Translating SQL into the Relational Algebra

Jan Van den Bussche
Stijn Vansummeren

Required background

Before reading these notes, please ensure that you are familiar with (1) the relational data model as defined in Section 2.2 of "Database Management Systems: The Complete Book (second edition)" (hereafter abbreviated as "TCB"); (2) the set-based relational algebra as defined in section 2.4 of TCB; its bag-based variant and extension as defined in sections 5.1 and 5.2 of TCB; and (3) the SQL query language as defined in chapter 6 of TCB.

Throughout these notes we will use the following example database schema about movies, as introduced in TCB Figure 2.5. The attributes of the primary key are underlined.

- Movie(<u>title: `string`, year: `int`</u>, length: `int`, genre: `string`, studioName: `string`, producerC#: `int`)
- MovieStar(<u>name: `string`</u>, address: `string`, gender: `char`, birthdate: `date`)
- StarsIn(<u>movieTitle: `string`, movieYear: `string`, starName: `string`</u>)
- MovieExec(name: `string`, address: `string`, <u>CERT#: `int`</u>, netWorth: `int`)
- Studio(<u>name: `string`</u>, address: `string`, presC#: `int`)

1 Introduction

Translating an arbitrary SQL query into a logical query plan (i.e., a relational algebra expression) is a complex task. In these course notes we try to explain the most important elements of this translation by making the following simplifying assumptions:

- Since the latest version of SQL is a very large and complex language (including features like recursion, stored procedures, user defined functions, . . .) we will focus here on the so-called SQL-92 subset of the language, which can be considered as the traditional "heart" of SQL (comprising only the traditional select-from-where queries, aggregation, etc).

- In addition, we will consider a *set-based* semantics of SQL. Recall that in a set-based semantics, no duplicates occur in input relations or query results. The real semantics of SQL, in contrast, is *bag-based.* In a bag-based semantics duplicates *do* occur in relations and query results. Although the presence or absence of duplicates can often be ignored (as we do in these notes), they *can* lead to different results for queries involving aggregation (e.g., when we want to sum the query results). In practice, therefore, the translation of SQL into a logical query plan is even more involved than described here. It is nevertheless founded on the same principles.

We will use expressions in the *extended* relational algebra (see section 5.2 in the book) interpreted over *sets* as logical query plans.

Provisio To exclude ambiguities, we will assume without loss of generality in what follows that all occurrences of relation symbols in a SQL statement are assigned a distinct name through the alias mechanism of SQL. In other words: we hence assume implicitly that SQL statements in which a relation symbol occurs multiple times, like, for example,

```
SELECT * FROM R, R
```

is rewritten into a SQL statement of the form

```
SELECT * FROM R, R R2
```

in which each occurrence is given a distinct (alias) name. Such rewriting is straightforward to do algorithmically, and can hence be done as a preprocessing step *before* execution of the algorithm described below.

2 Select-from-where statements without subqueries

Consider a general `SELECT-FROM-WHERE` statement of the form

`SELECT` Select-list
`FROM` $R_1, \ldots, R_2\ T_2, \ldots$
`WHERE` Where-condition

When the statement does not use subqueries in its where-condition, we can easily translate it into the relational algebra as follows:

$$\boldsymbol{\pi}_{\text{Select-list}}\, \boldsymbol{\sigma}_{\text{Where-condition}}(R_1 \times \cdots \times \boldsymbol{\rho}_{T_2}(R_2) \times \cdots).$$

Note that:

1. An alias R_2 T_2 in the `FROM`-clause corresponds to a renaming $\boldsymbol{\rho}_{T_2}(R_2)$.
2. It is possible that there is no `WHERE` clause. In that case, it is of course unnecessary to include the selection $\boldsymbol{\sigma}$ in the relational algebra expression.
3. If we omit the projection ($\boldsymbol{\pi}$) we obtain the translation of the following special case:

   ```
   SELECT *
   FROM R1, ..., R2 T2, ...
   WHERE Where-condition
   ```

Example 1. Consider the following `SELECT-FROM-WHERE` statement.

```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND birthdate = 1960
```

Its translation is as follows:

$$\boldsymbol{\pi}_{\texttt{movieTitle}}\,\boldsymbol{\sigma}_{\substack{\texttt{starName}=\texttt{name}\\ \wedge\texttt{birthdate}=1960}}(\texttt{StarsIn} \times \texttt{MovieStar}).$$

□

3 Normalizing Where-subqueries into Exists and Not Exists form

In Section 2 we have considered the special case of translating select-from-where statements in which subqueries do not occur. In general, however, we also have to be able to translate statements in which such subqueries do occur. Subqueries can occur in the `WHERE` clause through the operators `=`, `<`, `>`, `<=`, `>=`, `<>`; through the quantifiers `ANY`, or `ALL`; or through the operators `EXISTS` and `IN` and their negations `NOT EXISTS` and `NOT IN`. We can easily rewrite all of these cases using only `EXISTS` and `NOT EXISTS`, however, as illustrated next.

