Rapid Bushy Join-order Optimization with Cartesian Products

Bennet Vance
Oregon Graduate Institute of Science & Technology
bennet@cse.ogi.edu
http://www.cse.ogi.edu/DISC

David Maier
Oregon Graduate Institute of Science & Technology
maier@cse.ogi.edu
http://www.cse.ogi.edu/DISC

Abstract

Query optimizers often limit the search space for join orderings, for example by excluding Cartesian products in subplans or by restricting plan trees to left-deep vines. Such exclusions are widely assumed to reduce optimization effort while minimally affecting plan quality. However, we show that searching the complete space of plans is more affordable than has been previously recognized, and that the common exclusions may be of little benefit.

We start by presenting a Cartesian product optimizer that requires at most a few seconds of workstation time to search the space of bushy plans for products of up to 15 relations. Building on this result, we present a join-order optimizer that achieves a similar level of performance, and retains the ability to include Cartesian products in subplans wherever appropriate. The main contribution of the paper is in fully separating join-order enumeration from predicate analysis, and in showing that the former problem in particular can be solved swiftly by novel implementation techniques. A secondary contribution is to initiate a systematic approach to the benchmarking of join-order optimization, which we apply to the evaluation of our method.

1 Introduction

Because join-order optimization is NP-complete [IK84], practical solutions to the problem typically involve engineering trade-offs. This paper presents a new perspective on the engineering trade-offs that arise when join orders are optimized by exhaustive search.

One way that optimizers cope with the intractability of join optimization is to exclude some classes of plans from consideration. Plans with Cartesian products are frequently excluded on the grounds that they are unlikely to be optimal, and so it is not worth expending effort in analyzing such plans. Many optimizers also restrict their attention to the space of left-deep plans [IK91], on the grounds that (a) the space of bushy plans is vastly larger, and hence much more expensive to search; and (b) there are nonetheless plenty of left-deep plans to choose from, and so the best among them will likely suffice. We explore a different tack: how to make optimization that considers all join orders (including Cartesian products) run fast.

We approach the bushy-join optimization problem obliquely, first showing that cost-based optimization of multiway Cartesian products (i.e., expressions of the form $A \times B \times C \times D$) is not especially difficult—even when bushy product trees are considered. Although Cartesian product optimization via exhaustive enumeration has exponential complexity, it lends itself to very low-overhead implementation. Our timings for optimization of multiway Cartesian products are several orders of magnitude lower than we have seen reported for join-optimization problems of comparable size.

That result is interesting not because Cartesian product optimization is useful, but because it has much in common with join-order optimization. We show that our Cartesian product optimizer can be extended to accommodate join predicates (and hence to optimize multiway joins) without large loss of performance. This approach to join optimization carries the side benefit that, since our optimizer is at heart a Cartesian product optimizer, plans with Cartesian products will be chosen should they be optimal.

In benchmarking our join-optimization method, we attempt to isolate the characteristics of the input that influence performance. Our preliminary analysis indicates that with the number of base relations held fixed, the primary determinants of optimization time are the cost model and join-graph topology, while the base-relation cardinalities have a smaller but still important influence.

The work described here assumes that predicates are uncorrelated, and the cost models we consider are relatively simple. (Such simplifications are not unusual in the research literature [CM95, GLPK94].)

The paper proceeds as follows. Section 2 discusses related work. Section 3 presents an algorithm for Carte-

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

sian product optimization and analyzes its complexity. Section 4 shows how that algorithm can be efficiently implemented. Section 5 revises the algorithm to incorporate join predicates. Section 6 reports on performance measurements. We conclude in Section 7.

2 Related Work

Selinger et al. [SAC+79] applied dynamic programming to join-order optimization in the System R optimizer. Their optimizer focused on left-deep plans and excluded (or deferred) Cartesian products, but searched exhaustively subject to those restrictions.

Ono and Lohman [OL90] point out that the optimal plan for a multiway join may contain Cartesian products, and note that it is desirable for an optimizer to permit flexibility in optimization trade-offs. Their Starburst optimizer will search left-deep or bushy plan spaces, as requested, and also leaves consideration of Cartesian products as an option. They report the time complexity of their optimizer under various circumstances. The complexity is $O(n^3)$ (where n is the number of relation variables) to find optimal bushy plans for linear (or *chain*) join graphs without Cartesian products. In this special case, their optimizer performs spectacularly—the authors note that they can find optimal plans for joins of as many as 100 relations. Complexity for star queries is $O(n2^n)$, again without Cartesian products. When Cartesian products are considered, the number of joins enumerated is $O(n2^n)$ for left-deep search, and $O(3^n)$ for bushy search, regardless of the join graph. However, the underlying worst-case complexity of the enumerator itself is $O(4^n)$.

Cluet and Moerkotte [CM95] study the feasibility of optimization with Cartesian products from a theoretical standpoint. Their work, which considers only left-deep plan spaces, addresses the question of extending a join-optimization result from Ibaraki and Kameda [IK84]. That result showed polynomial complexity for optimization of queries with acyclic graphs under a restricted class of cost models. But even with these restrictions, optimization again becomes NP-complete when Cartesian products are allowed in subplans. Thus, some join-optimization problems are fundamentally harder with Cartesian products than without.

Graefe and McKenna [GM93] describe the extensible, rule-based Volcano optimizer. Their test runs allow bushy plans but not Cartesian products, and yield timings that, to our knowledge, reflect the state of the art in rule-based optimization. Like Starburst, Volcano adapts to the join-graph topology. In the worst case, Volcano optimizes joins in $O(3^n)$ time and $O(3^n)$ space.

Galindo-Legaria, Pellenkoft, and Kersten [GLPK94] propose an unconventional stochastic approach to join-order optimization. Whereas previous stochastic techniques (e.g., simulated annealing and iterated improvement [Ste96]) used transformations on plan trees to move from one state of the plan space to a neighboring state, the method of Galindo-Legaria et al. instead probes points of the plan space at random, avoiding the expense of navigating the space by applying transformations. The technique reportedly outperforms its predecessors, but as presented is applicable only to queries with acyclic graphs.

Steinbrunn [Ste96] surveys a variety of heuristic and stochastic techniques, and includes extensive empirical measurements. The stochastic searches appear to converge on plans of high quality, but the time it takes them to do so is often substantial. The evidence suggests that exhaustive search is the method of choice for n into the mid-teens; however, any such judgment is subject to revision as more becomes known about the problem.

3 Cartesian Product Optimization

We present below a straightforward dynamic programming algorithm for optimizing Cartesian products. This algorithm will serve as the foundation for the join-order optimization method we present subsequently.

Our algorithm differs structurally from dynamic programming algorithms used previously in join optimization; complexity measures, discussed later, are one manifestation of the difference. We begin by presenting the conception behind the algorithm; we then give a pseudocode version and study its complexity.

