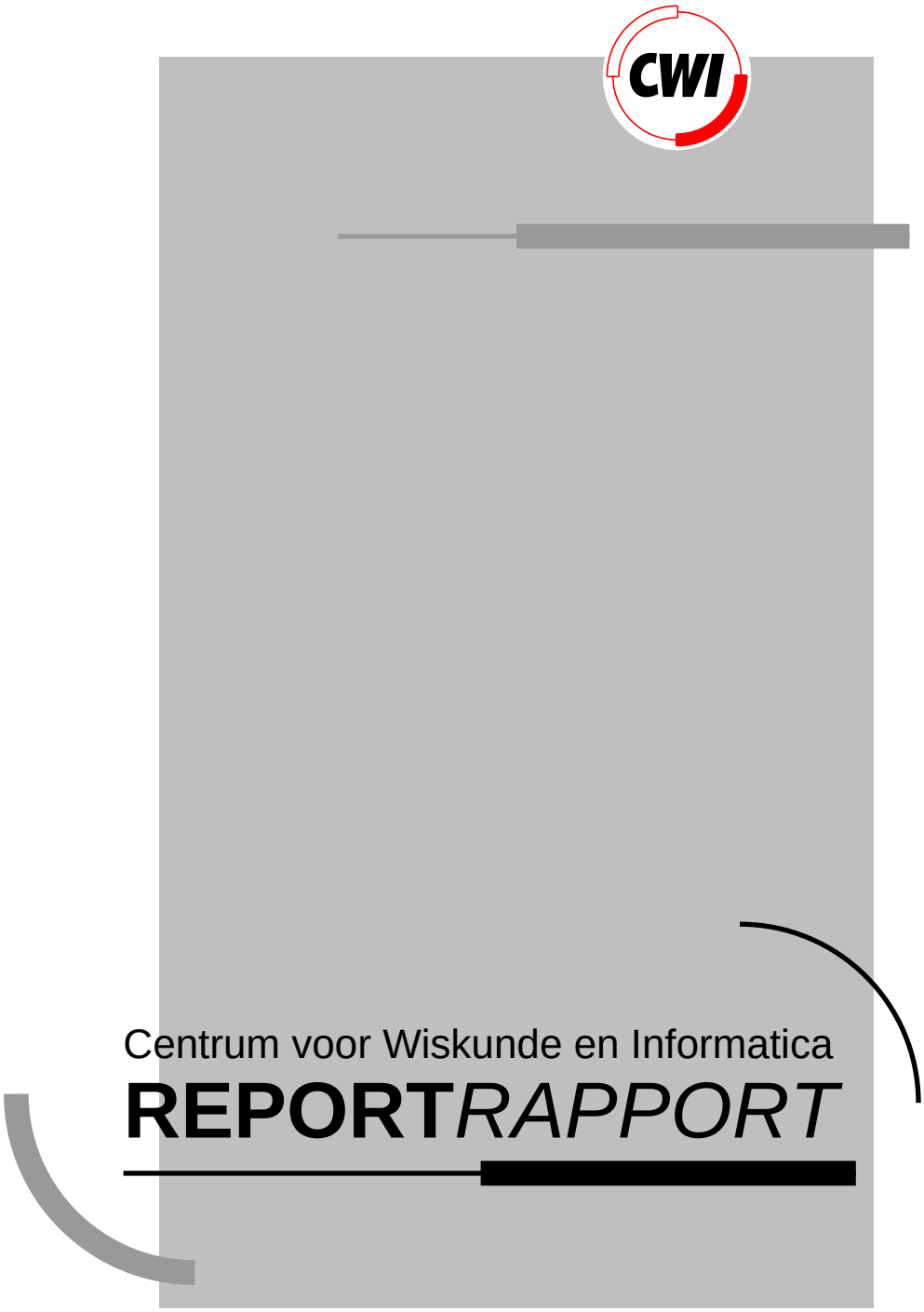


Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Uniformly-distributed random generation of join orders

C.A. Galindo Legaria, J. Pellenkoft, M.L. Kersten

Computer Science/Department of Algorithmics and Architecture

CS-R9431 1994

Uniformly-Distributed Random Generation of Join Orders

César A. Galindo-Legaria[1,2] cesar@acm.org
Arjan Pellenkoft[1] arjan@cwi.nl
Martin L. Kersten[1] mk@cwi.nl

[1] *CWI*
P. O. Box 94079, 1090 GB Amsterdam, The Netherlands

[2] *SINTEF DELAB*
N-7034 Trondheim, Norway

Abstract

In this paper we study the space of operator trees that can be used to answer a join query, with the goal of generating elements form this space at random. We solve the problem for queries with acyclic query graphs. First, we count the exact number of trees that can be used to evaluate a given query. Then, we establish a mapping between the n operator trees for a query and the integers 1 through n —i. e. a *ranking*— and describe efficient ranking and unranking procedures. The generation of random, uniformly distributed operator trees follows from the unraking.

CR Subject Classification (1991): [H.2.4] Database Systems, Query Processing; [G.2.2] Graph Theory; [F.2.2] Non-numerical algorithms and problems.
Keywords and Phrases: Join queries, query graphs, operator trees.
Note: C. A. Galindo-Legaria had an ERCIM postdoctoral fellowship while conducting this work.

1 Introduction

1.1 Background

The selection of a join evaluation order is a major task of relational query optimizers [Ull82, CP85, KRB85]. The problem can be stated as that of finding an operator tree to evaluate a given query, so that the estimated evaluation cost is minimum. In practice, the combinatorial nature of the problem prevents finding exact solutions, and both heuristics and randomized algorithms are considered as viable alternatives.

Two basic questions related to the space of operator trees of interest have remained open for some time now: What is the exact size of the space? And, how to generate a random element from the space efficiently? In this paper we answer those questions for the class of *acyclic queries* —those whose query graph, defined below, is acyclic. The answer to the second question has a direct application to randomized query optimization, as selection of a random item in the search space is a basic primitive for most randomized algorithms [SG88, Swa89b, Swa89a, IK90, IK91, Kan91, LVZ93, GLPK94].

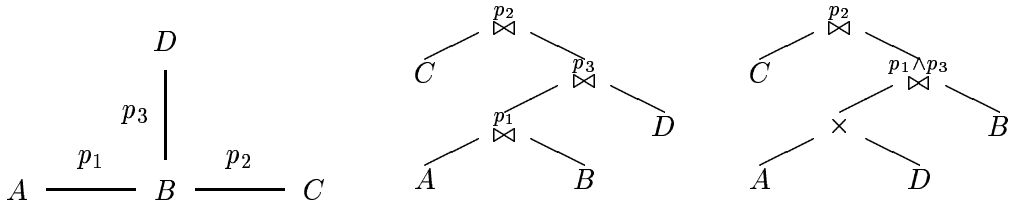


Figure 1: Query graph and operator trees.

