Principles of probabilistic query optimization
Waas, F.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Enumerating Techniques

In Chapter 1, we briefly sketched the notion of cost distributions already indicating that they are the key to the analysis of both the problem itself and the application of certain optimization techniques. The primary goal will be the analytical modeling of distributions. But in order to verify the generality of those results, we need means to extract actual cost distributions from search spaces used in both simplified and full-blown query optimizers. Having such a verification mechanism will also be handy to test for the limitations of the modeling techniques.

In this chapter we present three techniques that enable enumeration and sampling of search spaces and to obtain actual cost distributions. The three search spaces being scrutinized are (a) the cross product optimization problem, (b) join ordering—including problems with cyclic join graphs—and (c) full-blown query optimization as implemented in Microsoft SQL Server.

The enumeration and sampling framework in the third case is not only of interest for our experiments concerning cost distributions, but has also a very practical application: the verification of the query optimizer and the execution engine in the development of commercial database products. It extends the regular optimizer in a way that potentially all plans, the optimizer has to consider for a particular query, can be obtained. The optimality of the regular optimization result can be easily verified but more important, optimizer components like transformations and optimization strategies, which are otherwise hard to assess, can be tested: selecting plans other than the optimal provides a multitude of test cases for the execution engine. Plans the optimizer would only choose if the underlying catalog fulfills certain conditions can now be tested independently of whether or not these conditions—possibly hard to trigger otherwise—are fulfilled.
3.1 Labeled Binary Trees

The Cross Product Optimization Problem, in the following abbreviated as XOPT, is a special case of the general Join Order Problem in that all operand selectivities are equal to one. The relations are only characterized by their sizes as other parameters like attribute value distribution, are irrelevant. The cost function reduces to a pairwise multiplication of the size of the relations involved. Thus every solution can be split into two components: a binary tree, which implements the multiplication schema and the assignment of leaves.

Let us first focus on the binary tree structure and devise counting and generating schemas.

**Definition 3.1.1**

Let $t_1$ and $t_2$ be binary trees with $n$ leaves. $t_1$ and $t_2$ are *isomorphic* if there exists a bijective mapping $h$ between the nodes of $t_1$ and $t_2$ such that

$$w_j \text{ is son of } w_i \text{ in } t_1 \iff h(w_j) \text{ is son of } h(w_i) \text{ in } t_2.$$  

We denote tree isomorphism with $t_1 \cong t_2$, or $t_1 \equiv_h t_2$ if we want to specify a mapping explicitly.

Isomorphic trees do not add any additional information to the cost distribution as every tree has the same number of isomorphic duplicates. Hence, we can simply omit them in the enumeration schema and focus on the set of non-isomorphic trees only. We denote the set of non-isomorphic trees with $n$ leaves by $M_n$.

### 3.1.1 Counting

To count the number of trees in $M_n$ we use a local argument that considers only the root of a tree and applies recursively to its subtrees. The formula is a well-known result in combinatorics, however, we think it helpful to describe the approach in detailed manner as we will use a similar argument to generate these trees.

Considering the root of a non-isomorphic tree, it is characterized by the number of leaves in its left and its right subtree, $n_r$ and $n_l$ respectively. Apparently, there is more than one tree per such configuration, in general. The number of trees with $n$ leaves in total and $n_l$ in its left subtree, which induces $n_r$ immediately, can be denoted by the recurrence formula

$$B(n_l) \cdot B(n - n_l)$$

where $B(n)$ is the number of non-isomorphic trees with $n$ leaves. To avoid isomorphic duplicates we need to consider only the cases

$$n_l \geq n - n_l.$$
In case \( n \) is an odd number, \( B(n) \) is
\[
B(n) = \sum_{1 \leq i < n-i} B(n-i)B(i)
\]
with \( B(0) = 0 \) and \( B(1) = 1 \).

In case \( n \) is even, we have to correct this expression by a term that determines the number of non-isomorphic combinations of trees with \( \frac{n}{2} \) leaves. We will give a more detailed explanation of this term when we discuss the unranking of trees.

The number of non-isomorphic trees is
\[
B(n) = \begin{cases} 
\sum_{i=1}^{n} B(n-i) \cdot B(i), & \text{if } n \text{ is odd} \\ 
\sum_{i<n-i} B(n-i) \cdot B(i) + \frac{\binom{n}{2}}{2} \cdot (B(\frac{n}{2}) + 1), & \text{if } n \text{ is even}
\end{cases}
\]
with \( B(0) = 0 \) and \( B(1) = 1 \). Using a lookup table, \( B(n) \) can be computed in \( O(n^2) \).

### 3.1.2 Generating Trees

**Unranking.**

The idea behind unranking is to have a mechanism, which, given a rank and the number of leaves, creates a tree recursively by deciding locally, i.e., for the root node of each subtree, how many leaves are to be in the left subtree, and how many in the right. Then, subranks for both subtrees are derived and the unranking is called recursively. A recursion terminates, returning a leaf only, when the procedure is called with \( r = 0 \). Figure 3.1 shows an outline of the algorithm.

Given \( n \) and \( r \) we determine \( n_l \) as
\[
n_l = \max \{ i | \sum_{1 \leq i \leq n-1} B(i)B(n-i) \leq r + 1 \}.
\]

The subranks \( r_r \) and \( r_l \) for the right and the left subtree respectively compute to
\[
r_r = \left\lfloor \frac{r}{B(n_l)} \right\rfloor
\]
and
\[
r_l = r - r_rB(n_l).
\]

Recursive application unfolds the complete tree of rank \( r \). Once we set up the lookup table for \( B(n) \) unranking a pair \((n,r)\) is in \( O(n^2) \). The recursion terminates properly as \( n \leq 3 \) implies a rank of zero. We refer to the resulting tree as \( t_n(r) \), or simply as \( t(r) \) if \( n \) is clear from the context.
Algorithm UNRANK
Input $n$ number of leaves, $r$ rank
Output $t$ tree with $n$ leaves and rank $r$

if $r = 0$ then
   $t \leftarrow \text{LINEAR_TREE}(n)$
else
   compute $n_l, n_r$
   compute $r_l, r_r$
   $t \leftarrow \text{MAKE_TREE(UNRANK}(n_l, r_l), \text{UNRANK}(n_r, r_r))$
endif
return $t$