3.1 Conception of the Algorithm

Suppose we wish to find the optimal expression for computing the Cartesian product $A \times B \times C \times D$. (Assume that only a dyadic $\times$ operator is available.) Before proceeding, we need a cost model and some information about A, B, C, and D (e.g., their cardinalities). Let the cost of evaluating an expression be defined recursively as

$$cost(R) = 0 \tag{1}$$

$$cost(E \times E') = cost(E) + cost(E') + \kappa([\![E \times E']\!], [\![E]\!], [\![E']\!]), \tag{2}$$

where R names a base relation, E and E' are subexpressions, κ is a cost function that depends on the cost model, and $[\![E]\!]$ is the *value* (i.e., the relation) denoted by the expression E. Note that while the assumption $cost(R) = 0$ is unrealistic for some cost models, it is convenient and causes no harm, since the costs assigned to base relations do not affect the outcome of optimization. Note also that to compute $\kappa([\![E \times E']\!], [\![E]\!], [\![E']\!])$ it is not really necessary to evaluate $E \times E'$, E, and E'; the denotation $[\![E]\!]$ can just as well be an abstract interpretation of E. For example, for some cost models it suffices to obtain an estimate of the cardinality of the

Relation Set	Cardinality	Best LHS	Cost
$\{A\}$	10	none	0
$\{B\}$	20	none	0
$\{C\}$	30	none	0
$\{D\}$	40	none	0
$\{A,B\}$	200	$\{A\}$	200
$\{A,C\}$	300	$\{A\}$	300
$\{A,D\}$	400	$\{A\}$	400
$\{B,C\}$	600	$\{B\}$	600
$\{B,D\}$	800	$\{B\}$	800
$\{C,D\}$	1200	$\{C\}$	1200
$\{A,B,C\}$	6000	$\{A,B\}$	6200
$\{A,B,D\}$	8000	$\{A,B\}$	8200
$\{A,C,D\}$	12000	$\{A,C\}$	12300
$\{B,C,D\}$	24000	$\{B,C\}$	24600
$\{A,B,C,D\}$	240000	$\{A,D\}$	241000

Table 1: Dynamic programming table

relation denoted by E. Other cost models require additional information, but no sensible model will require complete knowledge of the relations under consideration.

Now let A, B, C, and D have cardinalities 10, 20, 30, and 40, respectively, and consider the naive cost model given by $\kappa_0(R_{out}, R_{lhs}, R_{rhs}) = |R_{out}|$. (That is, the cost of computing a given Cartesian product is equal to the cardinality of the result.) Table 1 illustrates the application of dynamic programming to the given optimization problem. The table is constructed in such a way that we can extract an optimal expression for the four-way product as follows. First we consult the final row of the table, which says that the Best LHS (best left-hand side) for $\{A,B,C,D\}$ is $\{A,D\}$, i.e., that the product of $\{A,B,C,D\}$ is best computed as $E \times E'$, where E computes the product of $\{A,D\}$ and E' computes the product of $\{B,C\}$. We then find optimal subexpressions for the product of $\{A,D\}$ and the product of $\{B,C\}$ by recursively consulting the table in the same manner, and finally obtain the expression $(A \times D) \times (B \times C)$.

To construct such a table one proceeds row by row, starting with sets of fewer relations and moving on to larger sets. Consider for example filling in the final row. The Cardinality for $\{A,B,C,D\}$ can be obtained by arbitrarily splitting $\{A,B,C,D\}$ into two nonempty subsets, say $\{A\}$ and $\{B,C,D\}$, and multiplying together the Cardinality fields associated with those subsets. To find the Best LHS of $\{A,B,C,D\}$, it is necessary to consider all possible splits of $\mathcal{S} = \{A,B,C,D\}$ into two nonempty subsets $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$, and to choose an $\mathcal{S}_{lhs}$ that minimizes $\text{Cost}(\mathcal{S}_{lhs}) + \text{Cost}(\mathcal{S}_{rhs}) + \bar{\kappa}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs})$. The minimum for this sum becomes the Cost field, in accordance with Equation (2).

Here $\bar{\kappa}$ is a variant form of the cost function κ appropriate to the dynamic programming context. The arguments to κ were relations or abstractions of relations, whereas each argument to $\bar{\kappa}$ is a *set* of relations that designates a table entry. For example, given our cost function $\kappa_0(R_{out}, R_{lhs}, R_{rhs}) = |R_{out}|$, the variant form becomes $\bar{\kappa}_0(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs}) = \text{Cardinality}(\mathcal{S})$, where $\text{Cardinality}(\mathcal{S})$ refers to the Cardinality field corresponding to Relation Set $\mathcal{S}$.

3.2 Algorithm *blitzsplit*

We now give our algorithm. The pseudo-code at the top of Figure 1 consists of two declarations. The first introduces a type *rel_data* that describes the information we need to know about the relations whose product is to be optimized. (With our simple cost model, we just need to know their cardinalities.) This declaration assumes that the type *rel_name* has been defined previously. The second declaration allocates a global variable *table*, which holds an array shaped like Table 1, with the Relation Set column acting as the array's index.

The pseudo-code following the declarations consists of procedures for filling in the array *table*. The top-level procedure *blitzsplit* takes as arguments a set $\mathcal{R}$ of relation names and an array *rel_data* containing information about the relations named in $\mathcal{R}$. The objective of *blitzsplit* is to find the least costly way of computing the Cartesian product of those relations.

The body of procedure *blitzsplit* consists of two **for**-loops. The first **for**-loop fills in the table entries for the singleton subsets of $\mathcal{R}$. The second **for**-loop (which has yet another **for**-loop nested inside of it) successively fills in the table entries for subsets of $\mathcal{R}$ consisting of 2 relation names, then for subsets consisting of 3 relation names, and so on. At the completion of these two loops, *table* has been entirely filled in.

The real work in filling in the table is done by the subprocedures *init_singleton*, *compute_properties*, and *find_best_split*. Procedure *find_best_split* deserves a word of explanation. Its role is to fill in the *best_lhs* and *cost* fields of table entries for non-singleton sets. To do so, it examines all splits of a given set $\mathcal{S}$ into pairs of nonempty subsets, and selects as the best split the pair that yields the lowest total cost. The left-hand component of that pair is recorded in the *best_lhs* field for $\mathcal{S}$; the corresponding cost is placed in the *cost* field for $\mathcal{S}$ when the loop completes. To reduce the effort needed to compute costs, we permit the cost function κ to be broken apart into a *split-independent* component κ' and a *split-dependent* component κ'', so that $\kappa(R_{out}, R_{lhs}, R_{rhs}) = \kappa'(R_{out}) + \kappa''(R_{out}, R_{lhs}, R_{rhs})$. Performance is best when the decomposition is such that κ'' is both cheap to compute and small in magnitude (we assume it is nonnegative).

```
type rel_data = array indexed by rel_name of
     record
         card : real
     end

var  table : array indexed by set[rel_name] of
     record
         card     : real
         best_lhs : set[rel_name]
         cost     : real
     end
```

```
proc blitzsplit(R : set[rel_name], rel_data : rel_data)
    for each R ∈ R do
        init_singleton(R, rel_data)
    end for
    for m := 2 to |R| do
        for each S ⊆ R such that |S| = m do
            compute_properties(S)
            find_best_split(S)
        end for
    end for
end proc

proc init_singleton(R : rel_name, rel_data : rel_data)
    table[{R}].card := rel_data[R].card
    table[{R}].best_lhs := ∅
    table[{R}].cost := 0.0
end proc

proc compute_properties(S : set[rel_name])
    choose U such that ∅ ⊊ U ⊊ S
    V := S − U
    table[S].card := table[U].card * table[V].card
end proc

proc find_best_split(S : set[rel_name])
    best_cost_so_far := ∞
    for each S_lhs such that ∅ ⊊ S_lhs ⊊ S do
        S_rhs := S − S_lhs
        oprnd_cost := table[S_lhs].cost + table[S_rhs].cost
        dpnd_cost := oprnd_cost + κ''(S, S_lhs, S_rhs)
        if dpnd_cost < best_cost_so_far then
            best_cost_so_far := dpnd_cost
            table[S].best_lhs := S_lhs
        end if
    end for
    table[S].cost := best_cost_so_far + κ'(S)
end proc
```

Figure 1: Algorithm *blitzsplit* for Cartesian product optimization

For example, the cost function $\kappa_0(R_{out}, R_{lhs}, R_{rhs}) = |R_{out}|$ can be decomposed into $\kappa'_0(R_{out}) = |R_{out}|$ and $\kappa''_0(R_{out}, R_{lhs}, R_{rhs}) = 0$.

3.3 Complexity

Here we consider first the space complexity, and then the time complexity of our algorithm. Let n be the number of relation names in $\mathcal{R}$. Then $\mathcal{R}$ has 2^n subsets, and there is an entry in *table* for each of these (except the empty set). Algorithm *blitzsplit* uses no other data structure of any significant size. Hence its space complexity is $O(2^n)$.

