A new compositional approach to SQL compilation

Studienarbeit

Lehrstuhl für Datenbanken und Informationssysteme
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften
Universität Tübingen

von

Hans-Joachim Ruscheweyh

Betreuer:

Prof. Dr. Torsten Grust, Jan Rittinger

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Tübingen, den 14. April 2009

Contents

1 Introduction **5**

2 Relational Algebra **6**

2.1 Relational Instance - Table . . . 6
2.2 Operators . . . 6
2.2.1 Projection . . . 6
2.2.2 Selection . . . 6
2.2.3 Equijoin . . . 7
2.2.4 Attach . . . 7
2.2.5 RowID . . . 7
2.2.6 Cartesian Product . . . 8
2.2.7 Difference . . . 8
2.2.8 Union . . . 8
2.2.9 Intersection . . . 9
2.2.10 Comparisons and basic arithmetic operations . . . 9
2.2.11 Negation . . . 9
2.2.12 Distinct . . . 10

3 Translation **11**

3.1 Data Structures . . . 11
3.1.1 Data Environment . . . 11
3.1.2 Metadata loop . . . 12
3.2 Rules . . . 12
3.2.1 Table Inferencerules . . . 12
3.2.2 Column Inferencerules . . . 14
3.3 Translation in detail . . . 14
3.3.1 `SELECT...FROM...WHERE...` . . . 14
3.3.2 `FROM` . . . 16
3.3.3 `WHERE` . . . 17
3.3.4 `SELECT` . . . 18
3.3.5 Table . . . 18
3.3.6 Lookup . . . 19
3.3.7 Constant . . . 19
3.3.8 $\odot$ . . . 19
3.3.9 `NOT` . . . 20
3.3.10 `EXISTS` . . . 20
3.3.11 `SOME/ANY` . . . 21
3.3.12 `IN` . . . 22
3.3.13 `ALL` . . . 23
3.3.14 `SELECT DISTINCT` . . . 23
3.3.15 `e AS s` . . . 24
3.3.16 SERIALIZE . . . 25

4 Example Translation **26**

1 Introduction

SQL is the most widely used database language. Even though it may not be apperant on the first sight, its fundament is based on relational algebra introduced by Edgar F. Codd in the 1970s. In our approach we aim to translate SQL into another relational algebra following in many points the standard algebra but alter some operators. Section 2 presents and explains all operators needed for our translation.

In section 3 we will be introduced to the rules for the translation. Each rule is triggered by the appearence of SQL structure, like `EXISTS` calls the `EXISTS` rule. Our approach aims to be highly compositional. That means that we have for each construct appearing in SQL exactly one rule. For example the appearence of a '+' in `SELECT` calls the same rule as in `WHERE`. This advantages to observe any query expressions immediatly and the count of rules shrinks.

For a SQL query our expected outcome is a directed acyclic graph, DAG containing relational operators as nodes. However, this structure is highly accessible for optimization. A DAG resulting from our translation means also to be able to translate SQL to several other database languages able to 'speak' relational algebra fluenty.

In section 4 we will run the translation for a query with a join on two tables. We will see first how the translation works and second how a DAG looks after translating a query with our rules.

2 Relational Algebra

As the theoretical fundament for all modern query languages, the relational algebra is used in our approach to transform SQL queries into equivalent relational expressions. Since relational expressions are compositional, there will always be a relational expression matching the input query. The fact that these expressions can be composed from relational algebra operators easily is based on a simple idea. Every operator needs as input and as output relational instances e.g. tables. The table as the main instance is shifted from the innermost operator, modified by operators higher hierarchies and becomes the input for the last operator which carries the result of the whole expression. However we will be shown later this is just a naive view on the evaluation of a query.

2.1 Relational Instance - Table

The table is our representation of data. A table has a fixed number of columns which are addressed by their names. The rows, called tuples, hold data which must have the data type assigned to the respective column.

2.2 Operators

This section introduces the relational operators. They are derived from the well known standard operators and extended by several new ones. Alltogether they are sufficient for the translation from SQL queries to relational expressions.

These operators mainly work on algebraic plans. These can either consist of other relational operators or tables. We use R respectively S to abbreviate algebraic plans and tables.

2.2.1 Projection $\pi_{(l)}\ R$

Input of the *projection* is R and the list l. l carries columns of R, appearing in the result-table. In addition to the standard operator we allow renaming of columns. Renaming is realized by the use of ':'.

$$\pi_{(\mathsf{n:a,b})} \left(\begin{array}{c|c|c} \mathsf{a} & \mathsf{b} & \mathsf{c} \\ \hline\hline \vdots & \vdots & \vdots \end{array} \right) \equiv \left(\begin{array}{c|c} \mathsf{n} & \mathsf{b} \\ \hline\hline \vdots & \vdots \end{array} \right)$$

Figure 1: Projection

2.2.2 Selection $\sigma_{\mathsf{c}}\ R$

In order to keep just specific tuples we provide the unary operator *selection*. It is sufficient to test rows of column c on the boolean values $\mathsf{c} \in \{true, false\}$. Just rows containing *true* in c will appear in the result-table.

$\sigma_{(\mathsf{ITEM})}$

KEY	ITEM
1	true
2	false
3	false
4	true

$\equiv$

KEY	ITEM
1	true
4	true

Figure 2: Selection

2.2.3 Equijoin $R \bowtie_{(R.\mathsf{a}=S.\mathsf{a})} S$

In order to compare rows of two tables or plans the relational algebra provides the binary *equijoin*-operator. It receives R and S as input and a single equality predicate consisting of two arguments refering to columns of both, R and S. All other kinds of joins e.g. the theta-join are simulated in our translation by *equijoin* and a consecutive test carrying the predicate.

KEY	ITEM
1	10
2	20

$\bowtie_{(\mathsf{KEY}=\mathsf{KEY}')}$

KEY′	ITEM′
1	a
3	b

$\equiv$

KEY	ITEM	KEY′	ITEM′
1	10	1	a

Figure 3: Equijoin

2.2.4 Attach $@_{\mathsf{c}:\ v}R$

Operator @ attaches a new column c to R. The name of the column and a constant value v are provided as arguments to @.

$@_{(\mathsf{col}:'true')}$

a	b	c
2	3	a
5	1	b
6	2	c
7	2	r

$\equiv$

col	a	b	c
true	2	3	a
true	5	1	b
true	6	2	c
true	7	2	r

Figure 4: Attach

2.2.5 RowID $\#_{\mathsf{c}}\ R$

Operator # creates a new column c with an enumeration of unique values. Each row in R is then distinguishable by the value of c.

$\#_{(\text{KEY})}$

a	b	c
2	3	a
5	1	b
6	2	c
7	2	r

$\equiv$

KEY	a	b	c
1	2	3	a
2	5	1	b
3	6	2	c
4	7	2	r

Figure 5: RowID

2.2.6 Cartesian Product $R \times S$

For given R and S, $R \times S$ returns a table containing all columns of R and of S. The result of $R \times S$ is a table consisting of tuples of all combinations made up by $r \in R$ and $s \in S$.

a	b
2	3
7	2

$\times$

c	d
6	2
7	2

$\equiv$

a	b	c	d
2	3	6	2
2	3	7	2
7	2	6	2
7	2	7	2

Figure 6: Cartesian Product

2.2.7 Difference $R \backslash S$

$R \backslash S$ returns a new table that contains all tuples from R not appearing in S. The schema of R must be the identical to the schema of S. In contrary to the standard *difference*, no duplicate elimination is provided.

a	b	c
2	3	a
2	3	a
6	2	c
7	2	r

$\backslash$

a	b	c
2	3	a
5	1	b
10	5	c
7	2	r

$\equiv$

a	b	c
2	3	a
6	2	c

Figure 7: Difference

2.2.8 Union $R \cup S$

$R \cup S$ returns a table containing all tuples of R and of S. Same as for *difference* the schemas of R and S must be identical. Compared to the standard operator the operator doesn't eliminate duplicates.

a	b	c
2	3	a
2	3	a
6	2	c
7	2	r

$\cup$

a	b	c
2	3	a
5	1	b
10	5	c
7	2	r

$\equiv$

a	b	c
2	3	a
2	3	a
6	2	c
7	2	r
2	3	a
5	1	b
10	5	c
7	2	r

Figure 8: Union

2.2.9 Intersection $R \cap S$

$R \cap S$ returns a table containing all tuples occuring in R and in S. Both schemas must be identical. Again no duplicate elimination is provided. As we will see later *intersection* is used only to simulate equijoins on multiple columns.

a	b	c
2	3	a
2	3	a
6	2	c
7	2	r

$\cap$

a	b	c
2	3	a
5	1	b
10	5	c
7	2	r

$\equiv$

a	b	c
2	3	a
2	3	a
7	2	r

Figure 9: Intersection

2.2.10 Comparisons and basic arithmetic operations $\odot_{(\mathsf{c}:<\mathsf{c}',\mathsf{c}''>)}R$

Comparisons and *basic arithmetic operations* are added to the standard algebra. They are generalized in $\odot$ because they all follow the same principle in translation and are all binary. They consume R and as arguments an operator token from ($\circledless$, $\circledgtr$, $\circledleq$, $\circledgeq$, $\oplus, \ominus, \oslash, \circledast$, $\circleddash$) plus two columns c' and c'' from the given table. $\odot$ generates a new column c which carries the evaluated result.