Figure 3.1. Generic unranking schema

Ranking.
Conversely, the rank of a given tree $b \in M_n$ computes to:

$$r(b) = r_r B(n_l) + r_l.$$

We can now close the gap of computing $B(n)$ in case $n$ is even. Consider a tree with $\frac{n}{2}$ leaves in both subtrees. Non-isomorphic combinations are only those with

$$r_l > r_r$$

as further restriction. With $0 \leq r_l < B(\frac{n}{2})$ we get

$$\frac{B(n/2)}{2} \sum_{i=1}^{B(n/2)} i$$

as the total number of possible combinations when $n_l = n_r = B(n/2)$. Rewriting the term gives

$$\frac{1}{2} B(\frac{n}{2})(B(\frac{n}{2}) + 1).$$

3.1.3 Labeling
For a tree $t$ of $M_n$, a function $f$ on the leaves of $t$ into $\{1, \ldots, n\}$ is called \textit{labeling} of $t$. A labeling can be interpreted as a permutation of the leaves. In principle, there are $n!$ possibilities to label a tree with $n$ leaves, however,
due to the symmetry of subtrees, we may encounter duplicates as shown in Figure 3.2.

The left tree can be transformed into the right one by means of commutative exchange of (isomorphic) subtrees.

**Definition 3.1.2**

Let $t_1$ and $t_2$ be isomorphic trees, and $f$ a labeling of $t_1$. $f$ is called isomorphic labeling, if there exists a bijective mapping $h$ such that

1. $t_1 \cong_h t_2$
2. if $w$ is leaf in $t_1$, $f(h(w)) = f(w)$

For every labeling there exist $2^k$ isomorphic labelings, $k$ being the number of inner nodes with isomorphic subtrees. In Figure 3.3 these nodes are indicated by circles. There are $\frac{n!}{2^k}$ non-isomorphic labelings for each tree. During enumeration of all labelings, isomorphic combinations can be avoided by skipping permutations.

For every pair of isomorphic subtrees rooted in the same node let $N_1$ and $N_2$ denote the labels of the left and the right subtree respectively. We call a labeling monotonic if

$$\min N_2 < \min N_1$$

holds for all such pairs. Enumerating monotonic labelings only is therefore equal to enumerating non-isomorphic ones.

If we think of the labels as id numbers of the database relations, we are able to count and generate execution plans consisting of cross products only. To obtain the cost distribution we could either enumerate the whole set $M_n$ or sample form it with uniform probability by generating random numbers between 0 and $B(n) - 1$ and unrank the associated tree.
3.2 Non-isomorphic Processing Trees

In the next step, we leverage the previous model with the information encoded in the join graph. Instead of enumerating all cartesian products we now generate only those trees that solely consist of join operators. The set of non-isomorphic processing trees is a subset of $M$. Equality between the sets holds only if the join graph is a completely connected graph. In practice, the sets differ substantially.

The previous approach was based on determining and grouping sets of leaves, which was sufficient as any relation can be freely combined with any other in a cross product. Here, we use additional information describing the inner nodes of the tree. Our algorithm is based on edges of the join graph. Every edge corresponds to an inner node of the tree, though not uniquely. First, we develop an algorithm that turns a sequence of edges into a processing tree. Through this connection, the concept of isomorphism extends immediately from trees to sequences. Using this close connection it is sufficient to enumerate non-isomorphic sequences to obtain the desired set of processing trees.

3.2.1 Sequences

For a set $E = \{e_1, \ldots, e_n\}$ a sequence $L$ over $E$ is denoted by

$$L = (e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(n)}) = (e_{\pi(i)})$$

where $\pi$ is a permutation function on $\{1, \ldots, n\}$. The set of all sequences over $E$ is denoted by $E^*$ and contains $n!$ elements. If $E$ is the empty set $E^*$ contains only the empty sequence $\langle\rangle$. Sequences are permutations of a set. We prefer the term sequence, however, since it induces intuitively the notion of subsequences, which we will use extensively below.
For convenience, we introduce the following operations in analogy to sets. \(|L|\) denotes the length of a sequence; \(L \setminus E'\) is the sequence without the elements of \(E'\) but retaining the order of the residual elements. As an example, consider \(\langle e_1, e_2, e_3 \rangle \setminus \{e_2\}\) which is \(\langle e_1, e_3 \rangle\). Finally, we use :: to describe the concatenation of a sequence with either a single element or another sequence.

In order to avoid any confusion of the indices of sequences with those of sets, we introduce a labeling \(\eta : E \to \{1, \ldots, n\}\). Throughout all examples we will refer to the \(e_i\) by their labels, i.e., for \(L = \langle e_1, e_2, e_3, e_4 \rangle\) with \(\eta(e_1) = 3, \eta(e_2) = 1, \eta(e_3) = 4, \eta(e_4) = 2\) we simply write \(L = \langle 3, 1, 4, 2 \rangle\).

To establish an order on the set \(E^*\), we introduce a ranking which assigns each sequence a unique number as follows:

**Definition 3.2.1**
For a sequence \(L = \langle e_1, \ldots, e_n \rangle\) of length \(n\), the *rank* of \(L\) to a base \(b\) with \(b \geq n\) is

\[
r_b(L) = \sum_{i=1}^{n} \eta(e_i) \cdot b^{n-i}
\]

The rank of the empty sequence is 0.

For instance, the rank of \(L = \langle 3, 1, 4, 2 \rangle\) to the base \(b = 10\) computes to

\[
r_{10}(L) = 3 \cdot 10^3 + 1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0 = 3142.
\]

Since we demand \(b \geq n\), every rank function is injective and rank functions to different bases define the same order on \(E^*\), i.e., we can omit the explicit notation of \(b\).

