Uniformly-Distributed Random Generation of Join Orders

César A. Galindo-Legaria[1,2] * Arjan Pellenkoft[1] Martin L. Kersten[1]

[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. We first count, in $O(n^3)$ time, the exact number of trees that can be used to evaluate a given query on n relations. The intermediate results of the counting procedure then serve to generate random, uniformly distributed operator trees in $O(n^2)$ time per tree. We also establish a mapping between the N operator trees for a query and the integers 1 through N —i. e. a *ranking*— and describe ranking and unranking procedures with complexity $O(n^2)$ and $O(n^2 \log n)$, respectively.

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.

This paper addresses two basic questions related to the space of operator trees of interest: What is the exact size of the space? And, how to generate a random element from the space efficiently? 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].

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

* C. Galindo-Legaria was supported by an ERCIM postdoctoral fellowship.

expressions on an associative operator, etc. A variety of techniques are used to enumerate graphs and trees [Knu68, HP73, RH77, GLW82, VF90], but none of them seems to apply directly to our problem.

Previous work has identified restricted classes of queries for which valid operator trees map one-to-one to permutations or to unlabeled binary trees —the first class known as *star* queries, and the second as *chain* queries, see for example [OL90, IK91, LVZ93]— thus solving the counting and random generation problems for those classes. For the general case, since it is easy to generate any valid operator tree non-deterministically, *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 labeled binary trees renders this method impractical [Swa89a, Swa91].

The paper is organized as follows. The remainder of this introduction defines the space of valid operator trees, and presents some notation and basic properties. Section 2 presents primitives for the construction of operator trees and shows how to efficiently count the number of trees for a given query. Section 3 is devoted to ranking of trees and random generation. Section 4 presents our conclusions.

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.

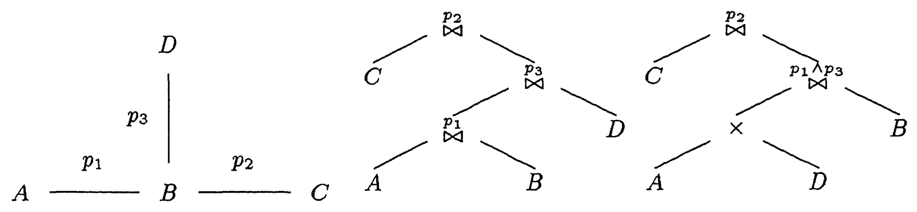

Fig. 1. Query graph and operator trees.

In the query graph, nodes correspond to relations of the database, and undirected edges correspond to join predicates of the query. The graph shown denotes the query $\{(a,b,c,d) \,|\, a \in A \wedge b \in B \wedge c \in C \wedge d \in D \wedge p_1(a,b) \wedge p_2(b,c) \wedge p_3(b,d)\}$, 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 usually evaluated by means of binary operators, and the two operator trees of Fig. 1 can be used to answer this query. The first operator tree requires only relational joins (denoted "$\bowtie$"), while the second requires Cartesian products (denoted "$\times$"). For a description of relational operators and query graphs, see, for example, [Ull82, CP85, KRB85]. A Cartesian product is required in the second tree of Figure 1 because we start by combining information from relations A, D, but there is no edge (i. e. predicate) between them in the query graph.

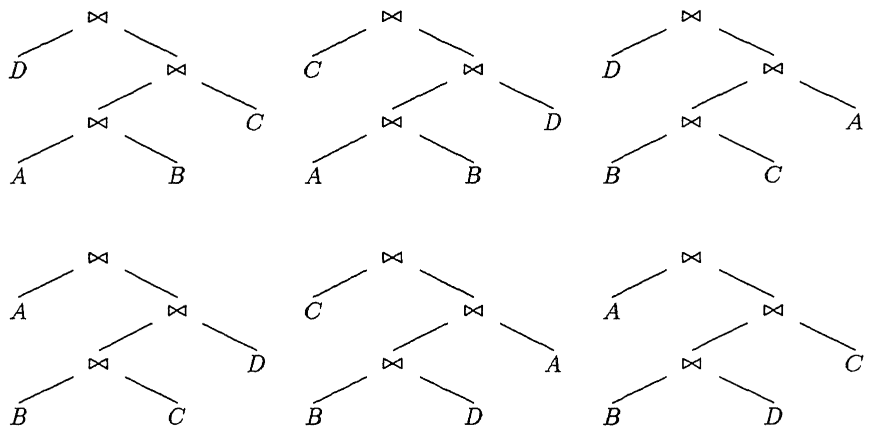

Fig. 2. All join trees of the query graph.