The type of the result is ambigious. For $\circledless$, $\circledgtr$, $\circledleq$, $\circledgeq$ we get a boolean $\mathsf{c} \in \{true, false\}$ and for $\oplus, \ominus, \oslash, \circledast$, $\circleddash$ a number.

2.2.11 Negation $\neg_{(\mathsf{c}:\mathsf{c}')}R$

The negation works similar to the previous operator but is unary and works on columns carrying a boolean value. For R and the boolean column c' in R we generate a new column c. $\neg$ simply writes into c the different negated value of column c'.

$\oplus_{\text{RES:<a,b>}}$

a	b	c
2	3	a
5	1	b
6	2	c
7	2	r

$\equiv$

a	b	c	RES
2	3	a	5
5	1	b	6
6	2	c	8
7	2	r	9

Figure 10: Plus

$\neg_{\text{RES:b}}$

a	b
2	true
5	false
6	false
7	false

$\equiv$

a	a	RES
2	true	false
5	false	true
6	false	true
7	false	true

Figure 11: Negation

2.2.12 Distinct $\delta\ R$

The *Distinct* operator filters out duplicate tuples from R. The output is a duplicate free table of the same schema as R.

δ

a	b	c
2	3	a
5	1	b
2	3	a
7	2	r

$\equiv$

a	b	c
2	3	a
5	1	b
7	2	r

Figure 12: Distinct

3 Translation

To translate a SQL query, we need a predefined structure of rules. The relational algebra operators presented in section 2 are just a brief introduction to the 'backbone' of the translation. To ensure correctness of the translation of SQL queries conforming the SQL99 standard, we need certain rules, which are derived from the relational algebra. Shortened that means, that each rule consists of an composed sequence of algebraic operators.

The rules don't check the correctness of SQL code. They simply describe how a SQL subset can be transformed into relational algebra in compositional manner. That allows to extend the translation to rules which cope also ORDER BY and GROUP BY not presented in this approach.

To evaluate the rules correctly and have an defined in- and output we need data structures able to carry algebraic plans. In section 3.1 we introduce the set *environment* abbreviated as Γ and the *loop* relation. Both together are sufficient to carry all data needed for the correct translation.

3.1 Data Structures

As already mentioned, there are some data structures needed to carry data and meta-data. In our case the data is defined by algebraic plans. Carrying this data is the job of the *environment*. When requested, the data is always accessible, if present in the *environment*.

Metadata is needed to ensure right mapping of queries into subqueries and back. The metadata is managed by the *loop* relation.

3.1.1 Data Environment Γ

Γ is a set carrying algebraic plans. To facilitate we provide in Fig 13 the materialized view of an entry in the *environment*.

$s.t.\mathsf{c} \mapsto$

KEY	ITEM
1	10
2	20
3	15

Figure 13: Materialized structure of an 'entry' in Γ

The structure allowed, is a table (algebraic plan) referenced by its identifier. The identifier is the name of the column c which is now renamed after ITEM and the surrounding table t and schema s in which the column has been stored before. The KEY-column is the identity of the row. Each KEY-column in the whole *environment* holds the same identity.

Fig. 14 shows the 'real' structure of entries in the *environment*. Each entry is an algebraic plan composed from relational operators.

A table R with n columns is presented by an *environment* with n entries. On the bottom of the DAG we reference the table R. As row identifier we attach the column KEY. The *environment* contains 3 entries derived by the algebraic structure of the DAG below. These expressions would evaluate to columns shown.

A more general approach of how data is stored in the *environment* is given in Fig. 15.

3.1.2 Metadata loop

Loop is the essential metastructure. *Loop* ensures right mappings from queries into subqueries and back. The IK column (inner key) of *loop* is derived by the KEY column of the referenced table and equal to the KEY column of the current *environment*. The OK column (outer key) refers to the IK column of loop_0 provided of the surrounding query. After renaming we apply the cross product on both columns to ensure correct mapping.

3.2 Rules

The rules can be seen as instructions how to compose algebraic operators. To translate SQL queries we provide a predefined structure. In detail, we need rules to form the input structure of SQL queries into a DAG derived by relational algebra operators. As already seen in building up the *environment* we produce rules in order to map SQL queries into algebraic expressions.

To maintain the SQL structure we provide two kinds of rules - the table inferencerules and the column inferencerules. Both handle the input consisting of Γ, loop and an expression e in the same way but can be distinguished by their output. The table inferencerules emit a new Γ' and loop$'$, whereas the column inferencerules emit a single column KEY|ITEM. The expression e is an abbreviation for any kind of SQL - either control structure like a `SELECT` clause or data carried with the control structure like columns.

3.2.1 Table Inferencerules

The table inferencerule is one of two available rules. The rule appears once you need an *environment* as output. Internally we process the expression e needing the *environment* and *loop* to a new or manipulated *environment*$'$ and changed *loop*$'$.

$$\Gamma; loop \vdash e \Mapsto (\Gamma'; loop')$$

For example '`SELECT` $\mathsf{c}', \mathsf{c}''$' needs an *environment* as output. This *environment* has exactly two entries, c' and c''.