For the time complexity we shall give more than just a big-O analysis, since we are interested in the detailed performance characteristics of the algorithm. We assume that sets of relation names are of bounded size, and hence that primitives on them are constant-time operations; we also assume that per-iteration loop overheads are constant. One way to satisfy these assumptions is shown in Section 4 below.

Observe first that procedure *blitzsplit* makes n calls to *init_singleton*, and approximately 2^n calls to each of *compute_properties* and *find_best_split* (one call for each subset of $\mathcal{R}$). The net contribution of *init_singleton* is plainly insignificant, while the net contribution of the straightline code in *compute_properties* and *find_best_split* is $2^n T_{subset}$ for some constant T_{subset}.

There is, in addition, a loop in *find_best_split*. Consider the execution of *find_best_split* for some particular argument $\mathcal{S}$, and let $m = |\mathcal{S}|$. Then the loop in *find_best_split* iterates approximately 2^m times. Now consider all executions of *find_best_split*. The set $\mathcal{R}$ has $\binom{n}{m}$ subsets $\mathcal{S}$ with $|\mathcal{S}| = m$, for each m from 2 to n, hence the number of executions of *find_best_split* with $|\mathcal{S}| = m$ is also $\binom{n}{m}$. In aggregate, therefore, the number of iterations of the loop across all executions of *find_best_split* is $\sum_{m=2}^{n} \binom{n}{m} 2^m$; approximate this sum by $loop\_count = \sum_{m=0}^{n} \binom{n}{m} 2^m$. Next recall the binomial expansion $(a+b)^n = \sum_{m=0}^{n} \binom{n}{m} a^m b^{n-m}$. Taking $a = 2$ and $b = 1$, we obtain $(2+1)^n = \sum_{m=0}^{n} \binom{n}{m} 2^m \cdot 1^{n-m} = \sum_{m=0}^{n} \binom{n}{m} 2^m = loop\_count$. That is, $loop\_count = (2+1)^n = 3^n$. Thus, in aggregate, the time contributed by the loop in *find_best_split* is $3^n T_{loop}$ for some constant T_{loop}.

To assess the contribution of the conditionally executed code within the loop body, we use a statistical argument. Consider a particular execution of *find_best_split*. On each iteration through the loop, the **if** condition is satisfied only when the split under consideration improves upon the best so far. Assume the splits are examined in random order. Then the probability that the split considered on the ith iteration is better than the first $i-1$ is $1/i$, since any of the first i splits is equally likely to be the best among the i. Hence the expected number of executions of the conditionally

executed code is given approximately by the harmonic series $H_{2^m} = \sum_{i=1}^{2^m} 1/i$, where again $m = |\mathcal{S}|$. In aggregate, the number of executions of this code across all calls to *find_best_split* is $cond\_count = \sum_{m=2}^{n} \binom{n}{m} H_{2^m}$. Using the fact $H_k \approx \ln k + \gamma$ where $\gamma = 0.57721\ldots$ [Knu73], we obtain $cond\_count \approx \sum_{m=0}^{n} \binom{n}{m} m \ln 2 + \sum_{m=0}^{n} \binom{n}{m} \gamma = (\ln 2/2) n 2^n + \gamma 2^n$. Let us disregard the $\gamma 2^n$ term and view the net contribution of the conditionally executed code as $(\ln 2/2) n 2^n T_{cond}$ for some constant T_{cond}.

Then for some T_{loop}, T_{cond}, and T_{subset}, the execution time of the algorithm is closely approximated by

$$3^n T_{loop} + (\ln 2/2) n 2^n T_{cond} + 2^n T_{subset}. \qquad (3)$$

4 Lightweight Implementation

Algorithm *blitzsplit*'s $O(3^n)$ time complexity is somewhat lower than Starburst's $O(4^n)$, and its $O(2^n)$ space complexity is below Volcano's $O(3^n)$. However, when n is of modest size, constant factors can be just as important as complexity. It is therefore significant that, because of its simple structure, our algorithm can be implemented with very low overhead, resulting in small values for the time constants T_{loop}, T_{cond}, and T_{subset}.

Below we discuss first the representation of the algorithm's data, then the realization of critical details of the algorithm's procedures, and finally, performance measurements on an implementation that uses the techniques we describe.

4.1 Representation of Data Types

Let us refer to the relation names in $\mathcal{R}$ as R_0, R_1, ..., R_{n-1}. Then an implementation of the algorithm may as well identify these names by their integer indexes; R_i will be just i. Similarly, *sets* of relation names may be represented as bit-vectors in the obvious way, and (provided $n \leq 32$) may be held in 32-bit integers. This representation is not only compact, but also provides for extremely rapid execution of the set manipulations we require in the algorithm.[1]

One consequence of the compactness of the integer representation for sets is that each row of our dynamic programming table need occupy only 16 bytes: 8 bytes for the real *card*,[2] 4 bytes for the real *cost*, and 4 bytes for the bit-vector *best_lhs*. The $O(2^n)$ space complexity estimate given previously may now be refined to $16 \cdot 2^n$ bytes. Most modern workstations can accommodate this space requirement for n up to at least 20.

[1] Note that a given small integer can have two completely different interpretations as a relation name on the one hand, and as a set of relation names on the other hand. The integer 5 representing the relation name R_5 should not be confused with the integer 5 representing the set $\{R_0, R_2\}$.

[2] We sometimes require a wide dynamic range, as in the double-precision format of the IEEE floating-point standard.

4.2 Realization of Procedures

Several details of the pseudo-code of Figure 1 require special attention to achieve high efficiency in implementation. For example, the second loop in procedure *blitzsplit* need not be realized exactly as specified. The pseudo-code dictates that 2-relation sets be dealt with before 3-relation sets, and so forth; but maintaining this order is unimportant, so long as we stipulate that before constructing the table entry for a set $\mathcal{S}$, we must have previously constructed the table entries for all subsets of $\mathcal{S}$. The simplest way to satisfy this requirement is to process the sets in the order of their integer representations: first $\{R_0, R_1\}$ (represented by $\mathcal{S} = 3$), then $\{R_0, R_2\}$ ($\mathcal{S} = 5$), $\{R_1, R_2\}$ ($\mathcal{S} = 6$), $\{R_0, R_1, R_2\}$ ($\mathcal{S} = 7$), and so on. Thus, the loop in question might be rewritten as follows:

```
for k := 2 to |R| do
    for S := 2^(k-1) + 1 to 2^k - 1 do
        compute_properties(S)
        find_best_split(S)
    end for
end for
```

The variable $\mathcal{S}$ will successively take on all values up to $2^{|\mathcal{R}|}$ except those that are powers of two—those values represent singleton sets and must be skipped.

Next observe that efficient realization of the **for-each-such-that** loop in procedure *find_best_split* is critical to overall performance of the algorithm, since it is this loop that iterates 3^n times in aggregate. The pseudo-code for this loop calls for $\mathcal{S}_{lhs}$ to be bound successively to each nonempty, proper subset of $\mathcal{S}$. Then the realization of this loop, given an *integer* $\mathcal{S}$, must step through all the integers whose 1-bits are a nonempty, proper subset of the 1-bits in $\mathcal{S}$. Imagine a *dilation* operator $\delta_{\mathcal{S}}(i)$ on $|\mathcal{S}|$-bit binary representations that inserts 0-bits into its argument according the bit-pattern prescribed by $\mathcal{S}$. For example, if a, b, and c are bits, then $\delta_{11001}(abc) = ab00c$. Then inside the loop, $\mathcal{S}_{lhs}$ must take on the values $\delta_{\mathcal{S}}(1)$, $\delta_{\mathcal{S}}(2)$, ..., $\delta_{\mathcal{S}}(2^{|\mathcal{S}|} - 2)$.

