Optimal Top-Down Join Enumeration (extended version)

David E. DeHaan
Frank Wm. Tompa

Technical Report CS-2007-02
March 15, 2007

Optimal Top-Down Join Enumeration (extended version)

David DeHaan and Frank Wm. Tompa
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{dedehaan, fwtompa}@uwaterloo.ca

ABSTRACT

Most contemporary database systems perform cost-based join enumeration using some variant of System-R's bottom-up dynamic programming method. The notable exceptions are systems based on the top-down transformational search of Volcano/Cascades. As recent work has demonstrated, bottom-up dynamic programming can attain optimality with respect to the shape of the join graph; no comparable results have been published for transformational search. However, transformational systems leverage benefits of top-down search not available to bottom-up methods.

In this paper we describe a top-down join enumeration algorithm that is optimal with respect to the join graph. We present performance results demonstrating that a combination of optimal enumeration with search strategies such as branch-and-bound yields an algorithm significantly faster than those previously described in the literature. Although our algorithm enumerates the search space top-down, it does not rely on transformations and thus retains much of the architecture of traditional dynamic programming. As such, this work provides a migration path for existing bottom-up optimizers to exploit top-down search without drastically changing to the transformational paradigm.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems—*query processing, relational databases*; F.2.0 [Analysis of Algorithms and Problem Complexity]: General

General Terms: Algorithms, Design, Experimentation, Performance

Keywords: join enumeration, query optimization, top-down, dynamic programming, memoization, branch-and-bound

1. INTRODUCTION

Cost-based query optimization is central to the performance of database management systems. Traditionally, either logical optimization is completed prior to physical optimization [17], or the logical optimizer invokes the physical optimizer as needed to optimize isolated select-project-join query blocks within a more complex query [1]. With this stratified approach, the enumeration strategy of the logical optimizer (heuristic vs. cost-based, bottom-up vs. top-down) may differ completely from that of the physical optimizer. Whereas top-down application of transformations is a natural paradigm for logical optimization, physical optimization is still widely performed using variations of the bottom-up dynamic programming method pioneered by Selinger et al. in System-R [19]. In contrast, the Volcano [8]/Cascades [6] model seamlessly integrates both logical and physical steps into a single top-down application of transformations; however, this introduces a significant paradigm shift for physical optimization.

Top-down search enables several enhancements that are not compatible with bottom-up dynamic programming:

1. demand-driven interesting orders
2. branch-and-bound pruning
3. exploiting partial information

In this paper we emphasize performance benefits resulting from branch-and-bound, although interesting orders and exploiting partial information will be discussed briefly in Sections 3 and 5, respectively. These benefits have been pointed out before within other top-down systems, but our paper is novel in showing that they can be attained within an optimizer that retains many characteristics of the traditional bottom-up dynamic programming paradigm. In particular, ours is the first paper to demonstrate that top-down search can be wedded with optimal enumeration of the space of join plans that avoid cartesian products—a space with ongoing importance for large queries because of the $\Omega(3^n)$ cost of considering cartesian products [14, 13].

We focus narrowly on the traditional problem of enumerating join orders for a single select-project-join query block. The computational complexity of this problem was detailed by Ono and Lohman [14] for a variety of join graph shapes. However, recent analysis by Moerkotte and Neumann [11] shows that traditional bottom-up join enumeration algorithms do not achieve Ono and Lohman's lower bounds. Moerkotte and Neumann go on to describe a new bottom-up enumeration algorithm that is optimal over any join graph.

Join enumeration has received significant attention in the literature, and so we begin with a taxonomy of existing algorithmic approaches (Section 2). Following this, our paper makes several contributions. We present the first join enumeration algorithm that both achieves the computational lower bound for any query graph and—unlike Moerkotte and Neumann's algorithm—enumerates the search space top-down

University of Waterloo Technical Report CS-2007-02.
This is an extended version of work published in
SIGMOD'07, June 11-14, 2007, Beijing, China.
That work is copyright ACM, 2007, and the definitive version
is available from the ACM Digital Library.

instead of bottom-up (Section 3). Second, we demonstrate empirically that combining optimal enumeration with branch-and-bound pruning made possible by the top-down search yields algorithms that are significantly faster than previous approaches for search spaces that avoid cartesian products (Section 4). Third, we point out a tension between dynamic programming and the style of branch-and-bound implemented in several existing systems, opening the possibility that the branch-and-bound pruning in those systems does not behave as expected (also Section 4). Finally, we describe why ours is the first dynamic programming/memoization algorithm to provide a flexible trade-off between CPU time and memory usage, and we detail how the top-down search can be used to exploit partial information that may be available to an optimizer (Section 5).

2. EXISTING APPROACHES

This section gives an overview of existing join enumeration algorithms, classified according to search strategy. An algorithm is called *bottom-up* if it considers small logical expressions before large ones; otherwise it is *top-down*. We restrict our focus to search strategies that can guarantee optimality of the result. We assume that the input to each algorithm is a connected join graph $G = (V, E)$ where V represents the set of relations to be joined, and each edge in E represents a join predicate.

2.1 Compositional Dynamic Programming

Compositional dynamic programming is a bottom-up technique that represents logical expressions as sets of relations, and then enumerates over choices for already optimized sets V_1 and V_2 to obtain a new plan for $V = V_1 \cup V_2$. System-R [19] used this approach for left-deep plans without cartesian products (CPs hereafter), relying solely upon expression size to dictate the enumeration order of V_1 and V_2; the generalization of this strategy to bushy plans is given by Moerkotte and Neumann [11].

Although widely used, size-driven enumeration is far from optimal because of attempted compositions of overlapping sets [22]. For CP-free search spaces the inefficiency is even worse because of the generate-and-test approach to avoiding CPs [11]. Moerkotte and Neumann propose another compositional dynamic programming algorithm for the space of bushy plans without CPs that achieves Ono and Lohman's lower bounds on the number of join operators considered. Their algorithm drives the enumeration of V_1 and V_2 by using the graph structure to incrementally grow "Connected-subgraph Complement Pairs".

2.2 Partitioning Dynamic Programming

Partitioning dynamic programming drives the enumeration by the choice of V, which is then partitioned into all choices for V_1 and V_2 such that $V = V_1 \cup V_2$. This is still a bottom-up framework, however, so the enumeration order of V must be carefully designed so that for any valid partition of V the sub-expressions have already been optimized.

This method was invented by Vance and Maier [22], who proposed searching the space of bushy plans with CPs using an efficient algorithm for generating subsets to drive the ordering and partitioning of V. Moerkotte and Neumann present a modification of this algorithm for the space of bushy plans without CPs, but the algorithm is inefficient for this space because the subset generation is naive—that is, oblivious to the query graph—and hence generates large numbers of CPs which all are discarded.

2.3 Prefix Search

Prefix search is a bottom-up algorithm, but uses backtracking search instead of dynamic programming. Left-deep join trees are abstracted as sequences of relations joined left to right, and the plan space is enumerated by extending prefixes incrementally. Bushy join trees can be abstracted as sequences containing nested parenthesis; however, the search space becomes too large for this approach to be practical. Unlike other algorithms discussed in this section, prefix search does not use divide-and-conquer and so is incompatible with dynamic programming/memoization. It is used in the Sybase SQL Anywhere optimizer [3] which targets small-footprint environments that cannot afford the memory cost of dynamic programming. The lack of dynamic programming leads to a search space containing $\Theta(n!)$ left-deep join trees. To address this, the SQL Anywhere optimizer relies heavily on very aggressive branch-and-bound pruning that sacrifices optimality for the sake of efficiency.

2.4 Transformational Search

Transformational search represents logical expressions as trees of *ordered* binary joins. After converting a query to an arbitrary logical (ordered) join tree, the space of possible join plans is searched by top-down application of logical-to-logical (join ordering) and logical-to-physical (join method selection) transformations. Memoization—the top-down variant of dynamic programming—avoids redundant work.

Transformational search is used in the Volcano [8], Cascades [6], OPT++ [9], and Columbia [20] systems, as well as in Microsoft SQL Server [7]. EROC [10] is a hybrid system that uses (top-down) transformational search for join method selection and costing but (bottom-up) compositional dynamic programming for the enumeration of join orders.

Transformational search is amenable to top-down enhancements such as demand-driven interesting orders and branch-and-bound. Strengths such as extensibility of the optimizer, online modification to the search space, and early generation of complete physical plans are additional features specifically due to the transformational paradigm.

Unfortunately, this paradigm has two weaknesses compared to traditional bottom-up join enumeration. The first weakness is well documented in the literature: memory cost. Transformational search requires storing *all* generated plans, not just optimal plans. For search spaces including CPs, this results in $\Omega(n2^n)$ (left-deep) or $\Omega(3^n)$ (bushy) storage, compared to the $\Omega(2^n)$ (left-deep or bushy) storage required by bottom-up join enumeration [14].

The second weakness of the transformational approach is less well known, but relevant to the contribution of our paper: there is no published transformational method that optimally enumerates CP-free join plans (either left-deep or bushy) of arbitrary queries. Pellenkoft et al. [15, 16] published one of the few analyses of transformational search; they show that by using a mechanism governing the application of transformations to guarantee that every expression has a unique derivation path, transformational search can achieve the lower bounds derived by Ono and Lohman [14] for both left-deep and bushy plans containing CPs. They also describe mechanisms for *acyclic* queries that guarantee the unique derivation path of every CP-free plan contains only CP-free plans. When combined with a simple generate-and-test approach to discarding plans containing CPs, the number of discarded CPs is proportional to the number of join operators; hence, for acyclic queries the search is optimal with respect to number of operators enumerated. It is easy to show, however, that that there exist simple cyclic queries for which the derivation path of at least one bushy CP-free plan must pass through a plan containing a CP; thus the generate-and-test approach fails to enumerate the complete space of CP-free plans. It is an open problem whether transformational search can enumerate either left-deep or bushy CP-free plans optimally for arbitrary join graphs.

Because of the prominence of transformational systems relative to the remaining top-down algorithms discussed in this section, top-down transformational search has been referred to by the unfortunate name "top-down search" in some of the literature (e.g. [20]). This can lead to strengths and weaknesses that specifically result from the transformational nature of these particular systems being incorrectly attributed to all top-down search methods.

2.5 Partitioning Search

Partitioning search is a top-down algorithm similar to the bottom-up algorithm described in Section 2.2: a given V is partitioned into all choices for V_1 and V_2 such that $V = V_1 \cup V_2$. Instead of requiring a strict enumeration order for V that obeys the bottom-up constraints of dynamic programming, the enumeration order follows directly from recursion on V_1 and V_2 and is combined with memoization.

Chaudhuri et al. [4] present a top-down partitioning algorithm for left-deep trees with CPs. The purpose of their paper is to integrate join enumeration with cost-based rewritings over materialized views, and so they ignore the top-down nature of their algorithm, referring to it as a "simplification and abstraction" of the System-R algorithm. This characterization is not quite apt, because the algorithmic complexity is actually akin to bottom-up partitioning, not size-driven enumeration—an important distinction when the approach is extended to bushy plans [11]. The authors do not consider interesting orders, branch-and-bound, or avoiding cartesian products.

3. OPTIMAL TOP-DOWN PARTITIONING

In this section we study in more detail the top-down partitioning search strategy just introduced. After describing the basic algorithmic framework, we show that achieving optimality in the join enumeration requires a non-trivial partitioning algorithm that exploits the graph structure.

The main contribution of this section is to prove both analytically and empirically that the performance of top-down partitioning search with memoization is comparable to bottom-up search with dynamic programming. This point deserves emphasis, because there have been several previous comparisons drawn between bottom-up dynamic programming and top-down *transformational* search [10, 9, 20] that can lead to the misconception that bottom-up search strategies are inherently more efficient—especially for CP-free search spaces—and that top-down search algorithms need to rely on branch-and-bound or demand-driven interesting orders to be competitive. In contrast, our results show that there is no inherent performance penalty in changing from bottom-up to top-down search.

3.1 Algorithmic Framework

Algorithm 1 provides a skeleton framework for join enumeration via top-down partitioning search. This skeleton generalizes the algorithm given by Chaudhuri et al. [4] both by incorporating demand-driven interesting orders and by replacing the enumeration over singleton relations with an abstract PARTITION function on line 4 of CALCBESTJOIN. The search space of the optimizer—left-deep vs. bushy, with or without cartesian-products—is controlled simply by changing the implementation of PARTITION.

As mentioned, our skeleton incorporates demand-driven interesting orders which are desirable because they limit optimization to expressions that are actually useful [8]. However, because the approach is essentially the same as has been demonstrated in transformational systems and because it is somewhat orthogonal to the actual enumeration of join orders which is the focus of this paper, we will not consider interesting orders further in this paper. In particular, we intentionally did not implement interesting orders as part of our experimental apparatus so that our comparisons between comparable bottom-up and top-down search algorithms are testing the enumeration of the search space only, not the handling of interesting orders. This is a valid abstraction because although top-down search enables demand-driven interesting orders, it does not require them; one could integrate a top-down search for enumerating join orders with existing techniques for bottom-up handling of interesting orders within each enumerated join operator.

