Local Logical Query Optimization
for a Relational Database

3rd February 2004
Annamaria Siegler
Abstract

The aim of the query evaluation in relational data base systems consists of generating an optimal execution plan for a given query. The first part of developing execution plans is the logical optimization followed by the physical optimization where an execution plan is provided and selected.

This project presents an introduction to the local logical query optimization.

At logical optimization equivalence rewritings are accomplished by transformation rules. These transformation rules must be examined, whether they are always an improvement or heuristically an improvement for query rewriting and if they are useful or not. But they can also be classified concerning their rewriting algorithm. At the end an overview is given about the algorithm, implementation and code file structure of the local logical optimizer.
# Table of contents

Chapter 1 ......................................................................................................................... 4  
  Introduction .................................................................................................................... 4  
  1.1 Aim of this project .................................................................................................... 5  
  1.2 Structure of this project ......................................................................................... 5  

Chapter 2 ......................................................................................................................... 6  
  Logical optimization ...................................................................................................... 6  
  2.1 Example for logical optimization .......................................................................... 6  
  2.2 General transformation rules for relational algebra operations: ......................... 8  
      Category 1: always an improvement ......................................................................... 8  
      Category 2: nearly always an improvement .............................................................. 9  
      Category 3: heuristically probably an improvement (rule-of-thumb) ...................... 10  
      Category 4: heuristically an improvement but need to check at meta DB ............ 11  

Chapter 3 ......................................................................................................................... 12  
  Classification of the rules concerning the re-writing algorithms ............................... 12  

Chapter 4 ......................................................................................................................... 20  
  4.1 Purpose .................................................................................................................. 20  
  4.2 Overview of the code file structure and algorithms ............................................... 20  
      4.2.1 Algorithm Details ......................................................................................... 21  
  4.3 Advancement of the optimizer .............................................................................. 22
Chapter 1

Introduction

Queries, which are requested to a data base system, define what the user wanted to know, but not, how the system is to determine the wanted information. There are many different possibilities to evaluate a query and to choose the right one, a query optimization should take place as early as possible in query processing.

At first the typical steps in query processing:

**Fig 1.1: Process of query optimization**

In the first step the request, the user entered, is brought into an internal form by a Parser, which is then validated with help of a Data Dictionary, in order to guarantee the fact that the request contains only permitted and validated references on existing data base objects.

The resulting algebraic expression is then submitted of an query optimization, which is divided into algebraic, physical optimization, whereby the transiton of the implementation are flowing.

The algebraic optimization consists of using arithmetic rules of relation algebra that is presented later.

Afterwards the physical optimization takes place under help of concrete storage and accesses techniques (e.g. indices and cluster) and use of alternative implementation for algebraic operations.

At the end of this phase several semantically equivalent access-plans are passed on, which suggest a smaller expenditure. This access-plan is estimated and eventually modified at the cost-based optimization by cost estimation. Afterwards the optimal plan is selected.
At the optimization of requests in classical relational systems two problem areas can be identified:
(a) On the one hand in the algebraic and physical optimization from all kinds of evaluation-plans must be extracted a small number of plans, which promise minimum costs at the evaluation, but as possible without omit very good plans;
(b) on the other hand the for the cost-based optimization must be developed a procedure, which helps to estimate the evaluated costs of a plan exactly, without actually implement the request.

There are three different times, on which the optimizations can be take place:
(a) statically: Here the optimization is done before the realisation.
(b) dynamic: In this case optimization takes place “during” the implementation.
(c) hybrid: Combined insert of the two above procedures.

1.1 Aim of this project

The aim of this work is the development of an optimizing generator (rule-based algorithm), that applies certain laws/rules of transformation to rewrite queries in a most efficient way and producing the result of the query.

1.2 Structure of this project

The following chapters are arranged as follows:
In Chapter 1 a short introduction is given into the topic of query optimization. Afterwards follows in Chapter 2 a representation of the rewriting rules of the relation algebra, which thereupon serve as basis for query heuristics. Chapter 3 shows a classification of the rewriting rules concerning their rewriting algorithms. At least Chapter 4 gives an overview about the implementation process of the local optimizer.
Chapter 2

Logical optimization

Logical optimization is a high-level-optimization. At this an expression transformation ("query rewriting") takes place, that means an optimizer applies certain laws/rules of transformation to rewrite queries in a most efficient way and produce the result of the query. The new query version is logically equivalent but can be executed more quickly.