Use of δ in an implementation would be cumbersome; it is preferable to define an operator $succ_{\mathcal{S}}$ such that $succ(\delta(i)) = \delta(i+1)$, which makes it possible to step from one $\mathcal{S}_{lhs}$ value to the next without evaluating δ. To that end, consider the *contraction* operator γ that is the left-inverse of δ; e.g., $\gamma_{11001}(abcde) = abe$. Assuming two's-complement arithmetic, observe that (with subscript $\mathcal{S}$ implicit)

$$\gamma(\delta(i) - \delta(j)) = i - j \qquad (4)$$

$$\delta(\gamma(w)) = \mathcal{S} \,\&\, w \qquad (5)$$

$$\delta(-1) = \mathcal{S}, \qquad (6)$$

where & denotes the bit-wise *and* operator. (Equation (4) is easiest to see by first considering examples

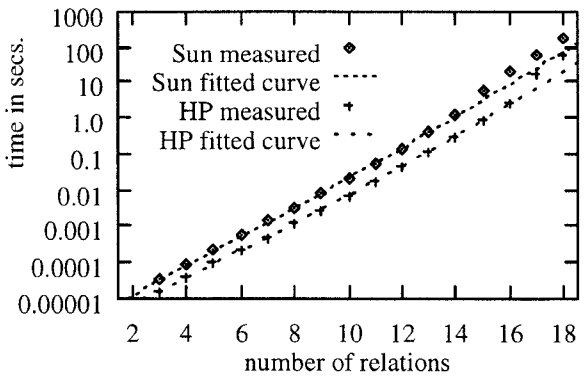


Figure 2: Cartesian product optimization times

where j has just one 1-bit; e.g., $\gamma_{11001}(\delta_{11001}(100) - \delta_{11001}(001)) = \gamma_{11001}(10000 - 00001) = \gamma_{11001}(01111) = 011$.) Now, using (4)–(6) we have $\delta(i+1) = \delta(i-(-1)) = \delta(\gamma(\delta(i) - \delta(-1))) = \mathcal{S} \& (\delta(i) - \delta(-1)) = \mathcal{S} \& (\delta(i) - \mathcal{S})$. In other words, we can obtain successive values of $\mathcal{S}_{lhs}$ through the simple computation $succ(\mathcal{S}_{lhs}) = \mathcal{S} \& (\mathcal{S}_{lhs} - \mathcal{S})$, without ever evaluating δ or γ.[3]

A final critical detail is that it is profitable to replace the **if** statement in *find_best_split* with a series of nested **if** statements. For example, computation of $\bar{\kappa}''$ can be predicated on the condition *oprnd_cost* < *best_cost_so_far*, since failure of this condition also implies *dpnd_cost* $\not<$ *best_cost_so_far*. The number of executions of $\bar{\kappa}''$ is then reduced from 3^n to some value intermediate between $(\ln 2/2)n2^n$ and 3^n. (Note that $\bar{\kappa}'$, being outside the loop, has a fixed execution count of just 2^n.)

4.3 Performance

Figure 2 shows typical performance of a Cartesian product optimizer based on the ideas above. The cost model is the naive model of Section 3.1; timings are reported for both a Sun SPARCstation 2 and a Hewlett-Packard Series 9000/755.[4] Formula (3) is fitted to the measured timings and tracks them closely until $n \approx 15$ (at which point cache effectiveness declines); we infer T_{loop} is about 180 nsec. on the Sun, and about 50 nsec. on the HP. Comparison of Figure 2 against exhaustive join-optimization timings show the Cartesian product timings to be lower apparently by several orders of magnitude. The immensity of this disparity motivates the approach to join-order optimization explored in the next section.

[3] One can equally easily visit the $\mathcal{S}_{lhs}$ in alternative orders—some of which may better conform to the randomness assumption of Section 3.3—by taking $succ(\delta(\imath)) = \delta(\imath + k)$ for arbitrary odd k. Iteration may begin with any valid $\mathcal{S}_{lhs}$, and proceed until that value is reached again (but 0 and $\mathcal{S}$ must be skipped over).

[4] Sun timings were taken on a lightly loaded Sun 4/75 running under SunOS 4.1.3_U1 Version B at circa 40MHz with a 64KB unified instruction + data cache; Hewlett-Packard timings were taken on a lightly loaded HP 9000/755 running under HP-UX 09.03 at 97MHz with 256KB each of instruction and data cache. Each timing point t represents an average over k executions of the algorithm, where k is such that $kt \geq 30$ seconds.

5 Join-order Optimization

We now build on the preceding results by observing that join-order optimization is essentially the same as Cartesian product optimization, except that intermediate-result cardinalities are computed differently. Here we address the computation of intermediate-result cardinalities in the presence of simple, uncorrelated join predicates. We present a technique that factors in the effects of predicate selectivities with minimal changes to our algorithm. Similar techniques can accommodate implied or redundant predicates and join hypergraphs, but we shall not discuss those topics here.

5.1 Join Graphs and Induced Subgraphs

Consider the join graph in Figure 3. Its nodes are labeled with the relation names A, B, C, and D; we will identify its edges as $\widehat{AB}$, $\widehat{AC}$, $\widehat{BC}$, and $\widehat{AD}$, and these names will also serve to identify the corresponding predicates. Following graph-theoretic convention, we may characterize the graph as an ordered pair $\mathbf{G} = (\mathcal{R}, \mathcal{P})$, where $\mathcal{R}$ is the node set $\{A, B, C, D\}$, and the edge set $\mathcal{P}$ is the set of predicate names $\{\widehat{AB}, \widehat{AC}, \widehat{BC}, \widehat{AD}\}$.

Now suppose we are interested in the cardinality that results from a join over the subset $\mathcal{S} = \{A, B, C\}$. Let $\mathcal{Q}$ be the set of edges *wholly contained* in $\mathcal{S}$ (i.e., those with both endpoints in $\mathcal{S}$)—namely $\{\widehat{AB}, \widehat{AC}, \widehat{BC}\}$. Then the subgraph $(\mathcal{S}, \mathcal{Q})$ of $\mathbf{G}$ is called the subgraph of $\mathbf{G}$ *induced by* $\mathcal{S}$. (See Figure 3.) In the course of a join of the relations named in $\mathcal{S}$, the predicates applied will be exactly those in the subgraph $(\mathcal{S}, \mathcal{Q})$: no more, because predicates not in $\mathcal{Q}$ cannot be evaluated with only the relations in $\mathcal{S}$; and no fewer, because there is no benefit in deferring the application of a predicate once its referent relations have become available.[5] It follows that the join cardinality of $\mathcal{S}$ can be computed by multiplying together the cardinalities and selectivities of the relations and predicates in the induced subgraph $(\mathcal{S}, \mathcal{Q})$.

5.2 Cardinality Recurrence

The observations of the foregoing paragraph give us the means to compute the join cardinality of $\mathcal{S}$. However, if we wish to take advantage of dynamic programming

[5] This assertion rests on the assumption that predicate evaluation is cheap; most other optimizers make the same assumption. However, Hellerstein and Stonebraker [HS93] describe a cost-based predicate-placement technique that achieves huge gains when expensive predicates are deferred.

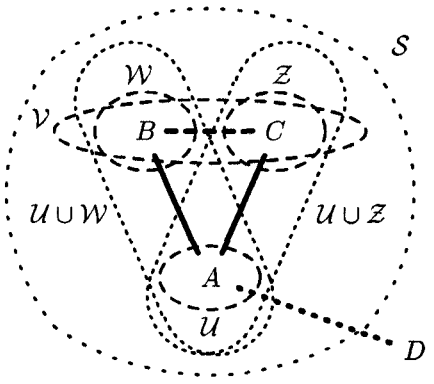


Figure 3: Subsets of relations in a join graph

to perform this computation more easily, we need a way to compute the join cardinality of $\mathcal{S}$ in terms of the join cardinalities of subsets of $\mathcal{S}$. What we desire is a *recurrence relation* for join cardinality.