Algorithm 1 represents a join query as a graph $G = (V, E)$; therefore, its analysis depends upon how G is encoded. The traditional edge-list encoding is asymptotically optimal; however, we contend that for most join enumeration contexts it is valid to assume that the ratio of query size (typically much smaller than 100 relations) to machine word size (typically 32 or 64 bits) is bounded by a small constant. Under this assumption, the most efficient encoding for G is as an array of bitmaps, which is the encoding we used in our implementation. To account for this, our analysis in this section uses a "bitmap" model of computation that assumes that the set operations *containment*, *union*, *intersection*, and *difference* can be performed in constant time using bit-wise machine instructions. Although we will use asymptotic notation, the accuracy of our computational model (and hence our analysis) depends upon the assumption that $\frac{|V|}{|\text{machine word}|}$ is bounded; for the true asymptotic case one should use an edge-list encoding and charge set operations at higher than constant cost, which would introduce at most a linear factor.

Our model of computation has two notable implications. First, testing graph connectivity (time $\Theta(|E|)$ with an edge-list encoding) costs only time $\Theta(|V|)$ because using depth-first search, it takes constant time on each step to remove edges pointing to previously-visited nodes using set difference. Second, given a graph $G = (V, E)$ and some $V' \subset V$, the induced subgraph $G|_{V'}$ can be generated in constant time by lazily intersecting V' with each original bitmap on demand, adding constant overhead to each read. In contrast, generating an induced subgraph using an edge-list encoding eventually incurs $\Theta(|E|)$ cost whether the generation is eager or lazy.

Finally, we need to define what we mean by an "optimal" join enumeration algorithm. Ono and Lohman's lower bounds [14] quantify only the number of join operators that must be enumerated and are not necessarily tight in terms of time complexity. Following the lead of Moerkotte and Neumann [11], we will call an enumeration algorithm "optimal" if it incurs no more than linear time overhead between enumerated join operators[1]. Algorithm 1 invokes PARTITION at most once for each unique V (due to the memoization); for each returned partition, the loop at line 5 of CALCBESTJOIN considers a unique join operator in the search space (by enumerating a constant number of physical join methods). Hence, an algorithm constructed in this framework is optimal if the partitioning function requires no more than linear time between outputting successive partitions. For comparison, Moerkotte and Neumann's bottom-up algorithm for bushy join trees requires $O(|V|)$ per join operator under our bitmap model, or $O(|E|)$ per join operator using an edge-list encoding. Their algorithm is thus optimal for any join graph, and so in this section our goal is simply to create an algorithm that is competitive with theirs.

3.2 Naive Partitioning

Consider the graph partitioning algorithm shown in Algorithm 2; this algorithm is "naive" in the sense that it ignores the edges of the join graph. When substituted into Algorithm 1, the resulting search algorithm enumerates left-deep operator trees with cartesian products. Each invocation of Algorithm 2 outputs $|V|$ partitions and has a total cost of $\Theta(|V|)$; hence, the search algorithm is optimal for the space of left-deep operator trees with CPs.

Now consider avoiding cartesian products. The obvious approach is to add to Algorithm 2 a test that $G|_{(V\setminus\{v\})}$ remains connected. This increases the total work per invocation of PARTITION to $\Theta(|V|^2)$; however, the number of partitions output now depends upon the graph structure and could be as few as two (chains). In the worst-case, the partitioning algorithm uses quadratic time between outputting successive partitions, and so the search algorithm is a linear factor worse than optimal for left-deep CP-free join trees.

To enumerate bushy plans with CPs, we modify line 1 of Algorithm 2 to enumerate all non-empty strict subsets of V. The total work per invocation of PARTITION is $\Theta(2^{|V|})$, and the number of partitions output is $2^{|V|} - 2$; therefore, the search algorithm is optimal for bushy join trees with CPs.

The obvious extension for bushy CP-free plans is to insert two connectivity tests. The total partitioning cost per invocation increases to $\Theta(|V| * 2^{|V|})$, and the number of partitions output again depends upon the graph structure, ranging from $2^{|V|}-2$ (cliques) to $|V|-1$ (acyclic graphs). Hence, the search algorithm is optimal only for near cliques and can be exponentially sub-optimal in general.

In summary, naive partitioning strategies provide optimal top-down search algorithms for both left-deep and bushy spaces containing cartesian products, but provide sub-optimal search algorithms for CP-free spaces. This is not surprising because there is a one-to-one correspondence between each pair (V_1, V_2) considered by top-down naive partitioning a pair enumerated by the bottom-up naive partitioning from Section 2.2. Moerkotte and Neumann present a more precise analysis of the number of pairs enumerated by bottom-up naive partitioning.

3.3 Minimal Cuts

The generate-and-test approach for avoiding cartesian products used by the naive partitioning algorithms does not exploit information present in the join graph. In this subsection we present algorithms that utilize graph-theoretic properties to partition a graph in linear time per partition.

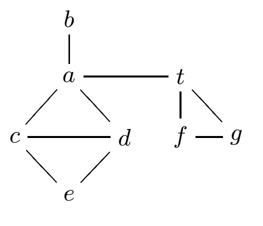
Consider the left-deep plan space. Graph $G|_{(V\setminus\{v\})}$ is disconnected precisely when v is an *articulation vertex* of G. Using the DFS algorithm of Aho et al. [2, p. 180–187] the set of articulation vertices can be identified (and hence avoided) in $\Theta(|E|)$ time, eliminating the need for a connectivity test. The resulting search algorithm is optimal for left-deep trees without cartesian products.

Now consider the bushy space. Whereas left-deep partitioning is naturally viewed as choosing individual vertices to delete from G without disconnecting it, this vertex-deletion perspective is a bit clumsy for bushy partitioning because it implies an undesired asymmetry between what is deleted and what remains. An equivalent edge-centric way to perceive a partition is as a graph *cut*—that is, a set of edges whose deletion divides G into two or more components. A cut is called *minimal* if it contains no other cut as a subset; a well-known corollary is that a minimal cut divides G into precisely two connected components. Hence, there is a duality between the set of all minimal cuts of G and the set of all partitions of V into (V_L, V_R) such that $G|_{V_L}$ and $G|_{V_R}$ are each connected. For the left-deep case considered above, each non-articulation vertex is the dual of a minimal cut in which one component is unary.

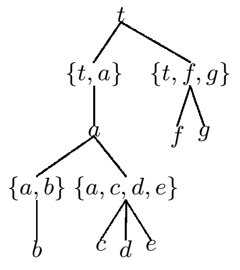
Minimal cuts are of long-standing interest to the networking community because of the obvious applications to network reliability [12]. Within that community, much attention has been given to the (s,t)-*cut* problem, formulated as follows: given a connected graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$, find all minimal subsets of E which, if removed from G, would disconnect s from t. One of the first linear-delay algorithms for the (s,t)-cut problem is described by Tsukiyama et al. [21]. Provan and Shier [18] present a recursive paradigm and show how it can be used to solve several variations of the (s,t)-cut problem. One such variation—K-*connectivity cutsets*—finds all minimal cuts that partition an arbitrary $K \subseteq V$. For our purposes, we are interested in the case when $K = V$, which corresponds to finding all minimal cuts of G.

Provan and Shier's algorithm depends upon a data structure called a *biconnection tree*. The biconnection tree $\mathbb{T}$ for a given connected graph $G = (V, E)$ and vertex $t \in V$ contains two types of nodes: *vertex nodes* correspond one-to-one with the vertices in V, and *set nodes* correspond one-to-one with the *biconnected components* of G [2]. For every vertex $v \in V$ that participates in a biconnected component β, there is an edge in $\mathbb{T}$ between the corresponding vertex and set nodes. Finally, the vertex node corresponding to t is designated as the root of $\mathbb{T}$. A biconnection tree can be built in time $\Theta(|E|)$ the procedure BUILDBCCTREE shown in Algorithm 3, which is a straightforward modification of Aho et al.'s DFS algorithm for identifying biconnected components of a graph. The reader is referred to Aho et al.'s text [2] for an explanation of how the algorithm works. Figure 1 shows an example of a graph G and its corresponding biconnection tree $\mathbb{T}$. Observe that the non-articulation nodes of G

[1]Simply identifying a logical expression requires linear bits, so this definition of optimality is tight under the traditional model of computation. We concede that it may be slightly generous for certain graph topologies under an amortized analysis within our bitmap model.



(a) G



(b) $\mathbb{T}$

Figure 1: Graph G with corresponding tree $\mathbb{T}$

are precisely the leaves of $\mathbb{T}$, and would also include the root if t had only one child in $\mathbb{T}$.

In the remainder of this section, we present our own modification of Provan and Shier's algorithm which we have tailored to the join enumeration context by tuning it specifically for the common case of graphs with low cyclicity.

3.3.1 Lazy Tree Building

Algorithm 4 is a recursive algorithm that maintains two disjoint, connected sets of vertices S and T. S is grown incrementally on each recursive invocation. Meanwhile T, seeded with an arbitrary vertex t, records vertices that have already been added to S in a different branch of the recursion. $\mathcal{N}(S)$ is the set of neighbours of S within G. Each recursive invocation outputs one minimal cut (two symmetric partitions) corresponding to the given S and then attempts to extend S with one of its neighbours while not causing $G|_{(V \setminus S)}$ to become disconnected. For each vertex node v in the biconnection tree $\mathbb{T}$ (we'll ignore $\mathbb{T}^{old}$ for the moment) the set $\mathcal{D}_{\mathbb{T}}(v)$ contains v and all its descendant vertex nodes in $\mathbb{T}$, and the set $\mathcal{A}_{\mathbb{T}}(v)$ contains v and all its ancestor vertex nodes in $\mathbb{T}$. More formally,

$$\begin{aligned}\mathcal{D}_{\mathbb{T}}(v) &:= \{u \in V \mid u \text{ occurs in subtree of } \mathbb{T} \text{ rooted at } v\} \\ \mathcal{A}_{\mathbb{T}}(v) &:= \{u \in V \mid u \text{ is a } \textit{vertex node} \text{ on path } t \rightsquigarrow v\}\end{aligned}$$

For example, in Figure 1 the nodes a, b, c have the following descendant and ancestor sets.

$$\begin{array}{ll}\mathcal{D}_{\mathbb{T}}(a) = \{a,b,c,d,e\} & \mathcal{A}_{\mathbb{T}}(a) = \{a,t\} \\ \mathcal{D}_{\mathbb{T}}(b) = \{b\} & \mathcal{A}_{\mathbb{T}}(b) = \{a,b,t\} \\ \mathcal{D}_{\mathbb{T}}(c) = \{c\} & \mathcal{A}_{\mathbb{T}}(c) = \{a,c,t\}\end{array}$$

P is the "pivot set" containing the neighbours of S not yet in T that are maximally-distant from t in $\mathbb{T}$ (line 8 exploits the fact that T is connected in order to identify the maximally-distant pivots). For each pivot, the extension of S using that pivot is then explored recursively.

Algorithm 4 has been created by tuning Provan and Shier's generic paradigm for our particular graph cutting problem. Some simple structural changes (replacing pure binary recursion with some iteration, adding the early exit test on line 3, etc.) yield at least a factor of four improvement in execution time over their original algorithm. However, the most expensive operation is the call to BUILDBCCTREE; therefore, the most important optimization we have added to MINCUTLAZY is to pass in the tree $\mathbb{T}^{old}$ from the parent invocation and to lazily construct a new tree only if $\mathbb{T}^{old}$ is not "usable" for $G|_{(V \setminus S)}$. For the purposes of Algorithm 4, $\mathbb{T}^{old}$ is usable if it allows us to compute $\mathcal{D}_{\mathbb{T}}(v)$ and $\mathcal{A}_{\mathbb{T}}(v)$. Although not shown, we define MINCUTEAGER as a version of Algorithm 4 which does not pass $\mathbb{T}^{old}$ forward for re-use and instead calls BUILDBCCTREE on every invocation.

Definition 3.1 (Usability) *Given a graph $G = (V, E)$, two sets $V_2 \subseteq V_1 \subseteq V$ that induce connected graphs $G|_{V_1}$ and $G|_{V_2}$, a distinguished vertex $t \in V_2$, and the biconnection tree $\mathbb{T}_1$ rooted at t corresponding to $G|_{V_1}$: we say that $\mathbb{T}_1$ is usable for $G|_{V_2}$ if the mapping from each biconnected component in $G|_{V_2}$ to the set node in $\mathbb{T}_1$ that contains it is injective.*

The shape of a biconnection tree is determined only by the choice of t and the (singleton) intersections between the biconnected components. Therefore, Definition 3.1 guarantees that the mapping into $\mathbb{T}_1$ preserves the same relative positions of the biconnected components as constructing $\mathbb{T}_2$ from $G|_{V_2}$. Hence, $\mathcal{D}_{\mathbb{T}_2}(v)$ and $\mathcal{A}_{\mathbb{T}_2}(v)$—which depend upon the relative positions of the set nodes in $\mathbb{T}_2$—can be calculated lazily in constant time as $\mathcal{D}_{\mathbb{T}_2}(v) := \mathcal{D}_{\mathbb{T}_1}(v) \cap V_2$ and $\mathcal{A}_{\mathbb{T}_2}(v) := \mathcal{A}_{\mathbb{T}_1}(v) \cap V_2$

Using Definition 3.1 to test usability requires computing the biconnected components of $G|_{V_2}$ which is as expensive as constructing $\mathbb{T}_2$. The following lemma provides a conservative test for usability that requires examining $\mathbb{T}_1$ only.