A non-algebraic query like SQL must be transformed at first into some internal representation, which is still equivalent to the original query but is more suitable for machine manipulation. An internal representation of the query is typically some kind of abstract syntax tree or a query tree or a query based on the relational algebra.

In this case the query language RAQUEL is used. It is formulated according to the principles of algebra and there is no need of transformation into standard forms any more. It is parsed and developed into a query tree.

Next an example of local optimization by showing how to use rewriting rules:

2.1 Example for logical optimization

- Given are the relation $R1, R2$

<table>
<thead>
<tr>
<th>$R1$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R2$</th>
<th>$C$</th>
<th>$D$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>z</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>x</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>y</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- The query which is worked on is:

$((R1 \text{ CProd } R2) \text{ Restrict}[R1.A = c \text{ AND } R2.E = 2 \text{ AND } R1.C = R2.C]) \text{ Project}[D, B]$

- With the following result:

$\begin{array}{c|c}
D & B \\
\hline
x & 2 \\
\end{array}$

- query tree:

```
Project[D, B]
```

```
```

```
CProd
```

```
R1
```

```
R2
```

In the beginning the Cartesian Product is formed by $R1$ and $R2$, thereafter the tuple required are Restricted. Finally the required Projection is done.
Altogether \((1)+(25)+(5+5\times 5)=56\) tuple are worked on. For the Cartesian Product each tuple of relation \(R1\) is read once, in order then connected with each tuple from \(R2\). That means that the 5 tuple read in \(R1\) once and everyone of the 5 tuple in \(R2\) is so often read, like \(R1\) gives (thus 5 times). That means 5 tuple accesses \(5+5\times 5\) for the product. An intermediate relation with 25 tuple develops. At the restriction everyone is once worked on this tuple, further 25 accesses. The next intermediate result contains only one tuple, on this used the Projection, so that still another tuple access is added.

- Algebraic optimized results after the following transformations:

\[
\begin{align*}
((R1 \times R2) \text{Restrict}([R1.A=c \text{ AND } R2.E=2 \text{ AND } R1.C=R2.C]) \text{ Project}[D, B] \\
\equiv((R1 \text{ Restrict}([R1.C=R2.C]) \text{ Restrict}([R1.A=c \text{ AND } R2.E=2]) \text{ Project}[D, B] \\
\equiv((R1 \text{ Join } R2) \text{ Restrict}([R1.A=c \text{ AND } R2.E=2]) \text{ Project}[D, B] \\
\equiv((R1 \text{ Restrict}([R1.A=c])) \text{ Join } (R2 \text{ Restrict}([R2.E=2])) \text{ Project}[D, B]
\end{align*}
\]

- Query tree:

```
Project[D, B]
  /   \
Join
   /  \
Restrict[R1.A=c]  Restrict[R2.E=2]
   R1           R2
```

Here Restrictions are shifted to the leafs and the Cartesian Product is replaced by a Join operation.

In this case \((1)+(1+1\times 3)+(5+5)=15\) tuple are worked up. At first to each relation \(R1\) and \(R2\) a Restriction is applied. Therefor must be read in each relation each tuple once, result is each 5 thus \(10=5+5\) accesses. The intermediate result of \text{Restrict}[R1]\) contains only one tuple, \text{Restrict}[R2]\) however 3. To Join the two Restrictions the result is \(4=1+1\times 3\) accesses. In addition still one comes for the Projection of the only tuple in the developed intermediate result.

We see a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. The single steps were executed until no replacements are left. Through the algebraic transformation of the query tree thus the number of tuple accesses could be reduced by approx. 73%. With this computation it becomes also evident that the size of the intermediate result represents an important indication for optimization.

Relating to the rules which are chosen and the implementation strategy is usually no guarantee that is truly optimal; it might in fact be so. Some rewriting is always going to be an improvement and some (heuristic) usually are, but there may need cost estimates.