Figure 2 shows all 6 operator trees for the query of Fig. 1 in which only join is required, called *join trees* here. A purely graph-theoretical definition of join trees is given next.

Definition 1. An unordered binary tree T is called a *join tree* of query graph $G = (V, E)$ when it satisfies the following:

- The leaves of T correspond one-to-one with the nodes of G —i. e. leaves$(T) = V$— and
- the leaves of every subtree T' of T induce a connected subgraph of G — i. e. $G|_{\text{leaves}(T')}$ is connected.

Join trees are unordered —i. e. do not distinguish left from right subtree—

because the operator is commutative. There are implementation algorithms that do not distinguish a left and right argument [Gra93], so the selection of left/right argument does not affect the query execution cost. If needed, ordering an operator 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 $(T_1.T_2)$. Also, we assume that query graphs are connected and acyclic, i. e. we deal with *acyclic queries.*

1.3 Notation and Basic Properties

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 the set of join trees in which a given leaf v is at level k (the level of a leaf v in a tree is the length of the path from the root to v). For example, for the query graph of Fig. 1, Fig. 2 shows that $\mathcal{T}_G$ consists of six trees, $\mathcal{T}_G^{D(1)}$ consists of only two trees, and $\mathcal{T}_G^{B(3)} = \mathcal{T}_G$.

Since our constructions often rely on paths from the root of the join tree to a specific leaf, we introduce an *anchored list* representation of trees. Elements of the anchored list are the subtrees found while traversing the path from the root to some anchor leaf. For list notation, we use square brackets as delimiters and the list construction symbol "|" of Prolog —i. e. $[x|L]$ is the list obtained by inserting a new element x at the front of list L.

Definition 2. 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 $L = []$.
- Otherwise, 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 $T = ([T_1, T_2, \ldots, T_k], v)$ is an element of $\mathcal{T}_G$, then $T \in \mathcal{T}_G^{v(k)}$; that is, the length of anchored list coincides with the level of leaf v in T. In addition, every tree T_i, as well as every suffix-based $T_i' = ([T_i, \ldots, T_k], v)$, for $i = 1, \ldots, k$, is a join tree of some subgraph of G.

The following straightforward observations serve as base cases for our tree counting scheme. Let $G = (V, E)$ be a query graph with n nodes, and let $v \in V$.

- 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, $|\mathcal{T}_G| = \left|\mathcal{T}_G^{v(0)}\right| = 1$, 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, $\left|\mathcal{T}_G^{v(0)}\right| = 0$, for $n > 1$.
- There is no association tree in which v is at level greater than or equal to n; that is, $\left|\mathcal{T}_G^{v(i)}\right| = 0$, for $i \geq n$.

- Since v appears at some unique level in any association tree of G, the total number of association trees is

$$|\mathcal{T}_G| = \sum_i \left|\mathcal{T}_G^{v(i)}\right| \ .$$

Our algorithms compute the size of each subset $\mathcal{T}_G^{v(0)}, \mathcal{T}_G^{v(1)}, \ldots, \mathcal{T}_G^{v(n-1)}$ of $\mathcal{T}_G$, for a graph G of n nodes. Therefore, we use a *v-level-partitioned cardinality* $|\mathcal{T}_G|_v = \left[\left|\mathcal{T}_G^{v(0)}\right|, \left|\mathcal{T}_G^{v(1)}\right|, \ldots, \left|\mathcal{T}_G^{v(n-1)}\right|\right]$. Clearly, $|\mathcal{T}_G|$ can be computed (in linear time) given $|\mathcal{T}_G|_v$.