Example 2. The SQL-statement

```
SELECT movieTitle FROM StarsIn
WHERE starName IN (SELECT name
                   FROM MovieStar
                   WHERE birthdate = 1960)
```

can be rewritten equivalently as

```
SELECT movieTitle FROM StarsIn
WHERE EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate = 1960 AND name = starName)
```

$\square$

Example 3. The SQL-statement

```
SELECT name FROM MovieExec
WHERE netWorth >= ALL (SELECT E.netWorth
                       FROM MovieExec E)
```

can be rewritten equivalently as

```
SELECT name FROM MovieExec
WHERE NOT EXISTS(SELECT E.netWorth
                 FROM MovieExec E
                 WHERE netWorth < E.netWorth)
```

$\square$

Example 4. Consider relations $R(A, B)$ and $S(C)$. Then

```
SELECT C FROM S
WHERE C IN (SELECT SUM(B) FROM R
            GROUP BY A)
```

can be rewritten as

```
SELECT C FROM S
WHERE EXISTS (SELECT SUM(B) FROM R
              GROUP BY A
              HAVING SUM(B) = C)
```

$\square$

Without loss of generality we will hence assume in what follows that all subqueries in the `WHERE` conditions are of the form `EXISTS` or `NOT EXISTS`.

4 Context relations

To translate a query with subqueries into the relational algebra, it seems a logical strategy to work by recursion: first translate the subqueries and then combine the translated results into a translation for the entire SQL statement. If the subqueries contain subqueries themselves, we again translate the latter first — continuing recursively until we reach a level that does not contain subqueries.

For subqueries that do not contain subqueries themselves, we could think that we can simply apply the method from Section 2. There is one complication, however: the subquery can refer to attributes of relations appearing in the `FROM` list of one of the outer lying queries. This is known as *correlated subqueries.*

Example 5. The following query contains a subquery that refers to the `starName` attribute of the outer relation `StarsIn`.

```
SELECT movieTitle
FROM StarsIn
WHERE EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate = 1960 AND name = starName)
```

□

We call the outer relations from which a correlated subquery uses certain attributes *context relations* for the subquery. Note that a subquery can have multiple context relations. We call the attributes of the context relations the *parameters* of the subquery. Note that not all of the parameters must actually occur in the subquery.

Example 6. In Example 5, `StarsIn` is hence a context relation for the subquery. (In this example, it is also the only context relation.) The corresponding parameters are all attributes of `StarsIn`, i.e., `movieTitle`, `movieYear`, and `starName`. □

Translating select-from-where subqueries To translate a `SELECT`–`FROM`–`WHERE` statement that is used as a subquery we must make the following modifications to the method from Section 2:

- We must add all context relations to the cartesian product of the relations in the `FROM` list;
- We must add all parameters as attributes to the projection π.

Example 7. With these modifications, the subquery from Example 5:

```
SELECT name FROM MovieStar
WHERE birthdate = 1960 AND name = starName
```

is translated into

$$\pi_{\texttt{movieTitle},\texttt{movieYear},\texttt{starName},\texttt{name}}\ \sigma_{\substack{\texttt{birthdate}=1960\\ \wedge\texttt{name}=\texttt{starName}}} (\texttt{StarsIn} \times \texttt{MovieStar}).$$

Note how we have added the context relation `StarsIn` to the cartesian product, and how we have added the parameters `movieTitle`, `movieYear` and `starName` (the attributes of `StarsIn`) to the projection π. □

5 De-correlation of subqueries appearing in a conjunctive Where condition

Consider again a general statement of the form

`SELECT` Select-list
`FROM` From-list
`WHERE` Where-condition

in which subqueries may occur in the `WHERE` condition. In the previous two examples we have only translated the *subqueries*. In this section we discuss how we can translate the select-statements in which these subqueries occur. For the time being we will make the following simplifying assumption (it will be lifted in Section 7):

> *The `WHERE`-condition is a conjunction (`AND`) of select-from-where subqueries, possibly with an additional condition that does not contain subqueries.*

In other words, the `WHERE`-condition is of the form

$$\psi \texttt{ AND EXISTS}(Q)\texttt{ AND } \cdots \texttt{ AND NOT EXISTS}(P)\texttt{ AND } \cdots$$

where ψ denotes the subquery-free condition and Q and P are select-statements (possibly with further select-statement subqueries of their own).