With help of transformation rules, rules-of-thumb can be formulated which are meaningful with nearly all requests.

An important rule-of-thumb, already mentioned, is to hold all intermediate results as small as possible because the size of the intermediate result has a very large influence on the operating time. At this it is e.g. exploit that Restriction and Projection make generally provisional results smaller.

Restrictions reduce the number of the tuple, whereas Projection at first shortened the length of the individual tuples and reduced in the case of a duplicate elimination some additional tuple.
2.2 General transformation rules for relational algebra operations:

For optimization of relational expressions we need some transformation rules to convert these into some equivalent form. They are categorised into different kind of improvements.

\[ R, R_1, R_2, \ldots \quad \text{Relations} \]
\[ q, p, p_1, p_2, p_3, \ldots \quad \text{Conditions} \]
\[ l, l_1, l_2, \ldots \quad \text{Set of attributes} \]

Operations:

- **Intersect**: Intersection of two relations
- **Union**: Union of two relations
- **Diff**: Difference of two relations
- **CProd**: Cartesian product of two relations
- **Project\[l\]**: Projection of a relation to the set of attributes \( l \)
- **Restrict\[p\]**: Restrict of a relation to the condition \( p \)
- **Join\[p\]**: Join of two relations to the condition \( p \)

Algebraic transformation rules and their use:

**Category 1: always an improvement**

**Rule 1:** *Converting/Replacing* a Restrict with Product sequence into Join: If the condition \( p \) of a Restrict that follows a Product corresponds to a Join condition, convert the Restrict and Product into a Join:

\[(R_1 \ CProd \ R_2) \ \text{Restrict}[p] \equiv R_1 \ \text{Join}[p] \ R_2\]

Using this rule, combine a Cartesian Product operation with a subsequent Restrict operation in the tree into a Join operation, if the condition represents a join condition.

i.e. a query tree:

```
          | Restrict[p]
          |           
          |           
          |           
          |           
          |           
          |           
          |           
          |           
          |           
          |           
CProd

            | Join[p]
            |
            |
            |
            |
            |
            |
            |
            |
            |
            |
            |
            |
            |
            |
R1           R2
```

**Rule 4:** *Sequence* of Projection: In a sequence of projection operations, all but the last one can be ignored, if \( l_1 \subseteq l_2 \subseteq \ldots \subseteq l_n \subseteq \text{set of attributes of } R \):

\[ (((R \ \text{Project}[l_1]) \ldots) \ \text{Project}[l_2]) \ \text{Project}[l_2] \equiv R \ \text{Project}[l_2] \]

Projections shorted the length of the individual tuple and reduce in the case of duple elimination additional the number of the tuple.
Rule 7: Sequence of Restrict: A conjunctive Restrict conditions can be broken up into a cascade of individual Restrict operations respectively a successive executed Restrict operations can be coalesced by conjunctions:

\[ ((...((R \text{Restrict}[p_n])...) \text{Restrict}[p_2]) \text{Restrict}[p_1]) \equiv R \text{Restrict}[p_1 \text{ AND } p_2 \text{ AND } ... \text{AND } p_n] \]

i.e query tree:  
\[
\text{Restrict}[p_1] \rightarrow \text{Restrict}[p_1 \text{ AND } p_2 \text{ AND } p_3 \text{ AND } ... \text{AND } p_n] \\
\text{Restrict}[p_2] \\
\text{Restrict}[p_n]
\]

In a conjunctive Restrict operation the conditions must be checked at first. If any of the condition is wrong, so the whole term is going to be false and then there is no need to execute the Restrict operation any more.

Category 2: nearly always an improvement

At category 2 in some cases attributes or conditions of rules must be checked at first in the meta DB before they are useful for optimization.

