# Estimating Compilation Time of a Query Optimizer

Ihab F. Ilyas
Purdue University
ilyas@cs.purdue.edu

Jun Rao
IBM Almaden Research Center
junrao@almaden.ibm.com

Guy Lohman
IBM Almaden Research Center
lohman@almaden.ibm.com

Dengfeng Gao
University of Arizona
dgao@cs.arizona.edu

Eileen Lin
IBM Silicon Valley Laboratory
etlin@us.ibm.com

## ABSTRACT

A query optimizer compares alternative plans in its search space to find the best plan for a given query. Depending on the search space and the enumeration algorithm, optimizers vary in their compilation time and the quality of the execution plan they can generate. This paper describes a compilation time estimator that provides a quantified estimate of the optimizer compilation time for a given query. Such an estimator is useful for automatically choosing the right level of optimization in commercial database systems. In addition, compilation time estimates can be quite helpful for mid-query reoptimization, for monitoring the progress of workload analysis tools where a large number queries need to be compiled (but not executed), and for judicious design and tuning of an optimizer.

Previous attempts to estimate optimizer compilation complexity used the number of possible binary joins as the metric and overlooked the fact that each join often translates into a different number of join plans because of the presence of "physical" properties. We use the number of plans (instead of joins) to estimate query compilation time, and employ two novel ideas: (1) reusing an optimizer's join enumerator to obtain actual number of joins, but bypassing plan generation to save estimation overhead; (2) maintaining a small number of "interesting" properties to facilitate plan counting. We prototyped our approach in a commercial database system and our experimental results show that we can achieve good compilation time estimates (less than 30% error, on average) for complex real queries, using a small fraction (within 3%) of the actual compilation time.

## 1. INTRODUCTION

A query optimizer in a database system translates a non-procedural query into a procedural plan for execution, typically by generating many alternative plans, estimating the execution cost of each, and choosing the plan having the lowest estimated cost. The complexity of an optimizer is mainly determined by a *search space* and an *enumeration algorithm* enumerating plans in the search space. In general, increasing the search space of an optimizer improves the chances–but does not guarantee–that it will find a better plan, while increasing the cost (compilation time) for optimizing the query. In this paper, we study the problem of estimating quantitatively how long it takes an optimizer to compile a query. We first motivate the importance of this problem in Section 1.1 and then outline our solution to the problem in Section 1.2.


*SIGMOD* 2003, June 9-12, 2003, San Diego, CA.

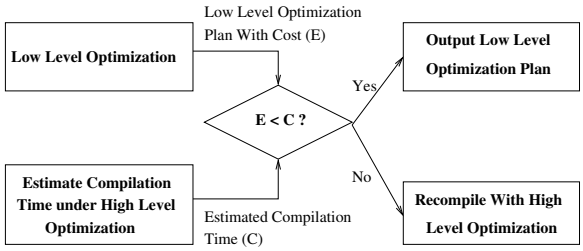




**Figure 1: A Simple Architecture of a Meta-optimizer**

### 1.1 Motivation

A major challenge in the design of a query optimizer is to ensure that the set of feasible plans in the search space contains efficient plans without making the set too big to be generated practically. For that purpose, most commercial database systems often have multiple levels of optimization. For example, a system can have a "low" level of optimization that employs a polynomial-time greedy method or a randomized algorithm, and a "high" level that searches all bushy plans using a conventional dynamic programming enumeration method [20]. "Knobs" within an optimizer, such as limits on the composite inner size and whether Cartesian products are allowed or not, essentially create many additional "intermediate" optimization levels. The higher the optimization level, the better the chance of getting a good execution plan, but the longer the compilation time. Currently, database administrators must decide what the right optimization level is by trying to trade off the estimated compilation time against possible improvements

in execution time. To automate such decisions, we will need to design a *meta*-optimizer (*MOP*).

Just as we estimate plan execution costs in a query optimizer, an essential component in a *MOP* is a compilation time estimator (*COTE*). Figure 1 describes how such an estimator can be used in a simple *MOP* to choose between two levels of optimization. *MOP* first compiles the query at the low level and obtains an estimate (measured by time) of the execution cost (call it $E$) of the best plan it finds. It then calls the *COTE* to obtain the compilation time estimate (call it $C$) of the query at the high level. *MOP* then compares $E$ with $C$ to decide whether to reoptimize the query at the high level. For example, if $C$ is larger than $E$, then there is no point in further optimization since the query can complete execution by the time high-level optimization finishes. It's possible that a query can take longer to compile than to execute, especially when the query is complex yet very selective, or when users only want to see the top $n$ rows. On the other hand, if $C$ is much smaller than $E$, reoptimizing the query at the high level gives the potential of further reducing $E$ with only a relatively small cost ($C$). A more advanced meta-optimizer can be built by exploiting additional information such as whether a query is static or dynamic (it's worthwhile to spend more time optimizing a static query since the query is likely to be executed repeatedly) and an estimate of potential gains in plan quality for each optimization level.

A *COTE* can be used for many other applications. It is useful in evaluating the need for mid-query reoptimization [15], in which an optimizer tries to generate a new plan in the middle of execution if a significant cardinality discrepancy is discovered. Since reoptimization itself takes time, the decision on whether to reoptimize or not is better made by comparing the execution cost of the remaining work with the estimated time to recompile.

Estimating the compilation time is also very useful for workload analysis tools. Examples of these tools are advisors for indexes [4, 22], materialized views [1], and partitioning [18] that have been built on top of commercial database systems. All these tools spend most of their time compiling (but not executing) a large number of queries in the input workload as part of their tuning analysis, and may run for hours or even days, depending on the workload. A *COTE* can be used to forecast how long such a tool will take to finish and possibly to show the progress of the tool as well.

Last, but not least, our study of compilation time discloses what the "real" cost of compiling a query is and where the time goes during optimization. Our results are useful to database builders for evaluating different heuristic rules for cutting down optimizer's search space.

## 1.2 Solution

One straightforward approach to estimating the compilation time is through to cache the compilation time for each compiled query in a statement cache and use it as an estimate for subsequent similar queries. However, this approach may not work well for a variety of complex ad-hoc queries, which are the focus of this paper.

Compilation time estimation has been studied to a certain extent, but not comprehensively in the literature. Most of the studies relate optimization complexity to the number of possible binary joins [17] (i.e., the compilation time should be proportional to the number of joins considered by the optimizer) and then try to efficiently count the number of joins. While our study confirms that join optimization dominates optimization cost, there are two major problems with this approach. First of all, counting the number of joins is hard for a general class of queries with cycles in the join graph. Second, all existing studies ignored the presence of "physical" properties (such as the order property) kept by an optimizer, which could disproportionally affect the time spent on optimizing each join. We will discuss these two problems in more detail in Section 2.2.

This paper shows that a better way to estimate optimization complexity is by estimating the number of generated join plans, rather than the number of joins. To do this for any possible query, we exploit the join enumerator in an optimizer to iterate all the joins and simply bypass plan generation. We accumulate a small number of relevant properties during enumeration to calculate the number of generated plans for each enumerated join. To validate our approach, we prototyped our methods in IBM Universal Database for Linux, Unix and Windows [6] (referred to simply as `DB2` in the rest of the paper). Our experimental results show that this approach provides compilation time estimates with an average error of 30% (much lower in many cases), using less than 3% of the actual compilation time.