The translation proceeds in four steps:

1. Translate the subquery-free part
2. De-correlate the `EXISTS` subqueries
3. De-correlate the `NOT EXISTS` subqueries
4. Apply the projection $\boldsymbol{\pi}_{\text{Select-list}}$

We detail these steps next.

Careful! Our statement contains subqueries, but we must not forget that the statement itself can be a subquery of a containing query!

5.1 Translating the subquery-free part

Consider the subquery-free part of our query, ignore the Select-list:

`SELECT *`
`FROM` From-list
`WHERE` ψ

We will translate it using the method of Section 2, but need to also include the following context relations:

- *When translating the subquery-free part we must include all context relations for which parameters occur in ψ. In addition, we must also include all context relations for which parameters only occur in* `NOT EXISTS` *subqueries.*
- Context relations whose parameters only occur in `EXISTS` subqueries need not be taken into account when translating the subquery-free part.

The relational algebra expression that we hence obtain is of the form

$$\boldsymbol{\sigma}_{\psi}(E),$$

where E is a cartesian product of all relations in the From-list, to which we add context relations for which parameters occur in ψ, or for which parameters occur in some `NOT EXISTS` subquery.

In what follows, we will gradually adapt and refine E when de-correlating the subqueries.

Example 8. Consider the following nested statement over the relations $R(A,B)$ and $S(C)$:

```
SELECT R1.A, R1.B
FROM R R1, S
WHERE EXISTS
  (SELECT R2.A, R2.B
   FROM R R_2
   WHERE R2.A = R1.B AND EXISTS
    (SELECT R3.A, R3.B
     FROM R R3
     WHERE R3.A = R2.B AND R3.B = S.C))
```

Let us denote the entire query by Q_1; the middle subquery by Q_2; and the inner subquery by Q_3. Now assume that we are currently translating Q_2. The subquery-free part of Q_2 is as follows:

```
SELECT *
FROM R R_2
WHERE R_2.A = R_1.B
```

Its translation is hence:

$$\boldsymbol{\sigma}_{R_2.A=R_1.B}(\boldsymbol{\rho}_{R_2}(R) \times \boldsymbol{\rho}_{R_1}(R)) \qquad (*)$$

Note that, although S is a context relation for Q_2, it does not appear in the translation. This is because the parameter $S.C$ does not occur in the subquery-free part of Q_2, but only in the subquery Q_3 itself. Since Q_3 is an `EXISTS` subquery, S does not need to be included in the cartesian product. In contrast, had Q_3 been a `NOT EXISTS` subquery, then we *would* have needed to include S. $\square$

5.2 De-correlating `EXISTS` subqueries

After we have translated the subquery-free part, we translate all subqueries `EXISTS`(Q) in turn. By applying the entire translation algorithm described in these notes recursively to Q, we can already translate Q into a relational algebra expression E_Q.

Example 9. Let us continue the translation of Q_2 from Example 8. We must then first translate Q_3 as follows:

$$\boldsymbol{\sigma}_{\substack{R_3.A=R_2.B\\ \wedge R_3.B=S.C}}(\boldsymbol{\rho}_{R_3}(R) \times \boldsymbol{\rho}_{R_2}(R) \times S)$$

Note that E_{Q_3} query already specifies for which tuples in R_2 and R_3 correct tuples in S exist (together with the values of these tuples). We can use this information to de-correlate Q_2 as follows. $\square$

Let $A_1, \ldots, A_p$ be the list of all parameters of context relations of Q. We can translate `EXISTS`(Q) by joining E with the "space of parameters" for E_Q, namely $\boldsymbol{\pi}_{A_1,\ldots,A_p}(E_Q)$:

$$E := E \bowtie \boldsymbol{\pi}_{A_1,\ldots,A_p}(E_Q).$$

Example 10. Let us continue the translation of Q_2 from Examples 8 and 9. We have so far:

$$\begin{aligned} E &= \boldsymbol{\rho}_{R_2}(R) \times \boldsymbol{\rho}_{R_1}(R) \\ E_{Q_3} &= \boldsymbol{\sigma}_{\substack{R_3.A=R_2.B\\ \wedge R_3.B=S.C}}(\boldsymbol{\rho}_{R_3}(R) \times \boldsymbol{\rho}_{R_2}(R) \times S) \end{aligned}$$

By joining E and E_{Q_3} on the parameters of Q_3 (i.e., $R_2.A$ and $R_2.B$) we ensure that we "link" the correct tuples from E with the correct tuples of E_{Q_3}. In particular, we calculate the tuples in R_1 for which tuples in R_2, R_3, and S exist that satisfy the requirements of Q_2 (together with the values of these tuples).