2 Construction and Counting of Join Trees

Our approach to counting join trees is based on two primitive operations that construct join trees of a graph G, given join trees of subgraphs of G. Those operations derive recurrence equations on the number of join trees of a query graph. Together with the base cases presented in Section 1.3, these recurrence equations are used to solve our tree counting problem.

2.1 Graph Extension / Leaf Insertion

Our first operation applies when a query graph G' is extended by adding a new node v and edge (v, w) to yield G. Then any join tree T' of G' can also be extended to a join tree T of G, by inserting a new leaf v somewhere in T'. But by the restrictions on join trees, the new leaf v can be inserted only in certain places.

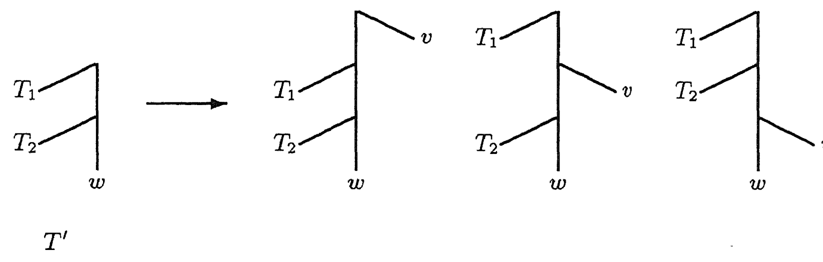

Fig. 3. Construction by leaf-insertion.

Figure 3 illustrates the situation. v has to be inserted somewhere *in the path from the root to w*. Inserting v somewhere else does not produce a valid join tree for G. For example, if in Fig. 3 v is inserted somewhere in T_1 to yield T_1' then $G|_{\text{leaves}(T_1')}$ is not connected, because it excludes node w and therefore edge (v, w), yet it includes v. For this reason, the valid tree T obtained from T' is uniquely determined by the level at which v is inserted. Note, however, that if two edges (v, w_1), (v, w_2) are added instead of just one, then v can be inserted either in the path to w_1 or the path to w_2, and therefore the new tree is not uniquely defined by the insertion level.

Definition 3. Let $G = (V, E)$ be an acyclic query graph. Assume $v \in V$ is such that $G' = G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$. We say G is the *extension by v adjacent to w* of G'.

Definition 4. Let G be the extension by v adjacent to w of G'. Let $T \in \mathcal{T}_G$, $T' \in \mathcal{T}_{G'}$, with anchored list representations $T = (L, w)$, $T' = (L', w)$. If L is the result of inserting v at position k in L', then T is constructed by leaf insertion from T', and we say T has *insertion pair* (T', k) *on* v.

The level at which a leaf can be inserted is also clearly restricted. For the same graphs G, G' and join trees T, $T' = (L', w)$ of the above definition, the length of L' is at least $k - 1$ —so that the insertion of a new element v in position k is feasible.

Lemma 5. *Let G be the extension by v adjacent to w of G'. Let $k \geq 1$. There is a bijection between the set $\mathcal{T}_G^{v(k)}$ and insertion pairs $\{(T', k) \mid T' \in \bigcup_{i \geq k-1} \mathcal{T}_{G'}^{w(i)}\}$.*

Proof. The bijection is given directly by the leaf-insertion operation. □

Lemma 6. *Let G be the extension by v adjacent to w of G'. Let $k \geq 1$. Then,*

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

Proof. Follows from Lemma 5, given that $\mathcal{T}_{G'}^{w(i)}$, $\mathcal{T}_{G'}^{w(j)}$ are disjoint for $i \neq j$. □

Example 1. Let G be a query graph with nodes $\{a, b, c, d, e\}$ and edges $\{(a, b), (b, c), (c, d), (d, e)\}$. From the base cases in Sect. 1.3, $\left|\mathcal{T}_{G|_{\{a\}}}\right|_a = [1]$. Then, using Lemma 6 we find $\left|\mathcal{T}_{G|_{\{ab\}}}\right|_b = [0, 1]$; $\left|\mathcal{T}_{G|_{\{abc\}}}\right|_c = [0, 1, 1]$; $\left|\mathcal{T}_{G|_{\{abcd\}}}\right|_d = [0, 2, 2, 1]$; and $|\mathcal{T}_G|_e = [0, 5, 5, 3, 1]$.