Acceptable operator trees are subject to restrictions on which relations can be joined together, and counting them does not reduce, in general, to the enumeration of familiar classes of trees —e. g. binary trees, trees representing equivalent expressions on an associative operator, etc. A variety of techniques are used to enumerate graphs and trees [Knu68, HP73, RH77, VF90]. The scheme we use is similar to that used, for example, in [GLW82], in the sense that an auxiliary structure serves to guide the counting and ranking of elements of the space, instead of applying a closed formula.

Previous work has identified restricted classes of queries for which valid operator trees map one-to-one to permutations or unlabeled binary trees —the first class known as *star* queries, and the second as *chain* queries, see for example [OL90, IK91]— thus solving the counting and random generation problems for those classes. Since it is easy to generate valid operator trees non-determinstically, even in the general case, *quasi*-random selection of operator trees has been used in some work on randomized query optimization [SG88, Swa89a]. The term *quasi*-random refers to the fact that every valid tree has a non-zero probability of being selected, but some trees have a higher probability than others and, furthermore, there is no precise characterization of the probability distribution.

Another approach to generate random operator trees is to generate labeled binary trees uniformly at random, until one of them turns out to be a valid operator tree for the query at hand. The validity of an operator tree can be checked efficiently, but the small ratio of valid trees with respect to the total binary trees renders this method impractical [Swa89a, Swa91].

Our work on acyclic queries covers the star and chain queries as particular cases, and provides polynomial time algorithms both to count the number of operator trees for a given query, and to generate one of those trees uniformly at random.

Although acyclic queries cover perhaps most of the queries posed in practice, cyclic queries are frequent enough to deserve attention. We are currently studying the class of cyclic queries, but the problem is more difficult. Many database problems become significantly more complex when cyclic structures are allowed (see for example [BFMY83]), and the techniques we use for the acyclic case do not seem to extend easily to cyclic queries.

1.2 Query graphs and join trees

Figure 1 shows the graph representation of a query, called a *query graph*, and two operator trees to answer the query. In the query graph, nodes correspond to rela-

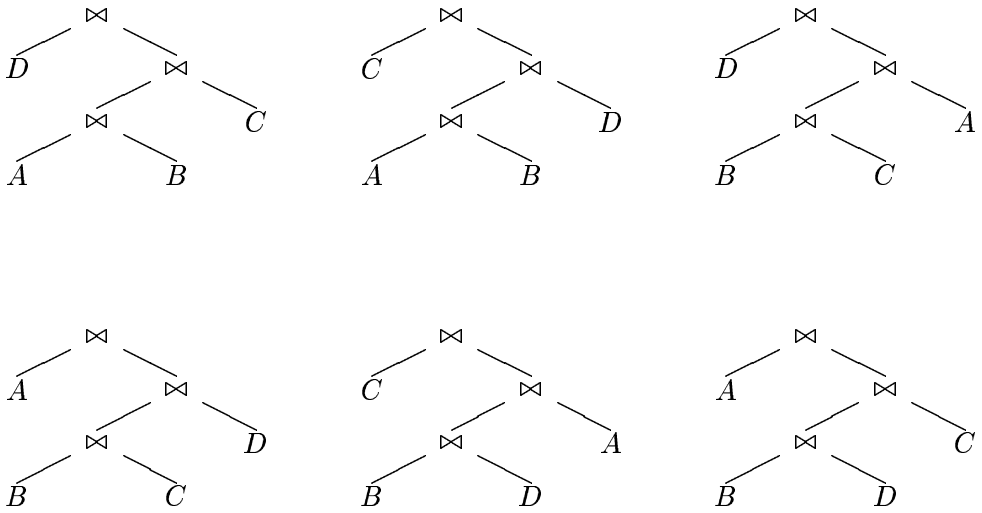


Figure 2: All join trees of the query graph.

tions of the database, and (undirected) edges correspond to selection conditions of the query. The graph shown corresponds to the query $\{(a,b,c,d) \mid a \in A, b \in B, c \in C, d \in D, p_1(a,b), p_2(b,c), p_3(b,d)\}$, where A, B, C, D are database relations and p_1, p_2, p_3 are binary predicates. In a database system, such a query is usualy evaluated by means of binary operators, and the two operator trees of Figure 1 can be used to answer this query. The first operator tree requires only relational joins, while the second requires Cartesian products. For a description of relational operators and query graphs, see, for example, [Ull82, CP85, KRB85]. The reason why a Cartesian product is required in the second tree is that we start by combining the information from relations A, D, but there is no edge (i. e. predicate) between them in the query graph. Figure 2 shows all 6 operator trees for this query in which only join is required, called *join trees* here. A purely graph-theoretical definition of join trees is given next.

Definition. An unordered binary tree T is called a *join tree* of query graph $G = (V, E)$ when it satisfies the following recursive definition:

- The leaves of T correspond one-to-one with the nodes of G; and
- every subtree of T is a join tree for connected subgraph of G.

Join trees are unordered —i. e. do not distinguish left from right subtree— because not all join-algorithm distinguish a left and right argument. Ordering a tree of n leaves requires a binary choice in each of the $n-1$ internal nodes, so there are 2^{n-1} ordered trees for each unordered tree of n relations. This mapping can be easily used to extend our counting and random generation of unordered join trees to the ordered variety.

In the sequel, we omit the operator $\bowtie$ when drawing join trees: A tree of the form $(T_1 \bowtie T_2)$ is written simply as $(T_1.T_2)$. We assume that query graphs are connected and

acyclic, i. e. we deal with *acyclic queries*.

We use $\mathcal{T}_G$ to denote the set of join trees of a query graph G, and $\mathcal{T}_G^{v(k)} \subseteq \mathcal{T}_G$ to denote join trees in which a given leaf v is at level k — with the root of a tree being at level 0. For example, for the query graph of Figure 1, Figure 2 shows that $\mathcal{T}_G$ consists of six trees, $\mathcal{T}_G^{D(1)}$ consists of only two trees, and $\mathcal{T}_G^{B(3)} = 6$, which happens to be equal to $\mathcal{T}_G$. An other way to compose $\mathcal{T}_G$ is by adding $\mathcal{T}_G^{C(1)}$, $\mathcal{T}_G^{C(2)}$ and $\mathcal{T}_G^{C(3)}$.

1.3 Lists

We introduce some notation and properties of lists that are used later in the paper. Square brackets are used as lists delimiters, as well as the list construction symbol "|" of Prolog —i. e. $[x|L]$ denotes the list obtained by inserting a new element x at the front of a list L. An array of values $x_0, \ldots, x_n$ in which index i stores value x_i, for $i = 0, \ldots n$, is represented as the list $[x_0, \ldots, x_n]$.

We say that a list L' is the *projection* of a list L on some property P of elements, if L' contains all the elements of L satisfying P, while also preserving the relative order of L —i. e. if x appears before y in L' then x appears before y in L. We say L is a *merge* of two lists L_1, L_2 without common elements, if the length of L is the sum of lengths of L_1, L_2, and both L_1, L_2 are projections of L.