Finally, we call a sequence \(\eta\)-sorted sequence, denoted by \(\tilde{E}\), if elements with higher \(\eta\) value occur later in the sequence than elements with a lower one, i.e., \(\eta(e_i) < \eta(e_j) \Rightarrow \pi(i) < \pi(j)\). For completeness, we define the \(\eta\)-sorted sequence of the empty set to be the empty sequence.
Chapter 3. Enumeration Techniques

3.2.2 Merging Processing Trees

Given a query graph $G(V, E)$, a sequence over $E$—i.e., a sequence of query predicates—can be turned into a processing tree as follows. For each edge of the sequence we add the respective join operator to the tree. If the edge's relations are already connected by a join added earlier, we only extend the predicate of that join.

Figure 3.5 shows the single steps necessary to convert the sequence $(1, 5, 2, 3, 4)$ using the query graph of Figure 3.4.

A detailed description of the algorithm is given in Figure 3.6. Starting with a forest of $|V|$ trivial trees that consist of root nodes only, trees are merged pairwise by adding a common root node, indicated by $\oplus$. For simplicity of presentation, we omitted the trivial trees in Figure 3.5. After every step, $T_i$ contains a forest. We refer to the output $T_{|L|}$ as $T(L)$.

Lemma 3.2.2

For a sequence $L$ over all edges of a query graph, algorithm MERGETREES constructs one single tree only, i.e., $T(L)$ contains only one tree.

Proof: Assume, to the contrary, $T(L) = \{t_1, t_2, \ldots, t_m\}$. Thus, no edges between leaves of $t_1, \ldots, t_m$ were in $L$, which means, that $G$ was not connected which contradicts the definition of query graphs. $\square$

Figure 3.5. Converting the sequence $(1, 5, 2, 3, 4)$ into a processing tree.
Note, in case $L$ is only a sequence over a subset of $E$, the result of MERGETREES $T(L)$ will be a forest.

**Lemma 3.2.3**
The processing tree computed by MERGETREES is a valid tree in the sense of Definition 2.2.1.

**Proof:** $T_i$ always contains valid processing trees for disjoint subgraphs of $G$:

1. $i = 0$: all elements $t_j$ of $T_0$ are trivial trees of height 0, thus valid trees for the Graphs $G_{t_i}((v_j), \emptyset)$.

2. $i \rightarrow i + 1$: let $(n_a, n_b)$ be the next edge that is to be added and $t_a$ and $t_b$ be elements of $T_i$ with leaves $n_a$ and $n_b$, respectively. Note, that $T_i$ contains at least a tree of height 0 with leaf $v$ for every possible leaf. If the edge connects nodes, that are leaves of the same tree the shape of the processing tree does not change, according to its definition, and thus the proposition holds. Otherwise, the edge connects the two graphs $G_{t_a}$ and $G_{t_b}$, with $G_t = G(\{\text{leaves of } t_i\})$. Therefore, $G_{t_a}$, $G_{t_b}$ and $G_{t_a \cup t_b}$ fulfill the conditions of Definition 2.2.1 which completes the proof. $\square$

**Proposition 3.2.4**
For a query graph $G(V,E)$, MERGETREES converts every sequence over $E$ into a valid processing tree.\(^1\)

**Proof:** Follows from Lemma 3.2.2 and 3.2.3. $\square$

The concept of isomorphism immediately extends to sequences: Two sequences $L_1$ and $L_2$ are isomorphic if $T(L_1) \equiv T(L_2)$. In the previous example $T((1, 5, 2, 3)) \equiv T((5, 1, 2, 3))$ holds, since both sequences yield the forest of Figure 3.5.d. On the other hand, $T((1, 5, 2, 3)) \neq T((2, 1, 5, 3))$.

During the enumeration we want to select one sequence of each class of isomorphic sequences. As a simple criterion for this selection, we can use the rank of the sequences as follows.

**Definition 3.2.5**
A sequence $L$ over $E$ is rank-minimal iff

$$\forall L' \in E^*, L' \neq L : \quad T(L) \equiv T(L') \Rightarrow r(L) < r(L')$$

i.e., every isomorphic sequence has greater rank. $\diamond$

In our example, $(2, 1, 5)$ is rank-minimal, since the only other isomorphic sequence over the same subset of $E$ is $(5, 1, 2)$.

Monotonicity of the sequence elements implies prefix monotonicity of rank-minimality as follows:

\(^1\)We present a discussion of the time complexity for all algorithms in Section 3.2.5.
Algorithm MERGETREES

Input
$L$ sequence of edges,
$n_i$ nodes of the query graph

Output
$T_{|L|}$ processing tree

$i \leftarrow 0$
for $j=0$ to $|L|$ do
  $t_j \leftarrow n_j$
done
$T_0 \leftarrow \bigcup t_j$

$L_0 \leftarrow L$
while $|L_i| > 0$ do
  let $e = (n_a, n_b)$ be the first element of $L_i$
  let $t_a \in T_i$ be tree where $n_a$ is leaf
  let $t_b \in T_i$ be tree where $n_b$ is leaf
  if $t_a \not= t_b$ do
    $t \leftarrow t_a \oplus t_b$
    $T_{i+1} \leftarrow (T_i \setminus \{t_a, t_b\}) \cup t$
done
  annotate deepest common ancestor of $n_a$ and $n_b$
  with predicate of $e$
  $L_{i+1} \leftarrow L_i \setminus \{e\}$
  $i \leftarrow i + 1$
done

Figure 3.6. Algorithm MERGETREES

Corollary 3.2.6
Let $L$ be sequence over $F \subseteq E$ and $e \in E \setminus F$ with $\eta(e) > \eta(f)$ for all $f$ in $L$ then

$L$ is rank-minimal $\iff L :: e$ is rank-minimal

holds.

Consequently, the $\eta$-sorted sequences are always rank-minimal.