Lemma 3.2 *A biconnection tree $\mathbb{T}_1$ for connected graph $G|_{V_1}$ is usable for connected graph $G|_{V_2}$ ($V_2 \subseteq V_1$) if for each set node $\beta \in \mathbb{T}_1$ such that $\beta \cap (V_1 \setminus V_2) \neq \varnothing$ all of the children of β in $\mathbb{T}_1$ are contained in $(V_1 \setminus V_2)$.*

Algorithm 5 implements the usability test from Lemma 3.2. For example, consider graph G and tree $\mathbb{T}$ from Figure 1. If vertex b is deleted from G, $\mathbb{T}$ would be still be usable because the deletion simply removed a biconnected component from G; the test using Lemma 3.2 returns true because b is the only child of set node $\{a, b\}$. If vertex c is instead deleted, $\mathbb{T}$ would not be usable because the resulting biconnected components $\{a, d\}$ and $\{d, e\}$ are both contained in $\{a, c, d, e\}$; our test catches this because d and e are un-deleted children of set node $\{a, c, d, e\}$. If e is instead deleted, $\mathbb{T}$ would still be usable (according to Definition 3.1) because the resulting biconnected component $\{a, c, d\}$ can map to set node $\{a, c, d, e\}$; however, our test returns a false negative because $\mathbb{T}$ does not contain enough information[2] to distinguish this from the deletion of c.

Analysis: Provan and Shier prove that their algorithm costs $\Theta(|E|)$ per cut, which is primarily due to the call to BUILDBCCTREE on every recursive invocation. Our refinements do not change worst-case complexity: the usability test on line 5 can be implemented in time $O(|V|)$ using Algorithm 5 and in the worst-case (a clique) $\mathbb{T}^{old}$ is never usable so BUILDBCCTREE gets called on (almost) every recursive invocation (except for those caught by line 3). Therefore, both MINCUTEAGER and MINCUTLAZY take time $\Theta(|E|)$ per cut.

Even though this bound is already optimal for our purposes, using an amortized analysis we can show a better result for MINCUTLAZY over certain acyclic graphs.

On each invocation, let S^{old} be the value of S in the parent invocation, which satisfies $S^{old} \subseteq S$. Then, $\mathcal{N}(S)$ can be

[2]This test can be tweaked to avoid false negatives for biconnected components of size three.

calculated incrementally as

$$\mathcal{N}(S) = (\mathcal{N}(S^{old}) \cup \{\mathcal{N}(v) \mid v \in (S \setminus S^{old})\}) \setminus S$$

which takes time $O(|S \setminus S^{old}|)$ because $\mathcal{N}(v)$ can be obtained in constant time from the bitmap representation of G. The usability test on line 5 can also be performed in time $O(|S \setminus S^{old}|)$ using Algorithm 5. On invocations in which $\mathbb{T}^{old}$ is reused, $\mathcal{D}_{\mathbb{T}}(v)$ and $\mathcal{A}_{\mathbb{T}}(v)$ can be calculated lazily as

$$\begin{aligned} \mathcal{D}_{\mathbb{T}}(v) &= \mathcal{D}_{\mathbb{T}^{old}}(v) \cap (V \setminus S) \\ \mathcal{A}_{\mathbb{T}}(v) &= \mathcal{A}_{\mathbb{T}^{old}}(v) \cap (V \setminus S) \end{aligned}$$

which incurs only a constant overhead. On invocations in which BuildBccTree is called to build a new $\mathbb{T}$, a single depth-first traversal of $\mathbb{T}$ suffices to precompute $\mathcal{D}_{\mathbb{T}}(v)$ and $\mathcal{A}_{\mathbb{T}}(v)$ for all $v \in (V \setminus S)$ in time $O(|E|)$; alternatively, this precomputation can be added into BuildBccTree without affecting its $O(|E|)$ time complexity. Calculation of P can be done in time proportional to $|\mathcal{N}(S)|$. Therefore, the total time for Algorithm 4 to output all of the cuts of an arbitrary graph has the form $O(s + N + t|E|)$ where s is the sum of the new nodes added to S over all invocations, N is the sum of $|\mathcal{N}(S)|$ over all invocations, and t is the total number of biconnection trees built.

Now consider acyclic graphs. Let c be the total number of minimal cuts. First of all, $|E| = c$. Next, it is easy to verify that the usability test always returns true; therefore, MincutLazy requires only one call to BuildBccTree (i.e. $t = 1$). Finally, when $\mathcal{D}_{\mathbb{T}}(v)$ is added to S on line 12, it entails that S will contain precisely a subtree of G rooted at v, and so $|\mathcal{N}(S)| = 1$. The only exception is the root invocation where $S = \varnothing$, in which case $|\mathcal{N}(\varnothing)| = |V \setminus \{t\} = c$. Therefore, the value of N does not exceed twice the number of cuts, which means that for any acyclic graph the total time required for MincutLazy is $O(s + c)$.

In the worst case, $s \in \Theta(|V|^2)$ for acyclic graphs. The reader can verify that if G is a left-deep binary tree rooted at t and line 10 always enumerates the child nodes in order from the top-right to the bottom-left, then there are $\frac{|V|+1}{2}$ recursive invocations at depth one which each add one node to S, but there are $\frac{|V|-3}{2}$ recursive invocations at depth two which each add an entire subtree of G to S. This yields $s = 1 + \left(\frac{|V|-1}{2}\right)^2$, which means that MincutLazy can require $\Omega(|V|)$ amortized time per cut even though only one biconnection tree is built.

For chains, the biconnection tree $\mathbb{T}$ is degenerate and never branches except at the root. This has the effect that on every recursive iteration the pivot set contains only leaves, and so $|\mathcal{D}_{\mathbb{T}}(v)| = 1$ on every execution of line 12. Therefore $s = c$, which means that MincutLazy runs in $O(1)$ amortized time per cut for chains.

Now consider stars. If t is the hub, then there are $|V| - 1$ recursive invocations at depth one which each add one node to S, and they all terminate at line 3. Therefore $s = |V| - 1 = c$. If t is not the hub, then there are $|V| - 2$ recursive invocations at depth one which each add one node to S, but only the first one makes it past line 3. On that invocation the only pivot is the hub, and so on line 12 the rest of the star is added to S, which causes the child invocation to terminate at line 3. Therefore $s = (|V|-2)+(|V|-2) = 2(c-1)$, which means that MincutLazy runs in $O(1)$ amortized time per cut for stars.

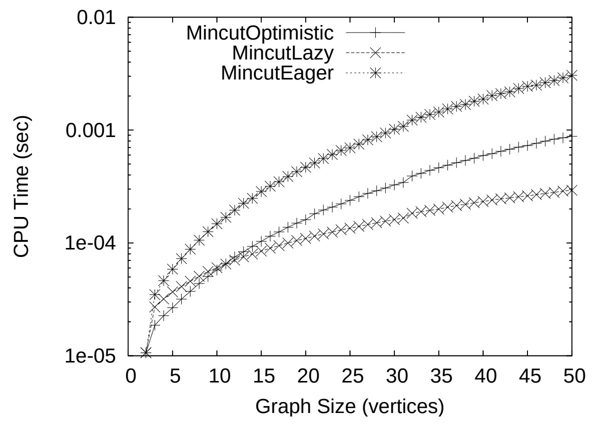


Figure 2: Minimal Cuts of Acyclic Graphs (C=0)

3.3.2 Optimistic Cutting

Algorithm 6 shows a much simpler partitioning algorithm that replaces the use of biconnection trees with a connectivity test. Although it uses a generate-and-test paradigm to some extent, the recursive backtracking used to grow S limits the number of failed tests to neighbours of S and so avoids the potential exponential number of failures of the naive partitioning algorithm in Section 3.2.

Analysis: Each invocation outputs one cut and performs up to $O(|V|)$ connectivity tests costing $O(|V|)$ each, yielding a bound of $O(|V|^2)$ per cut. This analysis is not necessarily tight in an amortized sense, however, because the cost of each successful connectivity test on line 6 can be charged forward to the recursive invocation on line 7.

The total cost to output all cuts has the form $\Theta((c{+}F)|V|)$ where c is the total number of cuts (and successful connectivity tests), and F is the total number of failed connectivity tests. F depends heavily on both graph topology and the order in which line 4 is enumerated. For cliques $F = 0$, while for acyclic graphs $F < c$; both of these cases the yield an amortized cost for Algorithm 6 of $\Theta(|V|)$ per cut. In the worst case (e.g. a spoked wheel in which the hub is the first element added to S), $F \in \Theta(c|V|)$ and so Algorithm 6 has an amortized cost of $\Theta(|V|^2)$ per partition.

3.3.3 Minimal Cut Evaluation

To validate the performance of Algorithm 4 empirically, we report on our experimental comparison between algorithms MincutEager, MincutLazy, and MincutOptimistic. Some of our datasets contain randomly-generated graphs. These random graphs are generated incrementally with different values for the factor C, which controls the degree of cyclicity—with probability C a generated edge connects two existing vertices, while with probability $1 - C$ it connects a new vertex to the graph.

Figures 2–5 compare the total CPU time required by each algorithm to enumerate all minimal cuts for random acyclic graphs (C=0), random cyclic graphs (C=.4), cliques, and spoked wheels. Note the drastically varying scales on these graphs, caused by the relationship between cyclicity and number of minimal cuts. Because the number of minimal cuts (and hence the CPU time) of random graphs can vary significantly, for Figures 2 and 3 we generated 100 random graphs for each size and report mean performance.

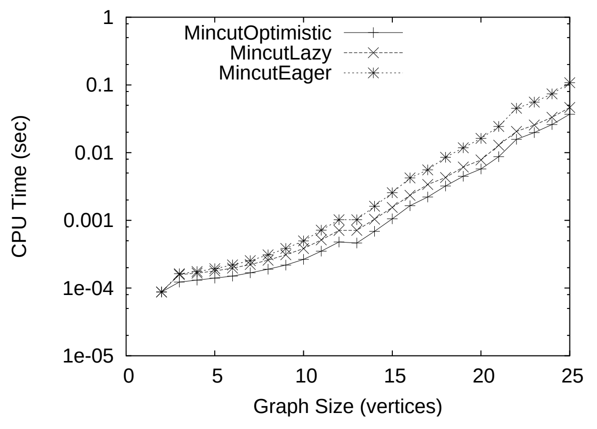


Figure 3: Minimal Cuts of Cyclic Graphs (C=.4)

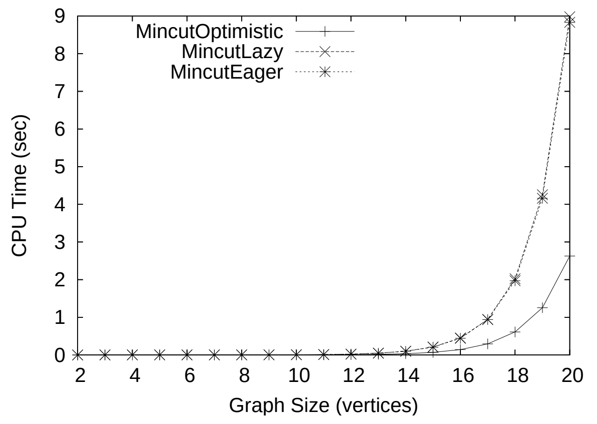


Figure 4: Minimal Cuts of Clique Graphs

As predicted by our previous analysis, MincutLazy is vastly superior for acyclic graphs. As the degree of cyclicity increases, the benefits of lazy construction degrade gracefully: for random cyclic queries (C=.4) MincutLazy is only slightly worse than MincutOptimistic but still much better than MincutEager, whereas in the extreme case of cliques MincutLazy performs identically to MincutEager (albeit much worse than MincutOptimistic). The experiments on spoked wheel graphs demonstrate that there do exist topologies for which the MincutOptimistic scales worse than both MincutEager and MincutLazy. However, the effect only shows up for large graphs; as well, in all experiments on random graphs MincutOptimistic always outperformed MincutEager.

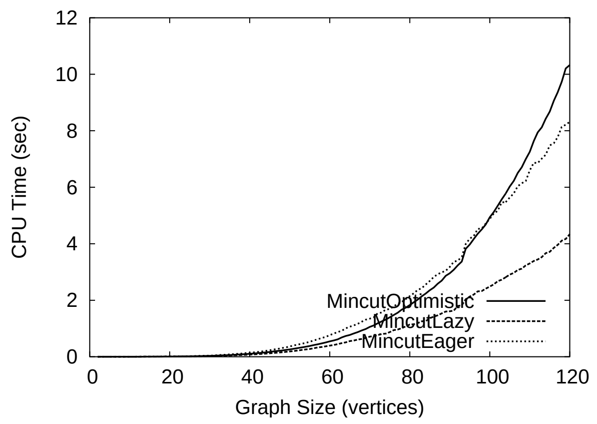


Figure 5: Minimal Cuts of Wheel Graphs

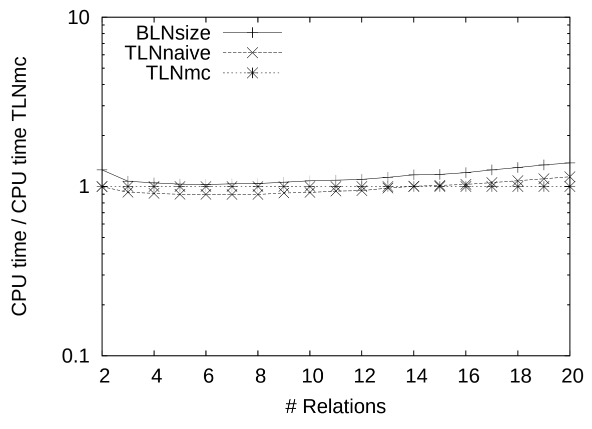


Figure 6: Left-Deep Optimization of Chain Queries

Because the degree of cyclicity in real-world database workloads is very low, we conclude that MincutLazy is the best partitioning algorithm for join enumeration. By combining Algorithm 4 with Algorithm 1 we obtain an optimal top-down join enumeration algorithm for the space of bushy trees without CPs. Note that MincutOptimistic would also be a viable alternative as it is much simpler to implement and appears to perform well in practice; however, in the worst case it is a linear factor worse than optimal.