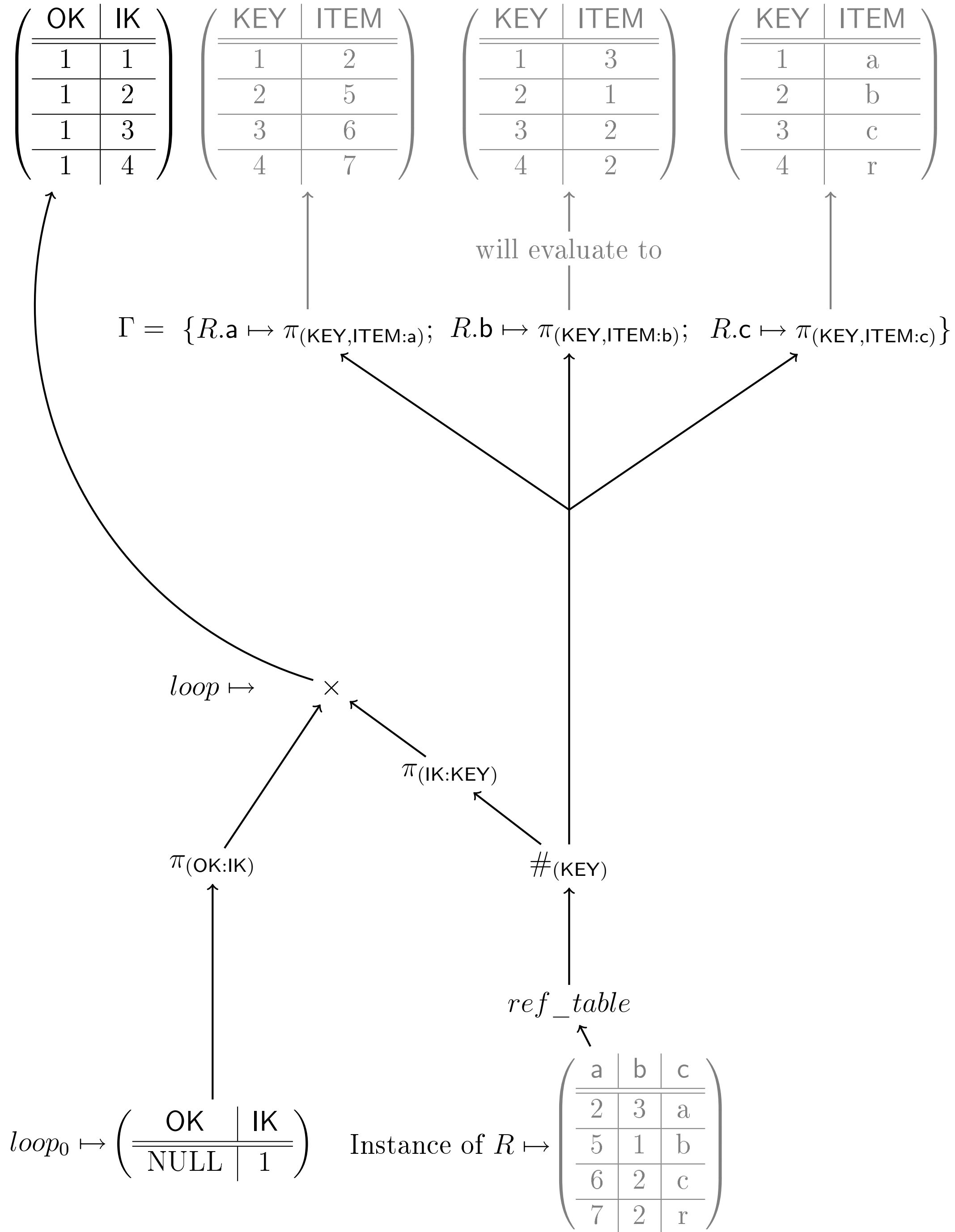

Figure 14: Example instance of *environment*

Γ={<schema.table |
table |
alias > .<column>$\mapsto$ algebraic plan

Figure 15: Entries in Γ

3.2.2 Column Inferencerules

Not every rule needs an *environment* as output. Column inferencerules satisfy the task given by expression e with a column q as output.

$$\Gamma; loop \vdash \ e \rhd \ q$$

$$q \mapsto \left(\begin{array}{c|c} \mathsf{KEY} & \mathsf{ITEM} \\ \hline\hline \vdots & \vdots \end{array} \right)$$

For example for the '`WHERE p`' clause q is a sufficient output. In KEY we have the row identifiers and in ITEM we have for each row one of the boolean values depending whether the predicate p is evaluated to *true* or *false*.

3.3 Translation in detail

The following translation of SQL queries lead to a relational algebra expression and so to a DAG structure. SQL is traversed topdown whereas the algebra is synthesized bottomup.

Apart from *SERIALIZE* which can be seen as the invisible seed of every query, every rule in our set of rules is called by a special expression. E.g. a standalone 'table' calls the rule *table* whereas '`FROM` table' calls the rule '`FROM` e' with table as e.

3.3.1 `SELECT`...`FROM`...`WHERE`...

`SELECT FROM WHERE` can be seen as the main block. Since there is no `GROUP BY`, `ORDER BY` and `HAVING` supported, `SELECT FROM WHERE` is sufficient to frame any SQL-query completely.

We introduce the *concat* operator ++. To concatenate two *environments* Γ' and Γ'' we use $\Gamma = \Gamma' \ ++ \Gamma''$. Any lookup on Γ first searches in Γ' and then subsequent in Γ''. This ensures correct column lookup and the outer scope is searched after the inner scope like in SQL.

$$\Gamma; loop \vdash \texttt{FROM e}_F \Mapsto (\Gamma_F; loop_F)$$

$$\Gamma = \{\ldots, e_x \mapsto q_x, \ldots\}$$

$$\Gamma' = \{\ldots, e_x \mapsto \pi_{(\mathsf{KEY:IK,ITEM})}(loop_F \underset{\mathsf{OK=KEY}}{\bowtie} q_x), \ldots\}$$

$$\Gamma_F = \{\ldots, e_y \mapsto q_y, \ldots\}$$

$$\Gamma'_F = \Gamma_F \; ++ \; \Gamma'$$

$$\Gamma'_F; loop_F \vdash \texttt{WHERE e}_W \rhd q_W$$

$$loop' \equiv \pi_{(\mathsf{OK,IK})}(loop_F \underset{\mathsf{IK=KEY}}{\bowtie} (\pi_{(\mathsf{KEY})} q_W)$$

$$\Gamma_W = \{\ldots, e_y \mapsto \pi_{(\mathsf{KEY,ITEM})}(loop') \underset{\mathsf{IK=KEY'}}{\bowtie} (q_y)), \ldots\}$$

$$\Gamma_W; loop' \vdash \texttt{SELECT e}_S \Mapsto (\Gamma_S; loop_S)$$

$$\Gamma; loop \vdash \texttt{SELECT e}_S \texttt{ FROM e}_F \texttt{ WHERE e}_W \Mapsto (\Gamma_S; loop_S)$$

For a given *environment* and a loop we expect an expression in the form of `SELECT` e_S `FROM` e_F `WHERE` e_W. This expression can be split into three subexpressions `SELECT` e_S, `FROM` e_F and `WHERE` e_W. They are called separately to evaluate the body of the rule.

- If this '`SELECT` $\texttt{e}_S$ `FROM` $\texttt{e}_F$ `WHERE` $\texttt{e}_W$' block is the outermost block in the query, we define $\Gamma = \{\}$ and $loop \equiv$

OK	IK
NULL	1

  . Otherwise these parameters are obtained from the calling rule.
- '`FROM` e_F' translates the expression e_F to a new Γ_F and emits the assigned $loop_F$.
- Γ representing the outer *environment* needs to be mapped to the scope of the inner query. We use $loop_F$ for a correct mapping. The outer key OK of $loop_F$ carries the same identities like q_x but already mapped to the inner scope. An equijoin on $\mathsf{OK} = \mathsf{KEY}$ will map q_x to the inner scope. The result is stored in Γ'
- Concatenate both *environments* Γ_F and Γ' and store it in Γ'_F.
- Run '`WHERE` e_S'. e_S emits q_w, carrying all keys fullfilling the predicate in e_S.
- We manipulate the entries q_y of Γ_F in order to delete keys not appearing in q_W. The result is stored to Γ_W. Apply the same to $loop_F$.
- Run '`SELECT` e_S'. The result Γ_S will just carry entries listed in e_S.
- The result '`SELECT` e_S' forms the result and is emitted to the calling rule.

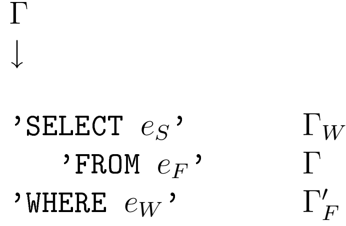

Figure 16: Visibility of Γ

As seen we deal with several *environments* representing different tables. Since the internal rules are called with different *environments* we introduce the visibility problem.

As input we receive Γ and *loop* from the calling rule. Leaving *loop* aside we feed '`FROM` e_F' with Γ. All data from the outer scope is visible in `FROM`. '`FROM` e_F' results to Γ_F. The data of Γ_F is disjunct to the data of Γ even though name conflicts can appear. To avoid name conflicts we apply the concat-function. Similar to SQL semantics, data emitted by `FROM` will be seen first and just if we can't find a matching entry we lookup the entries of the outer scope in Γ.

'`SELECT` e_S' is not allowed to see *environment*. Only expressions of the subquery is be visible. To ensure this we run `SELECT` with Γ_W representing the manipulated Γ_F

3.3.2 FROM

The main function of `FROM` in SQL is to supply other operations with tables. Since we internally do not work on tables we rewrite the function of `FROM`. `FROM` translates the data given as tables or *environments* from subqueries to exactly one *environment* carrying all references.

The '`FROM` e_F' of the previous rule is enlarged to '`FROM` $e_1, \ldots, e_n$'. For every $e_j,\ j = 1, \ldots, n$ we run the underlying rule, either '`SELECT`...`FROM`...`WHERE`...' or *table*3.3.5 resulting to Γ_j.

Because we need a common identity for all entries in our resulting *environment* we produce $loop_{1\ldots n}$. This *loop* has as inner key the overall identity.

Γ' is now produced by copying the entries of $\Gamma_j\ j = 1, \ldots, n$ and joining them with the respective columns of $loop_{1\ldots n}$. The inner key of $loop_{1\ldots n}$ is now the new key of every entry of Γ'.