Actually, we can simplify this expression somewhat. Indeed, note that the following are equivalent because we join R_2 with a subset of itself:

$$(\boldsymbol{\rho}_{R_1}(R) \times \boldsymbol{\rho}_{R_2}(R)) \bowtie \boldsymbol{\pi}_{R_2.A,R_2.B,S.C}\,\boldsymbol{\sigma}_{\substack{R_3.A=R_2.B\\ \wedge R_3.B=S.C}}(\boldsymbol{\rho}_{R_3}(R) \times \boldsymbol{\rho}_{R_2}(R) \times S)$$

and

$$\boldsymbol{\rho}_{R_1}(R) \bowtie \boldsymbol{\pi}_{R_2.A,R_2.B,S.C}\,\boldsymbol{\sigma}_{\substack{R_3.A=R_2.B\\ \wedge R_3.B=S.C}}(\boldsymbol{\rho}_{R_3}(R) \times \boldsymbol{\rho}_{R_2}(R) \times S)$$

This simplification can always be done: before joining with $\boldsymbol{\pi}_{A_1,\ldots,A_p}(E_Q)$, we can remove from E all context relations for Q. Indeed, all relevant tuples for these context relations are already present in the parameter space. Hence, if we denote the omission of the context relations in E by $\hat{E}$, we can summarize the adaption of E as follows:

$$E := \hat{E} \bowtie \boldsymbol{\pi}_{A_1,\ldots,A_p}(E_Q)$$

Example 11. By combining Examples 8 and 9 we can translate Q_2 from Example 8 as follows:

$$\boldsymbol{\sigma}_{R_2.A=R_1.B}(\boldsymbol{\rho}_{R_1}(R) \bowtie \boldsymbol{\pi}_{R_2.A,R_2.B,S.C}\, \boldsymbol{\sigma}_{\substack{R_3.A=R_2.B\\ \wedge R_3.B=S.C}}(\boldsymbol{\rho}_{R_3}(R) \times \boldsymbol{\rho}_{R_2}(R) \times S))$$

Notice that we have been able to remove the relation $\boldsymbol{\rho}_{R_2}(R)$ from the cartesian product $(*)$ that was the translation of the subquery-free part of Q_2 in Example 8. Because of this omission, the $\bowtie$ reduces to a cartesian product. (Indeed, recall that the natural join of two relations without common attributes is equivalent to the cartesian product of those two relations.)

Let us denote the expression above by E_2. The translation of the entire query Q_1 from Example 8 then is as follows.

$$\boldsymbol{\pi}_{R_1.A,R_1.B}(E_2)$$

Here, we have been able to remove both $\boldsymbol{\rho}_{R_1}(R)$ and S from the cartesian product originating from translation of the subquery-free part of Q_1. □

This finishes our discussion on how we can de-correlate subqueries of the form `EXISTS`(Q). We repeat this method for all `EXISTS` subqueries, one by one. For each such subquery, we hence add a join to E, and remove if possible context relations from the cartesian product. *It is important to stress that each relation can only be removed once; if a relation has only been removed when de-correlating a previous subquery, then we cannot remove this relation again when translating the current subquery.*

5.3 De-correlating `NOT EXISTS` subqueries

After de-correlation of the `EXISTS`(Q) subqueries we can de-correlate the `NOT EXISTS`(P) subqueries. Just as in the case of the `EXISTS` subqueries, this will require modifying E. We start by translating P into a relational algebra expression E_P. Again we consider the list $A_1, \ldots, A_p$ of all parameters of context relations of P. Since we now have to express `NOT EXISTS`(P) we do not join E with E_P, but perform an *anti-join* of E with E_P:

$$E := E \,\overline{\bowtie}\, \boldsymbol{\pi}_{A_1,\ldots,A_p}(E_P)$$

Here, the antijoin of two relations R and S is defined as $R-(R \bowtie S)$. In an antijoin, it is required that R contains all attributes of S.

We add such an anti-join to E for every `NOT EXISTS` subquery. Note, however, that in this case we cannot remove context relations from the cartesian product obtained by translating the From-list. (Why not?)

5.4 Translating the select list

Finally, we must apply to E the projection $\boldsymbol{\pi}_{\text{Select-list}}$. If the query that we are translating is used itself as a subquery in an outer lying query, then we must of course take care to add all parameters to Select-list.