Rule 3: Commuting Projection with a set operation: Suppose the Projection list is \( l=\{l_1,\ldots,l_n; r_1,\ldots,r_m\} \), where \( l_1,\ldots,l_n \) are attributes of \( R_1 \) and \( r_1,\ldots,r_m \) are attributes of \( R_2 \). The Projection operation commutes with union:

\( (R_1 \text{Union } R_2) \text{ Project}[l] \equiv (R_1 \text{ Project}[l_1,\ldots,l_n]) \text{ Union } (R_2 \text{ Project}[r_1,\ldots,r_m]) \)

By using these rules concerning the commuting of Project with other operations, break down and move list of projection attributes down the query tree as far as possible by creating new Project operations as needed. Only those attributes needed in the query result and in subsequent operations in the tree should be kept after each Project operation. This reduces the attributes (columns) of the intermediate relation, whereas the restrict operation reduce the number of tuples (records).

i.e a query tree:  
\[
\text{Project}[l] \text{Union} \\
\text{Union} \\
R_1 \rightarrow \text{Project}[l_1,\ldots,l_n] \text{Project}[r_1,\ldots,r_m] \\
R_2 \text{R1} \rightarrow \text{R2}
\]

Rule 5: Commuting Restrict with Join or Product: If all the attributes in the Restrict condition \( p \) involves only the attributes of one of the relations being joined (say \( R \)) the two operations can be commuted as:

\( (R_1 \text{ Join } R_2) \text{ Restrict}[p] \equiv (R_1 \text{ Restrict}[p]) \text{ Join } R_2 \)
\( (R_1 \text{ CProd } R_2) \text{ Restrict}[p] \equiv (R_1 \text{ Restrict}[p]) \text{ CProd } R_2 \)

If Restriction is done at first, mostly it reduces the number of tuples.
Rule 10: Alternatively, if the Restrict condition \( p \) can be written as \((p0 \text{ and } p1 \text{ and } p2)\), where condition \( p1 \) involves only the attribute of \( R1 \) and condition \( p2 \) involves only the attributes of \( R2 \), the operation commutes as:

\[
\begin{align*}
(R_1 \Join R_2) \text{ Restrict}[p] &= ((R_1 \text{ Restrict}[p1]) \Join (R_2 \text{ Restrict}[p2])) \text{ Restrict}[p0] \\
(R_1 \text{ CProd } R_2) \text{ Restrict}[p] &= ((R_1 \text{ Restrict}[p1]) \text{ CProd } (R_2 \text{ Restrict}[p2])) \text{ Restrict}[p0]
\end{align*}
\]

Rule 6: Commuting Restrict with set operations: The Restrict operation commutes with union, intersection and difference:

\[
\begin{align*}
(R_1 \text{ Union } R_2) \text{ Restrict}[p] &= (R_1 \text{ Restrict}[p]) \text{ Union } (R_2 \text{ Restrict}[p]) \\
(R_1 \text{ Intersect } R_2) \text{ Restrict}[p] &= (R_1 \text{ Restrict}[p]) \text{ Intersect } (R_2 \text{ Restrict}[p]) \\
(R_1 \text{ Diff } R_2) \text{ Restrict}[p] &= (R_1 \text{ Restrict}[p]) \text{ Diff } (R_2 \text{ Restrict}[p])
\end{align*}
\]

i.e a query tree:

\[
\text{Restrict}[p] \quad \text{Diff} \\
\Downarrow \quad \Downarrow \\
R_1 \quad R_2
\rightarrow
\begin{align*}
\text{Restrict}[p] \\
\Downarrow \\
R_1
\end{align*}
\]

Using these rules concerning commutativity of Restrict with other operations, move each Restrict operation down the query tree as is permitted by the attributes/conditions involved in the restrict condition. To do a Restriction at first it permits a smaller result.

Category 3: heuristically probably an improvement (rule-of-thumb)

Either at category 3 in some cases attributes or conditions of rules must be checked at first in the meta DB before they are meaningful for using them in optimization.

Rule 2: Commuting Projection with Join or Product: Suppose the Projection list is \( l = \{ l1, ..., ln; r1, ..., rm \} \), where \( l1, ..., ln \) are attributes of \( R1 \) and \( r1, ..., rm \) are attributes of \( R2 \). If the Join condition \( p \) involves only attributes in \( l \), the two operations can be commuted as:

\[
\begin{align*}
(R_1 \text{ CProd } R_2) \text{ Project}[l] &= (R_1 \text{ Project}[l1, ..., ln]) \text{ CProd } (R_2 \text{ Project}[r1, ..., rm]) \\
(R_1 \text{ Join}[p] R_2) \text{ Project}[l] &= (R_1 \text{ Project}[l1, ..., ln]) \text{ Join}[p] (R_2 \text{ Project}[r1, ..., rm])
\end{align*}
\]

If the Join condition \( p \) contains additional attributes not in \( l \), these must be added to the Projection list and a final Project operation is needed.