Finally we emit $loop''$ by a project on OK, IK of $loop_{1\ldots n}$.

$$\begin{array}{lll}
\Gamma; loop \vdash & e_1 & \mapsto (\Gamma_1 = \{e.\mathsf{c}_1 \mapsto q_1.\mathsf{c}_1, \ldots, e.\mathsf{c}_\mathsf{m} \mapsto q_1.\mathsf{c}_\mathsf{m}\}; loop_1) \\
 & \vdots & \\
\Gamma; loop \vdash & e_n & \mapsto (\Gamma_n = \{e.\mathsf{c}_1 \mapsto q_n.\mathsf{c}_1, \ldots, e.\mathsf{c}_\mathsf{k} \mapsto q_n.\mathsf{c}_\mathsf{k}\}; loop_n)
\end{array}$$

$$
\begin{aligned}
loop_{(12)} &\equiv (\pi_{(\mathsf{OK},\mathsf{IK}_1,\mathsf{IK}_2)}(\pi_{(\mathsf{OK},\mathsf{IK}_1:\mathsf{IK})} loop_1) \underset{(\mathsf{OK}=\mathsf{OK}')}{\bowtie} (\pi_{(\mathsf{OK}':\mathsf{OK},\mathsf{IK}_2:\mathsf{IK})} loop_2)) \\
loop_{(123)} &\equiv (\pi_{(\mathsf{OK},\mathsf{IK}_1,\mathsf{IK}_2,\mathsf{IK}_3)}(\pi_{(\mathsf{OK},\mathsf{IK}_1,\mathsf{IK}_2)} loop_{(12)}) \underset{(\mathsf{OK}=\mathsf{OK}')}{\bowtie} (\pi_{(\mathsf{OK}':\mathsf{OK},\mathsf{IK}_3:\mathsf{IK})} loop_3)) \\
&\vdots \\
loop_{(1\ldots n)} &\equiv (\pi_{(\mathsf{OK},\mathsf{IK}_1,\ldots,\mathsf{IK}_n)}(\pi_{(\mathsf{OK},\mathsf{IK}_1,\ldots,\mathsf{IK}_{n-1})} loop_{(1\ldots n-1)}) \underset{(\mathsf{OK}=\mathsf{OK}')}{\bowtie} (\pi_{(\mathsf{OK}':\mathsf{OK},\mathsf{IK}_n:\mathsf{IK})} loop_n))
\end{aligned}
$$

$$
loop' \equiv \#_{(\mathsf{IK})}(loop_{(1\ldots n)})
$$
$$
loop'' \equiv \pi_{(\mathsf{OK},\mathsf{IK})} loop'
$$

$$
\begin{aligned}
\Gamma' = \{(e_1.\mathsf{c}_1 &\mapsto \pi_{(\mathsf{KEY}:\mathsf{IK},\mathsf{ITEM})}(\pi_{(\mathsf{IK},\mathsf{IK}_1)} loop' \underset{(\mathsf{IK}_1=\mathsf{KEY})}{\bowtie} (q_1.\mathsf{c}_1)) \\
&\vdots \\
(e_1.c_m &\mapsto \pi_{(\mathsf{KEY}:\mathsf{IK},\mathsf{ITEM})}(\pi_{(\mathsf{IK},\mathsf{IK}_1)} loop' \underset{(\mathsf{IK}_1=\mathsf{KEY})}{\bowtie} (q_1.\mathsf{c}_\mathsf{m})) \\
&\vdots \\
&\vdots \\
(e_n.c_1 &\mapsto \pi_{(\mathsf{KEY}:\mathsf{IK},\mathsf{ITEM})}(\pi_{(\mathsf{IK},\mathsf{IK}_n)} loop' \underset{(\mathsf{IK}_n=\mathsf{KEY})}{\bowtie} (q_n.\mathsf{c}_1)) \\
&\vdots \\
(e_n.\mathsf{c}_\mathsf{k} &\mapsto \pi_{(\mathsf{KEY}:\mathsf{IK},\mathsf{ITEM})}(\pi_{(\mathsf{IK},\mathsf{IK}_n)} loop' \underset{(\mathsf{IK}_n=\mathsf{KEY})}{\bowtie} (q_n.\mathsf{c}_\mathsf{k}))\}
\end{aligned}
$$

$$
\Gamma; loop \vdash \texttt{FROM}\ e_1, \ldots, e_n \Mapsto (\Gamma'; loop'')
$$

In order to complete the visibility problem we introduced in Fig. 16, we now add the visibility between the *environments* resulting from the different expressions. Like in SQL they are not visible to each other until all expressions $e_i, i = 1, \ldots, n$ are evaluated completely.

3.3.3 `WHERE`

`WHERE` is a rule designed to obtain a predicate. The predicate e presenting a predicate of arbitrary nesting is directly passed through to the rules 3.3.8, 3.3.9, 3.3.10, 3.3.11, 3.3.12, 3.3.13. The result of e is defined as a KEY|ITEM column carrying in KEY identities of the *environment* entries. The rows of ITEM on the other hand are boolean values depending whether the predicate p is evaluated to *true* or *false*. The job of `WHERE` is to filter only rows containing *true* as entry in ITEM.

$$\frac{\Gamma; loop \vdash e \rhd q \qquad q' \equiv \sigma_{(\mathsf{ITEM})}(q)}{\Gamma; loop \vdash \texttt{WHERE } e \rhd q'}$$

3.3.4 `SELECT`

In addition to `WHERE` and `FROM`, `SELECT` is the last rule translated in the scope of the '`SELECT` $\mathtt{e}_S$ `FROM` $\mathtt{e}_F$ `WHERE` $\mathtt{e}_W$' block.

First we extend the expression e_S to the sequence of expressions 'e_1 `AS` $N_1, \ldots, e_n$ `AS` N_n'. Since $e_i,\ i = 1, \ldots, n$ is ambigious in it's meaning but needs to have the certain output in the form of a KEY|ITEM column we call each e_i seperate, resulting in q_i. Finally every q_i is stored in Γ'.

As reference and to provide renaming we attach the alias N_i. In the case of no renaming assume $e_i \in s.t.\mathsf{c}$ and therefor $e_i = N_i$

$$\frac{\begin{array}{rcl} \Gamma; loop & \vdash & e_1 \rhd q_1 \\ & \vdots & \\ \Gamma; loop & \vdash & e_n \rhd q_n \\ \Gamma' = \{N_1 \mapsto q_1, \ldots, N_n \mapsto q_n\} \end{array}}{\Gamma, loop \vdash \texttt{SELECT } e_1 \texttt{ AS } N_1, \ldots, e_n \texttt{ AS } N_n \Mapsto (\Gamma'; loop)}$$

3.3.5 Table

Rule *Table* transforms one table with n columns to a Γ with n entries.

To identify the entries in the *environment* we provide *schema.table* as name and referenced with the ref_table operator. To avoid name conflicts we directly rename every column $\mathsf{c_i},\ \mathsf{i} = 1, \ldots, \mathsf{n}$ to $\mathsf{ITEM_i}$ and attach a new column KEY as identity. To keep consistency for all e_j in `FROM` we connect the inner key of *loop* with the KEY-column of t by a cross product. The result is joined with t to keep the consistency to `FROM` also in Γ'. Finally we allocate the entries in Γ' with their respective name.

The names of the references have to be chosen carefully. Correct identifying of columns and tables in following parts of the query are directly dependent on a clear and distinct name. All information like names of tables, schemas, columns and aliases need to be present in the reference's name. Hence we denote the column col of table *tbl* in schema *schema* as *schema.tbl.col* abbreviated as *stc*.

$$t \equiv \#_{(\mathsf{KEY})}(\pi_{(\mathsf{ITEM_1:c_1,\ldots,ITEM_n:c_n})}(ref_table(schema.table.\mathsf{c_1} \ldots \mathsf{c_n})))$$

$$loop' \equiv \#_{(\mathsf{IK})}(\pi_{(\mathsf{OK:IK})}(loop) \times (\pi_{(\mathsf{KEY})}(t)))$$

$$stc \equiv \pi_{(\mathsf{KEY:IK,ITEM_1,\ldots,ITEM_n})}((\pi_{(\mathsf{KEY':KEY})}loop') \underset{\mathsf{KEY'=KEY}}{\bowtie} (t))$$

$$\Gamma' = \{schema.tbl.c_1 \mapsto \pi_{(\mathsf{KEY,ITEM:ITEM_1})}(stc)$$

$$\vdots$$

$$schema.tbl.c_n \mapsto \pi_{(\mathsf{KEY,ITEM:ITEM_n})}(stc)\}$$

$$loop'' \equiv \pi_{(\mathsf{OK,IK})}(loop')$$

$$\frac{}{\Gamma; loop \vdash schema.table.\mathsf{c_1} \ldots \mathsf{c_n} \mapsto (\Gamma'; loop'')}$$

3.3.6 Lookup

A lookup on the entries of the *environment* is sometimes needed. This operator obtains the entry by the reference and returns it to the calling structure.

$$\frac{\Gamma = \{\ldots, stc \mapsto q, \ldots\}}{\Gamma; loop \vdash stc \rhd q}$$

3.3.7 Constant

The *constant* rule is similar to the *table* rule. But instead of a table we allocate a single constant. Due to the fact that the calling rule needs a representation comparable to entries in an *environment*, we lift it to it's scope by applying the cross product on *loop* and the constant attached to the column ITEM.

$$\frac{q \equiv \pi_{(\mathsf{KEY:IK,ITEM})}\left(loop \times \frac{\mathsf{ITEM}}{\text{const}}\right)}{\Gamma; loop \vdash const \rhd q}$$

3.3.8 $\odot$

In order to evaluate simple predicates like comparisons we introduce the $\odot$-rule. Apart from reducing the amount of rules, $\odot$ can also be used to evaluate basic arithmetic operations. Alltogether $\odot$ is capable to represent $(<, >, \leq, \geq, \oplus, \ominus, \oslash, \circledast$, $\circleddash$, `AND`, `OR`) as supported operators already introduced in chapter two.