The result of merging two lists is not unique, in general. Let L_1, L_2 be lists of length l_1, l_2, respectively. There is a one-to-one mapping between the result of merging L_1, L_2 and the problem of non-negative integer decomposition of l_1 in $l_2 + 1$ —that is, a list of $l_2 + 1$ non-negative integers $\alpha = [\alpha_0, \ldots, \alpha_{l_2}]$ such that their sum is equal to l_1. Operationally, the mapping is as follows. To obtain a merge L from a decomposition $[\alpha_0, \ldots, \alpha_{l_2}]$, start with the first α_0 elements of L_1, then use the first element from L_2; now use the next α_1 elements of L_1, then one from L_2. The last α_{l_2} elements of L_1 follow the last element of L_2 in L. We then say that L is the result of *merging* L_1, L_2 *using* α.

Since the decomposition of n in k can be solved in $\begin{pmatrix} n+k-1 \\ k-1 \end{pmatrix}$ ways [NW78], there are $M(l_1, l_2) = \begin{pmatrix} l_1+l_2 \\ l_2 \end{pmatrix}$ acceptable results of the merge of lists L_1, L_2, each identified with a specific decomposition. Observe that $M(l_1, l_2) = M(l_1, l_2 - 1) + M(l_1 - 1, l_2)$. A table of size $N \times N$ can be constructed in $O(N^2)$ time so that $M(l_1, l_2)$ is found by a simple lookup, for $l_1, l_2 \leq N$. Such table can also be used to rank and unrank decompositions of l_1 in $l_2 + 1$ —i. e. lists of the form $[\alpha_0, \ldots, \alpha_{l_2}]$ that determine a specific merge of L_1, L_2— in $O(N)$ and $O(N \log N)$ time, respectively.

2 Decomposition and construction of trees

2.1 Anchored-list representation

Since our arguments and constructions often rely on paths from the root of the join tree to a specific leaf, we introduce an *achored list* representation of trees. Elements of the anchored list are those subtrees observed while traversing the path from the root to some anchor leaf.

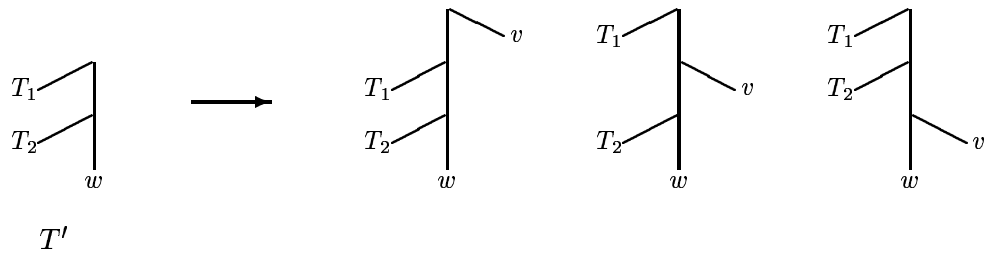


Figure 3: Construction by leaf-insertion.

Definition. Let T be a join tree and v be a leaf of T. The *list anchored on v of T*, call it L, is constructed as follows:

- If T is a single leaf, namely v, then $L = []$.
- Otherwhise, let $T = (T_l.T_r)$ and assume, without loss of generality, that v is a leaf ot T_r. Let L_r be the list of T_r anchored on v. Then $L = [T_l|L_r]$.

Then we say that $T = (L, v)$.

Observe that if $T = (L, v)$ is an element of $\mathcal{T}_G^{v(k)}$, then the length of the anchored list L is k.

2.2 Primitive operations

We now describe procedures that relate a join tree of a query graph G with some join trees of subgraphs of G. Applied in one direction, these procedures *construct* a join tree based on smaller join trees; applied in the other direction, they *decompose* join trees.

Our first procedure is *leaf insertion*. The idea is that two join trees are related by the insertion/removal of a leaf. The operation is stated as the insertion/removal of a one-leaf tree in the anchored list representation of join trees.

Definition. Let $G = (V, E)$ be a query graph and T be a join tree of G. Assume $v \in V$ is such that $G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$.

- Let $T = ([T_1, \ldots, T_{k-1}, v, T_{k+1}, \ldots, T_n], w)$.
- Let $T' = ([T_1, \ldots, T_{k-1}, T_{k+1}, \ldots, T_n], w)$.

We call (T', k) and *insertion pair on v*. We say that T is *decomposed into* pair (T', k) on v, or, equivalently, that T is *constructed from* pair (T', k) on v.

Example 1 Figure 3 shows a join tree $T' = ([T_1, T_2], w)$ and the join trees constructed from insertion pairs $(T', 1)$, $(T', 2)$, and $(T', 3)$ on v. ■

Observation 1 Let $G = (V, E)$ be a query graph with n nodes. Assume $v \in V$ is such that $G' = G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$, and $1 \leq k < n$. The leaf-insertion operation defines a one-to-one mapping between elements of $\mathcal{T}_G^{v(k)}$ and insertion pairs on v of the form (T', k), where T' is an element of the disjoint union $\cup_{i=k-1}^{n-2} \mathcal{T}_{G'}^{w(i)}$.

Our second procedure is *tree merging*. The idea is that a join tree can be obtained by merging two smaller join trees. The operation is stated as the merge/projection of the anchored list representation of join trees.

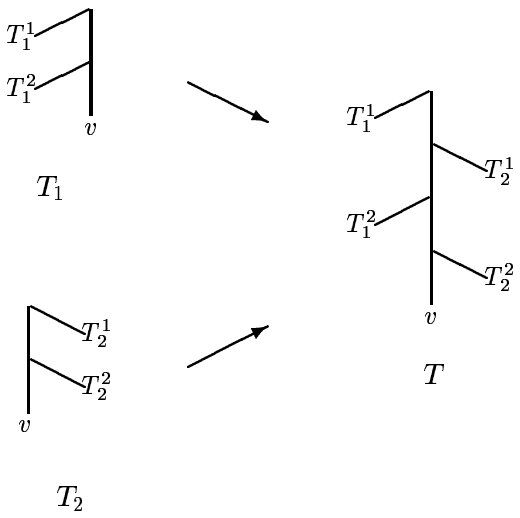


Figure 4: Construction by tree-merging.

Definition. Let $G = (V, E)$ be a query graph and T be a join tree of G. Assume sets of edges V_1, V_2 are such that $G|_{V_1}, G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$.

- Let $T = ([T_1, \ldots, T_n], v)$.
- Define property P_1 (respectively P_2) to be "every leaf of the subtree is in V_1 (V_2)."
- Let L_1, L_2 be the projection of L on properties P_1, P_2, respectively. Let α be an integer composition such that L is the result of merging L_1, L_2 using α.
- Let $T_1 = (L_1, w)$ and $T_1 = (L_2, w)$.

We call (T_1, T_2, α) a *merge triplet*. We say T is *decomposed into* triplet (T_1, T_2, α) on V_1, V_2, or, equivalently, that T is *constructed from* triplet (T_1, T_2, α) on V_1, V_2.