For expository purposes, we focus our discussions on a conventional dynamic programming style join enumerator because it is widely used in both commercial and research database systems. However, this approach is applicable to many other kinds of enumeration as well.

The remainder of the paper is organized as follows. Section 2 revisits the dynamic programming algorithm and describes the limitations of previous attempts to estimate optimization complexity. We present our general approach in Section 3 and our experience of prototyping on `DB2` in Section 4, respectively. Section 5 summarizes our experimental results on a wide range of workloads. We discuss other related work and potential extensions to our work in Section 6, and conclude in Section 7.

# 2. BACKGROUND AND PREVIOUS WORK

In this section, we first give an overview of how dynamic programming search algorithm works, and then explain previous attempts to estimate optimization complexity and why they are not sufficient.

## 2.1 Dynamic Programming Revisited

Since the join operation is implemented in most systems as a diadic (2-way) operator, the optimizer must generate plans that transform an n-way join into a sequence of 2-way joins using binary join operators. The two most important tasks of an optimizer are to find the optimal join sequence as well as the optimal join method for each binary join. Dynamic programming (*DP*) was first used for join enumeration in System R [20]. The essence of the *DP* approach is based on the assumption that the cost model satisfies the *principle of optimality*, i.e., the subplans of an optimal plan must be optimal themselves. Therefore, in order to obtain an optimal plan for a query joining $n$ tables, it suffices to consider only the optimal plans for all pairs of non-overlapping $m$ tables and $n-m$ tables, for $m = 1, 2, ..., n-1$.

To avoid generating redundant plans, *DP* maintains a memory-resident structure (referred to as *MEMO*, following the terminology used in [10]) for holding non-pruned

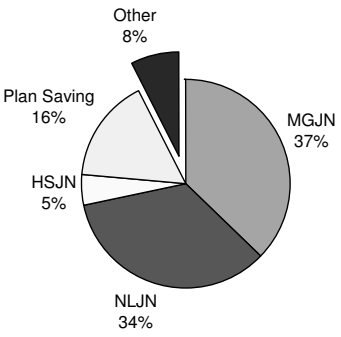



**Figure 2: Compilation Time Breakdown for a Customer Workload on DB2**

plans. Each MEMO entry corresponds to a subset of the tables (and applicable predicates) in the query. The algorithm runs in a bottom-up fashion by first generating plans for single tables. It then enumerates joins of two tables, then three tables, etc., until all $n$ tables are joined. For each join it considers, the algorithm generates join plans and incorporates them into the plan list of the corresponding MEMO entry. Plans with larger table sets are built from plans with smaller table sets. The algorithm prunes a higher cost plan if there is a cheaper plan with the same or more general properties for the same MEMO entry. Finally, the cheapest plan joining $n$ tables is returned.

## 2.2 Previous Work and its Limitations

Researchers have long observed that most of the compilation time is spent on join optimization (which includes both join enumeration and plan generation). Our experimental study on DB2 confirms such an observation. Figure 2 gives a breakdown of the compilation time for a real query workload. More than 90% of the time is either directly or indirectly spent on generating and saving join plans of different types. Ono and Lohman [17] introduced a key finding that optimization complexity is not determined by the total number of complete join trees that can be formed, but by the number of distinct binary joins. This is because the principle of optimality allows smaller subplans (cached in MEMO) to be "shared" by multiple larger plans. For example, for a query joining four tables–$A$, $B$, $C$, and $D$–together, the plan for a join between $A$ and $B$ can be used in join $(AB, C)$ as well as in join $(AB, D)$. Hence they attempt to estimate the compilation complexity of a query by the number of joins enumerated. The underlying assumption was that the cost of optimizing each join is approximately the same. These attempts suffer from the following limitations:

- Determining the number of joins from a general join graph is a hard problem. Although there are closed formulas for certain special classes such as the linear and the star-shaped queries [12, 17] and polynomial-time algorithms for counting the number of joins for queries with an acyclic join graph [7], counting the number of different joins with cycles in the join graph is as hard as counting *Hamiltonian* tours in a graph. The problem is #P-complete, which is even harder than NP-Hard [13]. Cycles are common in real queries because of automatic query generation tools as well as implied predicates computed through transitive closure in commercial systems. To make matters worse, optimizers in real systems are typically customized by various "knobs" and may not implement a full dynamic programming algorithm. For example, one such knob may limit the number of tables joined in the composite inner of bushy plans. Another example is that the optimizer may only support free-reordering plans for outerjoins, but not full reordering with compensation such as generalized outerjoins [8]. All these variants make estimating the number of joins a very difficult problem.

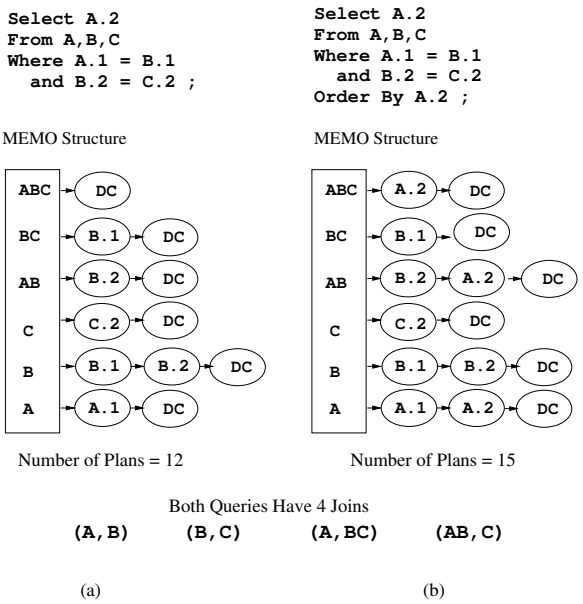



**Figure 3: Number of Joins vs. Number of Plans**

- A more severe problem is that, even if we are able to estimate the number of joins in a query correctly, the cost per join is far from uniform. As a matter of fact, each join typically generates a different number of plans depending on various "interesting" properties that need to be kept. Such properties are extensions of the important concept of *interesting orders* [20] introduced in System R. Suppose that we have two plans generated for table $R$, one ordered on $R.a$ (call it $p1$) and the other is not ordered (call it $p2$). Also suppose that $p1$ is more expensive than $p2$. Normally, $p1$ should be pruned by $p2$. However, if table $R$ can later be joined with table $S$ on attribute $a$, $p1$ can actually make the sort-merge join between the two tables cheaper than $p2$ since it doesn't have to sort $R$. To avoid pruning $p1$, System R identified orders of tuples that were potentially beneficial to subsequent operations for that query (hence the name interesting orders), and compared two plans only if they represented the same expression and had the same interesting order. This causes the number of plans generated for each join and stored in MEMO to vary. In Figure 3(a), we show a 3-way join query and the plans kept in the corresponding MEMO structure. For each MEMO entry, a list of plans is stored, each carrying a

different order property that is still interesting. We use *DC* to represent a "don't care" property value, which corresponds to all retired orders (orders no longer useful for subsequent operations such as joins) or no order at all. The cheapest plan with a *DC* property value is also stored in each MEMO entry if this plan is cheaper than any other plan with interesting orders. Modifying the query to that in Figure 3(b), by adding an *orderby* clause, increases the number of interesting order properties that need to be kept in all MEMO entries containing *A*. By comparing Figure 3(a) with Figure 3(b), we can see that the number of generated join plans changes, even though the join graph is still the same. We will elaborate more on plan properties in Section 3.2.