Suppose we split $\mathcal{S}$ into two disjoint subsets $\mathcal{U}$ and $\mathcal{V}$, as illustrated in Figure 3. Then what was true for $\mathcal{S}$ must also be true for $\mathcal{U}$ and $\mathcal{V}$: The predicates applied in the course of a join over $\mathcal{U}$ will be just those in the subgraph induced by $\mathcal{U}$, and similarly for $\mathcal{V}$. Now let $\mathcal{U}^*$ denote the best expression for joining $\mathcal{U}$, and $\mathcal{V}^*$ the best for $\mathcal{V}$, and consider a join of $\mathcal{U}^*$ and $\mathcal{V}^*$. Since all relations in $\mathcal{S}$ participate in this join, all predicates wholly contained in $\mathcal{S}$ should also participate. But some of those predicates may also be wholly contained in $\mathcal{U}$ and therefore already participate in $\mathcal{U}^*$, and similarly for $\mathcal{V}$. It is the predicates that are left over—those that *span* $\mathcal{U}$ and $\mathcal{V}$—that must qualify the join of $\mathcal{U}^*$ and $\mathcal{V}^*$. In our example, the predicates spanning $\mathcal{U}$ and $\mathcal{V}$ are $\widehat{AB}$ and $\widehat{AC}$, so the correct expression for joining $\mathcal{U}^*$ and $\mathcal{V}^*$ is $\mathcal{U}^* \bowtie_{\widehat{AB} \wedge \widehat{AC}} \mathcal{V}^*$.

Hence, to compute the join cardinality of $\mathcal{S}$, we may multiply together the join cardinalities of $\mathcal{U}$ and $\mathcal{V}$, and the selectivities of all predicates spanning $\mathcal{U}$ and $\mathcal{V}$:

$$card(\mathcal{S}) = card(\mathcal{U}) \cdot card(\mathcal{V}) \cdot \Pi_{span}(\mathcal{U}, \mathcal{V}) \qquad (7)$$

$$\Pi_{span}(\mathcal{U}, \mathcal{V}) \stackrel{\text{def}}{=} \prod \{ selec(p) \mid p \text{ spans } \mathcal{U}, \mathcal{V} \}, \qquad (8)$$

where $\mathcal{U} \cap \mathcal{V} = \emptyset$ and $\mathcal{U} \cup \mathcal{V} = \mathcal{S}$. If $\mathcal{U}$ and $\mathcal{V}$ are both nonempty, then the values $card(\mathcal{U})$ and $card(\mathcal{V})$ will be readily available in our dynamic programming table when it comes time to compute $card(\mathcal{S})$. Consequently, recurrence (7) gives us the means to compute join cardinalities easily, provided we can compute $\Pi_{span}(\mathcal{U}, \mathcal{V})$.

5.3 Products of Selectivities

In the interest of efficiently computing $\Pi_{span}(\mathcal{U}, \mathcal{V})$, we introduce the notion of a *fan*. We will then be able to state a *second* recurrence suited to the dynamic programming context. Use of the two recurrences together will permit quick computation of intermediate-result cardinalities in the presence of predicates.

The fan concept requires that we have a total order on relation names. This order has nothing to do with cardinality or any other property of the relations—it is just an arbitrary ordering on the names. In our example above, let us say that the order is $A < B < C < D$, so that (for example) $\min\{B, C, D\} = B$.

Definition *Let $\mathcal{S}$ be a nonempty set of relation names; let $\mathcal{U} = \{\min \mathcal{S}\}$ and let $\mathcal{V} = \mathcal{S} - \mathcal{U}$. Then the* fan *of $\mathcal{S}$ is the set of predicates that span $\mathcal{U}$ and $\mathcal{V}$.*

For example, in Figure 3, $\mathcal{U} = \{\min \mathcal{S}\} = \{A\}$ and $\mathcal{V} = \mathcal{S} - \mathcal{U} = \{B, C\}$, so the fan of $\mathcal{S}$ is $\{\widehat{AB}, \widehat{AC}\}$. The name *fan* derives from the fact that since $\mathcal{U}$ is a singleton, the edges emanating from $\mathcal{U}$ toward the relation names in $\mathcal{V}$ resemble the spokes of a fan.

Given a set $\mathcal{S}$, we define $\Pi_{fan}(\mathcal{S})$ to be the product of the selectivities of the predicates in the fan of $\mathcal{S}$; i.e.,

$$\Pi_{fan}(\mathcal{S}) \stackrel{\text{def}}{=} \Pi_{span}(\{\min \mathcal{S}\}, \mathcal{S} - \{\min \mathcal{S}\}). \qquad (9)$$

We now construct a recurrence for $\Pi_{fan}(\mathcal{S})$. Suppose as before that $\mathcal{S} = \{A, B, C\}$ has been split into $\mathcal{U} = \{A\}$ and $\mathcal{V} = \{B, C\}$. Figure 3 illustrates how $\mathcal{V}$ may be further subdivided into subsets $\mathcal{W}$ and $\mathcal{Z}$. Consider the sets $\mathcal{U} \cup \mathcal{W}$ and $\mathcal{U} \cup \mathcal{Z}$. Since A is least in $\mathcal{S}$, A must also be least in each of the sets $\mathcal{U} \cup \mathcal{W}$ and $\mathcal{U} \cup \mathcal{Z}$. It follows that the predicates spanning $\mathcal{U}$ and $\mathcal{W}$ constitute the fan of $\mathcal{U} \cup \mathcal{W}$, and those spanning $\mathcal{U}$ and $\mathcal{Z}$ constitute the fan of $\mathcal{U} \cup \mathcal{Z}$. Moreover, these fans are disjoint (since $\mathcal{W}$ and $\mathcal{Z}$ are disjoint) and their union is the fan of $\mathcal{S}$ (since $\mathcal{W} \cup \mathcal{Z} = \mathcal{V}$ and $\mathcal{U} \cup \mathcal{V} = \mathcal{S}$). From these facts we deduce the recurrence

$$\Pi_{fan}(\mathcal{S}) = \Pi_{fan}(\mathcal{U} \cup \mathcal{W}) \cdot \Pi_{fan}(\mathcal{U} \cup \mathcal{Z}), \qquad (10)$$

where $\mathcal{U} = \{\min \mathcal{S}\}$, $\mathcal{W} \cap \mathcal{Z} = \emptyset$, and $\mathcal{W} \cup \mathcal{Z} = \mathcal{S} - \mathcal{U}$. In the figure, both $\mathcal{W}$ and $\mathcal{Z}$ are singletons, but in general they need not be; (10) holds for any split of $\mathcal{V}$ into disjoint $\mathcal{W}$ and $\mathcal{Z}$. One may visualize the more general case by interpreting the solid lines in Figure 3 not as individual predicates, but as *bundles* of predicates connecting $\mathcal{U}$ to $\mathcal{W}$ and $\mathcal{U}$ to $\mathcal{Z}$.

Recurrence (10) gives an efficient way to compute $\Pi_{fan}(\mathcal{S})$. We can now efficiently compute $card(\mathcal{S})$ as well: Combining (7) and (9), we have

$$card(\mathcal{S}) = card(\mathcal{U}) \cdot card(\mathcal{V}) \cdot \Pi_{fan}(\mathcal{S}), \qquad (11)$$

provided $\mathcal{U} = \{\min \mathcal{S}\}$ and $\mathcal{V} = \mathcal{S} - \mathcal{U}$.

5.4 Implementation

To put the ideas of this section into practice, we must first add a column for Π_{fan} to our dynamic

programming table. Then procedure *blitzsplit* must be revised to initialize the Π_{fan} entries for all doubleton sets: $\Pi_{fan}(\{R, R'\})$ must be set to the selectivity of the predicate connecting R and R' (or to 1 if there is no such predicate). Finally, *init_singleton* and *compute_properties* must be revised; in the latter, $\Pi_{fan}(\mathcal{S})$ is computed using (10), and then $card(\mathcal{S})$ is computed using (11). (Note that $\mathcal{U} = \{\min \mathcal{S}\}$ can be computed as $\delta_{\mathcal{S}}(1) = \mathcal{S} \,\&\, -\mathcal{S}$.) No changes are required to *find_best_split*. Our code for Algorithm *blitzsplit*, including the revisions sketched here, amounts to about 150 lines of C.