6 Operations on relations

SQL-expressions do not simply consist of select-statements, but can also be formed using the operations `UNION`, `INTERSECT`, and `EXCEPT` (possibly in conjunction with the modifier `ALL`); and with the join-operations `CROSS JOIN`, `NATURAL JOIN`, `JOIN ON`, and `OUTER JOIN`. Notice that such expressions may also occur as subqueries. In this section, we discuss how to translate these operations. We begin with the join operations.

6.1 Join operations

Consider first the SQL-expression

$$Q_1 \texttt{ CROSS JOIN } Q_2 \tag{†}$$

where Q_1 and Q_2 are themselves also SQL-expressions. We first translate Q_1 and Q_2 into the relational algebra; which yields relational algebra expressions E_1 and E_2. At first sight we may think that we can then simply translate the expression (†) above as $E_1 \times E_2$.

This translation is incorrect, however, when Q_1 `CROSS JOIN` Q_2 is used as a subquery. Indeed, in that case, Q_1 and Q_2 can be correlated subqueries that may have parameters in common. In order to be able to continue using the translation strategy from Section 5, we need to synchronize the correlated subqueries on the common parameters. This can be done using a natural join over the common parameters, simply taking:

$$E_1 \bowtie E_2$$

instead of $E_1 \times E_2$. (Since, according to the SQL specification, the operands of a `CROSS JOIN` must have disjoint sets of attributes, any attributes common to both E_1 and E_2 must be parameters. Hence, it is not necessary to specify that only the parameters should be equal using a theta-join. A natural join suffices.)

Example 12. To illustrate this discussion, consider the relations $R(A, B)$ and $S(C)$, as well as the following query.

```
SELECT S1.C, S2.C
FROM S S1, S S2
WHERE EXISTS (
  (SELECT R1.A, R1.B FROM R R1
   WHERE A = S1.C AND B = S2.C)
  CROSS JOIN
  (SELECT R2.A, R2.B FROM R R2
   WHERE B = S1.C)
)
```

Let us translate its `EXISTS` subquery containing the cross join. The translations of Q_1 and Q_2 are as follows.

$$E_1 = \boldsymbol{\pi}_{S_1.C,S_2.C,R_1.A,R_1.B}\,\boldsymbol{\sigma}_{\substack{A=S_1.C\\ \wedge B=S_2.C}}\,(\boldsymbol{\rho}_{R_1}(R) \times \boldsymbol{\rho}_{S_1}(S) \times \boldsymbol{\rho}_{S_2}(S))$$
$$E_2 = \boldsymbol{\pi}_{S_1.C,R_2.A,R_2.B}\,\boldsymbol{\sigma}_{B=S_1.C}\,(\boldsymbol{\rho}_{R_2}(R) \times \boldsymbol{\rho}_{S_1}(S))$$

Notice that Q_1 and Q_2 have one context relation in common, namely S_1. The translation of the `EXISTS` subquery is then:

$$E_1 \bowtie E_2.$$

Notice that the natural join occurs only on the common parameter $S_1.C$. $\square$

In a similar manner we can translate the other SQL join-operations. We only have to take care to use the correct algebra operator in the translation: $\bowtie$ for `NATURAL JOIN`; theta-join for `JOIN ON`; and outer join $\overset{o}{\bowtie}$ for `OUTER JOIN`.

Example 13. Here is a simple example of a theta-join. The join expression is not a correlated subquery, and hence we do not need to take into account context relations.

```
Movie JOIN StarsIn ON
  title = movieTitle AND year = movieYear
```

It is translated as follows:

$$\texttt{Movie} \underset{\substack{\texttt{title=movieTitle}\\ \wedge\texttt{year=movieYear}}}{\bowtie} \texttt{StarsIn}.$$

$\square$

6.2 Union, Intersect and Except

Next consider a SQL-expression

$$Q_1 \texttt{ UNION } Q_2 \qquad (\ddagger)$$

As before, we first translate Q_1 and Q_2 into the relational algebra-expressions E_1 and E_2. Unfortunately, however, we cannot simply translate ‡ by $E_1 \cup E_2$: in the relational algebra, both operands of a union must have the same set of attributes in their schema (see Section 2.2 in TCB). When Q_1 and Q_2 have different context relations, they will also have different sets of parameters. In that case, E_1 and E_2 will have different sets of attributes in their schema.