# 3. OUR APPROACH

To overcome the problems illustrated in the previous section, we build a *COTE* that estimates the compilation time in terms of the number of plans. We should stress that it is the number of generated plans (which is much more than those kept in the MEMO structure) that we want to estimate. We focus our attention on only the number of join plans for two reasons. First of all, non-join plans (e.g., table scan plans and index scan plans) are typically much fewer than join plans, and thus take much less compilation time (a small fraction of the time in the "other" category in Figure 2). Commercial systems typically consider only a limited number of combinations of index plans (index ANDing and ORing) for practical reasons. Second, the number of non-join plans are much easier to estimate. For example, there are typically two *group-by* plans—one sort-based and one hash-based—for each aggregation that needs to be performed in the query. The number of index plans can be estimated by counting the set of applicable indexes (either because certain predicates can be pushed to indexes or an index-only plan can be used). Therefore, when necessary, we can always include non-join plans in our model. In the rest of the paper, we will use the term "plan" and "join plan" interchangeably.

Our goal is to be able to provide a reasonably accurate estimate of generated plans without using too much time and space. We first describe how to reuse an existing join enumerator to obtain all enumerated joins in Section 3.1. Next, we discuss physical properties that affect the number of plans generated in Section 3.2. Section 3.3 outlines our general approach of plan estimation considering a single type of physical property. We extend our approach to support multiple types of physical properties in Section 3.4. Finally, we describe how to convert plan estimates to time estimates in Section 3.5.

## 3.1 Reusing the Join Enumerator

While estimating the number of joins for any given query is quite hard, a join enumerator of an optimizer certainly will know how many joins it considers once it iterates through all possible joins. So, it's tempting to exploit the join enumerator to do the join counting. However, we have to answer two questions: (1) How expensive is a join enumerator? (2) How feasible is it to exploit the join enumerator in a real system?

Despite the fact that join enumeration time may grow exponentially to the number of tables in a query, join enumeration itself only takes a small fraction of the total compilation time. In Figure 2, join enumeration time is only a small portion (less than 20%) of the "other" category. Plan generation, on the other hand, dominates join optimization time. Our analysis shows that a large amount of time in generating a plan is spent on estimating the execution cost. Execution cost estimation is the foundation of any cost-based optimizer, and commercial systems build sophisticated execution cost models to make sure that their cost estimates are close to reality. Such developments as sophisticated disk drives, innovative buffer management, and new types of histograms all contribute to the complexity of a cost model. Therefore, as long as we bypass plan generation, reusing join enumeration is not as expensive as one might expect.

Is the join enumerator still usable after bypassing plan generation, i.e., do we still have the same set of joins enumerated as before? After all, we do miss certain information without generating the plans. A join enumerator is reusable as long as the number of joins enumerated doesn't depend on information present only in generated plans (such as estimated execution costs). Note that it is fine if a join enumerator only changes the relative order of joins enumerated based on plan-dependent information, since this doesn't affect the compilation complexity. Fortunately, join enumeration is usually performed on a logical basis, i.e., two sets of tables may be joined as long as they don't overlap and there is at least one feasible join predicate linking the two sets (assuming Cartesian product is not allowed).

Note that the design of many extensible optimizers [16, 10] decouples join enumeration from plan generation. The separation between the two aspects allows for greater flexibility and extensibility of the query optimizer. For example, the join enumeration algorithm can be modified without affecting how joins are implemented, and a new join method can be easily introduced without touching any enumeration code. There is only a thin interface between the enumeration and the plan generation component. In such systems, bypassing plan generation becomes much easier.

Reusing existing join enumeration gives us two advantages: (1) We can estimate the compilation time of any query without relying on any assumption about the shape (linear, star, etc.) or connectivity of the query graph. (2) The number of joins enumerated are consistent with all the customizations of the enumeration algorithm, and thus can reflect the real complexity of compiling a query.

## 3.2 Physical Plan Properties

The idea of interesting orders was later generalized to multiple physical properties in [11, 16] and is used extensively in modern optimizers. Intuitively, a physical property is a characteristic of a plan that is not shared by all plans for the same logical expression (corresponding to a MEMO entry), but can impact the cost of subsequent operations. We use the term physical property in a broader scope to refer to any property that violates the principle of optimality; such a property need not be physical. Hereafter, we may refer to "physical property" simply as "property" when the meaning is clear from the context.

The presence of physical properties can significantly affect the number of join plans generated and stored in the MEMO structure. The analysis in [9] shows that the number of bushy join plans generated when taking physical properties

| Physical Property | Its Application | What's Interesting |
|---|---|---|
| order [20] | optimizing queries relying on sort-based operations [21] | an order with columns matching the join column of a future join, the grouping attributes (from the *groupby* clause), and the ordering attributes (from the *orderby* clause) |
| (data) partition [2, 9] | optimizing queries in a parallel database | a partition with partitioning keys matching the join column of a future join, the grouping attributes, and/or the ordering attributes (depending on whether it's a range partition or a hash partition ) |
| pipelinable | optimizing queries asking for the first $n$ rows | if pipelinable, i.e., no `SORTs`, builds for hash joins or `TEMPs` that require full materialization |
| data source [14] | optimizing queries on heterogeneous data sources | any data source |
| expensive predicates [5] | allowing expensive predicates to be applied after joins | any subset of the expensive predicates |

**Table 1: Summary of Physical Properties**

into consideration is on the order of $O(kn^3)$ for a linear-shaped query and $O(kn2^n)$ for a star-shaped query, where $n$ is the number of tables and $k$ is the upper bound of physical properties associated with each join. Notice that $k$ is independent of $n$ (e.g., the number of interesting orders is determined primarily by the number of predicates, not by the number of tables) and can be an important or even dominant factor for relatively small $n$ (less than 100), which is typical for join queries. This is especially true when multiple types of physical properties are kept together, which creates a combinatorial explosion. In contrast, "logical" properties such as cardinalities, keys, and functional dependencies (FDs) in general have the same value for the same logical expression, and therefore don't increase the number of plans for a join. Logical properties can affect the time needed to generate a plan though. For example, a plan with many keys and FDs might take longer to generate than a plan with no keys and no FDs at all. However, since we only need to compute a logical property once for each MEMO entry (through property caching) instead of for each join, this impact is amortized.

We summarize a list of representative physical properties and their applications in the first two columns of Table 1. Similar to interesting orders, each physical property has its own concept of being interesting. This is described in the third column in Table 1. Interesting properties can "retire", i.e., they are no longer useful for any of the remaining operations. In Figure 3, we can see that retired orders are not carried around in the MEMO structure any more. Two important aspects of interesting properties are how they are generated during planning, and how they are propagated by various join methods. Both the property generation policy and the propagation character affect our plan estimation, as we will see in the next section.

We distinguish between two kinds of property generation policies. An interesting property can be generated in a *lazy* fashion, in which it is only generated naturally, or an *eager* fashion, in which the optimizer tries to force the generation of a property if it is not naturally present. Take the order property as an example. Under a lazy policy, interesting orders generated include those that result from an index scan or a sort-merge join. In contrast, under an eager policy, if an interesting order does not exist naturally, the optimizer will add a `SORT` operator on top of a plan to force that order to exist. There are tradeoffs between the two policies.

| Join Method/ Property | Order | Partition |
|---|---|---|
| NLJN | full | full |
| MGJN | partial | full |
| HSJN | none | full |

**Table 2: Property Propagation Classification**

The eager one generates a larger search space, and hence may significantly increase optimization time. The lazy policy reduces the number of plans generated, at the risk of missing good plans. The decision of which policy to take is system-dependent, and is typically made heuristically by considering the cost of enforcing a property and the potential benefit it can bring.