Support for uncorrelated predicates has only a modest effect on the time and space requirements of our algorithm. The dynamic programming table needs to grow by just one column, and the bulk of the additional computation occurs in *compute_properties*, which is executed just 2^n times. The original *compute_properties* contained one floating multiplication operation; with the revisions, it contains three (one to compute Π_{fan} and two to compute *card*), and these are all that is needed to incorporate predicate selectivities into the cardinality computations, regardless of the join graph. As will be seen in the next section, these selectivity computations do not significantly inflate the performance figures reported in Section 4.

More sophisticated cardinality-estimation schemes will increase both time and space requirements. However, the technique of using recurrences to achieve economies in property computations remains applicable in situations more general than the one illustrated here; it should continue to be possible to achieve these computations in time $O(2^n)$. Under no circumstances should changes in *find_best_split* be necessary to carry out cardinality estimation or other property computations.

6 Performance

Assessment of a join optimizer's performance is a nontrivial problem—especially comparative assessment against other methods. There is no standard set of benchmarks for join-order optimization, and the performance results reported in different papers are often difficult to compare with one another.

The root of the difficulty is that the space of possible test cases has a huge number of dimensions. In addition to the n base-relation cardinalities, each of which may be varied independently, there are up to $n(n-1)/2$ predicate selectivities, depending on the join-graph topology (which may be considered a variable in its own right); the cost model is yet another variable. If n is fixed at, say, 15, then the space of test cases has well over 100 dimensions. In choosing sample points from such a space, one hardly knows where to begin. Some assessments deal with the problem by reporting average performance over a mix of randomly selected test cases, in some instances reporting separate averages for each of several distinct join-graph topologies and cost models. But the use of random mixes introduces noise into the results, and obscures the patterns of variation in performance in different regions of the input space. In particular, some random mixes will fail to probe the more treacherous regions of the space.

Here we seek to determine the ways in which various characteristics of the input—not just the join-graph topology and cost model—influence optimization time under our algorithm. To that end, we generate all our test cases deterministically. Our performance analysis is a coarse first attempt and is subject to future refinement, but even this first attempt gives a fairly clear picture of the fundamental performance traits of our algorithm. We believe it also demonstrates the feasibility of systematically sampling a many-dimensional performance surface, provided the surface exhibits some regularity.

6.1 Empirical Measurements

Prior to running the measurements reported below, we conducted exploratory probing of the input space with the aim of reducing the number of independent dimensions needed to characterize performance. These initial probes led to the following observations, which provided the basis for our choice of test parameters. (For a full account of the test parameters, see the Appendix.)

- The *geometric mean* of the base-relation cardinalities is important in determining optimization time; the individual cardinalities are far less so. This observation, which is supported by the graphs below, justifies collapsing the n degrees of freedom represented by the base-relation cardinalities to two degrees of freedom—the geometric mean and the *variability* of the base-relation cardinalities.

- Large changes in predicate selectivities are tantamount to alteration of the join-graph topology, but moderate variation of the selectivities has only a mild effect on optimization time. For a fixed topology, the worst-case optimization time appears to arise when selectivities are allocated so as to minimize variability in the intermediate-result cardinalities. Our tests use four diverse topologies: *chain*, *cycle* augmented with three cross-edges ("*cycle*+3"), *star*, and *clique*. Preferring to err on the side of conservatism, we assign what we believe to be near-worst-case selectivities for each topology.

- Performance of the algorithm is very sensitive to the cost model; especially important is the question of whether κ has a significant κ'' component. Our tests use three cost models with different characteristics:

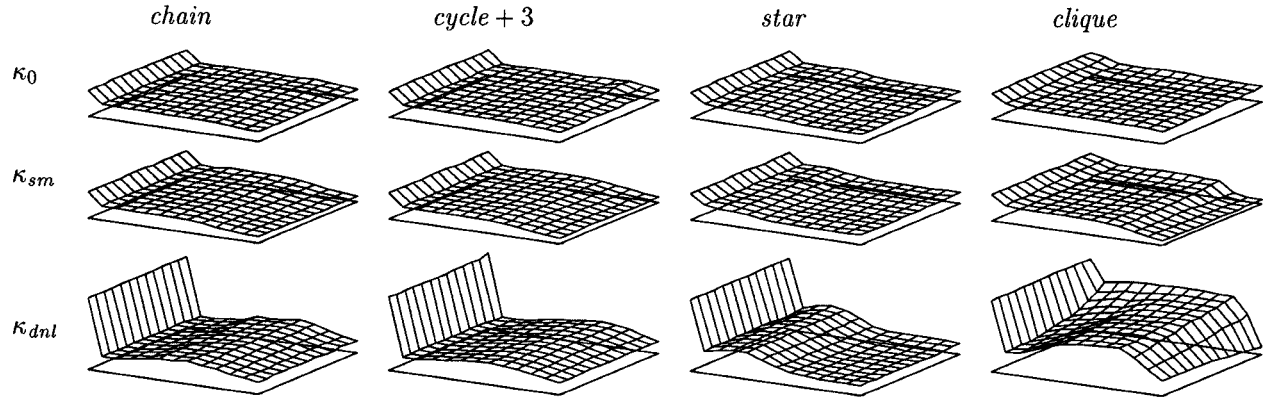


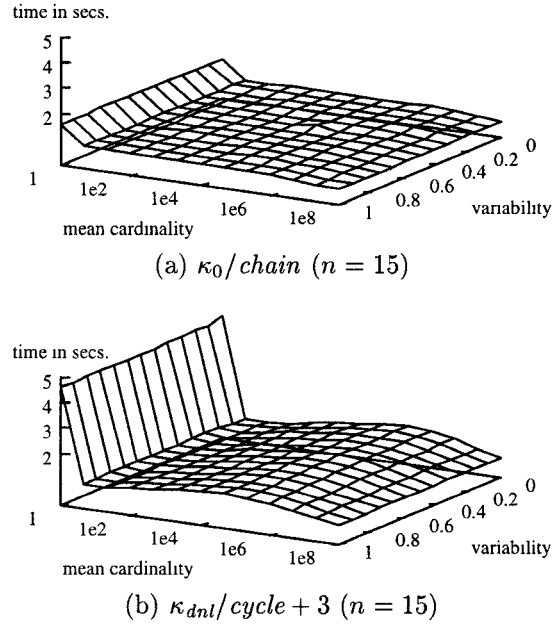
Figure 4: Four-dimensional summary of performance sensitivities ($n = 15$)

the naive model introduced in Section 3.1 (κ_0), a sort-merge cost model (κ_{sm}), and a disk-nested-loops model (κ_{dnl}).

- The observations above hold regardless of the choice of n; the graphs shown here for $n = 15$ are representative of the appearance of the corresponding graphs across a wide range of n-values.

In light of these observations, one can obtain an approximate characterization of our algorithm's performance by examining its behavior on a small subspace of the space of all possible inputs.

The array of graphs in Figure 4 presents HP 9000/755 timings taken over a 4-dimensional space with $n = 15$. The rows and columns of the array represent two axes of the space, a cost-model axis and a join-graph-topology axis; and within each cell inside the array, two more axes are represented—a long axis for mean base-relation cardinality, and a short axis for the variability among the base-relation cardinalities. Moving left-to-right along the long axis corresponds to increasing mean base-relation cardinality, and moving back-to-front along the short axis corresponds to increasing variability among the base-relation cardinalities. (The Appendix gives details.) Figure 5 shows two of the array cells in enlarged form, with labeled axes to give a sense of scale. Note that the vertical axis represents *optimization time* (and not plan cost).