The computation in the above example is isomorphic to the one used to count unlabeled binary trees in [RH77]. This is the case for *chain queries* —i. e. those with nodes $\{v_1, \ldots, v_n\}$ and edges $\{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$. Then, as shown in [RH77], the closed form for $|\mathcal{T}_G|$ is $1/n \cdot \binom{2n-2}{n-1}$, for a chain query of n nodes. Unfortunately, Lemma 6 is, by itself, insufficient to deal with non-chain queries, as shown in the next example.

Example 2. Take graph G of Example 1, and add a new node f and edge (c, f) to obtain a new graph H. To find $|\mathcal{T}_H|_f$ using Lemma 6 we need $|\mathcal{T}_G|_c$, but we obtained only $|\mathcal{T}_G|_e$ in Example 1. Independently of the order in which we consider the nodes of H, we face the same problem: After a sequence of extensions on a graph in a "chain" fashion, we need to come back to an earlier node to extend from there, but then the necessary counters are not available.

2.2 Graph Union / Tree Merging

A second operation helps to remove the limitation shown in Example 2. The case to consider now is when a query graph G results from the union of two graphs G_1, G_2 that share exactly one common node, say v. Then any two join trees $T_1 \in \mathcal{T}_{G_1}$ and $T_2 \in \mathcal{T}_{G_2}$ can be merged to obtain a join tree $T \in \mathcal{T}_G$. T is obtained by interleaving the subtrees of T_1, T_2 found in the path from the root to the common leaf v.

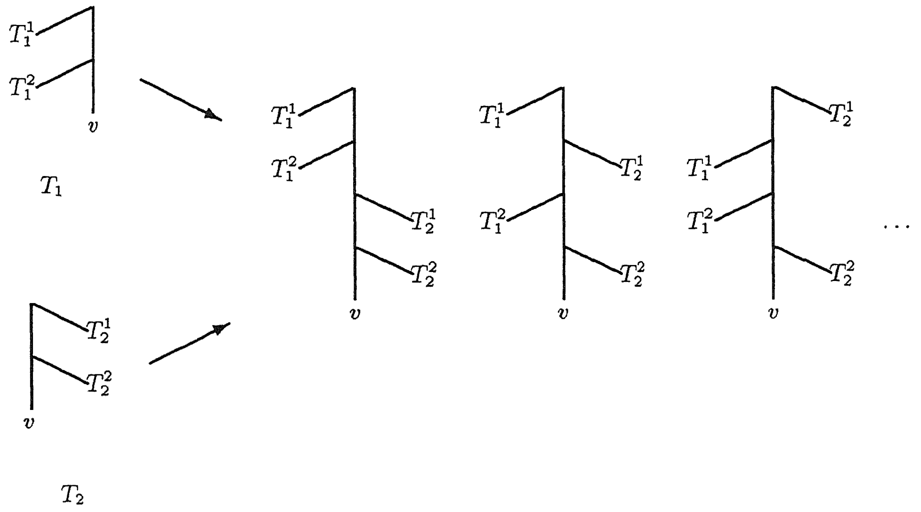

Fig. 4. Construction by tree-merging.

Figure 4 illustrates the situation. We can restrict our attention to the paths from the root to the common node v, in trees T_1, T_2 and the resulting tree T. The path to v in T contains the subtrees found in the paths in T_1, T_2, interleaved in some fashion. In terms of anchored lists, if $T_1 = (L_1, v)$ and $T_2 = (L_2, v)$, then $T = (L, v)$, where list L is a merge of lists L_1, L_2. The merging of two lists L_1, L_2 with respective lengths l_1, l_2 corresponds to 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 decomposition $[\alpha_0, \ldots, \alpha_{l_2}]$ indicates a merge of L_1, L_2 as follows: Take the first α_0 elements from L_1, then the first element from L_2; now take the next α_1 elements from L_1 and then the second element from L_2, and so on; the last α_{l_2} elements of L_1 follow the last element of L_2. In Fig. 4, for example, the trees shown are obtained by mergings $[2,0,0]$, $[1,1,0]$, and $[0,2,0]$, respectively. Note, however, that if G_1, G_2 share more than one node, then their corresponding trees can be merged in more elaborate ways.