Interesting properties are propagated differently by different types of join method. Again, take the order property for example. A nested-loops join (NLJN) can always propagate the order from its outer input. Since a sort-merge join (MGJN) requires both inputs to be sorted, it can only propagate orders corresponding to join columns of this particular join. Hash join (HSJN) in general destroys any order property, unless the inner table fits entirely in memory, which is hard to guarantee at compilation time. Accordingly, we categorize the way that a join method propagates properties into three classes: *full*, *partial* and *none*. Table 2 summarizes the propagation characterization of some physical properties.

## 3.3 Estimating the Number of Plans with One Type of Physical Property

In this section, we describe how to estimate the number of generated plans, assuming that there is only one type of physical property $P$. For each join being enumerated, the number of join plans generated depends on the number of distinct interesting properties of $P$ in both join inputs. One straightforward way to count the number of interesting properties for each enumerated join is to always recompute it on the fly from a collection of all possible property values of $P$, and filtering out retired properties. Even though this approach doesn't use any additional space beyond the MEMO structure itself, it is not efficient because it performs a lot of redundant computation. Observe that a property retired by a logical expression can never be interesting again for a subsuming logical expression. To exploit that feature, we instead accumulate an interesting property value list for

each MEMO entry bottom-up, and use the list from lower MEMO entries to compute the list for upper MEMO entries. This requires some additional space overhead and is a classical technique of trading space for time. Compared with the size of a full plan (typically in the order of hundreds of bytes), each property takes a much smaller amount of space (typically 4 bytes).

Initially, we have to populate the interesting property value list for MEMO entries for a single table. This actually depends on the property generation policy. If a lazy policy is used, we need to collect natural physical properties based on the physical configuration (such as indexes and partitions defined) of the underlying table. Otherwise, we will go through the logical expression of the query and push interesting properties down to base tables, which can normally be done in one top-down traversal of the internal representation of the query ([21] described such an approach of pushing interesting orders to base tables).

For each join $(S, L)$ (both $S$ and $L$ are sets of tables) enumerated, we propagate interesting property values of MEMO entries for $S$ and $L$ to that of the MEMO entry for $S \cup L$. We first make sure that a property value $p$ from the input can be propagated by at least one join method and then make sure that it is not retired by the join. Finally, we check to see if $p$ is redundant by testing its "equivalence" with other properties already in the interesting property list of $S \cup L$. Notice that joins can change property equivalence. For example, two distinct orders on $R.a$ and on $S.a$ become equivalent after the join predicate $R.a = S.a$ is applied. Therefore, equivalence needs to be checked for each enumerated join.

Instead of keeping one count for all join plans, we keep a separate count for each type of join method. This is because: (1) each type of join may generate a different number of plans, depending on its own property propagation policy; and (2) the cost of generating a join plan may not necessarily be the same for all join methods in real systems. For each candidate join, if a join method fully propagates $P$, we will use the number of distinct interesting properties from the join inputs as an estimate for the number of join plans. We further increment its count by one assuming that one additional plan will be generated for the $DC$ property value we introduced in Section 2.2. If a join method partially propagates $P$, we estimate the plans using a subset of interesting properties from the inputs (namely, those that can be propagated). Finally, for a non-propagating join, we always add its count by one.

One issue arises because of property subsumption [21]. We define a subsumption operator $\prec$ between two properties $p1$ and $p2$: $p1 \prec p2$ if $p2$ is more general than $p1$. For example, an order on $(R.a, R.b)$ ($o1$) is more general than that on $R.a$ ($o2$) and therefore $o2 \prec o1$. Property subsumption is relevant to join methods that partially propagate properties. Take for instance a MGJN between $R$ and $S$ using a join predicate $R.a = S.a$. Normally, only $o2$ can be propagated by the sort-merge join. However, if a subsuming interesting order property $o1$ is present (e.g., because of *orderby*), $o1$ will be propagated as well. This is because when the sort-merge join requests a plan ordered on $o2$, a plan ordered on $o1$ (since it is more general than $o2$) will be returned as well. As a result, two MGJN plans, instead of one, will be generated. To account for this, for a partial join, we compute a *coverage* list, which includes all interesting property values subsuming those propagatable by the join. The number of plans generated by the partial join is then increased by the length of the computed coverage list.

```
initialize(S)
input S: a MEMO entry
begin
  Allocate an interesting property list for S.
  If (S is for a single table)
    populate interesting property list for S based on
      the generation policy of P
end

accumulate_plans(S, L, J)
input S,L: MEMO entries of two table sets to be joined
output J: MEMO entry of the joined table sets
begin
  define list_s, list_l and list_j to be the interesting
    property list of S, L and J respectively
  For each property p in list_s and list_l
    if (p can be propagated by at least one join method)
      if (p has not been retired by the join AND
        p is not equivalent to any property in list_j)
        add p to list_j
  For each join type t
    accumulate join plans in join_t
    if (t fully propagates P)
      join_t += |list_s ∪ list_l|
      join_t += 1 (for DC property)
    if (t partially propagates P)
      list_p = {p|p ∈ list_s ∪ list_l, t propagates p}
      list_c = {p2|p1 ≺ p2, p1 ∈ list_p, p2 ∈ list_s ∪ list_l}
      (list_c is the coverage list)
      join_t += |list_p ∪ list_c|
    if (t doesn't propagates P)
      join_t += 1
end
```

**Table 3: Estimate Number of Plans Considering One Type of Physical Property $P$**

We are now ready to summarize our general framework in Table 3. We introduce two new functions *initialize()*, and *accumulate_plans()*. The first function is called every time the join enumerator creates a new MEMO entry. It allocates space for an interesting property list of $P$ and initializes the list for single-table MEMO entries, based on the generation policy of $P$. The second function is called for each enumerated join $(S, L)$ (assume that both $S$ and $L$ can serve as the outer for now). The function first tries to propagate interesting property values from the inputs. A property is propagated if it is propagatable by at least one type of join method, has not been retired by the join, and is not equivalent to any property already propagated. The algorithm then accumulates plan counts for each type of join, based on how it propagates properties. Even though our algorithm is based on a MEMO structure for a single query block, it can be easily extended to handle multiple query blocks for more complex queries.

## 3.4 Multiple Types of Physical Properties

We now extend our general framework to estimate generated plans for multiple physical properties. One simple solution is to treat multiple physical properties as a single

"compound" property, and our algorithm in Table 3 can easily be reused. In the algorithm, instead of storing single property values in the MEMO structure, we store vectors containing multiple property values, each corresponding to a different type of physical property. A compound property is retired only when all values in the vector are retired. The interesting property lists in the MEMO structure become longer because of the combinatorial effect of multiple properties.

We observe that certain types of physical properties are orthogonal to one another. Take the order property and the partition property in a shared-nothing parallel database system as an example. While the partition property designates how data is distributed across all the nodes, the order property specifies how data is sorted on each individual node. An interesting order property value can always coexist with any interesting partition property value and vice versa (we can always first redistribute data satisfying a specific partition followed by sorting data at each node to satisfy a specific order). In this situation, we alternatively maintain two separate interesting property lists, one for each orthogonal type of physical property. This saves both time and space during plan estimation, since we avoid generating and storing property combinations. The number of plans can then be estimated by multiplying the length of the two interesting property lists. Sometimes an interesting property combination can include a retired individual property. If we keep separate interesting property lists, no individual retired property value can exist in the MEMO structure. Therefore, this approach tends to underestimate the number of plans. We will see in Section 5 that this isn't a serious problem in general.