(a) $\kappa_0/chain$ ($n = 15$)

(b) $\kappa_{dnl}/cycle + 3$ ($n = 15$)

Figure 5: Optimization times (close-ups of Figure 4)

6.2 General Performance Traits

Figures 4 and 5 show that under the naive cost model, 15-way joins are optimized in times comparable to those we obtained for 15-way Cartesian products in Section 4.3. On the HP, the latter were typically optimized in about 0.9 seconds; here it is harder to say what is "typical," but optimization time under the naive model rarely falls outside the range 0.6–1.1 seconds. Incorporating predicates appears to make the execution time of Algorithm *blitzsplit* more variable, but not necessarily greater.

The chaise-longue-like shapes in the figures reflect two basic performance properties of our algorithm. First, performance degrades (sometimes dramatically) as the mean cardinality of the base relations approaches 1; but mean cardinality does not have to be large to escape this effect.[6] Second, performance is substantially affected by the cost model, but the performance differences

[6]The mean-cardinality axis in all the figures is logarithmic, and the sample points are taken at mean cardinalities 1, 4.64, 21.5, 100, 464, etc. Performance at mean cardinality 4.64 is seen to be comparable to that for much larger cardinalities.

diminish as mean cardinality increases (and also, in the case of cliques, as variability increases).

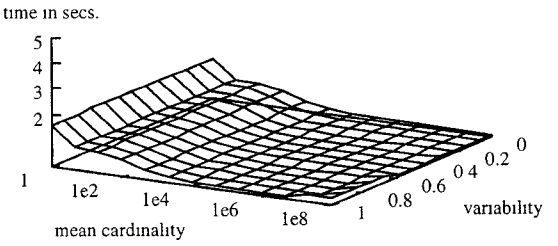
The explanation for the observed behavior is as follows. The nested **if** structure introduced in Section 4.2 makes execution of $\bar{\kappa}''$ conditional, and results in $\bar{\kappa}''$ having an execution count intermediate between $(\ln 2/2)n2^n$ and 3^n. Whether the count is closer to $(\ln 2/2)n2^n$ or 3^n depends on whether the costs of alternative subplans are spaced far apart or close together. When these costs are spaced far apart, as tends to be the case when cardinalities are large, the loop in *find_best_split* can often dismiss an uncompetitive split with a cursory glance at the operand costs alone, and so avoid computing $\bar{\kappa}''$ most of the time—giving a count approaching $(\ln 2/2)n2^n$. There is less opportunity for such savings when the costs of alternative splits are closely spaced, as tends to occur with very low cardinalities; in such situations, $\bar{\kappa}''$ may be computed fully 3^n times. Thus, the performance degradation caused by computation-intensive cost models is disproportionately large at low cardinalities. (We see some degradation at low cardinalities even for κ_0, because even our κ_0 implementation uses nested **if**'s: Computation of *oprnd_cost* is conditional on both $\text{Cost}(\mathcal{S}_{lhs})$ and $\text{Cost}(\mathcal{S}_{rhs})$ being below *best_cost_so_far*.)

Note that if the search were restricted to left-deep plans, the execution count for $\bar{\kappa}''$ would lie between about $(\ln n)2^n$ and $(n/2)2^n$ (for brevity we omit the derivation); the execution count for $\bar{\kappa}'$ would remain fixed at about 2^n. In the worst case, then, bushy search does far more work; but ordinarily, the $\bar{\kappa}''$ execution count is larger for bushy than for left-deep search by only a factor of $(\ln 2/2)n/\ln n$ (about 2 when $n = 15$).

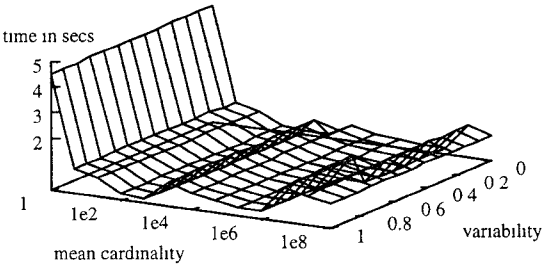
6.3 Effect of Join-graph Topology

From our algorithm's point of view, all join graphs are actually cliques, and are distinguished only by the selectivities associated with the predicates in these cliques. The join-graph topology is reflected in the variability in these selectivities, which affects the frequency with which the cost function $\bar{\kappa}''$ must be either partially or fully evaluated in the loop in *find_best_split*. More highly connected graphs tend to have an equalizing effect on the costs of alternative splits, especially when there is not much variability among the base-relation cardinalities. The clique is the extreme case; its total connectivity accounts for its elevated timings. Star queries also suffer relative to chain queries because they tend to yield a larger proportion of plans with competitive costs.

However, optimization time drops off regardless of join-graph topology when the mean base-relation cardinality is sufficiently large. The good performance at very large cardinalities can be partly explained in terms of variability in the intermediate-result cardinalities, but is also partly an artifact of implementation. We represent costs as single-precision floating-point values,



(a) $\kappa_0/chain$; threshold at 10^9 ($n = 15$)



(b) $\kappa_{dnl}/cycle + 3$; thresholds at 10^5, 10^{14} ($n = 15$)

Figure 6: Optimization times with plan-cost thresholds

and summarily reject plans whose cost overflows.[7] Our code for *find_best_split* computes $\bar{\kappa}'(\mathcal{S})$ *before* the loop, and on overflow avoids the loop entirely. When the base relations have large cardinalities, $\bar{\kappa}'(\mathcal{S})$ tends to overflow for many $\mathcal{S}$, and effort expended in *find_best_split* drops off.[8]

6.4 Pruning by Plan-Cost Thresholds

The beneficial consequences of overflow suggest an improvement to our implementation, namely to simulate the effect of overflow at a plan-cost threshold far below actual overflow. Best-split searches can then be avoided for a larger proportion of subsets $\mathcal{S}$. In those cases where no plan exists with cost below the threshold, optimization fails, and it is then necessary to reoptimize with a higher threshold. The net effect is that queries for which a low-cost plan exists are optimized more quickly, at the expense of queries for which the best plan has a high cost. Queries of the latter variety may require two or more passes through the optimizer before a plan is found; but since these queries are expected to be long-running at execution time, the extra investment they require at optimization time is not onerous.

Compare the graphs of Figure 6, which show optimization times with plan-cost thresholds, against the

[7] If cost is measured in, say, femtoseconds, then overflow occurs only in plans that would run for more than 10^{38} femtoseconds, or 3.2×10^{15} years. It is therefore unlikely than any useful plans could be overlooked because of overflow.

[8] If $\kappa' \equiv 0$, this effect will not occur. However, in a realistic cost model there is some cost for each output tuple (giving $\kappa' \not\equiv 0$). Even if small, such a cost suffices to induce the noted effect.

corresponding graphs of Figure 5. Figure 6(a), with cost model κ_0, a *chain* join-graph topology, and a plan-cost threshold of 10^9, exhibits optimization times that settle down to a scant 0.1 second; Figure 6(b), with cost model κ_{dnl}, topology *cycle*+3, and plan-cost thresholds of 10^5 and 10^{14}, also shows optimization times dropping off as cardinality rises, but then ripples appear where the plan-cost thresholds are exceeded, forcing multiple optimization passes at higher cardinalities.[9]

Plan-cost thresholds are effective for all join-graph topologies, but the benefits are most pronounced for chain-like graphs. With such graphs, and with plan-cost thresholds in place, the loop in *find_best_split* may be executed for only a tiny fraction of the 2^n sets of relations passed as arguments to *find_best_split*. As mean cardinality increases, the total number of executions of $\bar{\kappa}''$ tends to fall below $n^3/3$, reflecting the intrinsic polynomial complexity of chain-query optimization.[10]

6.5 Discussion

The foregoing analysis does not provide sufficient information to predict accurately the performance of our algorithm on arbitrarily chosen join-order optimization problems. But it does reveal the importance of base-relation cardinalities (and to a lesser degree, their variability) in determining the difficulty of optimization by our method. The importance of join-graph topology has been noted by others, and shows itself here as well. Further study will be needed to give a more complete description of the ways in which our method is sensitive to the characteristics of the input.