3.4 Join Enumeration Experiments

We have just shown how to optimally generate the set of partitions that a top-down join enumeration algorithm needs to examine on each recursive call. In this subsection we demonstrate empirically that top-down search algorithms created using Algorithm 1 are competitive with comparable bottom-up dynamic programming algorithms.

We implemented several algorithms within a shared optimization framework that we wrote in Java, using as many shared classes as possible to make comparisons equitable. Table 1 shows the search algorithms that we implemented and the names by which we will refer to them. Our framework implemented three different physical join operators, as well as a simple I/O cost model based on textbook formulae [5]; interesting orders were not implemented. All algorithms were tuned using a Java profiler.

For ease of comparison, throughout this subsection we report normalized CPU times relative to the optimal top-down partitioning search algorithm.

CP-Free Left-Deep Plans. Figure 6 compares the performance of the optimal top-down partitioning algorithm TLNmc with the algorithms TLNnaive and BLNsize over chain queries.

In theory, both TLNnaive and BLNsize are suboptimal for chain queries because for every enumerated join operator they incur a $\Theta(|V|^2)$ cost testing connectivity on a lin-

Type	Enumeration Style	Left-Deep		Bushy	
		No CPs	CPs	No CPs	CPs
Bottom Up	Size Driven (Section 2.1)	BLNsize	BLCsize	BBNsize	BBCsize
	Naive Partitioning (Section 2.2)	-	-	BBNnaive	BBCnaive
	Connected-subgraph Complement Pairs (Section 2.1)	-	-	BBNccp	-
Top Down	Naive Partitioning (Section 3.2)	TLNnaive	TLCnaive	TBNnaive	TBCnaive
	Minimal Cuts (Section 3.3)	TLNmc	-	TBNmc	-

Table 1: Implemented Algorithms

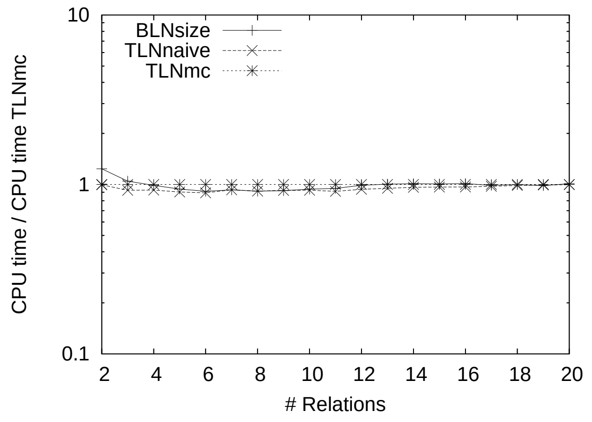


Figure 7: Left-Deep Optimization of Star Queries

ear number of cartesian products (compared with TLNmc's $\Theta(|E|)$ cost to identify the articulation points). In practise, for every join operator enumerated in our experimental framework an algorithm must expend a significant constant overhead using floating-point operations to cost three physical join operators. Figure 6 demonstrates that for query sizes of practical interest in join enumeration, the difference between expending $\Theta(|V|^2)$ and $\Theta(|E|)$ per join operator in the enumeration is modest due to the high constant overhead.

Figures 7 and 8 show similar experiments over star queries and randomly-generated cyclic queries, respectively. We observe that the results are similar to those observed for chain queries, and so we conclude that the added value of optimal partitioning is negligible for CP-free left-deep plans.

CP-Free Bushy Plans. Consider the bottom-up algorithms BBNsize, BBNccp, and BBNnaive in comparison to the top-down algorithms TBNnaive and TBNmc. A comparison of their performance on star queries is shown in Figures 9. Based on these results, we make two observations.

First, the behaviour of the BBNsize, BBNnaive, and BBNccp matches published analytical and empirical results [11], although the effect of the suboptimal enumeration takes longer to show up in our experiments. This latent effect is likely due to the (realistic) constant overhead for plan generation and costing in our optimization framework.

Second, the performance of each top-down algorithm exactly mirrors the analogous bottom-up algorithm. TBNnaive and BBNnaive have the same suboptimal time complexity, and their performance is indistinguishable. TBNmc and BBNccp are both optimal, and their performance is also indistinguishable. There is no top-down algorithm analogous to BBNsize; it is shown here for comparison because

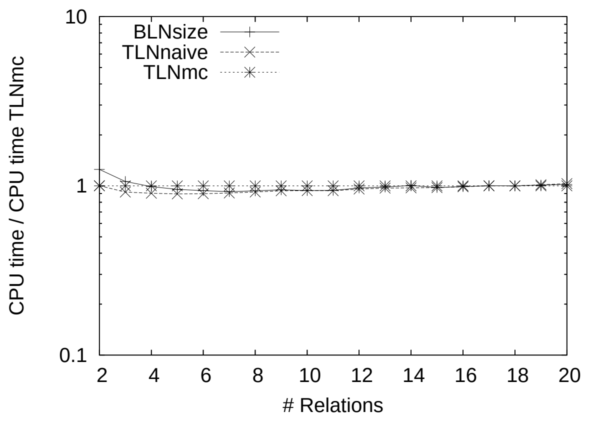


Figure 8: Left-Deep Optimization of Cyclic Queries ($C = .4$)

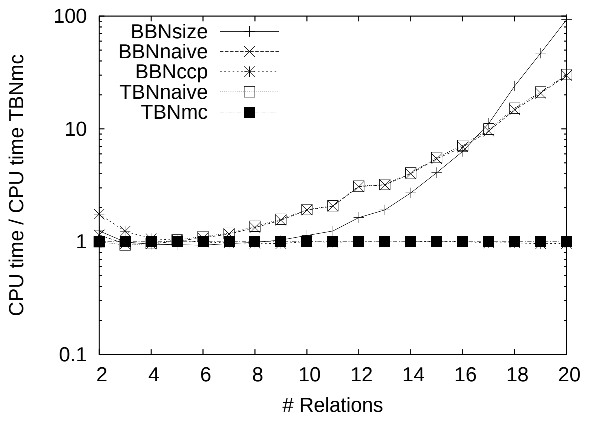


Figure 9: Bushy Optimization of Star Queries

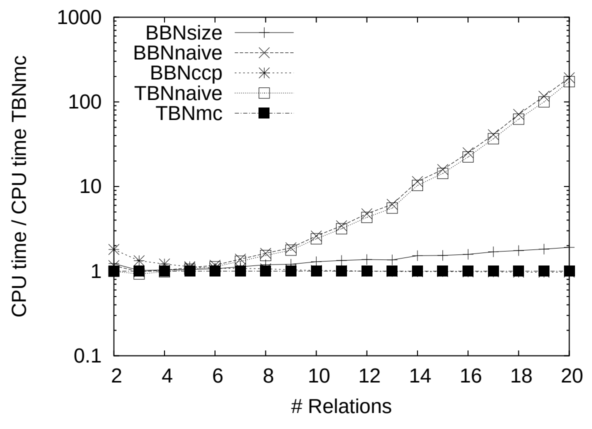


Figure 10: Bushy Optimization of Chain Queries

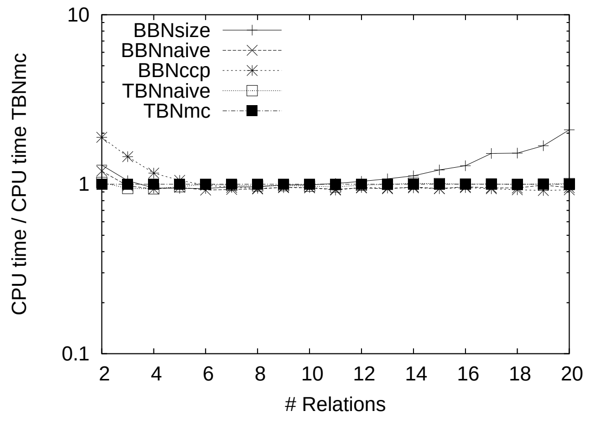


Figure 11: Bushy Optimization of Clique Queries

bottom-up size-based enumeration is widely implemented.

We also tested chains, cliques, cycles, and randomly generated queries ($C = 0, .1, .4$) and found the results to consistent with previous work [11]; some of these results are shown in Figures 10–12. Notably, for cliques the algorithms

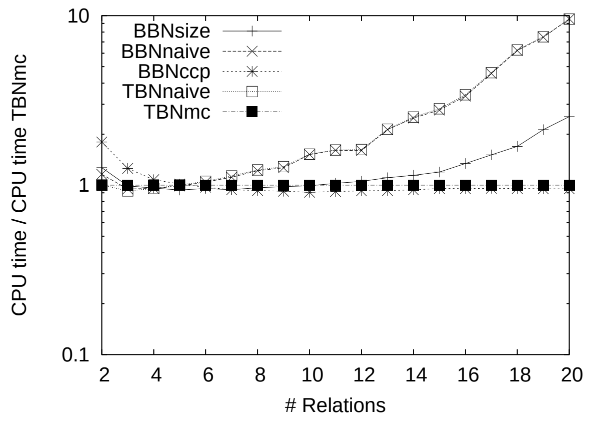


Figure 12: Bushy Optimization of Cyclic Queries ($C = .4$)

BBNNAIVE, TBNNAIVE, BBNCCP and TBNMC are all optimal, and Figure 11 shows them all to be competitive within a small constant (10–15%). This shows that the overhead in Algorithm 4 is negligible even for cliques, which are its worst-case input.

4. BRANCH-AND-BOUND PRUNING

In the previous section we showed that optimal bottom-up dynamic programming and top-down partitioning search algorithms have virtually identical performance when exhaustively enumerating a space of join plans. In this section we show that by integrating branch-and-bound pruning into the backtracking search, top-down partitioning search can avoid exhaustive enumeration while still guaranteeing optimality.

Branch-and-bound pruning is a well-known technique for backtracking search optimization algorithms. The algorithm maintains the cost of the best solution found so far (the upper bound U) and estimates the cost of the best solution obtainable from the current partial solution (the lower bound L). As long as L is a conservative estimate, the current partial solution can be safely abandoned whenever $U \leq L$. The beauty of branch-and-bound is that it is risk-free—in the worst case, it degrades gracefully into exhaustive search with a small constant overhead for the estimation function.

We report on our experience with two different bounding strategies found in the join enumeration literature, which we call *accumulated-cost* and *predicted-cost* bounding. One of these techniques yielded significant performance gains, while the surprisingly dismal performance of the other raises some interesting questions.

4.1 Accumulated-Cost Bounding

Backtracking search join enumeration algorithms build physical plans incrementally by optimizing one join operator at each node of the search tree. In accumulated-cost bounding, U is the cost of the best *complete* join plan found so far, and L is the accumulated cost of the join operators fixed as the algorithm traverses down its search tree; both of these quantities are passed down on each step. This approach is used explicitly in the prefix search of Sybase SQL Anywhere [3]. An equivalent variant preferable for divide-and-conquer approaches is implicit in the search of Volcano, Cascades, and Columbia [8, 6, 20]. Rather than passing down both U and L, those systems pass down a "budget" (i.e. the quantity $U - L$). At each step the budget is decremented by the cost of the current join operator, and search is curtailed if the budget reaches zero.

Partitioning search uses divide-and-conquer, so accumulated-cost bounding can be integrated by modifying each procedure in Algorithm 1 to accept a cost budget B and to return a failure if a plan with cost meeting B cannot be found. The resulting algorithm is shown in Algorithm 7. Observe that whenever the modified GETBESTPLAN receives a failure on lines 6 or 7, it stores B in the (empty) cell $Memo[V, o]$; this allows future invocations to immediately return failure if the given budget does not exceed B.

4.2 Predicted-Cost Bounding

Whereas accumulated-cost bounding is based on passing down information from above, predicted-cost bounding is based on hypothesizing what lies below. The upper bound U is the cost of the best plan found *for the current logical expression*; every time the search algorithm descends, it begins anew with $U = \infty$ until the first plan for that logical expression is obtained. A lower bound L is predicted for each possible branch, and only promising ones are explored. Without exploring a subtree, cost prediction can only be based on logical properties of the subexpression.

Given a lower bound function, adding predicted-cost bounding to top-down partitioning search requires only replacing line 5 of CalcBestJoin in Algorithm 1 as follows.

```
5.1         do if LowerBound($G_L, G_R$) < Cost($BestPlan$)
5.2              then for each operator $G_L \bowtie_i G_R$ satisfying $o$
```

Predicted-cost bounding can also be easily combined with accumulated-cost bounding. To do so, line 8 of CalcBestJoin in Algorithm 7 is modified as follows.

```
8.1         do if LowerBound($G_L, G_R$) ≤
                  Min($B$, Cost($BestPlan$))
8.2              then for each operator $G_L \bowtie_i G_R$ satisfying $o$
```

Clearly the definition of a lower bounding function is completely dependent upon the cost model. As mentioned earlier, our cost model was based on I/O [5]. Because our formulae include the I/O cost of reading the join inputs, we implemented a simple lower bound based on intermediate result size; in other words, the lower bound for $G_L \bowtie G_R$ is proportional to the I/O cost of scanning G_L and G_R, with base relations given a cost of zero[3].