Example 2 Figure 4 shows join trees $T_1 = ([T_1^1, T_1^2], v), T_2 = ([T_2^1, T_2^2], v)$ and the join tree $T = ([T_1^1, T_2^1, T_1^2, T_2^2], v)$ constructed from merge triplet $(T_1, T_2, [1, 1, 0])$. ∎

Observation 2 Let $G = (V, E)$ be a query graph with n nodes. Assume sets of edges V_1, V_2 are such that $G_1 = G|_{V_1}, G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$, and let $1 \leq k < n$. The tree-merging operation defines a one-to-one mapping between elements of $\mathcal{T}_G^{v(k)}$ and merge triplets on V_1, V_2 of the form (T_1, T_2, α), where $T_1 \in \mathcal{T}_{G_1}^{v(i)}, T_2 \in \mathcal{T}_{G_2}^{v(k-i)}$, and α specifies a merge of two lists of size i, $k - i$ respectively.

2.3 Standard decompositions

Join trees can be decomposed into a sequence of leaf-insertion and tree-merging operations, but these decompositions are not unique, in general. A key structure for our algorithms is the *standard decomposition graph*, which is obtained by selecting an arbitrary order of operations to construct the join trees of some graph G. Join decompositions are then unique with respect to the standard order defined.

A standard decomposition graph H of G can be viewed as a generic program (or operator tree) to build join trees of a given query graph. Unary nodes of H, labeled "$+_x$," construct

```
CONVERT-TO-SDG(v)
  Let x be the label of v.
  Let {w_1, ..., w_n} be the children of v.
  If n = 0
      Label v as "x".
  If n = 1
      Label v as "+_x";
      CONVERT-TO-SDG(w_1).
  If n > 1
      Label v as "×_x";
      create new nodes l, r, with label x;
      delete all edges (v, w_1), ..., (v, w_n);
      create new edges (v, l), (v, r), (l, w_1), (r, w_2), ..., (r, w_n);
      CONVERT-TO-SDG(l), CONVERT-TO-SDG(r).
```

Figure 5: Algorithm to obtain a standard decomposition graph.

a join tree by inserting a leaf x on its argument; binary nodes of H, labeled "$\times_x$," construct a join tree by merging two trees whose only common leaf is x.

Definition. A *standard decomposition graph* H of a query graph $G = (V, E)$ is obtained by modifying G as follows:

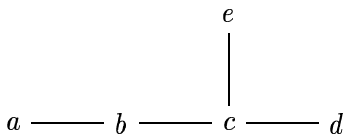
- Pick a node, say $v \in V$, as *root*. Direct the edges in E from the root v outwards to obtain G'. If there is a directed edge from u to w we say u is the *parent* of w. If there is a directed edge (of length zero or more) from u to w we say u is an *ancestor* of w. *Child* and *descendant* are the inverses of parent and ancestor, respectively.
- Transform G' using algorithm CONVERT-TO-SDG(r), shown in Figure 5, where r is the root chosen earlier. The result of this transformation is H.

The *labels of descendants* of a node v in H, denoted desc(v), is the set of node labels $\{w_i\}$ of G that appear in the descendants of v in the form "$\times_{w_i}$," "$+_{w_i}$," or "w_i."

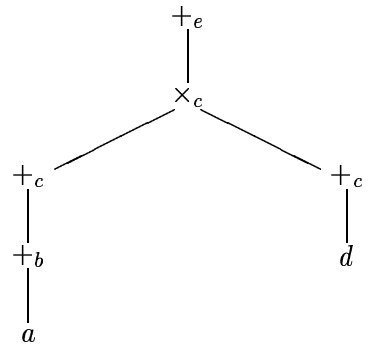
Example 3 Figure 6 shows a query graph and a standard decomposition graph obtained from it. In this case node e was selected as root. The labels of descendants of the node v labeled "$+_b$" in H is desc(v) = $\{a, b\}$. The labels of descendants of "$+_e$" is $\{a, b, c, d, e\}$. ■

When an insertion level k is selected at each node labeled "$+_x$," and a merge specification α is selected at each node labeled "$\times_x$," a standard decomposition graph become a complete "program" to construct a join tree. The annotations in a graph H necessary to construct T are called the *standard decomposition* of T in H.

Let r be the root of a standard decomposition graph H of G, and let T be a join tree of G. The standard decomposition of T in H is obtained by applying the procedure DECOMPOSE(T,r), defined in Figure 7.



Query graph G.



Standard decomposition graph H.

Figure 6: Query graph and standard decomposition graph.

DECOMPOSE(T, v)
  Let $\{w_1, \ldots, w_n\}$ be the children of v in H.
  If $n = 1$
    Let $+_x$ be the label of v.
    decompose T into an insertion pair, say (T', k), on x.
    annotate v as *insert-at* k;
    DECOMPOSE(T, w_1).
  If $n > 1$
    Let V_1, V_2 be the labels of desc(w_1), desc(w_2), respectively.
    decompose T into a merge triplet, say (T_1, T_2, α), on V_1, V_2.
    annotate v as *merge-using* α;
    DECOMPOSE(T_1, w_1), DECOMPOSE(T_2, w_2).

Figure 7: Algorithm to obtain the standard decomposition of a tree in H.