As pointed out earlier, the concept of rank-minimality is the key to a proper enumeration: Every class of isomorphic trees corresponds to exactly one single rank-minimal sequence. Therefore, enumerating the rank-minimal sequences only, would suffice.

In the next section we discuss criteria, on which we can identify prefixes that can be excluded from further considerations.
3.2.3 Redundancies

The fact that not every edge of a sequence prompts MERGETREES to change the shape of some tree already suggests that sequences may contain certain redundancies (cf. Fig. 3.5c and d). Besides the equivalences shown in previous examples which were caused solely by permutation of two elements of a sequence, cyclic graphs additionally contain redundancy.

Definition 3.2.7
For a sequence $L$ the set of redundant edges for every element is given by:

$$\rho_L(e_i) = \{ e_j \mid \eta(e_i) < \eta(e_j), T(L) \cong T(L \setminus \{e_j\}) \}$$

$$R(L) = \bigcup_{e \in L} \rho_L(e) \cup \bigcup_{e \notin L} \rho_L(e)$$

called the redundancy of $L$.

The first part means, that every edge $e$ can be replaced with every element of $\rho_L(e)$ while $T(L)$ remains unchanged.

The redundancy of a sequence, however, comprises more than just the redundant edges within the sequence itself. The rational behind this is as follows. When building a sequence incrementally, we need to know which of the edges that is not yet part of the sequence has any redundancy. Edges that are not adjacent to $G(L)$ cannot induce further redundancy. Hence, $R(L)$ is the set of edges either redundant to edges of $L$ or edges adjacent to $G(L)$.

The redundancy of $(1,2,4,5)$ is $\{3\}$, for instance. For the sequence $R((1))$ the redundancy computes to $\{3\}$ since 3 is redundant to an adjacent edge of $G((1))$, namely 2. Analogously, $R((3)) = 2$.

Furthermore, the redundant-edge property is transitive, i.e., for two edges $e_i$ and $e_j$ in $L$ the following holds:

$$\rho_L(e_i) \cap \rho_L(e_j) \neq \emptyset \implies e_j \in \rho_L(e_i) \lor e_i \in \rho_L(e_j)$$

Removing the redundancy from a sequence does not affect the shape of the resulting tree, i.e., $T(L) \cong T(L \setminus R(L))$. Hence, redundancy is prefix monotonic, in the following sense $R((e_1, \ldots, e_{n-1})) \subseteq R((e_1, \ldots, e_{n-1}, e_n))$.

Figure 3.7 shows three examples for different situations where redundancies may occur.

Proposition 3.2.8
Let $L$ be a sequence over $F \subset E$ and $e \in E \setminus F$. Algorithm COMPUTEREUNDANCY (see Fig. 3.8) computes $R(L :: e)$.

Proof: Since $R(L)$ is input parameter, only the redundancy added by $e$ has to be computed. Redundancy occurs if there is more than one edge between one component of $G(L)$ and $G_e$, and $V'$ and $G_e$, respectively. For every component of $G(L)$ the algorithm checks all remaining edges in $E \setminus L \setminus R(L)$ whether they connect the graphs and all but the first fulfilling the condition are added to $R'$. Thus $R'$ is $R(L :: e)$. \qed
Original join graph

$L = (5, 7, 8, 9, 10, 11, 1)$
$\rho(1) = 2, \rho(1) = 3$
$R = \{2, 3\}$

$L = (1, 2, 5, 9, 10, 11)$
$\rho(1) = 8, \rho(2) = 8, \rho(5) = 7$
$R = \{7, 8\}$

$L = (1, 5, 8, 9, 10, 11)$
$\rho(1) = 2, \rho(5) = 7$
$R = \{2, 7\}$

Figure 3.7. Examples of redundancies for different sequences $L$; Elements of sequence indicated by thick lines; Elements of redundancy indicated by gray lines.
Algorithm \texttt{COMPUTEREDUNDANCY}

Input
- \( L \) sequence of edges
- \( R \) set of redundant edges w.r.t. \( L \),
- \( e \) edge with \( e \in E \setminus R \)

let \( G_e \) be subgraph of \( G(L \cup e) \) containing \( e \)
let \( V' \) be set of nodes not connected to \( G(L \cup e) \)
foreach connected subgraph \( G' \in V' \cup (G(L \cup e) \setminus G_e) \) do
  \( \text{found} \leftarrow \text{false} \)
  foreach \( f \in E \setminus L \setminus R \) do
    if \( f \) connects \( G' \) with \( G_e \) do
      if \( \text{found} \) do
        \( R' \leftarrow R' \cup \{ f \} \)
      \( \text{found} \leftarrow \text{true} \)
    done
  done
done

\textbf{Figure 3.8.} Algorithm \texttt{COMPUTEREDUNDANCY}

### 3.2.4 Generating the Trees

In this section, we assemble the techniques presented so far. To generate all non-isomorphic processing trees for a given query graph \( G \), we generate the set \( \hat{L} \) of non-isomorphic edge sequences. This set is given by

\[
\hat{L} \subseteq E^*, \quad \text{with} \quad L_1, L_2 \in \hat{L} \Rightarrow T(L_1) \neq T(L_2)
\]

An algorithmically more practical form is the following one:

\[
\hat{L} = \{ L \in E^* | L \text{ rank-minimal} \}
= \{ L \in E^* | L = L' \setminus R(L') :: \hat{R}(L'), \quad L' \in E^*, \quad L' \setminus R(L') \text{ rank-minimal} \}
\]

The algorithm \texttt{RAPIDENUMERATION}, given in Figure 3.9, generates the sought sequences. The main loop iterates over the number of edges and calls itself recursively up to \( n \) times. The deepest recursion is reached once the sequence is either complete, or can be completed immediately by adding the redundancy as a \( \eta \)-sorted suffix. Since the basic design of the loop allows for the maximal possible set of \( n! \) sequences, the algorithm has only to avoid the generation of equivalents, that is, ignore redundant edges and discard not rank-minimal prefixes. According to Corollary 3.2.6, rank-
minimality is prefix monotonic, thus we can judge for each edge \( e \) whether \( L \triangleright e \) is a valid prefix or not.