Assume the simple case of comparing two entries of Γ. The lookups on e_1 and e_2 lead to their representation q_1 and q_2. Since each entry in Γ has the same scope their KEY columns are identical and we can apply an equijoin on them . After applying the

comparison operator the new column RES carries the results either '$true$' or '$false$'. q has after an projection just the common columns KEY|ITEM. Same procedure applies if either e_1 or e_2 are constants. Then the *Constant* rule is called instead of a lookup.

Now assume the that one or both of the expressions e_1 or e_2 are predicates again. Then we call recursively either the $\odot$ rule. With this knowlegde translation of arbitrarly nested predicates is feasible.

$$\Gamma; loop \vdash e_1 \rhd q_1$$

$$\Gamma; loop \vdash e_2 \rhd q_2$$

$$q_1' \equiv \pi_{(\mathsf{KEY'}:\mathsf{KEY};\mathsf{ITEM'}:\mathsf{ITEM})} q_1$$

$$\frac{q \equiv \pi_{(\mathsf{KEY},\mathsf{ITEM}:\mathsf{RES})}(\odot_{(\mathsf{RES}:<\mathsf{ITEM'};\mathsf{ITEM}>)}(q_1' \underset{\mathsf{KEY'}=\mathsf{KEY}}{\bowtie} q_2)))}{\Gamma; loop \vdash e_1 \odot e_2 \rhd q}$$

3.3.9 `NOT`

In addition to $\odot$ we need a rule presenting the logical `NOT`. The relational algebra provides $\neg$, doing that job. Similar to $\odot$ we lookup e and apply the $\neg$ operator. To reconstruct the standard output we delete column ITEM and substitute it by RES.

$$\Gamma; loop \vdash e \rhd q$$

$$\frac{q' \equiv \pi_{(\mathsf{KEY},\mathsf{ITEM}:\mathsf{RES})}(\neg_{(\mathsf{RES}:\mathsf{ITEM})}(q))}{\Gamma; loop \vdash \texttt{NOT}\ e \rhd q'}$$

3.3.10 `EXISTS`

The rule `EXISTS` expects e to be a structure leading to an *environment* Γ' with arbitrary many entries. For the translation we simply need the presence of $loop'$. $loop'$ will be tested for the occurence of distinct outer keys. Outer keys being in $loop'$ 'exist' in e and are associated to the inner key of *loop*. To these keys we attach the ITEM *true*. The complete translation expects column ITEM to have the correct cardinality, so all outer keys not appearing in $loop'$ but being present in *loop* as inner key are also part of result. To these keys we attach the ITEM $false$.

$$\Gamma; loop \vdash e \Mapsto (\Gamma'; loop')$$

$$q \equiv \delta(\pi_{(\mathsf{KEY}:\mathsf{OK})} loop')$$

$$q' \equiv (@_{(\mathsf{ITEM}='true')} q) \cup (@_{(\mathsf{ITEM}='false')}(\pi_{(\mathsf{KEY:IK})}(loop)) \backslash q)$$

$$\Gamma; loop \vdash \texttt{EXISTS}\ e \rhd q'$$

3.3.11 SOME/ANY

Similar to EXISTS in SOME/ANY we need to translate e first. The result of e is an *environment* with exactly one entry e_x. For correct translation e_x needs to be compared with e_y, provided by Γ. The comparison will be applied by the operator $\odot$. Since $\odot$ needs first, data from a common environment and second identical KEY columns, the algebraic plan q_y referenced by e_y will be mapped to the scope of $loop'$ and then concatenated with Γ' to Γ''

We are looking for inner keys of *loop* appearing at least once in result of $\odot$, q. $loop'$ grants the connection between q and *loop*. Just distinct outer keys of the join on $loop'$ and q having the ITEM-column *true* are equal to inner keys of *loop* satisfying $\odot$ and so appear in q'. To match the cardinality of the estimated result size we union all outer keys not appearing in q' but being present in *loop* as inner key to this result. To these keys we attach the ITEM $false$.

$$\Gamma; loop \vdash e \Mapsto (\Gamma' = \{e_x \mapsto q_x\}; loop')$$

$$\Gamma; loop \vdash e_y \rhd q_y$$

$$mapped_q_y \equiv \pi_{(\mathsf{KEY:IK,ITEM})}(q_y \underset{\mathsf{KEY=OK}}{\bowtie} loop')$$

$$\Gamma'' = \Gamma' \cup \{e_y \mapsto mapped_q_y\}$$

$$\Gamma''; loop' \vdash e_x \odot e_y \rhd q$$

$$q' \equiv \delta(\pi_{(\mathsf{KEY:OK,ITEM})}(loop' \underset{\mathsf{IK=KEY}}{\bowtie} \sigma_{(\mathsf{ITEM}='true')} q))$$

$$q'' \equiv q' \cup (@_{(\mathsf{ITEM}='false')}(\pi_{(\mathsf{KEY:IK})} loop) \backslash (\pi_{(\mathsf{KEY})} q'))$$

$$\Gamma = \{\ldots, e_y \mapsto q_y, \ldots\} \vdash e_y\ \odot \texttt{SOME/ANY}\ e \rhd q''$$

3.3.12 IN

A simple case of IN can be seen as a special case of SOME/ANY. If we reduce SOME/ANY to the equality comparision in $\odot$, then SOME/ANY is able to evaluate IN. If we now expand the count of entries in Γ' to n and provide the same count of arguments on the lefthandside of IN the impact of the rule gets clear. The whole row with n columns needs to be compared.

We look up e once. But for each entry in Γ' $e_{x_i},\ i = 1, \ldots, n$ and the associated e_{y_i} we call $\ominus$ separatly. To simulate the comparison of whole rows we intersect the results of the n evalutated q_i whose ITEM is *true*. The result of this operation, q can now be translated like the equivalent part in SOME/ANY.

$$\Gamma; loop \vdash e \Mapsto (\Gamma' = \{e_{x_1} \mapsto q_{x_1}, \ldots, e_{x_n} \mapsto q_{x_n}\}; loop')$$

$$\Gamma; loop \vdash e_{y_1} \rhd q_{y_1}$$

$$\vdots$$

$$\Gamma; loop \vdash e_{y_n} \rhd q_{y_n}$$

$$mapped_q_{y_1} \equiv \pi_{(\mathsf{KEY:IK,ITEM})}(q_{y_1} \underset{\mathsf{KEY=OK}}{\bowtie} loop')$$

$$\vdots$$

$$mapped_q_{y_n} \equiv \pi_{(\mathsf{KEY:IK,ITEM})}(q_{y_n} \underset{\mathsf{KEY=OK}}{\bowtie} loop')$$

$$\Gamma_1 = \Gamma' \cup \{e_{y_1} \mapsto mapped_q_{y_1}\}$$

$$\vdots$$

$$\Gamma_n = \Gamma' \cup \{e_{y_n} \mapsto mapped_q_{y_n}\}$$

$$\Gamma_1; loop' \vdash e_{x_1} \ominus e_{y_1} \rhd q_1$$

$$\vdots$$

$$\Gamma_n; loop' \vdash e_{x_n} \ominus e_{y_n} \rhd q_n$$

$$q \equiv (\sigma_{(\mathsf{ITEM})} q_1) \cap \ldots \cap (\sigma_{(\mathsf{ITEM})} q_n)$$

$$q' \equiv \delta(\pi_{(\mathsf{KEY:OK,ITEM})}(loop' \underset{(\mathsf{IK=KEY})}{\bowtie} q))$$

$$q'' \equiv q' \cup (@_{(\mathsf{ITEM}='false')}(\pi_{(\mathsf{KEY:IK})} loop) \backslash (\pi_{(\mathsf{KEY})} q'))$$

$$\Gamma; loop \vdash (e_{y_1}, \ldots, e_{y_n}) \ \texttt{IN} \ e \rhd q''$$

3.3.13 ALL

ALL uses the same approach as ANY/SOME but differs in the last step. Unlike ANY/SOME not just one row of e_x has to satisfy the comparison on e_y but all of them. This implies that if there is an outer key not evaluating to *true* in the join between $loop'$ and q, the outer key is deleted from the result.