### 3.5 Estimating Time from Plan Estimates

To translate the estimated number of plans to an estimate of the compilation time, we assume a simple model: $T = T_{inst} \times \sum(C_t \times P_t)$, where $T_{inst}$ is the time per instruction (a machine-dependent parameter), $C_t$ is a constant representing the number of instructions to generate a join plan of type $t$, and $P_t$ is the estimated number of join plans of type $t$. In order to obtain $C_t$, we can collect the real counts of generated join plans together with the actual compilation time for a set of training queries, and then calculate $C_t$ by running regression on our model. Note that we should rerun the regression to obtain a new set of $C_t$ for new releases of a database system, since the time to generate a join plan is likely to change.

## 4. IMPLEMENTATION EXPERIENCE ON A REAL SYSTEM

In order to validate our approach, we prototyped the methods described in the previous section on DB2. We share some of our experiences in this section, as they might be applicable to other systems as well.

DB2 uses a conventional bottom-up dynamic programming join enumeration algorithm. It has a *serial* version and a *parallel* version. Without losing generality, we focused on two most important kinds of physical properties: order and partition. In the serial version, we consider order as the only type of physical property. In the parallel version, we consider both the order and partition properties. To count the number of plans generated, we introduced a new *plan-estimate* mode in the optimizer and instrumented the code in the join enumerator to call the *initialize()* and the *accumulate_plans()* functions, instead of the normal plan generation function in this mode. We first summarize all the issues on the serial version.

1. DB2 uses an eager policy for generating order properties and precomputes a list of interesting orders for base tables. We naturally reuse the precomputed results to initialize the interesting order list for single-table MEMO entries.

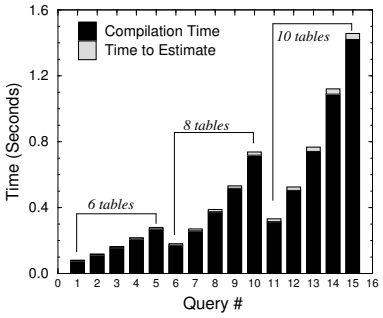
2. When computing the coverage list for a partial join (MGJN), we distinguish between an *orderby coverage* and a *groupby coverage*, because the relative column positions are important for the former, but not for the latter. Therefore, prefix subsumption is used to compute an *orderby coverage* list, whereas set subsumption is sufficient for *groupby coverage* [21].

3. Because of the presence of outer joins, correlations and subqueries, not all the table sets can be used as the "outer" in a join. The eligibility of a table set to be an outer is determined logically and marked by the join enumerator. A join method typically propagates the order property of its outer. To avoid overestimating, we only include order properties from an input marked as "outer enabled" to count join plans in *accumulate_plans()*, as described in Table 3.

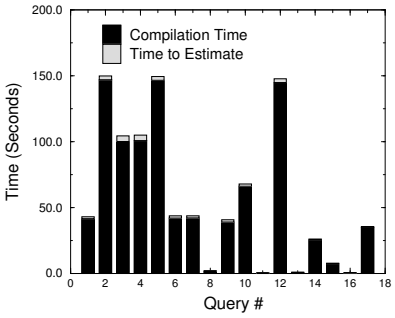
4. In the general framework, the propagation of interesting properties is performed for each join enumerated. We observe that order properties propagated to the same MEMO entry are hardly changed from join to join. For example, join $(RS, T)$ propagates to the MEMO entry for table set $\{R, S, T\}$ roughly the same set of orders as that propagated through join $(R, ST)$. The only differences are a small number of orders with columns from multiple tables. We therefore can simplify and speed the computation by propagating order property values for just the first join that produces plans for a MEMO entry. This simplification cuts down our estimation overhead without losing too much precision on plan counts.

5. DB2 allows a Cartesian product between two sets of tables if the estimated cardinality of one of the inputs is 1. This increases the number of joins enumerated and therefore our formula will underestimate the number of plans generated. To correct this, we store cardinality estimates in the MEMO structure for the join enumerator to make joins enumerated consistent. Cardinality is a logical property, and thus needs to be computed only once for each MEMO entry.

The parallel version of DB2 uses a typical shared-nothing architecture [2]. We treat the order property and the partition property as independent and keep two separate interesting property lists in the MEMO structure. The partition property is generated lazily, so we use the physical partition of each table to initialize the interesting partition value list of its MEMO entry. One subtlety arises during our implementation. Because of a heuristic rule exploited in DB2, additional interesting partitions can sometimes be generated. Consider a join between two tables $R$ and $S$. If




(a) *linear_s*




(b) *real2_s*

| Actual Time | Time to Estimate | Pctg |
|---|---|---|
| 17.010 | 0.115 | 0.7% |
| 1.103 | 0.031 | 2.8% |
| 4.353 | 0.022 | 0.5% |
| 12.670 | 0.085 | 0.7% |
| 1.771 | 0.0203 | 1.1% |
| 11.161 | 0.038 | 0.3% |
| 11.157 | 0.035 | 0.3% |
| 1.350 | 0.0173 | 1.3% |

(c) *real1_p*

**Figure 4: Overhead of Compilation Time Estimation Compared with Actual Optimization: (a) *linear* Workload on Serial Version, (b) *Real2* Workload on Serial Version, (c) *real1* Workload on Parallel Version**

neither $R$ nor $S$ is partitioned on the join column, the optimizer will try to repartition $R$ and $S$ on the join column, and thereby creating a new partition property value. To determine whether repartitioning might happen for a join, we test if any join column is used as the partitioning key in any of the interesting partition property values from the inputs. We propagate additional partitions on join columns if the test fails.

DB2 supports all three types of join methods we mentioned earlier. We ran regression tests based on the model described in Section 3.5 to obtain two sets of $C_t$, one for the serial version and the other for the parallel version, because generating a plan is typically more expensive in the latter. In the current release of DB2, the ratio of $C_m : C_n : C_h$ ($C_m$ for MGJN, $C_n$ for NLJN and $C_h$ for HSJN) is $5 : 2 : 4$ for the serial version and $6 : 1 : 2$ for the parallel version.

# 5. EXPERIMENTAL RESULTS

We present our experimental results on DB2 in this section. Our tests were conducted on a level of optimization that uses dynamic programming with certain limits on the composite inner size of a join. All experiments were performed on a UNIX machine with two 400 MHz processors and 1GB of RAM. The parallel version was set up on four logical nodes, all running on the same machine. We conducted tests on a wide range of workloads, but selected a representative subset to present here. The following summarizes the workloads we selected.

**Synthetic Workloads:** We generated workloads with two predefined join graphs: linear-shaped and star-shaped queries (we refer to them as *linear* and *star* respectively). In the linear workload, $N$ tables are joined in a sequential fashion (first table is joined to the second table and the second table is joined to the third, etc.). In the star workload, one table serves as the center table and all other tables are joined to the center table. In each workload, we varied the number of tables, the number of join predicates, and the number of columns in the *orderby* clause and the *groupby* clause. In the following experiments, we selected 15 queries from each workload. The queries are in three batches of five. Each batch joins the same number of tables, but varies the number of join predicates from 1 to 5. The number of tables joined for the three batches are 6, 8, and 10.