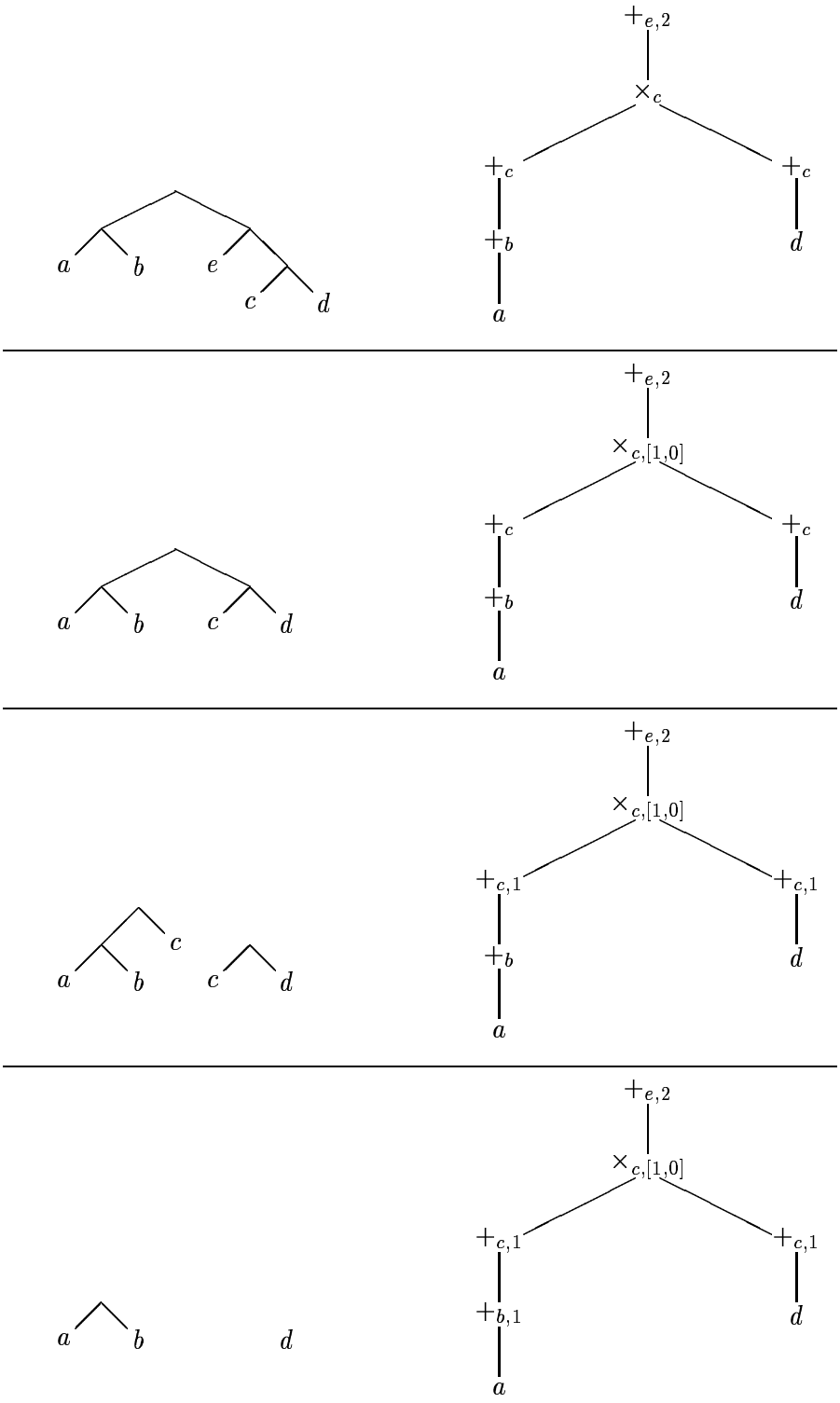


Figure 8: Obtaining the standard decomposition of a join tree.

Example 4 Figure 8 shows the process of obtaining a standard decomposition of a join tree T, using the standard decomposition graph of figure 6. First, the *insert-at* annotation of the root "$+_e$" is 2, because the join tree is the result of inserting e at level 2 in a smaller tree. This is shown as a label "$+_{e,2}$" in the first row of the figure. Then, the *merge-using* annotation of the node "$\times_c$" is $[1,0]$, because the anchored list on c of the join tree is $[(a.b),d]$, which results from the merge of lists $[(a.b)],[d]$ using $[1,0]$. This is shown as a label "$\times_{c,[1,0]}$" in the second row of the figure. The remaining annotations are obtained similarly. ■

3 Counting join trees

Our counting scheme is based on the tree decompositions described in section 2. We first derive recurrence equations relating the number of join trees of a query graph G with the number of trees of subgraphs of G. Then we apply these equations in the context of a standard decomposition graph.

3.1 Recurrence equations

Observation 3 The following equations serve as base cases for the computation of the number of join trees of a graph $G = (V, E)$, namely $|\mathcal{T}_G|$. Let $n = |V|$ and $v \in V$.

- If the graph has only one node, then it has only one join tree T, and v is at level 0 in T. That is,
$$|\mathcal{T}_G| = \left|\mathcal{T}_G^{v(0)}\right| = 1, \text{ for } n = 1.$$
- If the graph has more than one node, then it has no association tree in which v is at level 0. That is,
$$\left|\mathcal{T}_G^{v(0)}\right| = 0, \text{ for } n > 1.$$
- There is no association tree in which v is at level greater than or equal to n. That is,
$$\left|\mathcal{T}_G^{v(i)}\right| = 0, \text{ for } i \geq n.$$
- Since v appears at some unique level in any association tree of G, the total number of association trees is
$$|\mathcal{T}_G| = \sum_i \left|\mathcal{T}_G^{v(i)}\right| \ .$$

Now, the next two lemmas determine the number of join trees that can be constructed using our primitive operations of section 2.2.

Lemma 1. *Let $G = (V, E)$ be a query graph with n nodes. Assume $v \in V$ is such that $G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$, and $1 \leq k < n$. Then*

$$\left|\mathcal{T}_G^{v(k)}\right| = \sum_{i \geq k-1} \left|\mathcal{T}_{G'}^{w(i)}\right| \ .$$

Proof. The lemma follows from observation 1 in section 2.2.

COUNT-JT(v)
  Let $\{w_1, \ldots, w_n\}$ be the children of v.
  If $n = 0$
    annotate v with *count-array* $[1]$.
  If $n = 1$
    COUNT-JT(w_1);
    let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;
    annotate v with *count-array* $[0, z_1, \ldots, z_{n_1+1}]$,
      where $z_k = \sum_{i=k-1}^{n_1} x_i$, for $k = 1, \ldots, n_1 + 1$.
  If $n = 2$
    COUNT-JT(w_1), COUNT-JT(w_2);
    let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;
    let $[y_0, \ldots, y_{n_2}]$ be the *count-array* of w_2;
    annotate v with *count-array* $[z_0, \ldots, z_{n_1+n_2}]$,
      where $z_k = \sum_{i=0}^{n_1} x_i \, y_{k-i} \left(\begin{array}{c} k \\ i \end{array} \right)$, for $k = 1, \ldots, n_1 + n_2$.[a]

[a] To simplify the description, we assume that $y_i = 0$ for $i \notin \{0, \ldots, n_2\}$.

Figure 9: Algorithm to count the number of join trees.

Lemma 2. *Let $G = (V, E)$ be a query graph with n nodes. Assume sets of edges V_1, V_2 are such that $G|_{V_1}, G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$, and let $1 \leq k < n$. Then*

$$\left|\mathcal{T}_G^{v(k)}\right| = \sum_i \left|\mathcal{T}_{G_1}^{v(i)}\right| \cdot \left|\mathcal{T}_{G_2}^{v(k-i)}\right| \cdot \left(\begin{array}{c} k \\ i \end{array} \right) .$$

Proof. The lemma follows from observation 2 in section 2.2.

3.2 Counting standard decompositions

The standard decomposition graph defined in section 2.3 is used as an auxiliary structure in the computation of the number of join trees of a query graph. Viewing the standard decomposition graph again as a program (or operator tree), a bottom-up traversal is used to determine how many join trees can be constructed by a given operation, based on the number of trees that its children can construct. At each node we use either lemma 1 or 2 directly to determine the number of trees that can be constructed, and the result is incorporated in the graph as a *count-array* annotation of the node.

For a node u labeled $\odot_v$ (with $\odot \in \{+, \times\}$), the *count-array* annotation has the form $[x_0, \ldots, x_n]$. The interpretation is that node u can construct x_i different trees in which leaf v is at level i. To determine the total number of join trees for a query, just sum all entries of the *count-array* annotation in the root of the standard decomposition graph.