Definition 7. Let $G = (V, E)$ be an acyclic query graph. 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 $V_1 \cap V_2 = \{v\}$. We say G is the *union of G_1, G_2 with common node v.*

Definition 8. Let G be the union of G_1, G_2 with common node v. Let $T \in \mathcal{T}_G$, $T_1 \in \mathcal{T}_{G_1}$, $T_2 \in \mathcal{T}_{G_2}$, with anchored list representations $T = (L, v)$, $T_1 = (L_1, v)$, $T_2 = (L_2, v)$. If L is the result of a merging α of lists L_1, L_2, then T is constructed by tree merging from T_1, T_2, and we say T has *merge triplet* (T_1, T_2, α) *on* V_1, V_2.

Lemma 9. *Let G be the union of G_1, G_2 with common node v. Let $k \geq 1$. There is a bijection between the set $\mathcal{T}_G^{v(k)}$ and merge triplets $\{(T_1, T_2, \alpha) \mid T_1 \in \mathcal{T}_{G_1}^{v(i)}, T_2 \in \mathcal{T}_{G_2}^{v(k-i)}, \alpha$ is an integer decomposition of i in $k - i + 1\}$.*

Proof. The bijection is given by the tree merging operation. □

Lemma 10. *Let G be the union of G_1, G_2 with common node v. Let $k \geq 1$. Then*

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

Proof. Follows from Lemma 9. □

2.3 Counting Join Trees

Our tree-construction operations, and their corresponding count equations, can be applied on query graphs built using graph extension and graph union. We make this construction explicit by means of a *standard decomposition graph.* Algorithms to count and construct trees are implemented by traversals on this decomposition graph.

Definition 11. A *standard decomposition graph* is an operator tree H that builds a query graph G, using the following:

- Constant "v" delivers a graph G with one node v; v is the *distinguished node* of G.
- Unary "$+_v$" takes as input a graph $G' = (V', E')$ with distinguished node w, $v \notin V'$, and delivers a graph G that is the extension on v adjacent to w of G'. The distinguished node of G is v.
- Binary "$\times_v$" takes as input two graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ both with distinguished node v, $V_1 \cap V_2 = \{v\}$, and delivers a graph G that is the union of G_1, G_2. The distinguished node of G is v.

For example, Fig. 5 shows a query graph G and a standard decomposition graph H for G. It is easy to see that a linear time algorithm obtains standard decomposition graphs for acyclic query graphs. The number of nodes of the standard decomposition is linear in the number of nodes of the query graph it builds. Alternatively, operators in the decomposition graph of G can be interpreted as

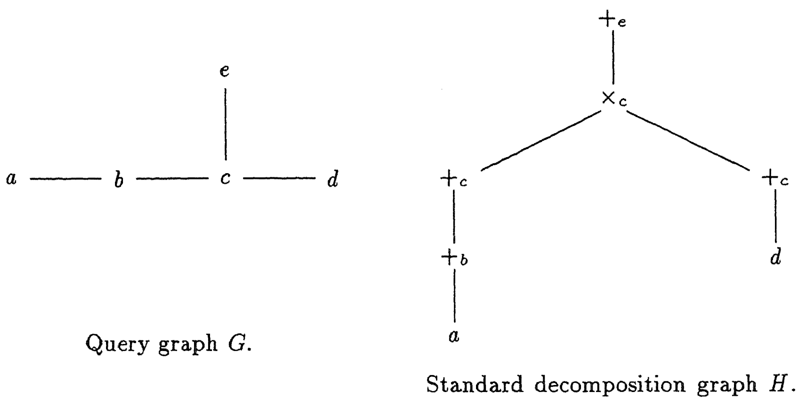

Fig. 5. Query graph and standard decomposition graph.

