Diploma thesis, Universität Tübingen, Germany, 2011.