Let r be the root of a standard decomposition graph H. To find the *count-array* annotations of H apply the procedure COUNT-JT(r), defined in Figure 9.

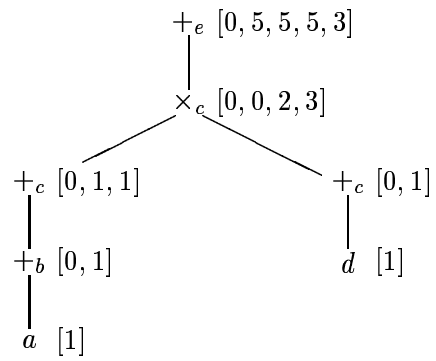


Figure 10: Standard decomposition graph with *count-array* annotations.

Example 5 Figure 10 shows the *count-array* annotations on the decomposition graph of Figure 6. The total number of different join trees for this query is 18. ■

Theorem 1. *The number of join trees of a given acyclic query graph G with n nodes can be computed in $O(n^3)$ time.*

Proof. A standard decomposition graph H of G can be constructed in linear time using algorithm CONVERT-TO-SDG in figure 5. The number of nodes of H is linear on n. The *count-array* annotations in H are obtained using algorithm COUNT-JT in figure 9 in $O(n^3)$ time, since the computation required per node is quadratic at worst. Finally, the number of join trees of G is the sum of the $O(n)$ values in the *count-array* of the root of H.

4 Ranking and unranking

Ranking is the process of mapping a set of n elements to the integers 1 through n. In our case the elements are trees and the set of trees is $\mathcal{T}_G$. A ranking function computes for a given tree, T, its rank, r, using the standard decomposition graph H of query G .

Unranking is the inverse of ranking. It determines which element of a set corresponds to a given number. The result of unranking k, using the standard decomposition graph H of query G, is a tree T which has rank k.

4.1 Mapping trees to integers

Our mapping between the N join trees of a query graph and the integers 1 through N is based on the recursive application of the following idea. Assume we want to rank an element $x \in S$, and S is partitioned into sets $S_0, \ldots, S_m$. If $x \in S_k$, for some $k \leq m$, and we can find a *local rank* of x in S_k, then we can set the rank of x in S to be local-rank$(x, S_k) + \sum_{i=0}^{k-1} |S_i|$. Conversely, to unrank some number y under our scheme, first find the set S_k from which the element must be retrieved, where $k = \min_j\ y \leq \sum_{i=0}^{j} |S_i|$. Then find the local rank $y' = y - \sum_{i=0}^{k-1} |S_i|$, and finally unrank-local(y', S_k).

RANK(T)
Let v be the root of the standard decomposition graph.
DECOMPOSE(T, v).
LOCAL-RANK(v).
Let (r, k) be the *local-rank* of v.
Let $[z_0, \ldots, z_n]$ be the *count-array* of v.
Return $r + \sum_{i=0}^{k-1} z_i$.

UNRANK(r)
Let v be the root of the standard decomposition graph.
Let $[z_0, \ldots, z_n]$ be the *count-array* of v.
Let k be $\min_j\ r \leq \sum_{i=0}^{j} z_i$.
Let r' be $r - \sum_{i=0}^{k-1} z_i$.
LOCAL-UNRANK(v, r', k).
The resulting annotations *insert-at* and *merge-using* define the tree whose rank is r.

Figure 11: Ranking and unranking algorithms.

In the case of join trees of a query of n relations, the set $\mathcal{T}_G$ is partitioned into sets $\mathcal{T}_G^{v(0)}, \ldots, \mathcal{T}_G^{v(n-1)}$, for any given leaf v. For example, for the annotated standard decomposition graph of figure 10, the numbers 1 through 5 are assigned to join trees in which leaf e is at level 1; numbers 6 through 10 are assigned to those in which e is at level 2; numbers 11 through 15 are assigned to those in which e is at level 3; and finally 16 through 18 are assigned to those in which e is at level 4.

Figure 11 shows algorithms to rank and unrank trees, based on a new annotation *local-rank* in the standard decomposition graph, as well as procedures LOCAL-RANK and LOCAL-UNRANK described below.

The procedure LOCAL-RANK operates on a standard decomposition graph H of a query graph G, with annotations *insert-at* and *merge-using* that define a tree T. In addition, H must also have annotations *count-array*. The procedure creates annotations *local-rank* on the nodes of the graph. The interpretation of a *local-rank* annotation of the form (r, k) in the root $\odot_v$ of H is that T has local rank r in the set $\mathcal{T}_G^{v(k)}$.

For the same graph H of G, but without *insert-at* and *merge-using* annotations, the procedure LOCAL-RANK finds (the *insert-at* and *merge-using* annotations that define) a tree with rank r in $\mathcal{T}_G^{v(k)}$, given r, k as input.

4.2 Local ranking

For the local ranking of a tree, we again use the standard decomposition graph and the primitive tree construction operations of section 2.2. The summands used to compute $\left|\mathcal{T}_G^{v(k)}\right|$ in lemmas 1 and 2 correspond to well-defined subsets of $\mathcal{T}_G^{v(k)}$. The partition defined by those subsets is appropriate for our needs.

Observation 4 Let $G = (V, E)$ be a query graph with n nodes. Assume $v \in V$ is such that $G' = G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$, and $1 \leq k < n$. The set $\mathcal{T}_G^{v(k)}$ can be partitioned into sets $\mathcal{T}_G^{v(k),k-1}, \mathcal{T}_G^{v(k),k}, \ldots, \mathcal{T}_G^{v(k),n-2}$, where $T \in \mathcal{T}_G^{v(k),i}$ if $T \in \mathcal{T}_G^{v(k)}$, the insertion pair on v of T is (T', k), and $T' \in \mathcal{T}_{G'}^{w(i)}$. The size of each partition is

$$\left|\mathcal{T}_G^{v(k),i}\right| = \left|\mathcal{T}_{G'}^{w(i)}\right|.$$

Observation 5 Let $G = (V, E)$ be a query graph with n nodes. Assume sets of edges V_1, V_2 are such that $G_1 = G|_{V_1}, G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$, and let $1 \leq k < n$. The set $\mathcal{T}_G^{v(k)}$ can be partitioned into sets $\mathcal{T}_G^{v(k),0}, \mathcal{T}_G^{v(k),1}, \ldots, \mathcal{T}_G^{v(k),k}$, where $T \in \mathcal{T}_G^{v(k),i}$ if $T \in \mathcal{T}_G^{v(k)}$, the merge triplet on V_1, V_2 of T is (T_1, T_2, α), and $T_1 \in \mathcal{T}_{G_1}^{v(i)}$. The size of each partition is

$$\left|\mathcal{T}_G^{v(k),i}\right| = \left|\mathcal{T}_{G_1}^{v(i)}\right| \cdot \left|\mathcal{T}_{G_2}^{v(k-i)}\right| \cdot \begin{pmatrix} k \\ i \end{pmatrix} .$$

Just as the annotations *count-array* provide the necessary set partition information in the RANK and UNRANK procedures of Figure 11, we use a new annotation *summands* to store information about the partitions introduced in observations 4 and 5.