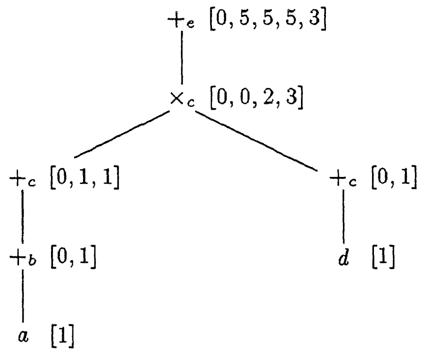

Fig. 6. Counting trees on the standard decomposition graph.

building join trees of G. Adding extra parameters "$+_{v,k}$" and "$\times_{v,\alpha}$" become operators for leaf insertion at k and tree merging on α.

Now, to count the number of join trees of a given acyclic query graph G, we first obtain a standard decomposition H of G, and then apply directly Lemmas 6 and 10 bottom-up on H, to compute a result for each node. In each subexpression H' of H, if H' constructs a graph G' (subgraph of G) with distinguished node v, then at the root of H' we compute $|\mathcal{T}_{G'}|_v$. For example, Fig. 6 shows the results of the computation for the query graph of Fig. 5. Note that this query is neither a chain nor a star. The total number of different trees is 18.

Theorem 12. *Let G be a connected, acyclic query graph on n relations. The number of join trees for G can be computed in $O(n^3)$ time.*

Proof. A standard decomposition graph H of G can be constructed in linear time, and the number of nodes of H is linear on n. Then we apply either Lemma 6 or Lemma 10 in each of the $O(n)$ nodes of H. Now, observe that the number of mergings $M(l_1, l_2)$, for lists of size l_1, l_2, satisfies $M(l_1, l_2) = M(l_1, l_2 - 1) + M(l_1 - 1, l_2)$; then a table for $M(l_1, l_2)$, where $l_1, l_2 \leq n$, can be precomputed in $O(n^2)$ time, and evaluating the expression $\binom{k}{i}$ required in Lemma 10 reduces to a simple table lookup. Computing the list of values $|\mathcal{T}_{G'}|_{v'}$ at each node of H by either Lemma 6 or 10 does not exceed $O(n^2)$ time. Then, the computation in all the nodes of H requires no more than $O(n^3)$ time. Finally, $|\mathcal{T}_G|$ is the result of adding the values in the list $|\mathcal{T}_G|_v$ obtained at the root of H. □

3 Random Generation and Ranking of Join Trees

The bijections given by Lemmas 5 and 9 and the standard decomposition graph can also be used to rank and to generate random join trees. We show how to do this next.

3.1 Generating Random Join Trees

To generate random join trees we can apply the following strategy recursively. Assume a set S is partitioned into sets $S_0, \ldots, S_m$. To generate a random element of S, first select a partition, say S_i, and then generate a random element within S_i. Using a biased probability of selection $|S_i|/|S|$ for each partition, and then generating uniformly from the partition, every element $x \in S_i$ is generated with probability $|S_i|/|S| \cdot 1/|S_i| = 1/|S|$ —i. e. the procedure generates elements from S uniformly at random.

Lemma 13. *Let G be a query graph with n nodes, obtained as the extension by v adjacent to w of G'. Let $1 \leq k_1 \leq k_2 < n$. A random, uniformly distributed T from $\bigcup_{k_1 \leq j \leq k_2} \mathcal{T}_G^{v(j)}$ can be obtained as follows:*

1. *Generate a random number r uniformly with $1 \leq r \leq \sum_{k_1 \leq j \leq k_2} \left|\mathcal{T}_G^{v(j)}\right|$.*
2. *Find $k = \min_j \left(r \leq \sum_{l=0}^{j} \left|\mathcal{T}_G^{v(l)}\right|\right)$.*
3. *Generate a random, uniformly distributed T' from $\bigcup_{i \geq k-1} \mathcal{T}_{G'}^{w(i)}$.*
4. *Build T from insertion pair (T', k).*

Proof. A set in the partition $\mathcal{T}_G^{v(k_1)}, \ldots, \mathcal{T}_G^{v(k_2)}$ is seleted with the appropriate bias in steps 1 and 2, and then an element of the selected partition is generated uniformly in steps 3 and 4 using the bijection of Lemma 5. □

Lemma 14. *Let G be a query graph with n nodes, obtained as the union of G_1, G_2 with common node v. Let $1 \leq k_1 \leq k_2 < n$. A random, uniformly distributed T from $\bigcup_{k_1 \leq j \leq k_2} \mathcal{T}_G^{v(j)}$ can be obtained as follows:*