**Real Workloads:** We selected two real customer workloads. The first one consists of 8 queries (we call it *real1*) and the second one consists of 17 queries (we call it *real2*). Queries from both workloads are complex data warehouse queries with inner joins, outerjoins, aggregations and subqueries. To get a sense of query complexity, one query from the *real2* workload consists of 14 tables constructed from 3 views, 21 local predicates and 9 *groupby* columns that overlap with the join columns.

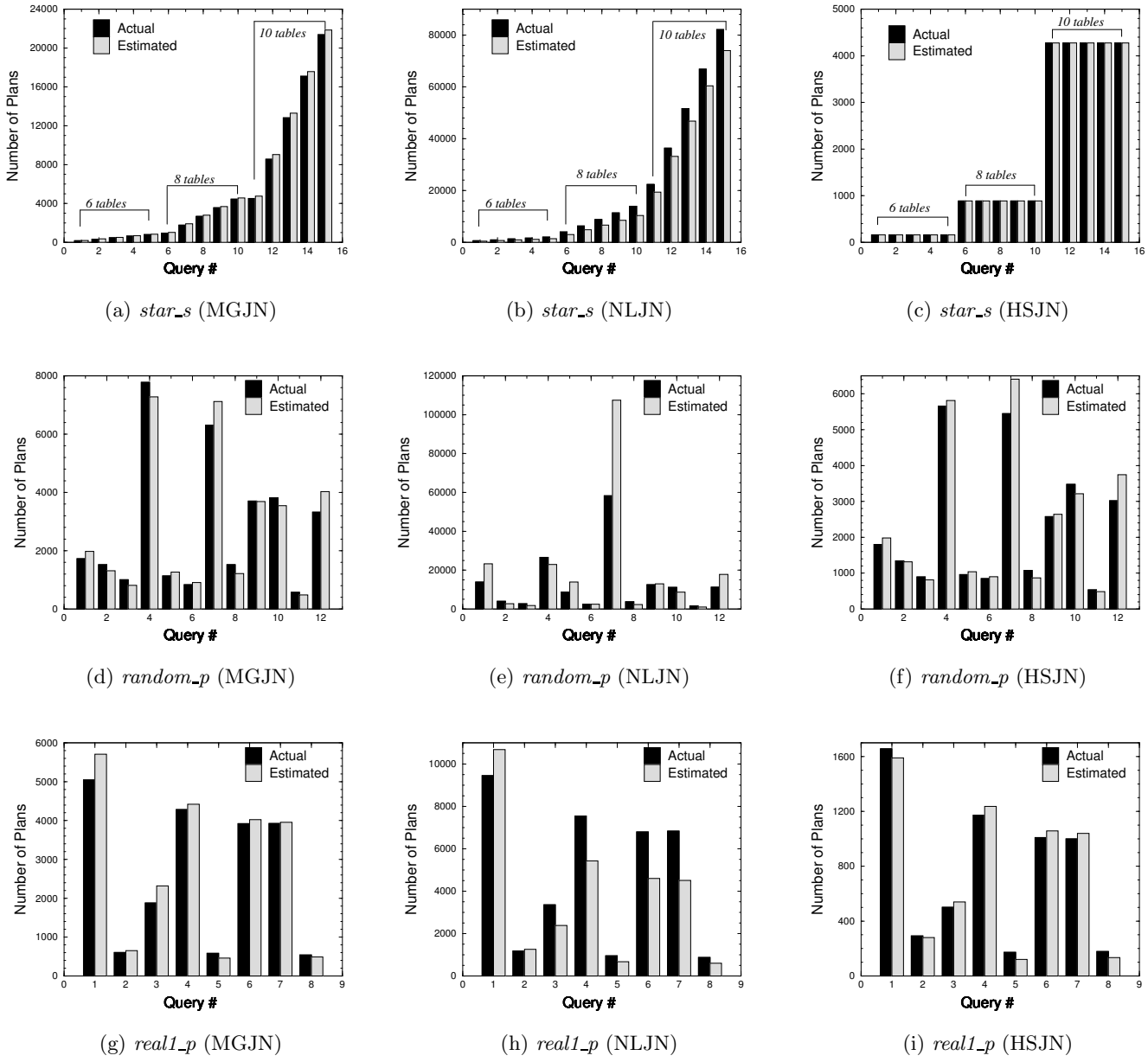
**Randomly Generated Workloads:** We employed a random query generator used for testing the robustness of DB2. The tool creates increasingly complex queries by merging simpler queries defined on a given database schema (the schema from *real1* was used in our test), using either subqueries or joins, until a specified complexity level is reached. One important feature of the generator is that it tries to join two tables with a foreign-key to primary-key relationship or having columns with the same name. As a result, the queries produced are relatively close to real customer queries. We refer to this workload as *random*.

**Benchmark Workloads:** We chose from the *TPC-H* [3] benchmark 7 queries that have the longest compilation time.

In the rest of this section, we apply postfixes of "_s" and "_p" to the name of each workload to refer to a workload being tested on the serial and the parallel version of DB2, respectively.

## 5.1 Plan Estimation Overhead

We first evaluate the overhead of our *COTE* compared to the actual compilation time. In Figure 4, we compare the two, for three different workloads. Figure 4(a) and Figure 4(b) give the results for *linear* and *real2*, respectively, both on the serial version. The time spent on *COTE* is between 1% and 3% of the actual compilation time. The overhead includes both the join enumeration cost and the cost of maintaining the interesting property value list in the MEMO structure. Figure 4(c) shows the overhead for the *real1* workload on the parallel version. The overhead is even smaller in the parallel version for two reasons. First of all, plan generation becomes more complicated in the parallel version and hence is more time consuming. Secondly, since




(a) *star_s* (MGJN) (b) *star_s* (NLJN) (c) *star_s* (HSJN)

(d) *random_p* (MGJN) (e) *random_p* (NLJN) (f) *random_p* (HSJN)