**Lemma 3.2.9**

Let \( L \) be a rank-minimal sequence over \( F \subset E \), and \( e \in E \setminus R(L) \) with \( \eta(e) > \eta(f) \) where \( f \) is the edge of \( L \) with \( \eta(f) = \max_{e' \in F} \eta(e') \). Then \( L \triangleright e \) is rank-

minimal.

**Proof:** Since rank-minimality is prefix monotonic, \( L \triangleright e \) is the rank-

minimal sequence containing all elements of \( L \) and \( e \).

**Lemma 3.2.10**

Let \( L \) be a rank-minimal sequence over \( F \subset E \), and \( e \in E \setminus R(L) \) with \( \eta(e) < \eta(f) \) where \( f \) is defined as above. The following holds

\[
L \triangleright e \text{ rank-minimal } \iff e \text{ is adjacent to } G_f
\]

where \( G_f \) denotes the component of \( G(L) \) that covers \( f \).

**Proof:** Let \( e_i \) be the elements of \( L \), i.e., \( L = \langle e_1, \ldots, e_{|L|} \rangle \).

Firstly, assume, to the contrary, \( e \) is not adjacent to \( G_f \). Let \( e_k \) be \( \min \{ \eta(e_i) \in L \mid \eta(e) < \eta(e_i) \} \), and \( G_a, G_b \) the components of \( G \) connected by \( e \). Furthermore, w.l.o.g. \( |G_a| \leq |G_b| \).

If \( |G_b| = 1 \), then \( e \) is not connected to any of \( G(L) \) components and inserting it at any position in the sequence does not change \( T(L) \).

Thus, \( r(\langle e_1, \ldots, e_{k-1}, e, e_k, \ldots, e_{|L|} \rangle) < r( L \triangleright e) \), i.e., \( L \triangleright e \) is not rank-

minimal.

If \( |G_b| > 1 \), let \( \langle b_1, \ldots, b_{|G_b|} \rangle \) be the subsequence that defines \( G_b \) only, i.e., \( L \setminus (E \setminus G_b) \). If \( |G_a| > 1 \), \( \langle a_1, \ldots, a_{|G_a|} \rangle \) is defined analogously. Otherwise, let \( a_{|G_a|} \) be \( b_{|G_b|} \). The first position in \( L \) where \( e \) can occur without affecting the equivalence is after both \( b_{|G_b|} \) and \( a_{|G_a|} \). Since both \( G_a \) and \( G_b \) are not connected to \( G_e \), \( r(\langle e_1, \ldots, e, e_{|L|} \rangle) < r(\langle e_1, \ldots, e_{|L|}, e \rangle) = r( L \triangleright e) \), i.e., \( L \triangleright e \) is not rank-

minimal.

To show the opposite direction, assume \( L \triangleright e \) is not rank-minimal. Then, a sequence \( L' \) with \( T(L \triangleright e) \equiv T(L') \) exists where \( e \) is not the last element, i.e., \( L' = \langle e_1, \ldots, e, \ldots, e_{|L|} \rangle \). However, inserting \( e \) before \( e_{|L|} \) does not effect the equivalence of \( T(L') \equiv T(L \triangleright e) \) unless \( e \) is adjacent to \( G_f \).

**Proposition 3.2.11**

Algorithm RAPIDENUMERATION computes all sequences of \( L \) with prefix \( L \).

**Proof:** The inner loop of the procedure potentially generates all prefixes. For every prefix, built incrementally, edges that are not already in the prefix or element of the redundancy of this prefix are checked for being added to the prefix. Thus, we show that only rank-minimal redundancy-free prefixes are generated by induction over the prefix length.

\( i = 1 \): trivial.
Algorithm \textsc{RapidEnumeration}

Input \hspace{1cm} L sequence of edges, 
\hspace{2cm} R set of redundant edges w.r.t. \(L\)

\begin{verbatim}
if \(E \setminus L \setminus R = \emptyset\) do
    \textsc{Mergetrees}(L :: \(\hat{R}\))
    return
done

\(\eta_{\text{max}} = \max \{\eta(e_i) | e_i \in L\}\)

let \(G_{\eta_{\text{max}}} \) be subgraph of \(G(L)\) covering \(e_{\eta_{\text{max}}}\)

\begin{verbatim}
foreach \(e \in E \setminus L \setminus R\) do
    if \(\eta(e) > \eta_{\text{max}} \vee e\) is adjacent to \(G_{\eta_{\text{max}}}\) do
        \(L' = L :: e\)
        \(R' = \text{Computeredundancy}(L', R, e)\)
        \textsc{Rapidenumeration}(L', R')

    done

done
\end{verbatim}

\end{verbatim}

Figure 3.9. Algorithm \textsc{RapidEnumeration}

\(i \rightarrow i + 1\): Let \(e \in E \setminus L \setminus R(L)\) be the edge we check. Either \(\eta(e) > \eta(e_i)\) for all \(e_i\) that are in \(L\) so far. The proposition follows with Lemma 3.2.9. Otherwise, if \(e\) is adjacent to the component covering \(\eta_{\text{max}}\) and Lemma 3.2.10 completes the proof.

Invoking \textsc{RapidEnumeration} with all possible prefixes \(L_i = \langle e_i \rangle\) yields \(\hat{L}\).

3.2.5 Discussion

In this section we scrutinize the techniques presented with respect to an efficient implementation and present a quantitative assessment.

Complexity of the Algorithms

One of the critical elements in the algorithm is the membership test for sets. But as our sets are limited to small sizes we can represent them by bit-vectors, which reduces both test and insert/remove operations to \(O(1)\).