Predicted-cost bounding was the main contribution of the Columbia system [20]. Because Columbia was built on top of Cascades, its branch-and-bounding is a actually combination of accumulated-cost and predicted-cost—notably, it passes in the upper bound as a cost budget, rather than initializing $U = \infty$ as we describe here.

4.3 Branch-and-Bound Experiments

In order to compare the two different bounding strategies just presented, we extended the optimal top-down algorithms TLNmc and TBNmc from Table 1 with accumulated-cost bounding, with predicted-cost bounding, and with a combination of accumulated- and predicted-cost bounding. We identify each variant by appending the algorithm name with an A, P, or AP, respectively. We present here only the results for CP-free search spaces; results for cartesian-products are reported in Section 5.

Unlike exhaustive enumeration, the enumeration time of a branch-and-bound algorithm depends not only on the graph topology, but also on the relation cardinalities and join selectivities. Hence, for all of our experiments involving branch-and-bound algorithms the input is given as a (vertex and edge)-weighted graph. Vertex weights are generated as 10^X, where X is drawn from a Gaussian distribution with $\mu = 5$ and $\sigma = 2$. This distribution gives a realistic mixture of relation cardinalities: 66% between 1k and 10M, 17% <1k, and 17% >10M (which roughly describes a TPC-H instance with scale-factor 10). The distribution of edge weights, representing join selectivities in the range [0,1), was carefully chosen based on the ratio of edges to vertices so that the expected cardinality of the final result ($\prod_{v \in V} \mathit{cardinality}(v) * \prod_{e \in E} \mathit{selectivity}(e)$) is described by 10^Y where Y follows a

[3]Base relations are given zero cost because by using an index, a join plan could avoid touching every tuple in the relation. This is not true for intermediate results.

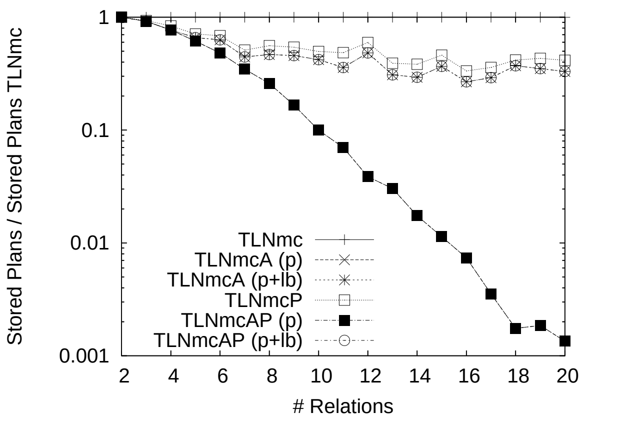


Figure 13: Storage Size: Star Queries, Left-Deep

Gaussian distribution with $\mu = 5$ (albeit with $\sigma > 2$, which is unavoidable). Such edge selectivities yield join inputs and join results with the same expected cardinalities, which is aesthetically pleasing, and also pose a worst-case scenario for branch-and-bound pruning. These selectivities minimize the impact of subexpression size (number of relations) on expected cardinality[4], thereby reducing the variance in the expected costs of the different possible partitions. In contrast, overly small edge selectivities bias the cost model towards left-deep join trees, while overly large edge selectivities bias towards balanced bushy trees.

4.3.1 Storage Size

We first consider how branch-and-bound pruning affects the number of plans stored in the memo table. Figures 13 and 14 show the (normalized) storage cost of optimizing star queries within the left-deep and bushy spaces, respectively. For the exhaustive and predicted-cost algorithms we count the number of table cells storing a plan, while for accumulated-cost algorithms the line labelled "(p)" counts cells storing a plan, while the line "(p+lb)" counts cells storing either a plan or a lower bound. We only report our results for star queries; experiments on other graph topologies yielded similar results. Because the branch-and-bound pruning depends upon the randomly generated weights, all data points for branch-and-bound algorithms reported in this paper are the mean value over 25 different weighted graphs, normalized by the mean value for the optimal exhaustive algorithm (which grows $\Theta(2^n)$ for both left-deep and bushy spaces).

The pattern is identical in the left-deep and bushy spaces: accumulated-cost bounding drastically reduces the number of stored plans, with the reduction steadily increasing with expression size. The total pruning of storage (plans + lower bounds) is much less, however, and reaches a plateau around 80% reduction in these experiments. This plateau is likely due to an increasing number of suboptimal plans that are good enough as to require exploration of most of the plan before the bound is strong enough to allow pruning. The number of stored plans for predicted-cost bounding has a curve similar to the total storage for accumulated-cost bounding, but the pruning is consistently weaker, reaching a plateau around 70% reduction in these experiments. The combina-

[4]This bias cannot be completely eliminated because the average edge-to-vertex ratio varies with subexpression size.

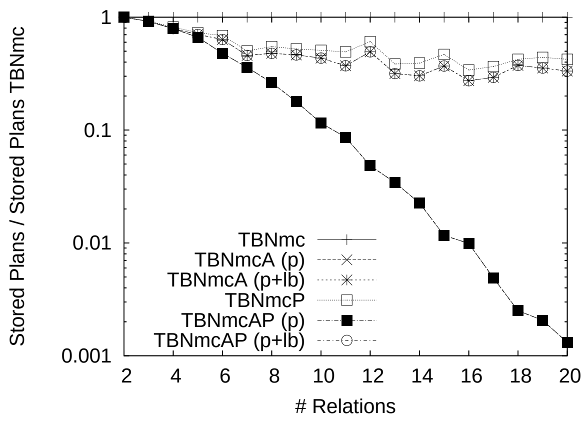


Figure 14: Storage Size: Star Queries, Bushy

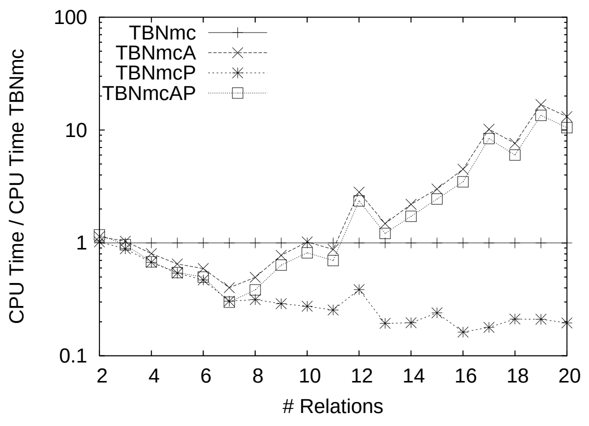


Figure 16: CPU Time: Star Queries, Bushy

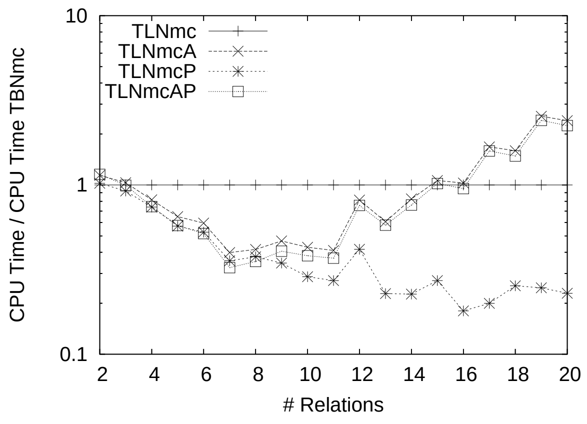


Figure 15: CPU Time: Star Queries, Left-Deep

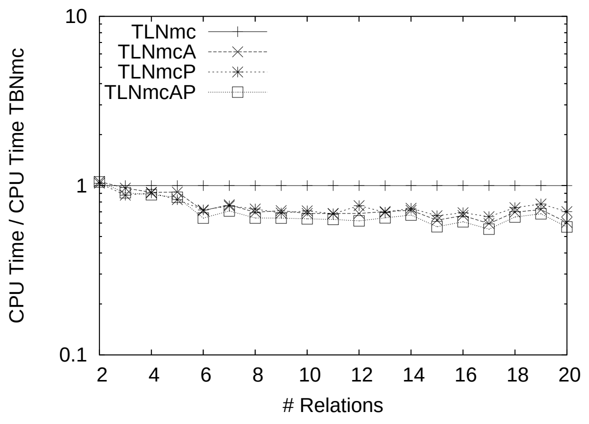


Figure 17: CPU Time: Chain Queries, Left-Deep

tion of the two techniques is no more effective than accumulated-cost bounding by itself.

It is to be expected that accumulated-cost bounding yields better pruning. Accumulated-cost lower bounds are derived from actual physical plan costs, while predicted-cost lower bounds are based solely upon logical properties and are necessarily weak in order to guarantee that the estimate remains conservative. Our experiments use a simple I/O cost model; as the complexity of the cost model increases, the difference in pruning between accumulated- and predicted-cost bounding is expected to grow (due to the increasing difficulty of predicting costs using only logical properties).

4.3.2 CPU Time

We now consider the effect of branch-and-bound pruning on algorithm run time. Figures 15 and 16 compare the CPU cost of optimizing star queries within the left-deep and bushy spaces, respectively. Because the exhaustive algorithms for both left-deep and bushy spaces consider $\Theta(n)$ join operators per cell in the memo table, it is reasonable to expect that pruning of storage directly translates into comparable reductions in CPU time. In fact, the study of branch-and-bound effectiveness within Columbia reports only the number of stored expressions, contending it allows for a platform-independent comparison of system performance [20].

In both the left-deep and bushy spaces, observed-cost bounding initially leads to improvements in CPU time, but as query size increases this improvement quickly disappears until the bounding has devastating *negative* effects! These results are shocking, considering both our earlier characterization of branch-and-bound as "risk-free" and the prevalence of accumulated-cost bounding in the literature! In contrast, predicted-cost bounding reduces CPU time approximately in proportion to the pruning of stored plans, hitting a plateau around 90% reduction in these experiments. The combination of the two methods is almost as bad as accumulated-cost bounding by itself.

Upon further study, we observe that by passing down upper and lower bounds from above (equivalently, a budget), accumulated-cost bounding contextualizes the optimization of each subexpression, which undercuts the divide-and-conquer foundation of memoization. The optimality of TBNmc depends upon exploring the search tree of each logical expression once only. By changing the search algorithm to leave a subtree in defeat when the budget is exhausted, we allow for the same logical expression to be re-optimized multiple times; this shatters the optimality of the enumeration.

Figures 17 and 18 present results for experiments over chain queries. In general, branch-and-bound pruning is not particularly effective for chain queries due to the small search space. For left-deep plans the three bounding strategies have

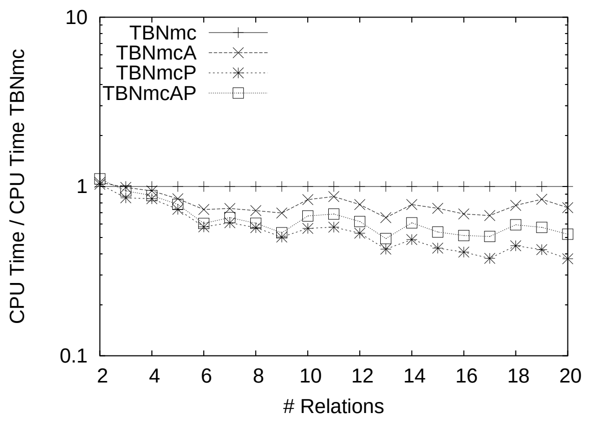


Figure 18: CPU Time: Chain Queries, Bushy

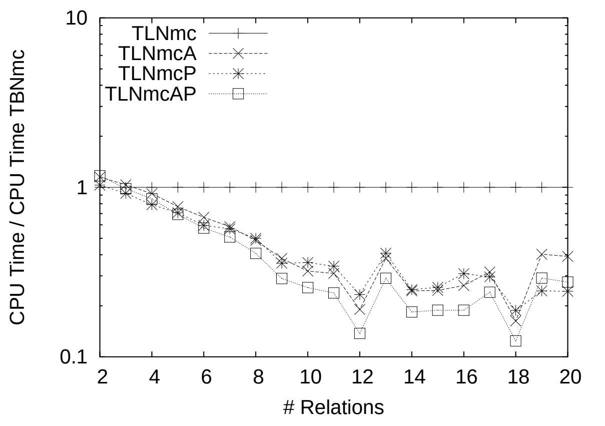


Figure 19: CPU Time: Cyclic Queries ($C = .4$), Left-Deep

similar performance, but for bushy plans we observe that accumulated-cost pruning performs worse than predicted-cost, although it never has negative effects. Figures 19 and 20 show the results for random cyclic queries. For left-deep plans the bounding strategies again perform similarly, with the combination performing slightly better than each strategy on its own. For bushy plans we again see that accumulated-cost bounding performs much worse, eventually achieving a negative effect.