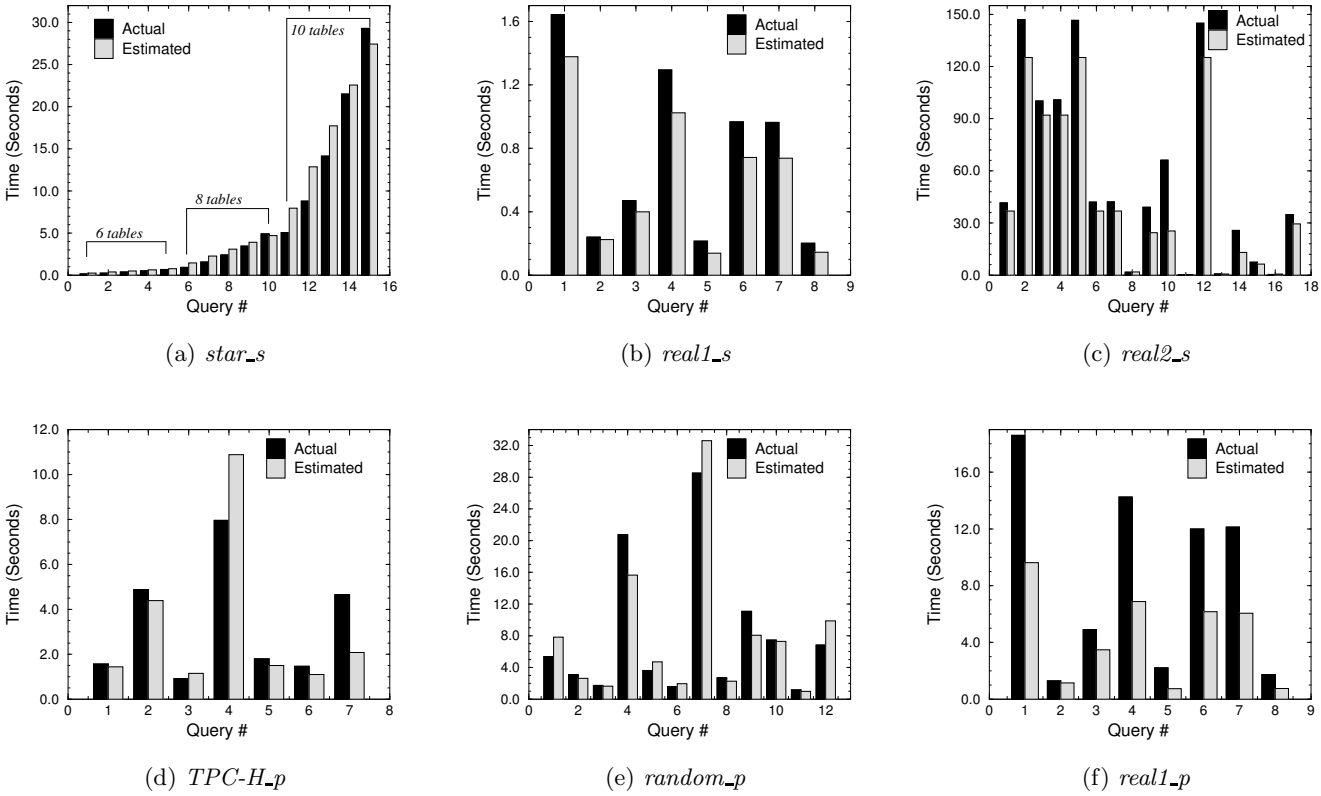
(g) *real1_p* (MGJN) (h) *real1_p* (NLJN) (i) *real1_p* (HSJN)

**Figure 5: Accuracy of the Estimated Number of Join Plans: (a)-(c), Plan Estimation for *star* Workload on Serial Version; (d)-(f), *random* Workload on Parallel Version; (g)-(i),*real1* Workload on Parallel Version**

we accumulate the partition property separately from the order property, we avoid the generation of all (partition,order) combinations during real optimization. This will affect our plan estimation though, as we will see in the next section. Although not shown here, the overhead percentages on other workloads are similar. To summarize, our results prove that join enumeration, together with property accumulation, although of exponential complexity, is not the primary consumer of time in query optimization.

## 5.2 Accuracy of the Estimated Number of Plans

In this section, we evaluate the accuracy of estimating the number of generated plans for each join method, using the approach outlined in Section 3.3. We first present the results on the serial version running the *star* workload in Figures 5(a)-5(c). Our estimates are exact in the case of HSJN plans (Figure 5(c)), because HSJN plans don't propagate interesting orders, and hence are exactly twice the number of enumerated joins. The estimated number of NLJN plans and MGJN plans are still close to the actual ones—less than 30% error for NLJN and less than 14% error for MGJN estimates. One problem affecting our estimates for NLJN and MGJN is a plan "sharing" problem between two property values, one more general than the other. Consider the following example. Suppose that table $R$ has two interesting orders: $(R.a)$ and $(R.a, R.b)$. We will assume that two plans




(a) *star_s* (b) *real1_s* (c) *real2_s*

(d) *TPC-H_p* (e) *random_p* (f) *real1_p*

**Figure 6: Compilation Time Estimation: on Serial Version, (a) *star* Workload, (b) *real1* Workload, (c) *real2* Workload; on Parallel Version, (d) *TPC-H* Workload, (e) *random* Workload, (f) *real1* Workload**

will be generated for NLJN (and for MGJN if $(R.a)$ is a join column). However, if the cost of a plan (for $R$) ordered on $(R.a, R.b)$ is cheaper, it will actually prune a plan ordered on $R.a$ (assuming everything else is the same) because the former is more general than the latter. We can think of the problem as the order on $R.a$ sharing the same plan as the order on $(R.a, R.b)$. The net result is that fewer join plans are actually generated than we estimate based upon just the number of interesting orders. Since the *COTE* bypasses plan generation and thus does not have plan cost, it's difficult for it to tell whether a plan might be pruned. This explains why we overestimate the number of MGJN plans (Figure 5(a)), but not our estimates for NLJN plans (Figure 5(b)), which turn out to be lower than the actual numbers. We discovered that this latter effect was caused by an implementation oversight in DB2 that generated redundant NLJN plans during the actual optimization. Once the implementation is fixed, we expect that the actual number of NLJN plans will better agree with our estimates. Finally, we want to point out the big difference between the number of MGJN (NLJN) plans and the number of HSJN plans among queries in the same batch, even though they all have the same number of joins (Figures 5(a) and 5(b), vs. Figure 5(c)).

Next, we present plan estimates and actuals for the *random* and *real1* workloads in the parallel environment in Figures 5(d)-5(f) and Figures 5(g)-5(i), respectively. The results in both sets of figures are quite similar. Notice that in both cases, our HSJN numbers are no longer the same as the actual ones. This is because the cardinality estimation we employed in *plan-estimate* mode is "simpler" than that used in real compilation. For example, it doesn't take into consideration the effect of keys and functional dependencies (since we don't propagate them in *plan-estimate* mode). Remember that joins enumerated in DB2 are affected by cardinality estimates due to heuristic rules such as allowing Cartesian products when one of the inputs has only one row. Therefore, sometimes we don't get the exact number of joins in *plan-estimate* mode, which directly translates into estimate errors of HSJN plans (between $-2\%$ to 24%). NLJN plans and MGJN plans are also affected, to a certain extent. In addition, because we maintain the order property and the partition property separately, we ignore plans generated carrying only an interesting order, but not an interesting partition, and vice versa. However, such errors tend to be relatively small. Finally, in Figure 5(e), NLJN has a few outliers where errors are more than 50%. These are the few cases where various kinds of errors accumulate, rather than cancel one another.

## 5.3 Accuracy of Compilation Time Estimation

We now present our compilation time estimation, based on the plan estimates using the model described in Section 3.5. Figures 6(a)-6(c) show the time estimates on the serial version, for the *star*, *real1* and *real2* workloads, respectively. All estimates are within 30% of the actual compilation time, but are much better in many cases. Observe that in Figure 6(a)

for the *star* workload, the actual compilation time significantly differs among queries within each batch, and yet our estimates are able to correctly predict the trend. Had we estimated compilation time using the number of joins only, we would have had errors of 20 times larger, no matter how we chose the time per join, because such a metric cannot distinguish queries within the same batch. Our plan-level metric provides much more accurate estimation.

Finally, Figures 6(d)-6(f) show our results on the parallel version for the *TPC-H*, *random*, and *real1* workloads. While we get less than 30% estimation error in the first two workloads, the estimates have larger errors in the *real1* workload (up to 66%). Recall that in Figures 5(g)-5(i), we get less than 30% error in estimating the number of generated plans for the same workload. This discrepancy is caused by a larger variation in the time to generate individual plans in the parallel environment. We plan in the future to investigate building a more sophisticated model to estimate the cost of generating a plan by taking into account other relevant factors such as the number of join predicates. Nevertheless, our estimation errors are still much lower than any estimation at the join level and, depending on the application, may still be acceptable. For example, an estimated plan execution cost can sometimes have comparable errors and thus our estimates are reasonable for the *meta*-optimizer.