However, in the last line of the rule we concatenate rows having keys with $false$ in their ITEM-column with rows having *true*.

$$\Gamma; loop \vdash e \Rrightarrow (\Gamma' = \{e_x \mapsto q_x\}; loop')$$

$$\Gamma; loop \vdash e_y \rhd q_y$$

$$mapped_q_y \equiv \pi_{\mathsf{KEY:IK,ITEM}}(q_y \underset{(\mathsf{KEY=OK})}{\bowtie} loop')$$

$$\Gamma'' = \Gamma' \cup \{e_y \mapsto mapped_q_y\}$$

$$\Gamma''; loop' \vdash e_x \odot e_y \rhd q$$

$$q' \equiv \zeta(\pi_{(\mathsf{KEY:OK,ITEM})}(loop' \underset{(\mathsf{IK=KEY})}{\bowtie} \texttt{NOT}\ (\sigma_{(\mathsf{ITEM}='true')}\ \texttt{NOT}\ q)))$$

$$q'' \equiv q' \cup (@_{(\mathsf{ITEM}='true')}(\pi_{(\mathsf{KEY:IK})}loop) \backslash (\pi_{(\mathsf{KEY})}q'))$$

$$\overline{\Gamma; loop \vdash e_y\ \odot \texttt{ALL}\ e \rhd q''}$$

3.3.14 SELECT DISTINCT

If instead of SELECT e, SELECT DISTINCT e is found in the query, then we translate SELECT e first and apply DISTINCT on the resulting *environment*. Since DISTINCT is an operator reading wholes rows and the entries in the *environment* are stored column-wise we need some reconstruction before applying DISTINCT.

First we lookup all entries in Γ', presenting the result of SELECT e. Then all entries $q_i,\ i = 1, \ldots, n$ are joined on their KEY-column. To keep the association to the outer key we also apply the join on *loop*. The table consisting of $\mathsf{OK}, \mathsf{ITEM_1}, \ldots, \mathsf{ITEM_n}$ is the input for DISTINCT. Because the old key is obsolete we attach a new key being the new row identity. Finally, we rebuilt the *environment*-structure in Γ'' and emit $loop''$ with the new key as well.

$$\Gamma, loop \vdash \texttt{SELECT e} \Rrightarrow (\Gamma' = \{e.c_1 \mapsto q_1, \ldots, e.c_n \mapsto q_n\}, loop')$$

$$\Gamma', loop' \vdash e.c_1 \rhd q_1$$

$$\vdots$$

$$\Gamma', loop' \vdash e.c_n \rhd q_n$$

$$q_1' \equiv (\pi_{(\mathsf{KEY_1:KEY,ITEM_1:ITEM})} q_1)$$

$$\vdots$$

$$q_n' \equiv (\pi_{(\mathsf{KEY_n:KEY,ITEM_n:ITEM})} q_n)$$

$$join_\Gamma \equiv (q_1') \underset{(\mathsf{KEY_1=KEY_2})}{\bowtie} (q_2') \underset{(\mathsf{KEY_2=KEY_3})}{\bowtie} \dots \underset{(\mathsf{KEY_{(n-1)}=KEY_n})}{\bowtie} (q_n')$$

$$join_{\Gamma,loop} \equiv join_\Gamma \underset{(\mathsf{KEY_1=IK})}{\bowtie} loop$$

$$distinct_values \equiv \#_{(\mathsf{KEY})}(\delta(\pi_{(\mathsf{OK,ITEM_1,\dots,ITEM_n})}(join_{\Gamma,loop})))$$

$$loop'' \equiv \pi_{(\mathsf{OK,IK:KEY})}(distinct_values)$$

$$\begin{aligned}\Gamma'' = \{(e.c_1 &\mapsto \pi_{(\mathsf{KEY,ITEM_1})}(distinct_values))\\ &\vdots\\ (e.c_n &\mapsto \pi_{(\mathsf{KEY,ITEM_n})}(distinct_values))\}\end{aligned}$$

$$\overline{\Gamma; loop \vdash \texttt{SELECT DISTINCT e} \Rrightarrow (\Gamma'', loop'')}$$

3.3.15 `e AS s`

An expression inside of `FROM` can be ambigious. Since some constructs behind expressions, like subqueries must be renamed after their evaluation we provide the `AS` rule. We simply lookup all entries of Γ', presenting the *environment* built up from expression e and name the identifier after the name provided on the righthandside of `AS`.

$$\frac{\Gamma; loop \vdash \texttt{e} \Rrightarrow (\Gamma' = \{e.c_1 \mapsto q_1, \dots, e.c_n \mapsto e_n\}, loop')}{\Gamma; loop \vdash \texttt{e AS s} \Rrightarrow (\Gamma'' = \{\mathsf{s.c_1} \mapsto q_1, \dots, \mathsf{s.c_n} \mapsto q_n\}, loop')}$$

3.3.16 SERIALIZE

SERIALIZE is the rule on the top of the directed acyclic graph resulting from the translation. The presence of a complete SQL query triggers *SERIALIZE*.

$$\Gamma; loop \vdash e \mapsto (\Gamma' = \{st\mathsf{c_1} \mapsto q_1, \ldots, st\mathsf{c_n} \mapsto q_n\}; loop')$$

$$\Gamma', loop \vdash st\mathsf{c_1} \rhd q_1$$

$$\vdots$$

$$\Gamma', loop \vdash st\mathsf{c_n} \rhd q_n$$

$$q_{12} \equiv \pi_{(\mathsf{KEY},\mathsf{c_1},\mathsf{c_2})}(\pi_{(\mathsf{KEY},\mathsf{c_1}:\mathsf{ITEM})}(st\mathsf{c_1}) \underset{\mathsf{KEY}=\mathsf{KEY'}}{\bowtie} \pi_{(\mathsf{KEY'},\mathsf{c_2}:\mathsf{ITEM})}(st\mathsf{c_2}))$$

$$q_{123} \equiv \pi_{(\mathsf{KEY},\mathsf{c_1},\mathsf{c_2},\mathsf{c_3})}((q_{12}) \underset{\mathsf{KEY}=\mathsf{KEY'}}{\bowtie} \pi_{(\mathsf{KEY'},\mathsf{c_3}:\mathsf{ITEM})}(st\mathsf{c_3}))$$

$$\vdots$$

$$q_{1\ldots n} \equiv \pi_{(\mathsf{KEY},\mathsf{c_1},\ldots,\mathsf{c_n})}((q_{12\ldots n-1}) \underset{\mathsf{KEY}=\mathsf{KEY'}}{\bowtie} \pi_{(\mathsf{KEY'},\mathsf{c_n}:\mathsf{ITEM})}(st\mathsf{c_n}))$$

$$q \equiv \pi_{(\mathsf{c_1},\ldots,\mathsf{c_n})}(q_{1\ldots n})$$

$$print(q)$$

$$\Gamma = \{\}, loop \equiv \frac{\begin{array}{c|c} \mathsf{OK} & \mathsf{IK} \end{array}}{\begin{array}{c|c} \text{NULL} & 1 \end{array}} \vdash e \rhd NULL$$

We start the translation with an empty *environment* and a literal *loop*. Table *loop* carries NULL as outer key. This value will never be read and just exists for consistency reasons. As inner key we provide just one entry. e is the abbreviation for the whole query to be translated.

Not an *environment* but a table is expected to be the output in the end of the query. Hence we need to rebuild the common table structure. To provide this structure we first lookup the columns $stc_i, i = 1, \ldots, n$. Then we rename the columns to their original names extracted by the reference's name and connect all columns pairwise by a join on their KEY column.

4 Example Translation

As roundup and to make the use of the rules clear we provide an example. First we will see how the rules can be composed and how they interact without showing the algebra underneath. From this basis shown in Fig. 18 we will translate Query Q2 in Fig. 17 to it's algebraic plan.