\textsc{Mergetrees} can transform a sequence into its corresponding processing tree within \(O(|E|^2)\) using a directory of leaves. The construction is in \(O(|E|)\) and requires no updates at run time. Another critical issue is

\(\hat{2}\)The actual implementation, in fact, expands all sequences from the empty sequence. However, for simplicity we omitted the parts necessary for proper treatment of \(\langle \rangle\).
the traversal of connected components in \textsc{ComputerRedundancy}. Naively identifying the components anew with every invocation is in $O(|E|)$. However, the incremental nature of the changes to the components—adding an edge leaves all other members of the component unchanged—suggests a directory of components. The overhead caused by its maintenance is in $O(|V|)$. This reduces the cost of \textsc{ComputerRedundancy} from $O(|E|^2)$ to $O(|V| \cdot |E|^2)$. Hence, the construction of a processing tree is in $O(|V| \cdot |E|^2)$.

We deliberately used $O(|V|)$ and $O(|E|)$ despite the fact that $O(|E|)$ is in $O(|V|^2)$ since the number of edges exceeds the number of nodes only marginally in typical database applications. Queries that correspond to clique graphs are of virtually no practical impact and solely used as worst-case scenarios. In the majority of all cases $|E|$ is close to $|V|$.

Finally, what is not expressed by the time complexity is the extent of re-use. For simplicity we presented the single procedures as separate from each other as possible. However, to gain the necessary performance, the call to \textsc{MergeTrees} in \textsc{RapidEnumeration} should not be postponed until the sequence is complete and—as the notation suggests—be discarded afterwards, but handled incrementally. We modify \textsc{MergeTrees} in a way that single edges can be added and removed from the tree or forest, respectively. The adding operation is then called before, the removing after the recursive invocation. With this modification, one tree is incrementally built, subsequently pruned, and merged again. Large parts of the tree and of the sequence prefix are not modified when going on to the next processing tree. In contrast to other enumeration techniques, the re-use is inherent in the method and does not require any additional memory nor running time spent on lookups.

In contrast to other enumeration techniques with exponential space requirements, \textsc{RapidEnumeration} needs only space in $O(|E|)$, since only one single tree is built and modified.

\section*{Sizes of Search Spaces}

The major advantage of the techniques presented is the reduction of the search space's size by the factor $2^k$. In Figure 3.10, the sizes of search spaces consisting of non-isomorphic trees only are contrasted with the ones including also isomorphic trees. The queries used in this experiment were taken from the query suite proposed in [GLPK94]. Note, search spaces of cyclic and tree-shaped query graphs are comparable in size, as long as the number of edges does not exceed the number of nodes substantially. The reduction by factor $2^k$ applies independent of the particular search space or query graph. We experimented with a large number of further queries. Since they are exactly in the line of the above we omit the results here.

\textbf{Corollary 3.2.12}

The number of non-isomorphic trees enumerated by \textsc{RapidEnumeration}
is \( N_{\text{clique}}(n) = \frac{(2n-2)!}{(n-1)!} \cdot 2^{1-n} \) if the query graph forms a clique of size \( n \), and \( N_{\text{chain}}(n) = \frac{(2n-2)!}{n!(n-1)!} \) in case the query graph is a chain.

**Proof:** Independent of the query graph’s shape, all processing trees have \( n \) leaves and \( n - 1 \) nodes inner nodes. Every isomorphic class has \( 2^{n-1} \) elements. The numbers for spaces including isomorphic trees [LVZ93] are known to be \( N_{\text{clique}}^{iso}(n) = \frac{(2n-2)!}{(n-1)!} \) and \( N_{\text{chain}}^{iso}(n) = \frac{(2n-2)!}{n!(n-1)!} \cdot 2^{n-1} \). Dividing those figures by \( 2^{n-1} \) yields the proposition.

Finally, with a simple modification we can restrict the algorithm to enumerate only linear processing trees—the most prominent group of trees since the early days of query optimization [SAC+79, IK91]. When dropping the condition \( \eta(e) > \eta_{\text{max}} \), we append only edges that are adjacent to \( G_{\text{max}} \), so at least one of its nodes is already part of the tree, i.e., we add either the bare predicate or a subtree that consists of a leaf only. Thus, the result is a processing tree of height \( n - 1 \).

**Corollary 3.2.13**

The number of non-isomorphic linear trees enumerated by RAPIDENUMERATION is \( N_{\text{clique}}(n) = \frac{n!}{2} \) if the query graph forms a clique of size \( n \). In case the query graph is a chain we enumerate \( N_{\text{chain}}(n) = 2^{n-2} \) trees.

**Proof:** In either case we need to focus on the non-redundant edges only. For the clique, every edge that connects to a node which is not part of the
prefix yet, is non-redundant. Thus, for every prefix of length 1, there are \((n - 2)\)! completions. The very first edge can be chosen in \(\binom{n}{2}\) ways. Therefore, the total number is \(\frac{n!}{2^{(n-2)!}} \cdot (n - 2)! = \frac{n!}{2}\).

The situation for a chain is as follows. Selecting a prefix of length 1 splits the query graph into two sub-chains that are to be merged by RAPIDENUMERATION. This can be done in \(\binom{L_r + L_l}{l_l}\) ways, with \(L_l\) and \(L_r\) is length of the left and right sub-chain, respectively. Applying this to all \(n - 1\) prefixes of length 1 yields \(N_{\text{chain}}(n) = \sum_{k=0}^{n-2} \binom{n-2}{k}\) which can be rewritten to \(2^{n-2}\).

### 3.2.6 Quantitative Assessment

To assess the efficiency of RAPIDENUMERATION is in so far difficult as there is no other method capable of enumerating the same spaces. However, in a different context Galindo-Legaria et al. developed counting, ranking, and unranking methods for non-isomorphic processing trees belonging to tree-shaped query graphs [GLPK95]. The techniques can be combined and after counting the trees each plan can be generated by unranking its ordinal number.

In Figure 3.11, the running time of RAPIDENUMERATION is compared to this combined technique—in the following referred to as GLPK—for the non-isomorphic search spaces belonging to tree-shaped join graphs (cf. Figure 3.11. Running time of RAPIDENUMERATION compared to GLPK.