Rule 2 is probably an improvement, because if the attributes of the Projection list contain a key attribute, in most cases there are no duplicate tuples in the attribute list and so maybe it is meaningful to execute the Project operation at first. But if the Projection list does not contains any key attributes the term must be optimized for removing duplicate tuples.
Category 4: heuristically an improvement but need to check at meta DB

At category 4 attributes or conditions of rules need to be checked at first in the meta DB before they are meaningful for using them in optimization or the parse tree has to be looked after different conditions.

Rule 8: Commuting Restrict with Project: If the Restrict condition $p$ involves only those attributes $l_i, ..., l_n$ in the Projection list, the two operations can be commuted:

$$(R \text{Restrict}[_p]) \text{Project}[l] \equiv (R \text{Project}[l]) \text{Restrict}_p$$

In most cases it is better to make first a Projection of $R$ as a Restriction because Projection removes duplicate tuples but a Restriction select all tuples that satisfy the Restrict condition from a relation $R$. But it can also be better on the other way round. It depends on the attributes and conditions which must be checked in the meta DB.

Rule 9: Commutativity of Restrict: Restrictions can be permuted among themselves.

$$(R \text{Restrict}_q) \text{Restrict}_p \equiv (R \text{Restrict}_p) \text{Restrict}_q$$

Useful for rearranging the leaf nodes of the tree. The leaf nodes relations with the most restrictive Restrict operations, the ones that produce a relation with the fewest tuples or with the smallest absolute size, are positioned so that they can be executed first in the query tree. But for these at first it is need to look at the parse tree after attributes are used in conditions.
Chapter 3

Classification of the rules concerning the re-writing algorithms

Class 1: contains the rules 2, 3, 6 and 10.

Rule 2: *Commuting* Projection with Join or Cartesian Product:

\[
\text{Project}[l] \quad \rightarrow \quad \text{CProd} \quad \rightarrow \quad \text{Project}[l1] \quad \text{Project}[l2]
\]

Result: set of attributes, includes as tuples all possible combinations from the resulting of Projection of R1 and R2, on which the Cartesian Product is executed.

\[
\text{Project}[l] \quad \rightarrow \quad \text{Join}[p] \quad \rightarrow \quad \text{Project}[l1] \quad \text{Project}[l2]
\]

Result: set of attributes, includes all the combinations of tuples from the resulting of Projection of R1 and R2, on which the Join is executed.

Rule 10: *Commuting* Restrict with Join or Product:

\[
\text{Restrict}[p] \quad \rightarrow \quad \text{Join} \quad \rightarrow \quad \text{Restrict}[p0]
\]

Result: set of attributes, includes all the combinations of tuples from the resulting of Restriction of R1 and R2, on which the Join is executed.

\[
\text{Restrict}[p] \quad \rightarrow \quad \text{CProd} \quad \rightarrow \quad \text{Restrict}[p0]
\]

Result: set of attributes, includes as tuples all possible combinations from the resulting of Restriction of R1 and R2, on which the Cartesian Product is executed.
Rule 3: *Commuting* Projection with a set operation:

```
   Project[\(l\)]
   /
  Union                 Union
 /                     /                      Project[\(l\)]
R1        \(\Rightarrow\)          Project[\(l\)]
  \(\Rightarrow\) R2            R1
```

Result: set of attributes, includes all the tuples from the resulting of Projection of R1 or R2 or both R1 and R2, on which Union is executed. Duplicate tuples are eliminated.