Example 14. Let us reconsider the query from Example 12. Suppose that we replace the `CROSS JOIN` by `UNION`. We cannot simply take the union $E_1 \cup E_2$: E_1 has four attributes, whereas E_2 only has three. □

The solution consists of "padding" E_1 with the context relations of Q_2 and, vice versa, "padding" E_2 with the contex relations of Q_1. This padding is done by means of a cartesian product. Formally, let:

- $V_1, \ldots, V_m$ be the context relations of Q_1 that do not occur in Q_2;
- $W_1, \ldots, W_n$ be the context relations of Q_2 that do not occur in Q_1.

We then translate SQL-expression (‡) as:

$$\boldsymbol{\pi}_{\ldots}(E_1 \times W_1 \times \cdots \times W_n) \cup \boldsymbol{\pi}_{\ldots}(E_2 \times V_1 \times \cdots \times V_m)$$

Here, the projections $\boldsymbol{\pi}_{\ldots}$ need to ensure that the parameters of a context relation occur in the same column position in both subexpressions, and that the other attributes have the same name.

Example 15. Consider the translation of the `EXISTS` subquery from Example 12, but with `UNION` instead of `CROSS JOIN`. There is only one V, namely S_2. There are no W's because the only context relation of Q_2 is S_1, which is also a context relation of Q_1. The translation of the `EXISTS` subquery is hence:

$$\boldsymbol{\pi}_{S_1.C,S_2.C,R_1.A\to A,R_1.B\to B}(E_1) \cup \boldsymbol{\pi}_{S_1.C,S_2.C,R_2.A\to A,R_2.B\to B}(E_2 \times \boldsymbol{\rho}_{S_2}(S))$$

□

In a similar manner we can translate `INTERSECT` and `EXCEPT` using $\cap$ instead of $\cup$ for `INTERSECT` and $-$ instead of $\cup$ for `EXCEPT`.

Example 16. Here is a simple example of an `EXCEPT` expression. This is not a correlated subquery, and hence there is no need to take context relations into account.

```
(SELECT name, address from MovieStar)
  EXCEPT
(SELECT name, address from MovieExec)
```

Its translation is

$$(\boldsymbol{\pi}_{\texttt{name,address}}(\texttt{MovieStar}) - \boldsymbol{\pi}_{\texttt{name,address}}(\texttt{MovieExec}))\,.$$

□

7 The non-conjunctive case

How can we translate `WHERE`-conditions that contain subqueries, but that are not of the conjunctive form required in Section 5? We first note that we can always rewrite any `WHERE` condition as a disjunction (`OR`) of the form:

$$\varphi_1 \texttt{ OR } \varphi_2 \texttt{ OR } \ldots$$

where every φ_i is a conjunction (`AND`) of subqueries, possibly with an additional subquery-free condition.

Rewriting an arbitrary `WHERE` condition into this so-called *disjunctive normal form* can be done by repeatedly applying the laws of De Morgan[1] and distributivity[2] of `AND` over `OR`.

Example 17.

- The `WHERE` condition

  `NOT EXISTS` Q `AND (A=B OR C<6)`

  with Q a subquery is already in normal form: it is a conjunction of a subquery and a condition without subqueries. In this case there is hence only one conjunction: the entire condition itself.

- The `WHERE` condition

  `A=B OR EXISTS` Q

  is also in normal form. In this case there are two conjunctions, each consisting of a single atomic condition.

- The `WHERE` condition

  `A=B AND NOT(EXISTS` Q `AND C<6)`

  is not in normal form. We can rewrite it into the desired form by one application of De Morgan followed by one application of distributivity:

  `A=B AND (NOT EXISTS` Q `OR C>=6)`
  `(A=B AND NOT EXISTS` Q`) OR (A=B AND C>=6)`

□

To translate an arbitrary select-statement we hence first rewrite its `WHERE` condition into disjunctive normal form:

[1] $\neg(p \lor q) = \neg p \land \neg q$, $\neg(p \land q) = \neg p \lor \neg q$.
[2] $(p \lor q) \land r = (p \land r) \lor (q \land r)$

```
SELECT Select-list
FROM From-list
WHERE $\varphi_1$ OR $\varphi_2$ OR ...
```

We then treat the disjunction as a union:

```
(SELECT Select-list  UNION  (SELECT Select-list  UNION ...
 FROM From-list              FROM From-list
 WHERE $\varphi_1$)                    WHERE $\varphi_2$)
```

Since every select-statement subquery of this union has a conjunctive `WHERE` condition φ_i, we can translate these subqueries using the method from paragraph 5. The `UNION` operations themselves can then be translated as discussed in Section 6.2.