1. *Generate a random number r uniformly with* $1 \leq r \leq \sum_{k_1 \leq j \leq k_2} \left|\mathcal{T}_G^{v(j)}\right|$
2. *Find* $k = \min_j \left(r \leq \sum_{l=0}^{j} \left|\mathcal{T}_G^{v(l)}\right|\right)$.
3. *Generate a random number r' uniformly with* $1 \leq r' \leq \left|\mathcal{T}_G^{v(k)}\right|$.
4. *Find* $i = \min_j \left(r' \leq \sum_{l=0}^{j} \left|\mathcal{T}_{G_1}^{v(l)}\right| \cdot \left|\mathcal{T}_{G_2}^{v(k-l)}\right| \cdot \binom{k}{l}\right)$.
5. *Generate random, uniformly distributed T_1 from $\mathcal{T}_{G_1}^{v(i)}$, T_2 from $\mathcal{T}_{G_2}^{v(k-i)}$, and integer partition of i in $k-i+1$ α.*
6. *Build T from merge triplet (T_1, T_2, α).*

Proof. A set in the partition $\mathcal{T}_G^{v(k_1)}, \ldots, \mathcal{T}_G^{v(k_2)}$ is selected with the appropriate bias in steps 1 and 2. The selected set is again partitioned and one of those partitions is in turn selected with the appropriate bias in steps 3 and 4. Finally, an element of the resulting set is selected uniformly in steps 5 and 6. The partition in step 3 and 4, and the uniform selection in steps 5 and 6 use the bijection of Lemma 9. □

Theorem 15. *Let G be a connected, acyclic query graph on n relations. After a preprocessing step of $O(n^3)$ time, uniformly-distributed random join trees for G can be generated in $O(n^2)$ time, given a source of random numbers.*

Proof. By Theorem 12, the standard decomposition graph H of G and the count arrays in the nodes of H can be computed in $O(n^3)$ time. This completes the preprocessing step. To generate a random tree, traverse the decomposition graph recursively from the top, applying the procedure of either Lemma 13 or 14 at each node (except for the "constant" nodes of H, which define a one-node query graph with only one join tree, in which case random selection is trivial). The time taken by either procedure at each node of H is bound by $O(n)$. Therefore, the total time required generate a random tree is $O(n^2)$.

The above scheme generates $O(n)$ random numbers to produce a random join tree. An alternative to obtain a random tree is to generate a single random number, and then unrank such number into a tree, as shown below. The complexity of unranking, however, is higher.

3.2 Ranking and Unranking Join Trees

Mapping the N join trees of a query graph to 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 $0 \leq k \leq m$, and we can find a *local rank* of x in S_k, then simply set the rank of x in S to be local-rank$(x, S_k) + \sum_{i=0}^{k-1} |S_i|$. Conversely, to unrank the element corresponding to number y under our scheme, first find the set S_k from which the element must be retrieved, where $k = \min_j \left(y \leq \sum_{i=0}^{j} |S_i|\right)$. Then find a local number $y' = y - \sum_{i=0}^{k-1} |S_i|$, and finally unrank-local(y', S_k).

Example 3. Applying the idea of ranking based on partitions to the query whose standard decomposition is shown in Fig. 6, 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.

Lemma 16. *Let G be a query graph with n nodes, obtained as the extension by v adjacent to w of G'. Let $1 \leq k_1 \leq k_2 < n$. The rank r of T in $\bigcup_{k_1 \leq j \leq k_2} \mathcal{T}_G^{v(j)}$ can be obtained as follows:*

1. *Find the insertion pair (T', k) of T, where $T' \in \mathcal{T}_{G'}$, $k_1 \leq k \leq k_2$.*
2. *Find the rank r' of T' in $\bigcup_{i \geq k-1} \mathcal{T}_{G'}^{w(i)}$.*
3. *The rank of T is $r = r' + \sum_{i=0}^{k-1} \left|\mathcal{T}_G^{v(i)}\right|$.*