Rule 6: *Commuting* Restrict with set operations:

```
   Restrict[\(p\)]
   /
  Union                 Union
 /                     /                      Restrict[\(p\)]
R1        \(\Rightarrow\)          Restrict[\(p\)]
  \(\Rightarrow\) R2            R1
```

Result: set of attributes, includes all the tuples from the resulting of Restriction of R1 or R2 or both R1 and R2, on which Union is executed. Duplicate tuples are eliminated.

```
   Restrict[\(p\)]
   /
  Intersect                 Intersect
 /                     /                      Restrict[\(p\)]
R1        \(\Rightarrow\)          Restrict[\(p\)]
  \(\Rightarrow\) R2            R1
```

Result: set of attributes, includes all the tuples from the resulting of Restriction of both R1 and R2, on which Intersection is executed.

```
   Restrict[\(p\)]
   /
  Diff                 Diff
 /                     /                      Restrict[\(p\)]
R1        \(\Rightarrow\)          Restrict[\(p\)]
  \(\Rightarrow\) R2            R1
```

Result: set of attributes, includes all the tuples from the resulting of Restriction of R1 that are not in R2, on which Difference is executed.
Pseudocode:

IF ((currentToken == Operation like Project[l] OR Restrict[p]) AND
    (currentToken has a child which is a Commutative Relation like CProd OR Join[p] OR Union OR
     Intersect OR Diff)) {
    IF (child of currentToken == Union OR Intersect OR Diff) {
        result = check at meta DB if l of currentToken involves the attributes of R1 or/and R2;
        IF (result == true) {
            split = false;
            split_and_change_tokens(currentToken, split);
        }
    } ELSE IF (child of currentToken == CProd OR Join[p]) {
        result1 = check at metaDB if l of currentToken are attributes of R1 and if l do not contain a key attribute
        result2 = check at metaDB if l of currentToken are attributes of R2 and if l do not contain a key attribute
        IF ((result1 and result2) == true) {
            IF (child of currentToken == Join[p]) {
                result3 = check at metaDB if p only contains attributes which are also in l
                IF (result3 == true) {
                    split_and_change_tokens(currentToken, split=true);
                }
            }
            ELSE {
                split_and_change_tokens(currentToken, split=true);
            }
        }
        ELSE {
            split_and_change_tokens(currentToken, split=true);
        }
    }
    ELSE {
        No pattern!
    }
}

split_and_change_tokens(root, split) {
    IF ((split == true) {
        split root Token into two Tokens so that;
        Token1 contains only the attributes of R1 and;
        Token2 contains only the attributes of R2;
        lift up root’s child one level;
        set Token1 as root’s left child;
        set Token2 as root’s right child;
        Token1 acquires R1 as its child;
        Token2 acquires R2 as its child;
    }
    ELSE {
        create a new Token;
        copy attributes of root and set these attributes to new Token;
        lift up root’s child one level;
        set root as left child;
        set new Token as right child;
        root acquires R1 as its child;
        new Token acquires R2 as its child;
    }
}
Class 2: contains rule 5

Rule 5: Commuting Restrict with Join or Product:

\[
\begin{align*}
\text{Restrict}[p] & \rightarrow \text{Join} \\
\text{Join} & \rightarrow \text{Restrict}[p] \\
R1 & \rightarrow \text{Restrict}[p] \\
R2 & \rightarrow R2
\end{align*}
\]

Result: set of attributes, includes all the combinations of tuples from the resulting of Restriction of R1 and of the relation R2, on which the Join is executed.

\[
\begin{align*}
\text{Restrict}[p] & \rightarrow \text{CProd} \\
\text{CProd} & \rightarrow \text{Restrict}[p] \\
R1 & \rightarrow \text{Restrict}[p] \\
R2 & \rightarrow R2
\end{align*}
\]

Result: set of attributes, includes as tuples all possible combinations from the resulting of Restriction of R1 and of the relation R2, on which the Cartesian Product is executed.