Our results raise the interesting question of why accumulated-cost bounding is implemented in several existing systems. SQL Anywhere's prefix search does not use memoization, and so our findings do not apply—without branch-and-bound that algorithm already incurs redundant computation. The more intriguing question is what the true effect of accumulated-cost bounding is within transformational optimizers. From the previous section we know that the number of unique logical expressions enumerated is remaining fairly constant at around 20% of the table, so the spike in CPU time in Figures 15 and 16 is due to a drastic increase in the number of times that each logical expressions is being re-enumerated; this pattern of increasing frequency of re-enumeration is likely to exist in any search algorithm combining accumulated-cost bounding with memoization. In top-down partitioning search, each time a logical expression

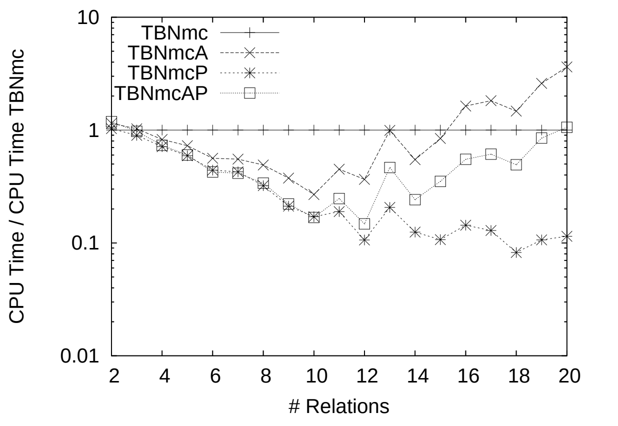


Figure 20: CPU Time: Cyclic Queries ($C = .4$), Bushy

is re-enumerated it incurs the full price of the partitioning function as well as the expense of creating and costing physical join operators that may have been previously discarded. Without a more detailed study, it is unclear the extent to which similar costs are incurred within transformational enumeration. On one hand, it might be possible that by saving *every* derived plan in the memo (as compared to only optimal plans) a transformational optimizer minimizes the redundant work performed by re-enumeration. On the other hand, previous experiments [20, 8] may not have been detailed enough to uncover adverse performance due to redundancy in the enumeration if it exists.

5. FURTHER EXTENSIONS

The foundation of bottom-up dynamic programming is the assumption that the optimal plan must be precalculated for every logical subexpression. In contrast, it is by making these calculations in a demand-driven fashion that top-down algorithms enable the branch-and-bound pruning discussed in the previous section. In this section we discuss another benefit of demand-driven computations: the ability to exploit pre-existing partial information to shape the search. As part of this discussion, we also give performance results over search spaces containing cartesian-products.

5.1 Flexible Memo Tables

Consider the chain query $Q_1 = A \bowtie B \bowtie C$. After optimization, the dynamic programming table contains optimal plans for the expressions A, B, C, AB, BC, and ABC. Now suppose that a second chain query $Q_2 = B \bowtie C \bowtie D$ is submitted to the optimizer. A bottom-up dynamic programming optimizer cannot avoid re-deriving plans for B, C, and BC. In contrast, consider a top-down optimizer that begins its search using the table left over from Q_1: upon descending from ABC to BC it finds a plan already in the table and so avoids optimizing an entire subtree.

The above example raises questions about the proper structuring of a dynamic programming/memo table. All algorithms presented in Section 2 require a data structure providing constant-time lookup by logical expression. Whereas all of the bottom-up methods write blindly and later perform a guaranteed read, top-down partitioning search uses the table as a cache. This caching analogy questions the

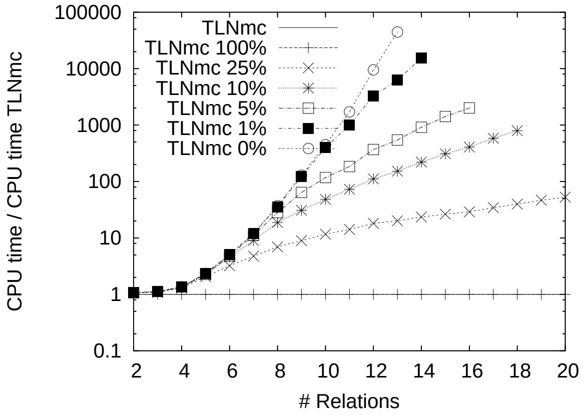


Figure 21: CPU-Storage Trade-off, TLNmc

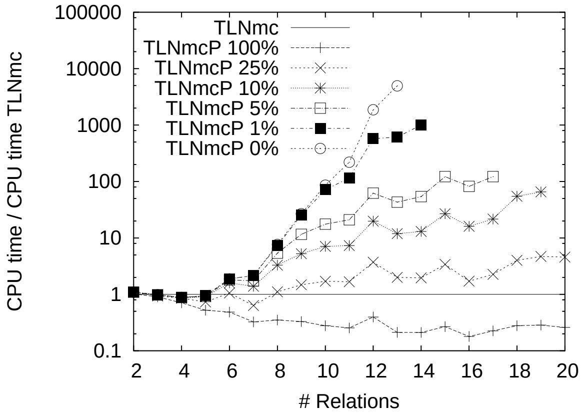


Figure 23: CPU-Storage Trade-off, TLNmcP

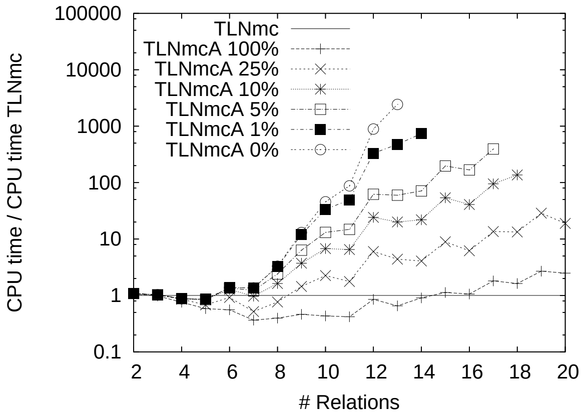


Figure 22: CPU-Storage Trade-off, TLNmcA

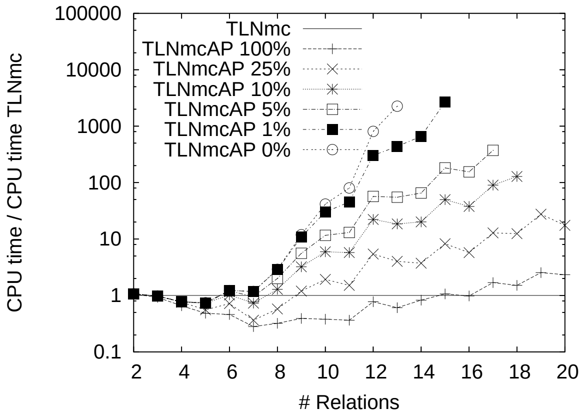


Figure 24: CPU-Storage Trade-off, TLNmcAP

traditional assumption that a fresh table should be used for optimizing each query.

One reason to use a separate data structure per query is to simplify the access. For example, in Algorithm 1 we assume that the memo is keyed only by the set of relations; obviously this would introduce errors if the queries Q_1 and Q_2 shown above had different predicates on tables B or C. Solving this problem simply requires keying the access using a canonical representation of the entire logical expression, which introduces only a minor penalty per access.

The other main reason to use separate tables is to make it easy to delete the contents after optimization completes. All previous algorithms in the literature *require* that all optimal subplans be retained until the algorithm has finished using them. Bottom-up methods would fail if an intermediate expression was deleted before a larger expression depending upon it was calculated. Even top-down transformational systems are rigid on this point, because transformations are applied against expressions already in the memo table, so deleting an expression too early causes transformations to be missed and the search space to be curtailed. In contrast, top-down partitioning search degrades gracefully if a subplan goes missing; it simply recalculates it.

A system could realize two potential benefits by treating the memo table as a global plan cache. First, it would provide a simple method for avoiding redundant computations between optimization of similar queries. In Section 1 we motivated our focus on SPJ queries by noting that we expect the physical optimizer to be called repeatedly by the logical optimizer. Chaudhuri et al. [4] noted that integrating view rewriting directly into the physical join enumeration led to sizable gains over generating all the logical rewritings and optimizing them independently. Using our proposed approach, however, view rewriting could be implemented as a logical optimization, yet avoid performing redundant work between the physical optimization of different rewritings.

The second benefit would be flexible memory management. There is a disparity in the memory cost of the approaches discussed in Section 2. Bottom-up dynamic programming uses $\Omega(2^n)$ memory for storing optimal plans, but top-down transformational algorithms are significantly worse, using $\Omega(3^n)$ memory because they require storing every plan. On the other extreme, prefix search uses $O(n)$ memory but blows up the search space of left-deep join trees to $\Omega(n!)$. Partitioning search requires $\Omega(2^n)$ memory to attain optimal time complexity, but the graceful memory degradation means that depending upon available memory it can be tuned to trade space for time. The memo could be implemented with any cache eviction policy, possibly using the logical descriptions of the current queries in the system to weight the eviction priorities.

To illustrate the effect of trading space for time, we performed a series of experiments using the four left-deep algorithms tested in Section 4.3. Each algorithm was modified to accept as input an integer indicating the maximum number of cells in the table it was allowed to populate, and a simple LRU eviction policy was added to keep the number of populated cells in the memo table below the given maximum. A cell is considered populated if it stores either a query plan or (for accumulated-cost variants only) a lower bound. For each query size, we precalculated the number of populated cells required for optimal enumeration of a star query using Ono and Lohman's formulae [14]. For each algorithm, we optimized star queries of varying size using thresholds that were 100%, 25%, 10%, 5%, 1%, and 0% of the precalculated requirement. When the threshold is 100% an algorithm is maintaining the LRU list but never evicts anything, while with 0% it never stores anything and recomputes every expression on demand.

Figures 21 through 24 show for each algorithm how the execution time increases as the allowable storage is reduced. For consistency with results previously reported in the paper, these graphs report the mean execution normalized against the optimal top-down left-deep algorithm TLNMC (original version). We make two observations. First, in Figure 21 we see that reduction of storage leads to an exponential increase in the CPU time of exhaustive enumeration due to re-computation of evicted plans, with the base of the exponential factor dependent upon the relative reduction in storage. Second, the other figures show that all three branch-and-bound algorithms exhibit a similar exponential increase as storage is reduced, although the relative increase is slightly less than for the exhaustive case. This is likely because the branch-and-bound pruning already reduces the number of cells populated in the table (see Section 4.3.1), so the impact of a fixed storage limit is lessened.

Figures 25 through 30 show the results of the same experiments, this time grouped by storage threshold and normalized by the execution time of the exhaustive algorithm with that threshold. This allows us to compare the four algorithms in each storage context. With 100% storage the results are identical to those in Section 4.3. As the amount of allowed storage is decreased to 25%, predicted-cost bounding improves relative to exhaustive enumeration because, as just described, the reduction in storage impacts it less. However, this effect tapers off as the storage limit decreases further, and the relative performance of TLNMCP is similar for all storage limits below 10%.

In contrast, accumulated-cost bounding (TLNMCA and TLNMCAP) exhibits a steady relative improvement as storage is decreased. There are several factors in play here. Much of the improvement from 100% to 25% is likely for the same reason as described above—the reduction has less impact because accumulated-cost algorithms never used 100% of the table in the first place due to pruning; however, as with predicted-cost pruning, we would expect that effect to taper off for lower storage limits. As described in Section 4.3.2, with 100% storage there is interference between accumulate-cost pruning and memoization because an accumulated-cost algorithm potentially visits a logical expression multiple times (with different cost budgets), while the exhaustive algorithm TLNMC only visits each expression once; this accounts for the poor performance of TLNMCA and TLNMCAP in Figure 25. However, as the storage limit decreases, TLNMC also begins re-visiting logical expressions