[a]
```
SELECT R.a
  FROM R,S
WHERE R.a>S.a
```

[b]

a	b
2	3
2	3
6	2
7	2

[c]

a	b
2	3
5	1
10	5
7	2

[d]

a
6
7
6
7

Figure 17: [a] Query Q2, [b], [c] Tables R and S, [d] q, Output of Q2

Q2 shown in Fig. 17 is a simple SQL-query on two tables R and S. Just one comparison on two columns is included to keep the query simple. The plan without the algebraic operators is shown in fFig. 18. Nodes 1-12 depict the rules included in the evaluation of the query with their scopes as dashed rectangular boxes underneath. Like *SERIALIZE* surrounds the whole query, so it's box does as well. *Lookup* on the other hand has no scope, so no box is provided.

First *SERIALIZE* as the invisible seed of the query is called and passes the SQL query to `SELECT`...`FROM`...`WHERE`..., expecting an algebra-plan as result. On the other hand `SELECT`...`FROM`...`WHERE`... relies on the correct translation of `FROM`, `WHERE` and `SELECT`. `FROM` needs the algebraplan of *table* whereas `WHERE` requires the result of $\odot$ to produce its output. As we can see SQL is traversed topdown whereas the algebra is synthesized bottomup.

Now to include the relational operatores we expand the rules given in Fig. 18 The overall plan would go over the scope of a page, so we split the plan into four parts. Since the algebra plan is built bottomup we start with *table* depict in Fig. 19. *Table* is called two times due to the fact that we have two tables R and S to be accessed - dashed blue arrows highlight that the marked structure is provided by the scope outside of the current `SELECT`...`FROM`...`WHERE`.... `S`...`F`...`W`... denotes the abbreviation of `SELECT`...`FROM`...`WHERE`.... The working range of the rules is shown by dotted lines and their names above and underneath.

However the translation is straighforward to the rule shown in section 3.3.5, so we will not go into details. As expected R and S lead to $loop_1$, Γ_1, $loop_2$ and Γ_2 processed in `FROM` (Fig. 20) to one common *environment* Γ_F. After leaving `FROM` Γ_F and Γ from the outer scope are concatenated to Γ_W forming the input for `WHERE`. `WHERE` is a column inferencerule and as consequence it produces just a column as output, shown by that the connection of $loop_F$ is broken between `WHERE` and `S`...`F`...`W`... in Fig. 21.

The output of `WHERE` is a single column transforming in `S`...`F`...`W`... Γ_F to the input of `SELECT`. `SELECT` simply looks up $R.a$. $Loops_S$ and Γ_S form the output of `S`...`F`...`W`.... Now as the final step, shown in Fig. 22 we translate *SERIALIZE*. Since there is just

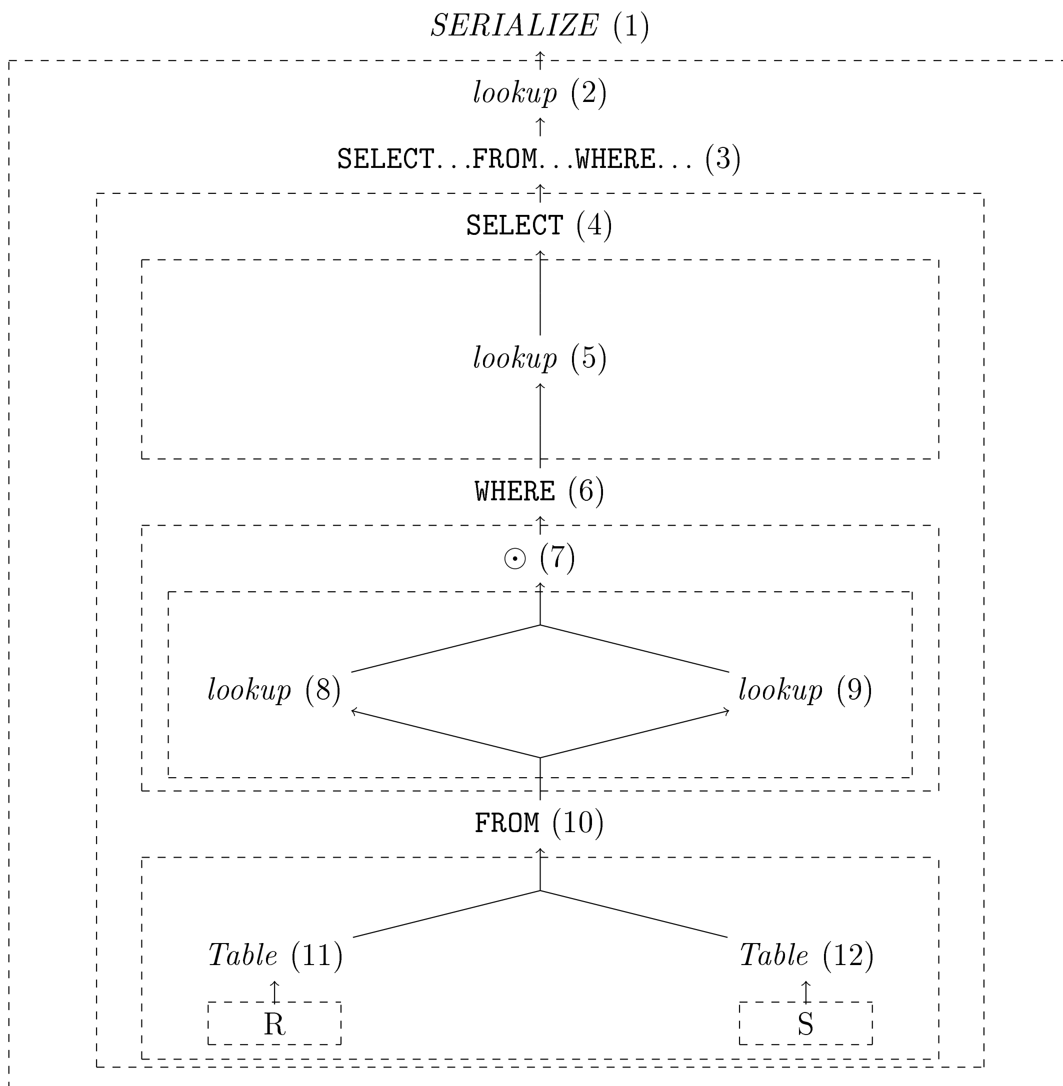

Figure 18: Rules called by Q2

one entry, $R.a$ in Γ' there is not much workload for *SERIALIZE*. Lookup $R.a$, rename and prune column KEY. The overall output q is shown in Fig. 4.