**Pseudocode:**

IF ((currentToken == Operation like Restrict[p]) AND (currentToken has a child which is a Commutative Relation like CProd OR Join)) {
    result1 = check at metaDB if p only contains attributes from R1
    result2 = check at metaDB if p only contains attributes from R2;
    IF (result1 == true)
        change_tokens(currentToken, result1);
    ELSE IF (result2 == true)
        change_tokens(currentToken, result2);
    ELSE
        no pattern!
}

change_tokens(root, result){
    IF (result==true){
        lift up child one level;
        set root as its left child;
        set R2 as its right child;
        set R1 as root’s child;
    }
    ELSE{
        lift up child one level;
        set root as its right child;
        set R1 as its left child;
        set R2 as root’s child;
    }
}
Class 3: contains rule 1

```
Restrict[p] -> Join[p]
CProd
R1       R2
```

Result: set of attributes, includes as tuples all possible combinations from R1 and R2, on which the Join is executed. Except that the Join attributes of R2 are not included in the resulting relation, if they have the same names.

**Pseudocode:**

IF ((currentToken == Operation like Restrict[p]) AND (currentToken has a child which is a CommutativeRelation like CProd)) {
    result = check at metaDB if p contains attributes from R1 or/and R2;
    IF (result == true)
        delete_and_replace(currentToken);
    ELSE
        no pattern!
}

deleate_and_replace(root){
    create a new Token;
    copy attributes of root and set these attributes to new Token;
    lift up root’s child one level;
    delete root;
    replace child by new Token;
    R1 and R2 are lifted up by one level;
    set R1 as child’s left child;
    set R2 as child’s right child;
}

Class 4: contains rule 4

Rule 4: Sequence of Projection:

```
Project[I1]    \rightarrow  Project[I1]
|
Project[I2]    \rightarrow  R
|
Project[ln]    \rightarrow  R
```

Result: is an new relation with only one of the attributes of R on which the Projection is executed.
Pseudocode:

IF (currentToken == Operation like Project[l1] AND (currentToken has a child which is also a Project[l2] Operation)) {
    DO {
        result = check at metaDB if the attributes of currentToken are in the attribute list of its child’s;
        IF(result == true){
            compare next two Tokens;
            counter +1;
        }
        ELSE
            no pattern!
    }
    WHILE ((currentToken == Project[ln]) AND (its child == Project[ln]));
    result = check at metaDB if attributes of child are in R;
    IF(result == true)
        delete_sequence(currentToken=Project[l1], counter);
    ELSE
        no pattern!;
}

delete_sequence(root, number_of_project_token){
    DO{
        count the number of project tokens;
        number_of_project_token –1;
    }
} while(number_of_project_token >1);
reset the pointer of root to its child;
set pointer to R;

Class 5 : contains rule 7

Rule 7: Sequence of Restrict:


Result: set of attributes of R, concatenated by AND conditions, on which the Restrict is executed.
**Pseudocode:**

IF ((currentToken == Operation like Restrict[p1]) AND (currentToken has a child which is also an Operation like Restrict[p2])) {
    DO {
        compare next two Tokens;
        counter +1;
    }
    WHILE ((currentToken == Restrict[pn]) AND (its child == Restrict[pn]));
    result = concatinate_sequence(currentToken, counter);
}

bool concatinate_sequence(root, number_of_restrict_token){
    create a new condition list;
    push p1 of root to new condition list;
    DO{
        go to nextToken;
        push pn to new condition list;
        concatenate the attributes by the condition ‘AND’;
        number_of_restrict_tokens –1;
    }
    WHILE(number_of_restrict_token >1);
    create a new Token;
    replace root by new Token;
    set the new condition list to new Token;
    set R as new Token’s child;
    result = check at meta DB if attributes in the new condition list are in R;
    IF (result == true)
    return true;
    ELSE{
        no executable restrict operation!;
        exit;
    }
}

**Class 6 : contains rule 8 and 9**

**Rule 8: Commuting Restrict with Project:**

```
<table>
<thead>
<tr>
<th>Project[l]</th>
<th>Restrict[p]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restrict[p]</td>
<td></td>
</tr>
</tbody>
</table>
```

Result: set of attributes of R, on which Restriction is executed.
Rule 9: Commutativity of Restrict:

\[
\begin{array}{c|c|c|c|c}
  & \text{Restrict}[p] & \text{Restrict}[q] \\
\hline
\text{Restrict}[q] & \rightarrow & \text{Restrict}[p] \\
\hline
R & & R
\end{array}
\]

Result: set of attributes of R, on which Restriction is executed.