Our favorable timings are due partly to the use of relatively simple cost models. More complicated models will degrade performance as the time devoted to the computation of $\bar{\kappa}''$ increases. The question then becomes whether the degradation is more severe for our method than for methods that exhaustively search a more restricted space; our observations regarding the execution counts for $\bar{\kappa}''$ suggest that it may not be. Cost computation may be *proportionately* more of a performance limit for our method than for others simply because we spend so little time on join-order enumeration.

Related to the issue of complicated cost models is the issue of multiple join algorithms. Dealing with the latter may be regarded as a special case of dealing with the former. For example, if both a sort-merge join and disk-nested-loops join are available, then the cost of a join is $\kappa(\ldots) = \min(\kappa_{sm}(\ldots), \kappa_{dnl}(\ldots))$. There is no need to keep track of which algorithm yields the minimum. On completion of optimization, a single traversal of the optimal plan suffices to attach the appropriate algorithm to each join node.

The issue of physical properties (e.g., "interesting" sort orders [SAC+79]) is trickier. Although we have a plausible strategy for accommodating physical properties in special cases, we have yet to develop a strategy for the general case.

7 Conclusions and Future Work

Two factors largely account for our algorithm's ability to search quickly through a large, unconstrained space of join orders: First, we generate the join orders themselves extremely rapidly, so that little time is lost on activities other than costing of plans. Second, we capitalize on the fact that little costing effort is required to reject exorbitantly expensive plans. Although our algorithm performs no explicit join-graph analysis, in a sense it "discovers" the join-graph topology and prunes the search space accordingly. It quickly dismisses Cartesian products that are obviously wasteful, while retaining optimal ones; to exclude Cartesian products *a priori* would be redundant at best, and potentially harmful. Confining the search to left-deep plans could also harm plan quality, and would likely offer only modest optimization-time savings for $n \leq 15$.

We will continue to investigate the performance characteristics and adaptability of our algorithm. But already it is apparent that optimization by our method is—somewhat perversely—most expensive in situations where finding a true optimum matters least. That is, when all plans are close in cost, and any plan would serve nearly as well as the others, the optimizer requires more effort to choose among them than when the costs are widely scattered. In addition, like any optimizer that performs exhaustive search, ours is limited in the number of relations it can handle in reasonable space and time. Stochastic methods offer a way around both these problems. We are currently experimenting with a hybrid method inspired by the Chained Local Optimization technique of Martin and Otto [MO]. Our hybrid combines dynamic programming with randomized search, and will be the subject of a future paper.

Acknowledgments We are grateful for helpful feedback from Roberto Bayardo, Goetz Graefe, Joe Hellerstein, Bala Iyer, Guy Lohman, Guido Moerkotte, Len Shapiro, and the referees. Michael Steinbrunn graciously provided a preprint of Chapter 4 of his book.

Support for this work was provided in part by the Advanced Research Projects Agency, ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518, and by NSF grant IRI 91 18360.

[9] The units of our cost models are left unspecified, but it is reasonable to suppose that the thresholds 10^9 for κ_0 and 10^5 for κ_{dnl} represent plan-execution times of many minutes.

[10] However, our algorithm's complexity remains exponential even under these conditions, since plan-cost pruning has no effect on the $2^n T_{subset}$ term in the execution time.

References

[CM95] Sophie Cluet and Guido Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proceedings of the 5th International Conference on Database Theory*, volume 893 of *Lecture Notes in Computer Science*, pages 54–67, Prague, Czech Republic, January 1995. Springer-Verlag.

[GLPK94] César Galindo-Legaria, Arjan Pellenkoft, and Martin Kersten. Fast, randomized join-order selection—why use transformations? In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 85–95, Santiago, Chile, September 1994. Morgan Kaufmann.

[GM93] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE Conference on Data Engineering*, pages 209–218, Vienna, Austria, April 1993.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 267–276, Washington, DC, May 1993.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[IK91] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 168–177, Denver, Colorado, June 1991.

[Knu73] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1973.

[MO] O. Martin and S. Otto. Combining simulated annealing with local search heuristics. To appear as a chapter of *Metaheuristics in Combinatorial Optimization*, volume 60 in the series *Annals of Operations Research*, edited by G. Laporte and I. Osman.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 314–325, Brisbane, Australia, August 1990.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM-SIGMOD 1979 International Conference on Management of Data*, pages 23–34, Boston, Massachusetts, May 1979.

[SMK93] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Optimizing join orders. Technical Report MIP-9307, Universität Passau, 1993.

[Ste96] M. Steinbrunn. *Heuristic and Randomised Optimisation Techniques in Object-Oriented Database Systems*. infix-Verlag, Ringstraße 32, 53757 St. Augustin, Germany, 1996. Dissertation, Universität Passau.

Appendix

This appendix gives details on each of the parameters that were varied in the measurements reported in Section 6. In all measurements, the number of base relations n was held fixed at 15; below we refer to these relations as R_0–R_{14}, with R_0 assuming the lowest cardinality, and R_{14} the highest.

Cost model Our cost models are drawn from the study by Steinbrunn et al. [SMK93]. The naive cost model (cf. Section 3.1) is defined by $\kappa_0(R_{out}, R_{lhs}, R_{rhs}) = |R_{out}|$. The sort-merge cost model is defined by $\kappa_{sm}(R_{out}, R_{lhs}, R_{rhs}) = |R_{lhs}| \cdot (1 + \log|R_{lhs}|) + |R_{rhs}| \cdot (1 + \log|R_{rhs}|)$; note that the expensive logarithm computation in this model can be memoized in the dynamic programming table.

We give the disk-nested-loops model a different formulation from Steinbrunn et al.: Here, $\kappa_{dnl}(R_{out}, R_{lhs}, R_{rhs}) = 2 \cdot |R_{out}|/K + |R_{lhs}| \cdot |R_{rhs}|/K^2(M-1) + \min(|R_{lhs}|, |R_{rhs}|)/K$, where K is the blocking factor of relation records per disk block (we make the simplifying assumption that K is a constant), and M is the number of disk blocks that can be held in main memory. In our measurements, we arbitrarily set $K = 10$ and $M = 100$. (In separate tests, we have found that actually computing the relation widths and blocking factors, rather than taking K to be constant, has little effect on the performance graphs.)

Join graph Our chain graphs have the following predicate connections: R_0–R_8–R_1–R_9–R_2–R_{10}–R_3–R_{11}–R_4–R_{12}–R_5–R_{13}–R_6–R_{14}–R_7. (We have tried chains with alternative cardinality orderings and have obtained essentially the same results.) The "*cycle* + 3" topology augments a chain with predicate connections at R_0–R_7, R_8–R_{14}, R_1–R_6, and R_9–R_{13}. Star graphs have predicate connections between the hub R_{14} and each other relation. (R_0 as hub gives similar results.) Cliques have predicate connections between every pair of relations.

In all graphs, the selectivity of the predicate (if any) connecting R_i and R_j is computed as $\mu^{1/k} \cdot |R_i|^{-1/k_i} \cdot |R_j|^{-1/k_j}$, where μ is the mean base-relation cardinality, k is the total number of predicates, and k_i is the number of predicates incident on R_i. Note that these selectivities yield a query result cardinality of μ.

Mean cardinality Mean base-relation cardinality is the geometric mean $(\prod_{i=0}^{14} |R_i|)^{1/15}$.

Variability Variability ranges from 0 to 1, with 0 indicating that all $|R_i|$ are equal. In general, $|R_0| = (\textit{mean cardinality})^{1-\textit{variability}}$, and the remaining R_i are such that $|R_i|/|R_{i-1}|$ is constant.