The *summands* annotation is an array $\sigma = [\sigma_0, \ldots, \sigma_n]$, whose elements in turn are arrays of the form $\sigma_k = [\sigma_{k0}, \ldots, \sigma_{km}]$. If σ is the *summands* annotation of a node whose *count-array* annotation is $[x_0, \ldots, x_m]$, then it holds that $x_k = \sum_{i=0}^{m} \sigma_{ki}$, for $k = 0, \ldots, n$.

Let r be the root of a standard decomposition graph H. To find the *summands* annotations of H apply the procedure SUMMANDS(r), defined in Figure 12.

Algorithms LOCAL-RANK and LOCAL-UNRANK are shown in Figures 13 and 14, respectively. They implement recursively the idea of ranking in terms of set partitions, whose one-level version is the basis of RANK and UNRANK. The necessary information is stored in the *count-array* and *summands* annotations.

We do not describe in detail the procedures RANK-DECOMPOSITION and UNRANK-DECOMPOSITION, but they are relatively straightforward. The issue is briefly mentioned at the end of section 1.3.

The procedure RANK-TRIPLET$(a, b, c; A, B, C)$ computes the rank r of a triplet (a, b, c) from the set $\{(x, y, z) \mid 1 \leq x \leq A, 1 \leq y \leq B, 1 \leq z \leq C\}$, and its inverse is UNRANK-TRIPLET$(r; A, B, C)$. These are also straightforward.

Example 6 Figure 15 shows the results of the local ranking for the tree T of example 4. The graph contains the annotations of the standard decomposition of T shown in figure 8, the *count-array* annotations of figure 10, and the *summands* annotations computed by SUMMANDS. Procedure LOCAL-RANK is used to compute annotations *local-rank*. The annotations of a node "$\odot_v$" of the decomposition graph are shown in the figure in the format

$$(k, l)\ \odot_{vp}\ [x_0, \ldots, x_n]\ \ \sigma_k = [\sigma_{k0}, \ldots, \sigma_{km}],$$

where p is the *insert-at* annotation if $\odot = +$, or else the *merge-using* annotation if $\odot = \times$; $[x_0, \ldots, x_n]$ is the *count-array* annotation; $\sigma = [\sigma_0, \ldots, \sigma_l]$ is the *summands* annotation, but only σ_k is shown. Finally, (k, l) is the local rank computed at the node; that is, the subtree computed at the given node has rank k in set $\mathcal{T}_{G'}^{v(l)}$.

SUMMANDS(v)

Let $\{w_1, \ldots, w_n\}$ be the children of v.

If $n = 0$

there is no *summands* annotation in v.

If $n = 1$

SUMMANDS(w_1);

let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;

annotate v with *summands* $[\sigma_0, \sigma_1, \ldots, \sigma_{n_1+1}]$,

where $\sigma_k = [\sigma_{k0}, \sigma_{k1}, \ldots, \sigma_{kn_1}]$, for $k = 1, \ldots, n_1 + 1$,

and $\sigma_{ki} = \begin{cases} x_i & \text{if } 0 < k \text{ and } k - 1 \leq i; \\ 0 & \text{otherwise.} \end{cases}$

If $n = 2$

SUMMANDS(w_1), SUMMANDS(w_2);

let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;

let $[y_0, \ldots, y_{n_2}]$ be the *count-array* of w_2;

annotate v with *summands* $[\sigma_0, \sigma_1, \ldots, \sigma_{n_1+n_2}]$,

where $\sigma_k = [\sigma_{k0}, \sigma_{k1}, \ldots, \sigma_{kn_1}]$, for $k = 1, \ldots, n_1 + n_2$,

and $\sigma_{ki} = \begin{cases} x_i \, y_{k-i} \begin{pmatrix} k \\ i \end{pmatrix} & \text{if } 0 \leq k - i \leq n_2; \\ 0 & \text{otherwise.} \end{cases}$

Figure 12: Algorithm to compute set partition information.

LOCAL-RANK(v)

Let $\{w_1, \ldots, w_n\}$ be the children of v.

If $n = 0$

annotate v with *local-rank* $(1, 0)$.

If $n = 1$

LOCAL-RANK(w_1);

let (r_1, k_1) be the *local-rank* of w_1;

let k be the *insert-at* of v;

let $[\sigma_0, \ldots, \sigma_n]$ be the *summands* of v;

annotate v with *local-rank* (r, k), where $r = r_1 + \sum_{i=0}^{k_1 - 1} \sigma_{ki}$.

If $n = 2$

LOCAL-RANK(w_1), LOCAL-RANK(w_2);

let (r_1, k_1) be the *local-rank* of w_1;

let (r_2, k_2) be the *local-rank* of w_2;

let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;

let $[y_0, \ldots, y_{n_2}]$ be the *count-array* of w_2;

let k be $k_1 + k_2$;

let α be the *merge-using* of v;

let q be RANK-TRIPLET $(r_1, r_2,$RANK-DECOMPOSITION$(\alpha); x_{k_1}, y_{k_2}, \begin{pmatrix} k \\ i \end{pmatrix})$.

annotate v with *local-rank* (r, k), where $r = q + \sum_{i=0}^{k_1 - 1} \sigma_{ki}$.

Figure 13: Algorithm for local ranking.

LOCAL-UNRANK(v, r, k)

Let $\{w_1, \ldots, w_n\}$ be the children of v.

If $n = 0$

arguments are consistent if $r = 1$, $k = 0$;

there is no additional annotation on v.

If $n = 1$

let $[z_0, \ldots, z_n]$ be the *count-array* of v;

let $[\sigma_0, \ldots, \sigma_n]$ be the *summands* of v;

arguments are consistent if $k \leq n$, $r \leq z_k$;

let k_1 be $\min_j\ r \leq \sum_{i=0}^{j} \sigma_{ki}$;

let r_1 be $r - \sum_{i=0}^{k_1-1}$;

annotate v with *insert-at* k;

LOCAL-UNRANK(w_1, r_1, k_1).

If $n = 2$

let $[z_0, \ldots, z_n]$ be the *count-array* of v;

let $[\sigma_0, \ldots, \sigma_n]$ be the *summands* of v;

let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;

let $[y_0, \ldots, y_{n_2}]$ be the *count-array* of w_2;

arguments are consistent if $k \leq n$, $r \leq z_k$;

let k_1 be $\min_j\ r \leq \sum_{i=0}^{j} \sigma_{ki}$;

let k_2 be $k - k_1$;

let q be $r - \sum_{i=0}^{k_1-1}$;

let (r_1, r_2, a) be UNRANK-TRIPLET$(q; x_{l_1}, y_{l_2}, \begin{pmatrix} k \\ i \end{pmatrix})$.

let α be UNRANK-DECOMPOSITION(a);

annotate v with *merge-using* α;

LOCAL-UNRANK(w_1, r_1, k_1), LOCAL-UNRANK(w_2, r_2, k_2).