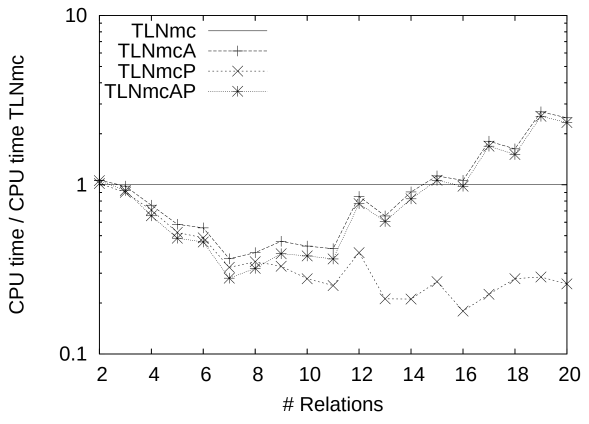


Figure 25: Star Queries, 100% Storage

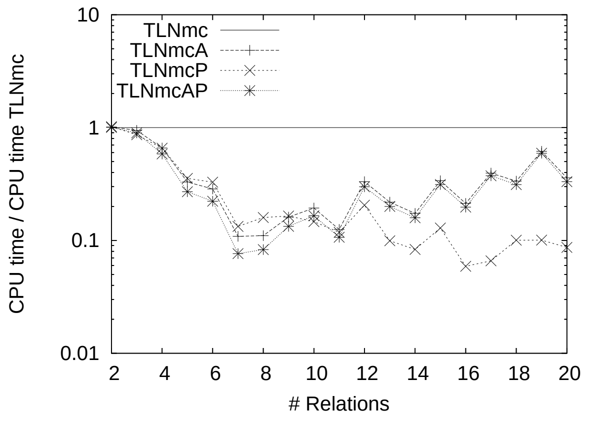


Figure 26: Star Queries, 25% Storage

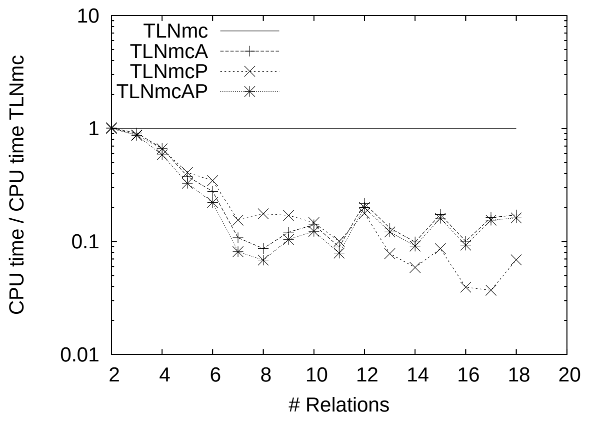


Figure 27: Star Queries, 10% Storage

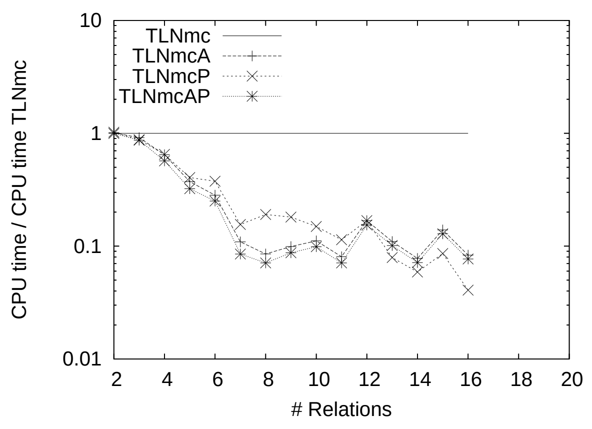


Figure 28: Star Queries, 5% Storage

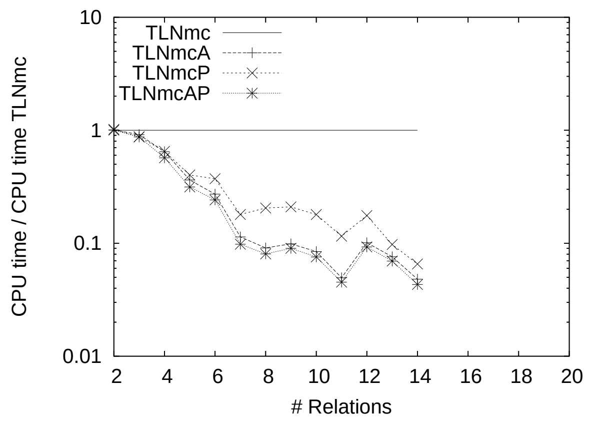


Figure 29: Star Queries, 1% Storage

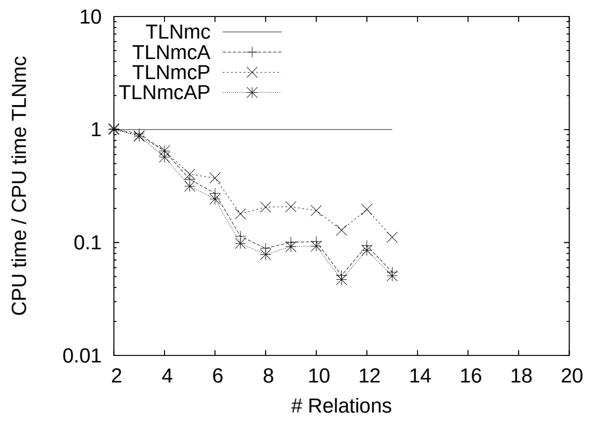


Figure 30: Star Queries, 0% Storage

that have been evicted. Therefore, as the storage limit decreases, the interference between accumulated-cost pruning and memoization also decreases because memoization is becoming less effective. In the extreme case of 0% storage (Figure 30), no memoization takes place and TLNMC revisits an expression for every possible use; in this case any pruning performed by accumulated-cost bounding is guaranteed to reduce the number of expressions visited.

As observed in Section 4.3.1, accumulated-cost bounding uses actual physical plans to derive lower bounds, and so the bounds are significantly stronger than those derived by predicted-cost bounding, which uses only logical properties. Therefore, with no storage we expect that accumulated-cost bounding will always be superior to predicted-cost bounding, which is indeed the case in Figure 30 (and in Figure 29 with 1% storage). One of the most interesting results of these experiments, however, is that accumulated-cost bounding is only dominant over predicted-cost bounding when using very low amounts of storage (¡5% in our experiments). For contexts where more memory is used, the interference between accumulated-cost bounding and memoization is significant enough that savings due to stronger pruning are eventually out-weighed by an increased frequency in re-visiting logical expressions.

5.2 Multi-Phase Optimization

Query optimization requires balancing the conflicting goals of optimization speed and plan quality. One way to manage this trade-off is to optimize a query in an iterative fashion, using successively larger search spaces until a plan of desired quality is found. This raises the interesting question of whether the optimizer can exploit the results of one phase to speed up a later phase. Before we explore that question, we will first compare the different search spaces directly.

Table 2 presents enumeration costs of various search spaces for stars and randomly generated queries. Enumeration algorithms are grouped by search space; the first row of each group shows the number of join operators in the search space[5] (for randomly generated queries, we report the mean across 25 random instances). Subsequent rows show the mean CPU time required to optimize the queries by different algorithms implemented in our Java prototype; timings are for a P4 3.0GHz machine with 1GB RAM.

The first algorithm in each group uses the optimal top-down partitioning algorithm from Section 3.4 to explore the respective space exhaustively. While making no claims about transferability of absolute CPU times, we posit that the first algorithm is conceptually competitive with any exhaustive algorithm over the same search space, which includes all bottom-up dynamic programming methods.

The second algorithm in each group extends the first with predicted-cost bounding (Section 4.2). Note that the pruning is dependent on many factors, including the cost model, the lower bound function, the statistical properties of both the queries and the data, and the distribution of plan quality within the chosen search space. Section 4 only showed results for the CP-free spaces, but pruning is actually much more effective in spaces containing CPs because CPs generally introduce very many bad plans that are easy to prune away early. Relative comparisons between predicted-cost pruning in the space of bushy plans with CPs (TBCNAIVEP)

[5]Numbers vary slightly from earlier formulae [14] because we count $A \bowtie B$ and $B \bowtie A$ separately.

Search Space (join ops)	Star Queries				Random Acyclic (C=0)				Random Cyclic (C=.4)			
Algorithm (sec)	5	10	15	20	5	10	15	20	5	10	15	20
Left-Deep CP-free	*36.0*	*2313*	*1.1e5*	*5.0e6*	*24.2*	*415*	*5.3e3*	*7.3e4*	*31.8*	*836*	*2.1e4*	*4.7e5*
TLNmc	1.2e-4	8.9e-3	0.51	21.0	7.5e-5	1.7e-3	3.6e-2	0.51	1.0e-4	4.2e-3	0.17	3.50
TLNmcP	7.1e-5	2.5e-3	0.14	4.81	5.6e-5	7.8e-4	1.4e-2	0.17	7.3e-5	1.5e-3	4.3e-2	0.85
Bushy CP-free	*64.0*	*4.6e3*	*2.3e5*	*1.0e7*	*47.0*	*1.2e3*	*1.8e4*	*2.5e5*	*63.5*	*3.0e3*	*1.4e5*	*5.0e6*
TBNmc	2.1e-4	1.7e-2	0.91	36.5	1.3e-4	4.0e-3	7.9e-2	1.14	2.0e-4	1.4e-2	0.88	30.6
TBNmcP	1.1e-4	4.6e-3	0.22	7.14	9.1e-5	1.5e-3	2.2e-2	0.26	1.2e-4	2.5e-3	9.4e-2	3.50
Left-Deep with CPs	*75.0*	*5.1e3*	*2.5e5*	*1.0e7*	*75.0*	*5.1e3*	*2.5e5*	*1.0e7*	*75.0*	*5.1e3*	*2.5e5*	*1.0e7*
TLCnaive	2.0e-4	2.4e-2	1.72	89.9	1.8e-4	2.3e-2	1.76	89.4	2.0e-4	2.5e-2	1.89	96.8
TLCnaiveP	8.0e-5	2.7e-3	0.18	5.60	8.0e-5	3.8e-3	0.15	3.63	9.8e-5	3.5e-3	0.16	5.03
TLNmc+TLCnaive	3.3e-4	3.6e-2	2.19	111	2.7e-4	2.6e-2	1.82	90.2	3.1e-4	3.1e-2	2.09	100
TLNmcP+TLCnaiveP	1.2e-4	4.5e-3	0.23	7.87	1.0e-4	2.8e-3	0.12	2.85	1.2e-4	2.5e-3	0.12	4.14
Bushy with CPs	*180*	*5.7e4*	*1.4e7*	*3.5e9*	*180*	*5.7e4*	*1.4e7*	*3.5e9*	*180*	*5.7e4*	*1.4e7*	*3.5e9*
TBCnaive	5.1e-4	0.27	99.9	3.1e4	3.8e-4	0.25	97.6	2.8e4	5.7e-4	0.31	97.3	3.1e4
TBCnaiveP	1.4e-4	1.4e-2	2.30	452	1.4e-4	1.4e-2	1.26	183	1.5e-4	1.0e-2	1.12	195
TBNmc+TBCnaive	7.1e-4	0.30	100	3.0e4	4.9e-4	0.26	97.0	2.8e4	8.1e-4	0.34	98.5	3.1e4
TBNmcP+TBCnaiveP	2.1e-4	1.6e-2	1.82	335	1.8e-4	1.1e-2	0.92	153	2.0e-4	6.9e-3	0.77	148

Table 2: Absolute Cost of Enumerating Various Search Spaces

versus exhaustive search of the left-deep (TLCnaive) or CP-free spaces (TBNmc) are consistent with those reported by Shapiro et al. [20] for the Columbia system (they report the number of join operators enumerated instead of CPU time, but similar patterns are evident).

The disparity between search space sizes in Table 2 reinforces the efficacy of incremental optimization. Consider, however, an application demanding the globally optimal plan. A bottom-up optimizer has no reason to search a smaller space before exploring bushy plans with CPs, because any optimal plan in a smaller space may be suboptimal relative to the larger space and needs to be recalculated. This is not true for a top-down algorithm that uses branch-and-bound. Any suboptimal plan provides an upper bound that could enhance the pruning, especially if this upper bound is somewhat close to globally optimal (as we might expect of the optimal plan in one of the smaller spaces).

For each space containing cartesian products we consider a two-phase optimization strategy where the first phase finds optimal CP-free plans that are used only as initial upper bounds for the second phase. The third algorithm in each group performs exhaustive enumeration in both phases, illustrating the worst-case scenario where no pruning occurs so the first phase was completely wasted. The only timings in Table 2 where a significant overhead is observed is for left-deep plans over star queries. In all other cases, the first phase adds only a small percentage to the overall cost. The fourth algorithm in each group uses predicted-cost pruning in both phases. Except for the left-deep plans over star queries, all other timings show that the first phase pays for itself with about a 20% improvement in the second phase for larger queries. Considering that there are other compelling reasons to optimize a query progressively, this performance gain is effectively free for a top-down optimizer that can leverage the partial information from a previous phase.

6. SUMMARY

Top-down partitioning search is a framework for join enumeration analogous to bottom-up dynamic programming. By taking existing algorithms for the minimal cut problem and tuning them for the join enumeration context, we have presented the first top-down join enumeration algorithm with space and time complexity that is optimal with respect to the join graph. Our optimal enumeration integrates easily with branch-and-bound pruning or demand-driven interesting orders, yielding an algorithm for enumerating bushy join trees that is significantly faster than previous methods. Finally, we have described techniques for exploiting partial information that are unavailable to bottom-up methods, and we have explained how top-down partitioning search is the first dynamic programming-based algorithm to be robust to memory constraints, allowing for a flexible trade-off between storage size and optimization time.

7. REFERENCES

[1] R. Ahmed, A. Lee, A. Witkowski, D. Das, H. Su, M. Zait, and T. Cruanes. Cost-based query transformation in Oracle. In *Proc. 32nd International Conference on Very Large Data Bases*, pages 1026–1036, 2006.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] I. T. Bowman and G. N. Paulley. Join enumeration in a memory-constrainted environment. In *Proc. 16th International Conference on Data Engineering*, pages 645–654, 2000.

[4] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. 11th International Conference on Data Engineering*, pages 190–200, 1995.

[5] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[6] G. Graefe. The Cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[7] G. Graefe. The Microsoft relational engine. In *Proc. 12th International Conference on Data Engineering*, pages 160–161, 1996.

[8] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. 9th International Conference on Data Engineering*, pages 209–218, 1993.

[9] N. Kabra and D. J. DeWitt. OPT++: An object-oriented implementation for extensible database query optimization. *VLDB J.*, 8(1):55–78, 1999.

[10] W. J. McKenna, L. Burger, C. Hoang, and M. Truong. EROC: A toolkit for building NEATO query optimizers. In

Proc. 22nd International Conference on Very Large Data Bases, pages 111–121, 1996.

[11] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proc. 32nd International Conference on Very Large Data Bases*, pages 930–941, 2006.

[12] E. F. Moore and C. E. Shannon. Reliable circuits using less reliable relays. *J. Franklin Institute*, 262(3,4):191–208, 281–297, 1956.

[13] S. Morishita. Avoiding cartesian products for multiple joins. *J. ACM*, 44(1):57–85, 1997.

[14] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. 16th International Conference on Very Large Data Bases*, pages 314–325, 1990.

[15] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. Complexity of transformation-based optimizers and duplicate-free generation of alternatives. Technical Report CS-R9639, CWI, 1996.

[16] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. The complexity of transformation-based join enumeration. In *Proc. 23rd International Conference on Very Large Data Bases*, pages 306–315, 1997.

[17] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 39–48, 1992.