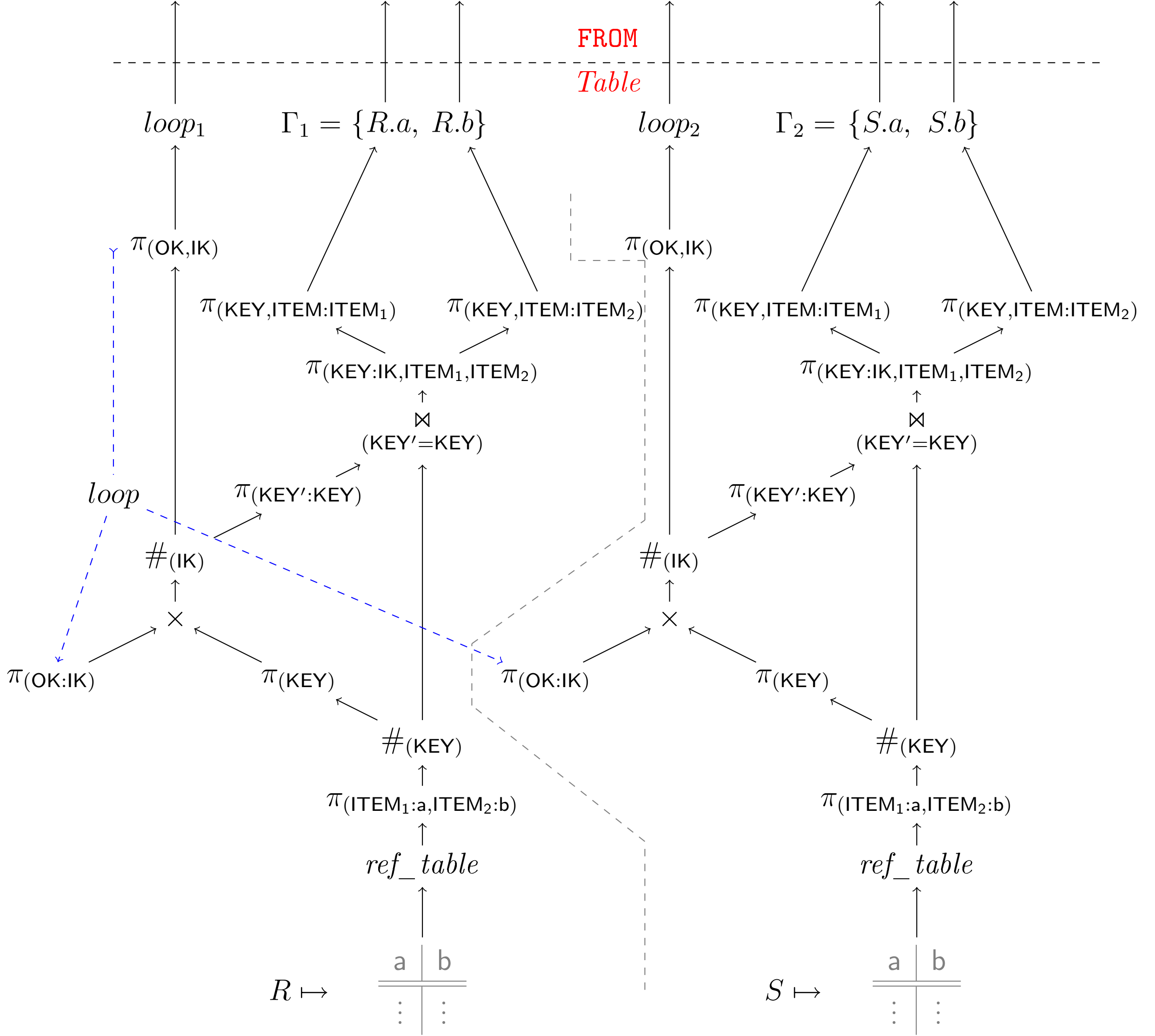

Figure 19: Table called by `FROM` for R and S

WHERE

S. . . F. . . W. . . $\Gamma_W = \{R.a,\ R.b,\ S.a,\ S.b\} ++ \{\}$

$\pi_{(\text{KEY:IK,ITEM})}$

$\bowtie_{(\text{OK=KEY})}$

$\Gamma = \{\}$

S. . . F. . . W. . .

FROM

$loop_F$

$\Gamma_F = \{R.a,\ R.b,\ S.a,\ S.b\}$

$\pi_{(\text{OK,IK})}$ $\pi_{(\text{KEY:IK,ITEM})}$ $\pi_{(\text{KEY:IK,ITEM})}$ $\pi_{(\text{KEY:IK,ITEM})}$ $\pi_{(\text{KEY:IK,ITEM})}$

$\bowtie_{(\text{IK}_1\text{=KEY})}$ $\bowtie_{(\text{IK}_1\text{=KEY})}$ $\bowtie_{(\text{IK}_2\text{=KEY})}$ $\bowtie_{(\text{IK}_2\text{=KEY})}$

$\#_{(\text{IK})}$

$\bowtie_{(\text{OK}'\text{=OK}')}$

$\pi_{(\text{OK,IK}_1\text{:IK})}$ $\pi_{(\text{OK}'\text{:OK,IK}_2\text{:IK})}$

$loop_1$ $\Gamma_1 = \{R.a,\ R.b\}$ $loop_2$ $\Gamma_2 = \{S.a,\ S.b\}$

Figure 20: FROM

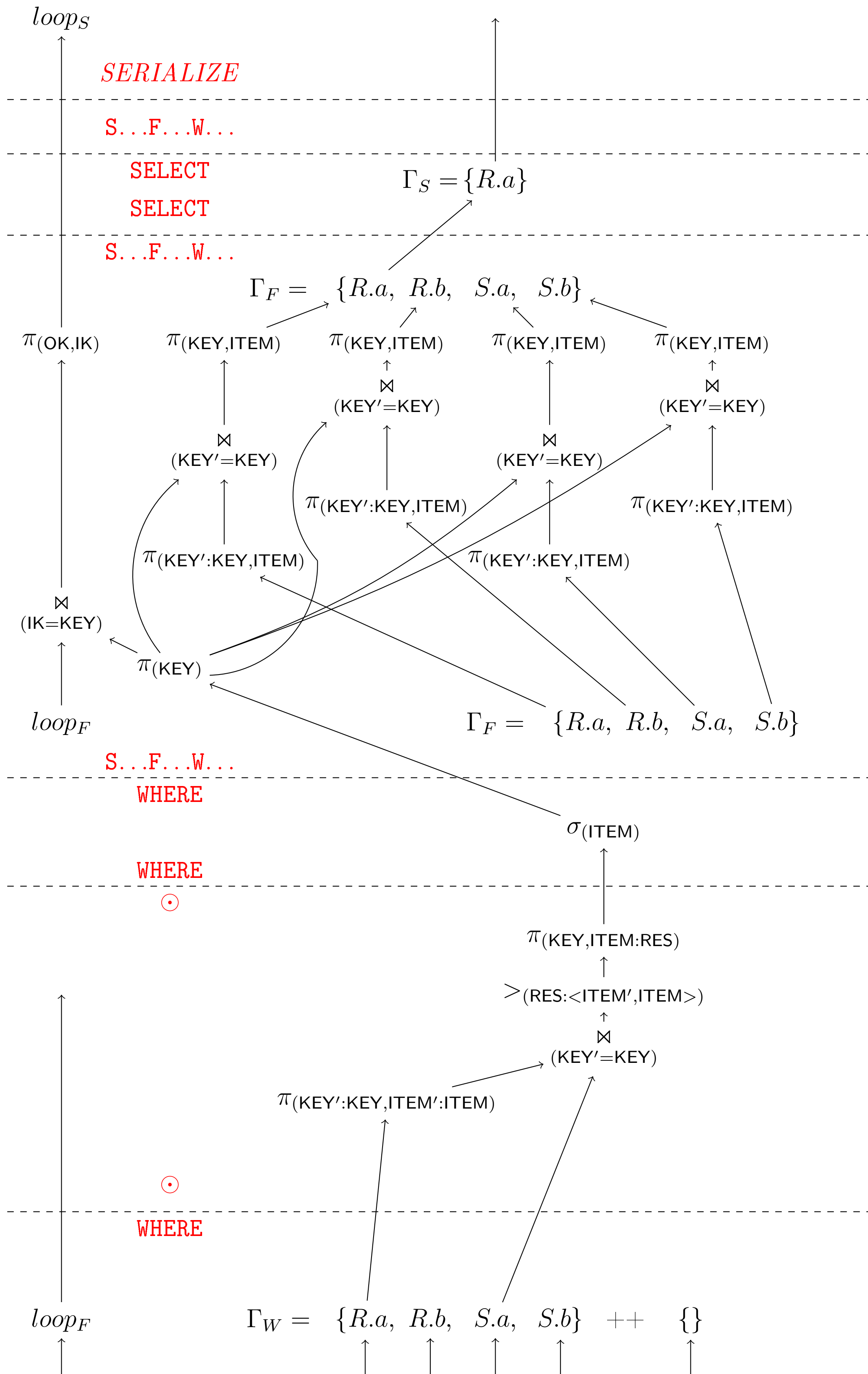

Figure 21: WHERE and ⊙

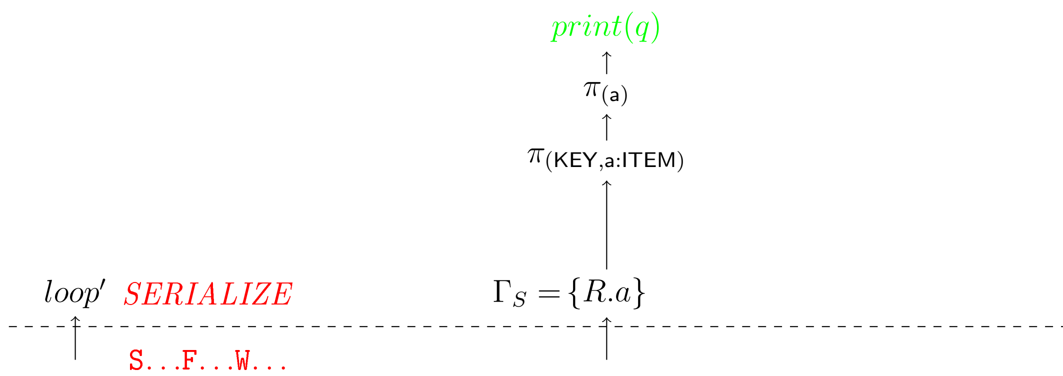

Figure 22: *SERIALIZE*