Figure 14: Algorithm for local unranking.

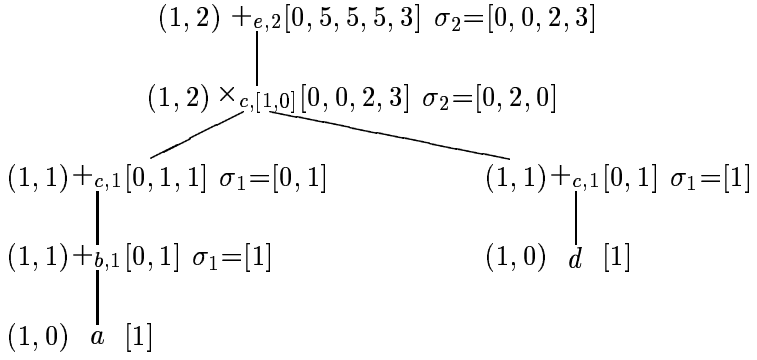


Figure 15: Local ranking of (the standard decomposition of) a join tree.

At node $\times_c$ the *merge-using annotation* is $[1, 0]$. For the purposes of this example we assume that RANK-DECOMPOSITION$([1, 0]) = 1$ and RANK-TRIPLET$(1, 1, 1; 1, 1, 2) = 1$.

Now, applying RANK on the graph resulting from LOCAL-RANK, we determine that the rank of T is 6. ∎

4.3 Efficiency of ranking and unranking

Once the *count-array* and *summands* annotations of a graph are available, ranking and unranking of join trees is based on traversing arrays, for the most part. Actually, the "bottleneck" of the process is the ranking and unranking of integer decompositions, since each decomposition may have as many as $O(n)$ elements. The relatively simple algorithms we use now take $O(n)$ time to rank and $O(n \log n)$ time to unrank.

Theorem 2. *Let G be a query graph on n relations. After a preprocessing step of $O(n^3)$ time, association trees of G can be ranked in $O(n^2)$ time and unranked in $O(n^2 \log n)$ time.*

Proof. By theorem 1, the standard decomposition graph H of G and its *count-array* annotations can be computed in $O(n^3)$ time. To compute the *summands* annotation we use algorithm SUMMANDS, which takes also $O(n^3)$ time, because it requires $O(n^2)$ per node. This completes the preprocessing step of $O(n^3)$ time. To rank a tree, the most expensive procedure is that of LOCAL-RANK. In the worst case, the time taken per node is $O(n)$ due to the ranking of integer decompositions. The total ranking time, then, is bounded by $O(n^2)$. To unrank a tree, the most expensive procedure is that of LOCAL-UNRANK. In the worst case, the time taken per node is $O(n \log n)$ due to the unranking of integer decompositions. The total unranking time, then, is bounded by $O(n^2 \log n)$

5 Generating random join trees

Uniformly distributed random generation of join trees follows from our results on counting and unranking. To generate random join trees for a given query graph G, first count the number of join trees in the space as described in section 3; say there are N join trees. Now, simply generate a random number r between 1 and N, and unrank the join tree of r as described in section 4. This can be done efficiently.

Theorem 3. *Let G be a query graph on n relations. Assuming a source of random bits, join trees for G can be generated at random with uniform distribution in time $O(n^2 \log n)$ per tree, after a preprocessing step of $O(n^3)$ time.*

Proof. To generate random join trees follow the procedure outlined above. Time bounds follow from theorem 2.

6 Discussion

In this paper we have described procedures to count the number of join trees that can be used to evaluate a given query, and to generate them uniformly at random. The difficulty

of those problems results from the fact that there is no one-to-one mapping between join trees and a simple combinatorial structure.

Our concept of a standard decomposition graph provides a supporting structure for counting and random generation, because it defines a canonical construction for each tree. In addition, computing an array of values that characterizes the number of canonical constructions can be computed bottom up in an efficient way.

We gave priority to clarity over efficiency when describing our algorithms, and the reader must be aware that there are obvious optimizations. None of those optimizations, however, seems to improve the time bounds stated in our theorems.

The integers required by our algorithms can become quite large, as is the case with other graph counting/generation problems [vL90],section 10.1.5. This eventually limits the applicability of our current results. Nevertheless, the algorithms can be used to a good extent on practical database queries (e. g. certainly for queries of 20 relations).

Acknowledgements. We are grateful to Zandra-Navarro-Villicaña for her suggestions and comments on preliminary drafts of this paper.

References

[BFMY83] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, July 1983.

[CP85] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, New York, 1985.

[GLPK94] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, randomized join-order selection —Why use transformations? In *Proceedings of the Twentieth International Conference on Very Large Databases, Santiago*, 1994. Also CWI Technical Report CS-R9416.

[GLW82] U. Gupta, D. T. Lee, and C. K. Wong. Ranking and unranking of 2-3 trees. *SIAM Journal of Computation*, pages 582–590, August 1982.

[HP73] F. Harary and E. M. Palmer. *Graphical Enumeration.* Academic Press, 1973.

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 312–321, 1990.

[IK91] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 168–177, 1991.

[Kan91] Y. C. Kang. *Randomized Algorithms for Query Optimization.* PhD thesis, University of Wisconsin-Madison, 1991. Technical report #1053.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 1968. Second edition, 1973.

[KRB85] W. Kim, D. S. Reiner, and D. S. Batory, editors. *Query processing in database systems.* Springer, Berlin, 1985.

[LVZ93] R. S. G. Lanzelotte, P. Valduriez, and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. *Proc. of the 19th VLDB Conference, Dublin, Ireland*, pages 493–504, 1993.

[NW78] A. Nijenhuis and H. S. Wilf. *Combinatorial algorithms.* Academic Press, New York, 2nd edition, 1978.

[OL90] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. *Proc. of the 16th VLDB Conference, Brisbane, Australia*, pages 314–325, 1990.

[RH77] F. Ruskey and T. C. Hu. Generating binary trees lexicographically. *SIAM journal of Computation*, 6(4):745–758, December 1977.

[SG88] A. N. Swami and A. Gupta. Optimization of large join queries. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 8–17, 1988.

[Swa89a] A. N. Swami. *Optimization of Large Join Queries.* PhD thesis, Stanford University, 1989. Technical report STAN-CS-89-1262.

[Swa89b] A. N. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 367–376, 1989.

[Swa91] A. N. Swami. Distribution of query plan costs for large join queries. Technical Report RJ 7908, IBM Research Division, Almaden, 1991.

[Ull82] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, 2nd edition, 1982.

[VF90] J. S. Vitter and Ph. Flajolet. Analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 9, pages 431–524. North Holland, 1990.

[vL90] J. van Leeuwen. Graph algorithms. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. North Holland, 1990.