Proof. In step 1, the partition where T belongs is identified by looking at the level of leaf v. In step 2, a local rank of T in its partition is obtained using the bijection of Lemma 5 (assuming some ranking function for the trees in G'). Finally, step 3 adjusts the local rank to the rank in the complete set. □

Lemma 17. *Let G be a query graph with n nodes, obtained as the union of G_1, G_2 with common node v. Let $1 \leq k_1 \leq k_2 < n$. The rank r of T in $\bigcup_{k_1 \leq j \leq k_2} \mathcal{T}_G^{v(j)}$ can be obtained as follows:*

1. *Find the merge triplet (T_1, T_2, α) of T, where $T_1 \in \mathcal{T}_{G_1}^{v(i)}$, $T_2 \in \mathcal{T}_{G_2}^{v(k-i)}$, and α is an integer decomposition of i in $k-i+1$.*
2. *Find the rank r_1 of T_1 in $\mathcal{T}_{G_1}^{v(i)}$, the rank r_2 of T_2 in $\mathcal{T}_{G_2}^{v(k-i)}$, and the rank r_3 of α. Set the local rank $r' = (r_3 - 1) \cdot \left|\mathcal{T}_{G_2}^{v(k-i)}\right| \cdot \left|\mathcal{T}_{G_k}^{v(i)}\right| + (r_2 - 1) \cdot \left|\mathcal{T}_{G_k}^{v(i)}\right| + r_1$.*
3. *The rank of T is $r = r' + \sum_{l=0}^{i-1} \left|\mathcal{T}_{G_1}^{v(l)}\right| \cdot \left|\mathcal{T}_{G_2}^{v(k-l)}\right| \cdot \binom{k}{l} + \sum_{l=0}^{k-1} \left|\mathcal{T}_G^{v(l)}\right|$.*

Proof. In step 1, the partition where T belongs is identified from the merge triplet. In step 2, a local rank is obtained, using the bijection of Lemma 9 (assuming some ranking function for trees in G_1 and G_2, and on integer decompositions). Finally, step 3 adjusts the local rank in the complete set. □

Theorem 18. *Let G be a connected, acyclic query graph on n relations. After a preprocessing step of $O(n^3)$ time, join trees of G can be ranked in $O(n^2)$ time and unranked in $O(n^2 \log n)$ time.*

Proof. By Theorem 12, the standard decomposition graph H of G and the count arrays in the nodes of H can be computed in $O(n^3)$ time. This completes the preprocessing step. To rank a tree, at each node of H we have to decompose the tree either into an insertion pair or into a merge triplet, and then use the rules of Lemma 16 or 17 (except for "constant" nodes of H, in which the ranking is trivial). The work per node does not exceed $O(n)$ time, and therefore the work to compute the local ranks for all the nodes takes time $O(n^2)$.

To unrank a tree, we first translate the rank in $\mathcal{T}_G$ to a local rank in some $\mathcal{T}_G^{v(k)}$. Then, using the rules of Lemmas 16 and 17, we find at each node of H an insertion level k or a merging specification α in a top-down pass of H (except for "constant" nodes of H, in which unranking is trivial). In a bottom-up pass we use the parameters at each node to build the tree. Unranking a merging specification α takes $O(n \log n)$ in the worst case, which bounds all the other computation performed at each node of H. Thus the time required to unrank is $O(n^2 \log n)$. □

4 Discussion

In this paper we described how to efficiently count the number of join trees that can be used to evaluate a given query, and how to generate them uniformly at random. The difficulty of these problems results from the fact that there is no natural 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. The tree constructions are such that they can be counted efficiently by means of a simple traversal of the decomposition graph.

The integers required by our algorithms can become quite large, as is the case with other graph counting/generation problems [vL90]. This eventually limits the applicability of our approach. Nevertheless, our algorithms can be used to a good extent on practical database queries (e. g. certainly for queries of 20 relations, using standard 64-bit integers).

We performed experiments on the selection of join evaluation orders, for query optimization, using the random generation of trees presented here. The results are encouraging. The interested reader is referred to [GLPK94] for more details.

The results we have presented apply only to queries whose graph is *acyclic*. 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.

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.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[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.

[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, chapter 10, pages 525–631. North Holland, 1990.