![Figure 3.11. Running time of RAPIDENUMERATION compared to GLPK.](image-url)
3.3. MEMO-BASED ENUMERATION

3.10. All experiments were carried out on a MIPS 10K/250MHz. The queries considered were the same as used for Fig. 3.10. As the graph shows, RAPIDENUMERATION is up to an order of magnitude quicker than GLPK—a gap that widens with increasing query size. For the largest queries run, GLPK is 20 times slower than RAPIDENUMERATION. As the reduction by factor $2^k$ applies to every query, we found almost the same ratio of running times for other query suites.

3.3 MEMO-based Enumeration

In an analytical analysis, abstract and simplified models as described in the previous sections of this chapter are definitely preferable for the reasons sketched at the very beginning. However, an analysis of a simplified model can only be conclusive and useful in practice if it stands the comparison with its real-world counterpart. In other words, we need to be able to examine cost distributions of a real, fully fletched database system in order to verify our results.

The method developed in this section is based on the MEMO framework as detailed in Chapter 1, but can be transferred to any other optimizer that deploys dynamic programming techniques.

Like with the simplified models, we will devise a ranking algorithm. However, since the term ranking has been used in a different notion in [IK84] or [KBZ86], it is important to point out, that our ranking technique is completely in the line of the previous chapter, i.e., with respect to the shape of execution plans rather than in terms of the above cited work.

The previously developed techniques do not extend to the more general problem as the space of alternatives considered by industrial query optimizers is not restricted to an abstract combinatorial problem, such as join reordering. Multiple execution algorithms, index utilization, reordering of grouping operators, special-purpose physical operators, and heuristics to control the time spent on searching, all make up for an actual space that is hard to describe succinctly using abstract, regular structures.

3.3.1 Preparatory Steps

Once alternatives are generated, the MEMO structure contains all operators but does not keep track of how many combinations of operators there are, and only the best possible plan is completely assembled. To illustrate the counting framework, let us assume a final state of the MEMO—after generation of alternatives is complete—as given in Figure 3.12.

In order to facilitate later operations we extract all physical operators and materialize the links between operators and their possible children. In Figure 3.13, the materialized links for all children of the previous example's root (operator 7.7) are shown. The resulting structure describes all possible execution plans that can be rooted in this operator. Due to the differences
in physical properties some operators of a group may qualify as potential children while others do not.

For instance operator 3.3 in Figure 3.13, can have any operator from group 1 and 2 as left and right child, respectively. Operator 3.4 however can use only the darkened operators 2.3 and 1.3 or 1.4.

### 3.3.2 Counting Query Plans

We compute the total number of possible plans bottom-up by computing the individual numbers of possible plans that can be extracted from each operator. We denote the number of children of operator $v$ by $|v|$ and the $j$-th alternative for the $i$-th child of $v$ by $w_{i,j}^{(v)}$. For example, in Figure 3.13, take $v = 7.7$, then $w_{1,1}^{(v)} = 4.2$, and $w_{2,2}^{(v)} = 3.4$.

To compute the number of plans $N(v)$ rooted in an operator $v$, we first determine the number of possible alternatives for each child $i$ as

$$b_v(i) = \sum_j N(w_{i,j}^{(v)}) .$$

Operator $v$ will take any of the available alternatives on each children, independently, so the number of combined choices is given by a product.
The numbers of plans we can generate using only the first $k$ children is

$$B_v(k) = \prod_{i=1}^{k} b_v(i).$$

Hence, the number of plans rooted in $v$ is

$$N(v) = \begin{cases} 1, & \text{if } |v| = 0 \\ B_v(|v|), & \text{otherwise} \end{cases}$$

In Figure 3.13, this process is illustrated for operator 7.7. The upper right corner of operators has the computation of the number of alternatives that can be extracted using it as a root.

The total number of plans is the sum of possible plans rooted in any of the root group’s operators:

$$N = \sum_{i} N(v_i), \quad v_i \in G_{\text{root}}$$

where $G_{\text{root}}$ denotes the root group.

Computing the counts for operators takes linear time, as each operator has to be visited exactly once.\(^3\)

\(^3\)For the number of logical operators for the problem of join reordering, see [OL90,
3.3.3 Unranking Plans

Before we describe the unranking mechanism in detail, it might be helpful to give a short outline of the idea:

Starting with the root group and the rank \( r \), we choose an operator of the group to be the root of the tree. We then compute a local rank for this operator. This local rank for an operator \( v \) is in the interval \( 0, \ldots, N(v) \). Now, assume operator \( v \) has children alternatives

\[
\{ w_{i,1}^{(v)}, \ldots, w_{i,j_i}^{(v)} \}, \ldots, \{ w_{n,1}^{(v)}, \ldots, w_{n,j_n}^{(v)} \},
\]

with \( n = |v| \). \( n \) subranks are computed, and used in each child choice to recursively unrank a subplan. The resulting tree is assembled from unranked suplans, using \( v \) as the root.

Detailed steps are described next.

1. Given a pair \((r, G)\) consisting of a rank and a group we determine which operator of this group becomes the root of the sub-plan.

The first physical operator in the group covers the plan numbers \( 0, 1, \ldots, N(v_1) - 1 \), the second \( N(v_1), N(v_1) + 1, \ldots, N(v_1) + N(v_2) - 1 \) and so on. Thus, the sought operator has index

\[
k = \max\{ i \mid \sum_i N(v_i) \leq r \}.
\]

\( v_k \) becomes the root of the (sub-)plan. The local rank \( r_l \) of \( v_k \) is

\[
r_l = r - \sum_{i=1}^{k-1} N(v_i)
\]

The local rank is necessary to determine the subranks for the children in the next step.