8 Group by and Having

Up until now we have only considered select-from-where statements without aggregation. In general, however, a select-statement can also contain `GROUP BY` and `HAVING` clauses:

```
SELECT Select-list
FROM From-list
WHERE Where-condition
GROUP BY Group-list
HAVING Having-condition
```

Note that such a statement can also occur as a subquery! To translate such a statement, we first translate its `FROM–WHERE` part:

```
SELECT *
FROM From-list
WHERE Where-condition
```

Let E be the relational algebra expression hence obtained. (Note in particular that the where-condition may contain subqueries in non-conjunctive form, in which case we have to apply the techniques from Section 7 to obtain E). Let $A_1, \ldots, A_n$ be the parameters of the statement (if it occurs as a subquery). The translation of the entire statement then is the following.

$$\boldsymbol{\pi}_{A_1,\ldots,A_n,\text{Select-list}}\,\boldsymbol{\sigma}_{\text{Having-condition}}\,\boldsymbol{\gamma}_{A_1,\ldots,A_n,\text{Group-list},\text{Agg-list}}(E).$$

Here, 'Agg-list' consists of all aggregation operations performed in the Having condition or Select-list. If the `HAVING` clause is absent, then the $\boldsymbol{\sigma}$ can be omitted.

Example 18. The SQL statement

```
SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE cert# = producerC#
GROUP BY name
HAVING MIN(year) < 1930
```

is thus translated into

$$\pi_{\texttt{name,SUM(length)}}\,\sigma_{\texttt{MIN(year)}<1930}\,\gamma_{\texttt{name,MIN(year),SUM(length)}}\\ \sigma_{\texttt{cert\#=producerC\#}}(\texttt{MovieExec} \times \texttt{Movie}).$$

□

Example 19. The subquery from Example 4:

```
SELECT SUM(B) FROM R
GROUP BY A
HAVING SUM(B) = C
```

is translated into

$$\pi_{C,\texttt{SUM}(B)}\,\sigma_{C=\texttt{SUM}(B)}\,\gamma_{C,A,\texttt{SUM}(B)}(R \times S).$$

Notice how we have added the context relation S to the cartesian product, and how we have added the parameter C to the operators γ and π. □

9 Subqueries in the From-list

Subqueries in SQL can not only occur in the `WHERE` condition of a query, but also in the `FROM` list.

Example 20. The following statement contains a subquery in its From-list:

```
SELECT movieTitle
FROM StarsIn, (SELECT name FROM MovieStar
               WHERE birthdate = 1960) M
WHERE starName = M.name
```

□

The translation of such `FROM`-subqueries is straightforward. Just like we normally translate a From-list without subqueries by means of a cartesian product:

$$\texttt{FROM}\ R_1,\ R_2,\ \dots \quad \Rightarrow \quad (R_1 \times R_2 \times \cdots)$$

we translate a From-list with subqueries analogously:

$$\texttt{FROM}\ (Q_1),\ (Q_2),\ \dots \quad \Rightarrow \quad (E_1 \times E_2 \times \cdots)$$

Here, E_1, E_2, ..., are the relational algebra translations of Q_1, Q_2, ...

Example 21. We hence translate the query from Example 20 as follows:

$$\pi_{\texttt{movieTitle}} \sigma_{\texttt{starName}=\texttt{M.name}}(\texttt{StarsIn} \times \rho_{\texttt{M}} \pi_{\texttt{name}} \sigma_{\texttt{birthdate}=1960}(\texttt{MovieStar})).$$

□

9.1 Lateral subqueries in SQL-99

SQL-99 also allows `FROM`-subqueries to be correlated with relations or subqueries that precede it in the From-list.

Example 22. The following statement is legal in SQL-99 (but not in SQL-92). It uses the keyword `LATERAL` to indicate that a subquery is correlated.

```
SELECT S.movieTitle
FROM (SELECT name FROM MovieStar
      WHERE birthdate = 1960) M,
     LATERAL (SELECT movieTitle
              FROM StarsIn
              WHERE starName = M.name) S
```

□

To translate such subqueries it suffices to replace the cartesian product (which we normally use to translate From-lists) by a natural join of the parameters. It is important to note, however, that context relations need no longer be base relations, but can themselves also be translations of earlier subqueries in the From-list. Just as with `EXISTS` subqueries, we can omit preceding context relations if those are already contained in the parameters space of a subsequent subquery.