[18] J. S. Provan and D. R. Shier. A paradigm for listing (s,t)-cuts in graphs. *Algorithmica*, 15(4):351–372, April 1996.

[19] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[20] L. D. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H. min Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. In *International Database Engineering & Applications Symposium*, pages 20–33, 2001.

[21] S. Tsukiyama, I. Shirakawa, H. Ozaki, and H. Ariyoshi. An algorithm to enumerate all cutsets of a graph in linear time per cutset. *J. ACM*, 27(4):619–632, 1980.

[22] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 35–46, 1996.

Algorithm 1 Top-Down Partition Search

```
GetBestPlan($G, o$)
    ▷ Input: join graph $G = (V, E)$
    ▷ Input: interesting order $o$
    ▷ Returns: optimal plan satisfying $o$
1   if $Memo[V, o]$ is empty
2     then if $|V| = 1$
3            then $Memo[V, o] \leftarrow$ CalcBestScan($G, o$)
4            else $Memo[V, o] \leftarrow$ CalcBestJoin($G, o$)
5   return $Memo[V, o]$

CalcBestScan($G, o$)
    ▷ Input: trivial join graph $G = (\{R\}, \varnothing)$
    ▷ Input: interesting order $o$
    ▷ Returns: optimal scan satisfying $o$
1   $BestPlan \leftarrow$ null                    ▷ Let Cost(null) $= \infty$
2   if $o \neq \varnothing$   ▷ Start with plan that enforces $o$ by sorting
3     then $BestPlan \leftarrow$ Sort$_o$(GetBestPlan($G, \varnothing$))
4   for each operator $OpScan_i(R)$ satisfying $o$
5       do $CurrPlan \leftarrow OpScan_i(R)$
6          if Cost($CurrPlan$) < Cost($BestPlan$)
7            then $BestPlan \leftarrow CurrPlan$
8   return $BestPlan$

CalcBestJoin($G, o$)
    ▷ Input: non-trivial join graph $G = (V, E)$
    ▷ Input: interesting order $o$
    ▷ Returns: optimal join plan satisfying $o$
1   $BestPlan \leftarrow$ null                    ▷ Let Cost(null) $= \infty$
2   if $o \neq \varnothing$   ▷ Start with plan that enforces $o$ by sorting
3     then $BestPlan \leftarrow$ Sort$_o$(GetBestPlan($G, \varnothing$))
4   for each partition $(G_L, G_R)$ in Partition(G)
5       do for each operator $G_L \bowtie_i G_R$ satisfying $o$
6              do $o_L \leftarrow$ order for $G_L$ required by $\bowtie_i$
7                 $p_L \leftarrow$ GetBestPlan($G_L, o_L$)
8                 $o_R \leftarrow$ order for $G_R$ required by $\bowtie_i$
9                 $p_R \leftarrow$ GetBestPlan($G_R, o_R$)
10                $CurrPlan \leftarrow p_L \bowtie_i p_R$
11                if Cost($CurrPlan$) < Cost($BestPlan$)
12                  then $BestPlan \leftarrow CurrPlan$
13  return $BestPlan$
```

Algorithm 2 Naive Partitioning

```
Partition($G$)
    ▷ Input: join graph $G = (V, E)$
    ▷ Output: left-deep partitions of $G$
1   for $v \in V$
2       do output $(G|_{(V \setminus \{v\})}, G|_{\{v\}})$
```

Algorithm 3 Build Biconnection Tree

```
BuildBccTree($G, t$)
    ▷ Input: connected join graph $G = (V, E)$
    ▷ Input: vertex $t \in V$
    ▷ Output: biconnection tree $\mathbb{T}$ for $G$ rooted at $t$
1   declare set of vertices VISITED = $\varnothing$
2   declare integer COUNT = 1
3   declare map:vertex→int DFNUM
4   declare map:vertex→int LOW
5   declare map:vertex→vertex FATHER
6   declare stack of graph edges ESTACK
7   declare stack of biconnection tree vertex nodes VSTACK
8   declare stack of biconnection tree set nodes SSTACK
9   return DFSearch($t$)

DFSearch($v$)
    ▷ Input: vertex $v \in V$
    ▷ Output: subtree of $\mathbb{T}$ rooted at $v$
 1  VISITED ← VISITED ∪ $\{v\}$
 2  DFNUM[$v$] ← COUNT++
 3  LOW[$v$] ← DFNUM[$v$]
 4  for each neighbour $w$ of $v$ in $G$
 5       do if $w \notin$ VISITED
 6            then push $(v, w)$ onto ESTACK
 7                 FATHER[$w$] ← $v$
 8                 push DFSearch($w$) onto VSTACK
 9                 if LOW[$w$] ≥ DFNUM[$v$]
10                   then                 ▷ completed bcc
11                        $bcc \leftarrow \varnothing$
12                        repeat  $(e_1, e_2)$ ← pop ESTACK
13                                add $e_1, e_2$ to $bcc$
14                          until $(e_1, e_2) = (v, w)$
15                        create set node $n_{bcc}$ for $bcc$
16                        while top of VSTACK ∈ $bcc$
17                              do $n_c$ ← pop VSTACK
18                                 make $n_c$ a child of $n_{bcc}$
19                        push $n_{bcc}$ onto SSTACK
20                 LOW[$v$] ← Min(LOW[$v$], LOW[$w$])
21          elseif $w \neq$ FATHER[$v$]
                 and DFNUM[$v$] > DFNUM[$w$]
22            then push $(v, w)$ onto ESTACK
23                 LOW[$v$] ← Min(LOW[$v$], DFNUM[$w$])
24  create vertex node $n_v$ for $v$
25  while set node at top of SSTACK contains $v$
26       do $n_{bcc}$ ← pop SSTACK
27          make $n_{bcc}$ a child of $n_v$
28  return $n_v$
```

Algorithm 4 Minimal Cut Partitioning

```
Partition($G$)
    ▷ Input: connected join graph $G = (V, E)$
    ▷ Output: all minimal cuts of $G$
1   $t$ ← arbitrary element of $V$
2   MincutLazy($\varnothing, \{t\}$, null)

MincutLazy($S, T, \mathbb{T}^{old}$)
    ▷ Input: disjoint sets $S, T \subseteq V$
    ▷ Input: biconnection tree from parent invocation
    ▷ Output: minimal cuts extended from $S$
 1  if $S \neq \varnothing$
 2     then output $(G|_S, G|_{(V\setminus S)})$ and $(G|_{(V\setminus S)}, G|_S)$
 3  if $\mathcal{N}(S) \subseteq T$                ▷ Define $\mathcal{N}(\varnothing) = V \setminus \{t\}$
 4     then return                    ▷ $S$ cannot be extended
 5  if $\mathbb{T}^{old}$ is usable for $G|_{(V\setminus S)}$
 6     then $\mathbb{T} \leftarrow \mathbb{T}^{old}$
 7     else  $\mathbb{T}$ ← BuildBccTree($G|_{(V\setminus S)}, t$)
 8  $P \leftarrow \{v \in \mathcal{N}(S) \mid v \in (V \setminus (S \cup T))$
                  $\wedge(\mathcal{D}_{\mathbb{T}}(v) \cap \mathcal{N}(S)) = \{v\}\}$
 9  $T' \leftarrow T$
10  for $v \in P$
11        do MincutLazy($S \cup \mathcal{D}_{\mathbb{T}}(v), T', \mathbb{T}$)
12           $T' \leftarrow T' \cup \mathcal{A}_{\mathbb{T}}(v)$
```

Algorithm 5 Biconnection Tree Usability

```
isUsable($\mathbb{T}, V_1, V_2$)
    ▷ Input: biconnection tree $\mathbb{T}$
    ▷ Input: set $V_1$ such that $\mathbb{T}$ is usable for $G|_{V_1}$
    ▷ Input: set $V_2 \subseteq V_1$
    ▷ Output: boolean indicating if $\mathbb{T}$ is usable for $G|_{V_2}$
 1  if $V_2 = \varnothing$
 2     then return true
 3  if $V_2$ does not contain root (vertex) node of $\mathbb{T}$
 4     then return false
 5  $V_{deleted} \leftarrow V_1 \setminus V_2$
 6  for $v \in V_{deleted}$
 7        do $n_v$ ← vertex node in $\mathbb{T}$ corresponding to $v$
 8           $n_{bcc}$ ← parent (set) node of $n_v$ in $\mathbb{T}$
 9           $n_u$ ← parent (vertex) node of $n_{bcc}$ in $\mathbb{T}$
10           if $(bcc \setminus \{u\}) \not\subseteq V_{deleted}$
11              then return false
12  return true
```

Algorithm 6 Optimistic Partitioning

```
PARTITION($G$)
    ▷ Input: connected join graph $G = (V, E)$
    ▷ Output: all minimal cuts of $G$
1   $t \leftarrow$ arbitrary element of $V$
2   MINCUTOPTIMISTIC($\varnothing, \{t\}$)

MINCUTOPTIMISTIC($S, T$)
    ▷ Input: disjoint sets $S, T \subseteq V$
    ▷ Output: minimum cuts extended from $S$
1   if $S \neq \varnothing$
2       then output $(G|_S, G|_{(V \setminus S)})$ and $(G|_{(V \setminus S)}, G|_S)$
3   $T' \leftarrow T$
4   for $v \in (\mathcal{N}(S) \setminus T)$            ▷ Define $\mathcal{N}(\varnothing) = V \setminus \{t\}$
5       do $S' \leftarrow S \cup \{v\}$
6           if $G|_{(V \setminus S')}$ is connected
7               then MINCUTOPTIMISTIC($S', T'$)
8           $T' \leftarrow T' \cup \{v\}$
```

Algorithm 7 Top-Down Partition Search with Accumulated-Cost Bounding

```
GETBESTPLAN($G, o, B$)
    ▷ Input: join graph $G = (V, E)$
    ▷ Input: interesting order $o$
    ▷ Input: cost budget $B$
    ▷ Returns: optimal plan satisfying $o$ with cost not
               exceeding $B$, or NULL if no such plan exists
1   $BestPlan \leftarrow$ NULL
2   if $Memo[V, o]$ contains plan with cost $\leq B$
3       then $BestPlan \leftarrow Memo[V, o]$
4   elseif $Memo[V, o]$ is empty or contains lower bound $< B$
5       then if $|V| = 1$
6               then $BestPlan \leftarrow$ CALCBESTSCAN($G, o, B$)
7               else $BestPlan \leftarrow$ CALCBESTJOIN($G, o, B$)
8            if $BestPlan =$ NULL
9               then $Memo[V, o] \leftarrow$ lower bound $B$
10              else $Memo[V, o] \leftarrow BestPlan$
11  return $Memo[V, o]$

CALCBESTSCAN($G, o, B$)
    ▷ Input: trivial join graph $G = (\{R\}, \varnothing)$
    ▷ Input: interesting order $o$
    ▷ Input: cost budget $B$
    ▷ Returns: optimal scan satisfying $o$ and $B$, or NULL
1   $BestPlan \leftarrow$ NULL                ▷ Let COST(NULL) $= \infty$
2   if $o \neq \varnothing$   ▷ Start with plan that enforces $o$ by sorting
3       then $C \leftarrow$ cost of sorting tuples in $G$ by $o$
4            $UnsortedPlan \leftarrow$ GETBESTPLAN($G, \varnothing, B - C$)
5            if $UnsortedPlan \neq$ NULL
6               then $BestPlan \leftarrow$ SORT$_o$($UnsortedPlan$)
7   for each operator $OpScan_i(R)$ satisfying $o$
8       do $CurrPlan \leftarrow OpScan_i(R)$
9            if COST($CurrPlan$) $<$ COST($BestPlan$)
             and COST($CurrPlan$) $\leq B$
10              then $BestPlan \leftarrow CurrPlan$
11  return $BestPlan$

CALCBESTJOIN($G, o, B$)
    ▷ Input: non-trivial join graph $G = (V, E)$
    ▷ Input: interesting order $o$
    ▷ Input: cost budget $B$
    ▷ Returns: optimal join plan satisfying $o$ and $B$, or NULL
1   $BestPlan \leftarrow$ NULL                ▷ Let COST(NULL) $= \infty$
2   if $o \neq \varnothing$   ▷ Start with plan that enforces $o$ by sorting
3       then $C \leftarrow$ cost of sorting tuples in $G$ by $o$
4            $UnsortedPlan \leftarrow$ GETBESTPLAN($G, \varnothing, B - C$)
5            if $UnsortedPlan \neq$ NULL
6               then $BestPlan \leftarrow$ SORT$_o$($UnsortedPlan$)
7   for each partition $(G_L, G_R)$ in PARTITION(G)
8       do for each operator $G_L \bowtie_i G_R$ satisfying $o$
9              do $C_{\bowtie_i} \leftarrow$ cost of operator $\bowtie_i$
10                $B' \leftarrow$ MIN($B$, COST($BestPlan$)) $- C_{\bowtie_i}$
11                $o_L \leftarrow$ order for $G_L$ required by $\bowtie_i$
12                $p_L \leftarrow$ GETBESTPLAN($G_L, o_L, B'$)
13                if $p_L \neq$ NULL
14                   then $B' \leftarrow B' -$ COST($p_L$)
15                        $o_R \leftarrow$ order for $G_R$ required by $\bowtie_i$
16                        $p_R \leftarrow$ GETBESTPLAN($G_R, o_R, B'$)
17                        if $p_R \neq$ NULL
18                           then $BestPlan \leftarrow p_L \bowtie_i p_R$
19  return $BestPlan$
```