## 5.4 Summary of Experiments

The experiments presented in this section show consistent and promising results. Our plan estimation is very close for both synthetic workloads and more complex random and real customer workloads. The formula that we used to convert the number of plans to time estimates, although simple, is quite effective for most of the workloads we tested. We want to emphasize the fact that our results reflect a commercial optimizer in two different environments.

We do observe errors in our estimates. Some of them are errors from the join enumerator and others are errors from how properties are used to estimate plans and how plans are converted to compilation time. Some of these errors are harder to correct than others because they depend on the estimates of execution cost of individual plans, which we don't generate. We summarize those errors below (a "+" sign meaning that we overestimate and a "-" sign meaning that we underestimate).

+/- inconsistent cardinality estimation affecting joins enumerated, because of cardinality-sensitive heuristic rules

\+ plan sharing between a more general property and a less general one

\- non-interesting properties surviving when multiple types of physical property are present (in the parallel version).

+/- variation of time to generate a plan of a specific join method

Finally, we observed during our experiments that the number of indexes present does not significantly affect the number of plans generated, because `DB2` uses an *eager* policy for order propagation. On the other hand, how data is initially partitioned in a parallel environment does affect plans generated and the compilation time because a *lazy* policy is employed for the partition property.

# 6. DISCUSSION

In this section, we first discuss related work, and then the possibilities of extending our work.

## 6.1 Other Related Work

[23] studied the problem of counting and sampling execution plans in a cost-based optimizer. The work tries to count the number of complete plans from counts of subplans stored in the MEMO structure. The authors proposed their work mainly for stress tests of an optimizer, and they do not bypass plan generation as we do.

Researchers have studied how to reduce the search space of a dynamic programming enumerator by taking advantage of a quickly precomputed initial plan for the query (a "*pilot-pass*" approach) [19]. During dynamic programming optimization, any generated subplan may be pruned if its estimated execution cost exceeds that of the full initial plan. This can affect our estimates on the number of plans generated, since our approach bypasses plan generation and thus doesn't estimate the execution cost of any subplan. However, this kind of pruning may not be very effective, even when the initial solution is quite good, since the cost of a (good) full plan typically exceeds that of most partial plans. Our preliminary analysis on `DB2` shows that no more than 10% of plans are pruned by the initial plan in real workloads.

## 6.2 Extension

Our model can be extended to optimization with materialized views as well. If materialized views are selected on a cost basis, optimization will take roughly the same amount of time as that without materialized views, if pruning is not very effective (same argument as in the previous paragraph). When materialized views are chosen heuristically in a query rewrite phase, our model can be used to estimate the compilation time for optimizing the rewritten query. In either case, we need to take into consideration the time spent on matching materialized views. We plan to investigate this in the future.

It's possible to estimate the compilation time of multiple levels of optimization in a single pass, as long as the search space of the highest level subsumes that of all other levels. For example, when estimating the compilation time for a level that considers all bushy trees, we can piggyback the estimation for a level that only considers left deep tree or that limits the size of composite inners. Such information is useful for a *meta*-optimizer to choose the right optimization among multiple levels. The overhead of estimating compilation time is also amortized.

Our work can also be extended to estimate memory consumption during optimization. Assuming that each plan takes roughly the same amount of space, the total amount of memory needed in a MEMO structure can be estimated by summing the length of the interesting property lists of all MEMO entries and multiplying that by the space required per plan. Note that this is a lower bound of the memory required by an optimizer. If it is already larger than the currently available memory, there is no point in starting optimization at that level since it is very likely to run out of memory.

In this paper, we focused on a bottom-up dynamic programming style join enumerator. A transformation-based optimizer [10] also uses a MEMO structure. However, MEMO entries can represent more logical expressions (not just joins)

and are not necessarily filled bottom-up, i.e., an entry for a larger logic expression might be populated before that for a smaller expression. Because of this, a transformation-based optimizer has to store many more generated plans in the MEMO structure, not just the best plan for each interesting property, as with the bottom-up approach. On the other hand, it can obtain a complete plan much earlier, and thus has the ability to stop optimization at any given time. Since the decision of when to stop optimization may depend on estimates of execution costs, this can affect the estimates of compilation time using our general framework. We'd like to investigate this problem further in the future.

# 7. CONCLUSION

In this paper, we presented how to build a *COTE* to accurately estimate the compilation time of a query optimizer. Our work is based on the main observation that most of the compilation time is spent on generating join plans. To overcome the limitations of previous work, we reused the join enumeration process to get a close estimate of the number of joins, and maintain interesting physical property value lists in the MEMO structure to facilitate counting join plans more accurately. Our approach avoids generating the actual plans and collects only a small set of relevant properties. As a result, it can provide good compilation time estimates using only a small fraction of the actual optimization time (less than 3% in all workloads). We conducted a comprehensive experimental study on a wide range of queries in a real system. The results show that our estimates are typically within 30% error compared with the actual compilation time. We believe that our work is important for many applications, including *meta*-optimization, mid-query re-optimization, monitoring workload analysis tools, and the design and tuning of an optimizer. In the future, we plan to account for more physical properties in our *COTE*.

## Acknowledgments

We'd like to thank Nimrod Megiddo for helping us understand the complexity of join enumeration and David Simmen for explaining some of the `DB2` code to us.

# 8. REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

[2] C. Baru, et al. DB2 parallel edition database systems: The future of high performance database systems. *IBM Systems Journal*, 34(2), 1995.

[3] TPC benchmark H (decision support) revision 1.1.0. http://www.tpc.org/.

[4] Surajit Chaudhuri and Vivek Narasayya. Microsoft index tuning wizard for SQL Server 7.0. *SIGMOD Record*, 27(2), 1998.

[5] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *TODS*, 24(2), 1999.

[6] IBM Corporation. DB2 Universal Database enterprise edition Version 8.1. 2002.

[7] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Uniformly-distributed random generation of join orders. In *ICDT*, 1995.

[8] César A. Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *TODS*, 22(1), 1997.

[9] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *ACM SIGMOD*, 1992.

[10] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[11] Goetz Graefe and David J. DeWitt. The exodus optimizer generator. In *ACM SIGMOD*, 1987.

[12] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *ACM SIGMOD*, 1991.

[13] Mark Jerrum. Counting trees in a graph is #P-complete. *Information Processing Letters*, 51(3), 1994.

[14] Vanja Josifovski, Peter M. Schwarz, Laura M. Haas, and Eileen Lin. Garlic: A new flavor of federated query processing for DB2. In *ACM SIGMOD*, 2002.

[15] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD*, 1998.

[16] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *ACM SIGMOD*, 1988.

[17] K. Ono and G. Lohman. Measuring the complexity of join enumeration in relational query optimization. In *VLDB*, 1990.

[18] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *ACM SIGMOD*, 2002.

[19] Arnon Rosenthal, Umershwar Dayal, and David Reiner. Speeding a query optimizer: the pilot pass approach. *unpublished manuscript*.

[20] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path election in a relational database management system. In *ACM SIGMOD*, 1979.

[21] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *ACM SIGMOD*, 1996.

[22] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

[23] Florian Waas and César A. Galindo-Legaria. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. In *ACM SIGMOD*, 2000.