Example 23. We translate the statement from Example 22 as follows. We start by translating the first subquery:

$$E_1 = \pi_{\texttt{name}}\, \sigma_{\texttt{birthdate}=1960}(\texttt{MovieStar}).$$

We then translate the second subquery, which has E_1 as a context relation:

$$E_2 = \pi_{\texttt{name,movieTitle}}\, \sigma_{\texttt{starName}=\texttt{name}}(\texttt{StarsIn} \times E_1).$$

We would then translate the entire `FROM` clause by means of a join and not by means of a cartesian product (because of the correlation):

$$\pi_{\texttt{movieTitle}}(E_1 \bowtie E_2).$$

In this example, however, all relevant tuples from E_1 are already contained in E_2, and hence we can omit E_1. The obtained end result is then $\pi_{\texttt{movieTitle}}(E_2)$. □

10 Subqueries in the Select-list

Finally, SQL also allows subqueries in the Select-list; the so-called *scalar* subqueries. These subqueries must always return a single value. They can nevertheless be correlated. In that case they must hence return a single value for each possible assignment of values to the parameters.

Example 24. Consider the relations $R(A, B)$ and $S(C)$, and assume that A is a key for R. Then the following query is allowed in SQL.

```
SELECT C, (SELECT B FROM R
           WHERE A=C)
FROM S
```

Consider in addition a relation $T(D, E)$. The following is also allowed.

```
SELECT D, (SELECT B FROM R
           WHERE A=D AND B>SUM(E))
FROM T
GROUP BY D
```

□

Scalar subqueries in the select-list can be rewritten as `LATERAL` subqueries in de From-list. Since we already know how to translate the latter, we hence also know how to translate the former. Formally, the rewriting of subqueries in the select-list goes as follows. Consider a general select-statement with a subquery in its select-list.

```
SELECT Select-list, (Q)
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
```

Since Q must be a scalar subquery, it can only have one output attribute, say attribute A. We then rewrite as follows:

```
SELECT Select-list, T.A
FROM (SELECT *
      WHERE ...
      GROUP BY ...
      HAVING ...),
LATERAL (Q) T
```

If the subquery has parameters that are aggregation operators (such as, e.g., the parameter `SUM(E)` in the second expression of Example 24) we need to fine-tune this rewriting by including these aggregations to the γ operator in the translation of the first subquery (which corresponds to the original statement).

Remark: In practice a query processor need not implement subqueries in the select list in this manner. It is often more efficient to leave the subquery correlated, and to execute it separately for each possible assignment of values to the parameters. In this way, it is also easier to verify that the subquery always returns only in a single value.

11 The Count bug

There is a type of subqueries for which the translation algorithm above is incorrect, namely the subqueries that use `COUNT` without an accompanying `GROUP BY`. To illustrate, consider the following example query over the relations $R(A, B)$ and $T(A, C)$:

```
SELECT T.A FROM T
WHERE T.C = (SELECT COUNT(R.B) FROM R
             WHERE R.A=T.A)
```

If T contains the tuple $(5, 0)$ and R contains no tuple with $A = 5$, then the output of the query certainly contains the value 5. Let us translate this query into the relational algebra. First, we rewrite the subquery into the `EXISTS` form:

```
SELECT T.A FROM T
WHERE EXISTS (SELECT COUNT(R.B) FROM R
              WHERE R.A=T.A
              HAVING COUNT(R.B) = T.C)
```

The relational algebra translation is then:

$$\boldsymbol{\pi}_{T.A}\, \boldsymbol{\sigma}_{\texttt{COUNT}(R.B)=T.C}\, \boldsymbol{\gamma}_{T.A,T.C,\texttt{COUNT}(R.B)}\, \boldsymbol{\sigma}_{R.A=T.A}(R \times T)$$

If R contains no tuple with $A = 5$, then the section $\boldsymbol{\sigma}_{R.A=T.A}$ will eliminate the T-tupel $(5, 0)$. Hence, in that case, 5 will *not* occur in the output, which is incorrect.

This problem is well-known and only poses itself with these kinds of subqueries. The solution here consists in replacing the cartesian product with the context relations (in our example: T) by an outer join. This way, the parameters for which the subquery yields an empty results are still retained in the parameter space. Concretely, our example subquery would then be translated as

$$\boldsymbol{\pi}_{T.A}\, \boldsymbol{\sigma}_{\texttt{COUNT}(R.B)=T.C}\, \boldsymbol{\gamma}_{T.A,T.C,\texttt{COUNT}(R.B)}\, (R \underset{R.A=T.A}{\overset{\mathrm{o}}{\bowtie}_{\mathrm{R}}} T).$$

(Here, $\overset{\mathrm{o}}{\bowtie}_{\mathrm{R}}$ denotes the right outer join.) On our hypothetical database, the outer join produces a tuple

$$(T.A = 5,\ T.C = 0,\ R.A = \bot,\ R.B = \bot).$$

If we then implement the aggregation operator γ in such a way that the `COUNT` of a single $\perp$ equals 0, the evaluation proceeds correctly.