2. Using the concepts introduced in the previous section, we can write the subrank for the \( i \)-th child as

\[
s_v(i) = \begin{cases} R_v(i), & \text{if } i = 1 \\ \left[ \frac{R_v(i)}{B_v(i-1)} \right], & \text{else} \end{cases}
\]

with

\[
R_v(i) = \begin{cases} r_l, & \text{if } i = |v| \\ R_v(i + 1) \mod B_v(i), & \text{otherwise} \end{cases}
\]

\[\text{PGLK97b}].\] There are a few physical operators for each logical joins, implementing different alternatives of hash join, merge join, and index lookups, so the number of physical joins is usually a small multiple of the count of logical joins.
We add the operator $v_k$ to our plan and repeat this step for each child, i.e., for the $i$-th child we unrank $(s_v(i), G_i)$ where $G_i$ is the group for this child. We repeat the steps recursively until we reach the terminal operators.

Unranking is in $O(m)$, $m$ being the number of operators in the tree, which is limited by the number of groups in the MEMO.

**Example.**

This example describes the steps necessary to unrank a plan in detail for the MEMO structure as shown in Figure 2.5. We unrank plan number 13 in group 7, i.e., we unrank the pair $(13, 7)$. First, we determine the operator which becomes the root (operators that become part of the plan are underlined):

$$k = 1, v_k = 7.7$$

since $v_1$ covers the plans $1, \ldots, 22$. The local rank computes to

$$r_l = 13.$$  

For the first operator in a group like 7.7, the local rank is always equal to the global rank within the group. With

$$R_{7.7}(2) = 13, R_{7.7}(1) = 1$$

the subranks for the children compute to

$$s_{7.7}(2) = 6, s_{7.7}(1) = 1,$$

i.e., we have to unrank the sixth possible subtree of the right child, and the first of the left. We unrank the subranks in the children's groups, i.e., $(s_{7.7}(1), 4)$ and $(s_{7.7}(2), 3)$. Unranking $(s_{7.7}(1), 4)$ gives

$$k = 1, v_k = 4.3$$

since 4.3 covers the first subplan, and 4.4 the second. As there are no further children, no subranks need to be computed and unranked. For the right son we have to unrank $(s_{7.7}(2), 3)$, which delivers

$$k = 1, v_k = 3.4$$

Here, the local rank computes to

$$r_l = 0.$$
With
\[ R_{3,4}(2) = 0, R_{3,4}(1) = 0. \]

The subranks for the children compute to
\[ s_{3,4}(2) = 0, s_{3,4}(1) = 0. \]

Finally, unranking \((s_{3,4}(2), 0)\) yields
\[ k = 1, v_k = 2.3 \]

and for \((s_{3,4}(1), 0)\) we obtain
\[ k = 1, v_k = 1.3 \]

In total we unranked the operators 7.7, 4.3, 3.4, 2.3, and 1.3. They span the tree shown by darkened operators in Figure 2.5.

### 3.3.4 Verifying Query Processors

Besides its use to analyze cost distributions be it by sampling or complete enumeration (see next chapter), the counting and un-ranking mechanism presented is also of very practical relevance to the ongoing development process of Microsoft SQL Server: The verification of both query optimization and query execution [5].

The choice of an execution plan is the result of various, interacting factors, such as database and system state, current table statistics, calibration of costing formulas, algorithms to generate alternatives of interest, and heuristics to cope with the combinatorial explosion of the search space. Normally, experimental validation and testing of the query processor is limited to consider the one plan that was chosen by the optimizer for execution. This is a severe limitation, as this plan is only a minuscule fraction of the space of alternatives. In fact, during regular development and maintenance of a query processor, it has been our experience that some code defects can remain undetected for a long time, until the right combination of factors steer the optimizer to chose a plan that exposes the problem.

In [Slu98], Slutz presents a tool to generate SQL statements probabilistically, to increase the test coverage of the database engine. One simple advantage of this approach is the sheer speed at which new, different tests are generated, making it a very effective testing tool. The same claim can be made for the selection and execution of multiple plans given a single query, which increases even further the coverage of the optimizer logic.

In the current implementation in Microsoft SQL Server, we extended the SQL syntax with an option to specify what plan to use for the execution.
3.3. MEMO-BASED ENUMERATION

```sql
SELECT *
FROM Professors P, Students S, Enrolled E, Courses C
WHERE S.Name = "Sam White" AND
S.SID = E.SID AND
E.Title = C.Title AND
C.By = P.PID
OPTION (USEPLAN 8)
```

*Figure 3.14. Extended SQL syntax to specify what plan to use for execution*

The SQL statement in Figure 3.14 causes the optimizer to build the MEMO structure, count the possible plans, and select plan number 8 for execution.

Using scripting primitives, any given query can be extended easily with the OPTION clause and a loop construct that iterates over a deterministically or randomly selected set of possible plans. This way developers are for instance able to generate test cases for specific queries.

Some advantages of using these techniques in testing are:

1. It is easy to generate large test sets for the engine to scrutinize both correctness of the query execution and its performance.

2. The results are simple to verify since all plans should deliver the same outcome. The probability that an incorrect result is overlooked is rather small as opposed to external testing where each result requires manual verification.

3. It is possible to test operator implementations that the optimizer would not chose with the catalog data in the test database.

4. Optimizer decisions and correct assembling of plans by the optimizer can be easily verified. This point is of particular importance when extending the set of both operators and their implementations.

5. The verification and calibration of cost formulas is no longer restricted to one single plan per query but can also check cost values of sub-optimal plans.

6. The enumeration of complete search spaces for small queries helps check and analyze optimizer principles like cost-bound pruning and search strategies.

The features described are part of the routine testing in the development of Microsoft SQL Server.
3.4 Summary

The query optimization problem can be abstracted in several ways. The most common approach is to focus on join ordering only, since joins are in most cases the cost dominating operators. Another important model is the further simplification using only cross products. Like its richer brother, it is still NP-hard indicating already that cartesian products play a particularly important role.

In this chapter, we presented techniques how to generate all possible solutions for the XOPT, and JOPT, and for cost based query optimization in general. The first as well as the last are based on counting and unranking of n-ary trees, whereas the second one, for the JOPT, utilizes an algorithmic schema of sequences.

The techniques presented can now be used to derive complete or partial cost distributions by enumerating or sampling the search spaces of different problem instances.