Pseudocode:

IF ((currentToken == Operation like Project[l] OR Restrict[p]) AND (currentToken has a child which is also an Operation like Restrict[p])) {
  IF (actualNode == Project[l] ) {
    result1 = check at metaDB if l involves the attributes of R;
    IF(result1 == true){
      result2 = check at metaDB if p involves only those attributes in the Projection list l;
      IF(result2 = true)
        change_Token(currentToken);
      ELSE
        p is not in l;
    }
  ELSE
    p is not in l;
  }
ELSE IF ((currentToken == Restrict[p]) AND (its child == Restrict[q])) {
  result1 = check at metaDB if p involves the attributes of R;
  size = check at metaDB which size the attribute in p has;
  IF (result1 == true) {
    result2 = check at metaDB if q involves the attributes of R;
    size2 = check at metaDB which size the attribute in q has;
    IF((result2 == true) AND (size<size2))
      change_Token(currentToken);
  }
  ELSE
    No pattern!;
}

change_Token(root) {
  change root about its child;
}
Chapter 4

4.1 Purpose

1. Logical optimization of a RAQUEL parse tree.
2. Output of an optimized RAQUEL parse tree.
   (remark: The input and output of a complete parse tree is not yet possible. It is only an input of smaller trees possible, which are noted as comments at module main.cxx.)

4.2 Overview of the code file structure and algorithms

The RAQUEL optimizer consists of the following modules:

- Module **main** is used for all screen in- and outputs.
  Input function **main**: different tokens can be entered by the user. These tokens are passed to function **add** and a parse tree is created. The parse tree and the pointer to its root **treeRoot** is then passed to module **patternrecogn**. Output: procedure **showParseTree** outputs the optimized tree and a pointer to its root on screen.
  **main.cxx** is only a scaffolding and can be removed.

- Module **metaDB** has only a supplementing function, until the real metaDB is finished. Its main task is to check parameters passed by different modules and to return boolean values.

- **patternrecogn** is the most important module. Here the entire parse tree is scanned for token combinations/patterns and which one fulfill one of the rules specified at chapter 2 and 3. If one of the pattern is found, it is assigned to the associated class.

- **Class1-Class6** are pattern rewriting modules. All classes have the same task: transforming the parse tree, which was passed over from **patternrecogn**, according to certain rules. All of these modules are crated in a way, that any time one can be added or removed.

Simple summary of code files:

- scaffolding main-programm: main.cxx
  main.o

- scaffolding metaDB: metaDB.h
  metaDB.cxx
  metaDB.o

- patternrecognition: patternrecogn.h
  patternrecogn.cxx
  patternrecogn.o

- pattern rewriting modules: Class1.h
  Class1.cxx
4.2.1 Algorithm details

- Of “find_pattern” at module patternrecogn

Procedure `find_pattern` gets the pointer onto the `root` and thus the parse tree from `main.cxx`. The variable `currentToken` is placed to the root of the tree, variable `nextToken` points on root’s next token and variable `help_pointer` or `help_pointer2` on the following token(s). Then the token are checked, whether they correspond to one of the rules in chapter 2/3. If a token combination is found, the parse tree is passed over upto `currentToken`, to an associated class. This way the entire tree should be scanned for pattern. However this is not yet implemented and would have to be supplemented.

**Input variable:**

`root`: parse tree.

**Output variable:**

`treeRoot`: a pointer to the root of the parse tree.
• of procedures and functions of Class1 to Class6
  In modules Class1 to Class6 the algorithms are implemented, which carry out the optimization. Descriptions of the functions and procedures at the individual modules are added as comments.

4.3 Advancements of the optimizer

As already remarked at different passages, the optimizer in not finished yet. Further additions must be made in example at module patternrecogn.cxx. The major task would be to go through any parse trees and to find pattern which must be optimized as long as no more optimization possibilities exist. A possibility, which can be considered, is a recursive call of the procedure find_pattern, in order to find all pattern in the parse tree.

Modules Class1 to Class6 can be removed or new modules can be added, if new rules are formed, any time.