Iterative Dynamic Programming: A New Class of Query Optimization Algorithms

Donald Kossmann Konrad Stocker

Universität Passau
Lehrstuhl für Informatik
94030 Passau, Germany
⟨lastname⟩@*db.fmi.uni-passau.de*

Abstract

The query optimizer is one of the most important components of a database system. Most commercial query optimizers today are based on a dynamic-programming algorithm, as proposed in [SAC+79]. While this algorithm produces good optimization results (i.e., good plans), its high complexity can be prohibitive if complex queries need to be processed, new query execution techniques need to be integrated, or in certain programming environments (e.g., distributed database systems). In this paper, we present and thoroughly evaluate a new class of query optimization algorithms that are based on a principle that we call *iterative dynamic programming*, or `IDP` for short. `IDP` has several important advantages: First, `IDP`-algorithms produce the best plans of all known algorithms in situations in which dynamic programming is not viable because of its high complexity. Second, some `IDP` variants are adaptive and produce as good plans as dynamic programming if dynamic programming is viable and as-good-as possible plans if dynamic programming turns out to be not viable. Three, all `IDP`-algorithms can very easily be integrated into an existing optimizer which is based on dynamic programming.

1 Introduction

The great commercial success of database systems is partly due to the development of sophisticated query optimization technology: users pose queries in a declarative way using SQL or OQL, and the optimizer of the database system finds a good way (i.e., plan) to execute these queries. The optimizer, for example, determines which indices should be used to execute a query and in which order the operations of a query (e.g., joins and group-bys) should be executed. To this end, the optimizer enumerates alternative plans, estimates the cost of every plan using a cost model, and chooses the plan with the lowest cost.

One central component of a query optimizer is its *search strategy* or *enumeration algorithm*. The enumeration algorithm of the optimizer determines which plans to enumerate, and the classic enumeration algorithm is based on *dynamic programming*. This

algorithm was pioneered in IBM's System R project [SAC+79], and it is used in most query optimizers today. Dynamic programming works very well if all queries are standard SQL-92 queries, the queries are moderately complex, and only simple textbook query execution techniques are used by the database system. Dynamic programming, however, does not work well if these conditions do not hold; e.g., if the database system must support very complex applications like SAP R/3 whose queries often involve many tables [DHKK97, KKM98] or new query optimization and execution techniques need to be integrated into the system in order to optimize queries in a distributed and/or heterogeneous programming environment [HKWY97, Kos98]. In these situations, the search space of query optimization can become very large and dynamic programming is not always viable because of its very high complexity.

In general, there is a tradeoff between the complexity of an enumeration algorithm and the quality of the plans generated by the algorithm. Dynamic programming represents one extreme point: dynamic programming has exponential time and space complexity and generates "optimal" plans.[1] Other algorithms have lower complexity than dynamic programming, but these algorithms are not able to find as low-cost plans as dynamic programming. Since the problem of finding an optimal plan is $\mathcal{NP}$ hard [IK84, SM97], implementors of query optimizers will probably always have to take this fundamental tradeoff between algorithm complexity and quality of plans into account when they decide which enumeration algorithm to use.

In this paper, we present and evaluate a new class of enumeration algorithms that are based on a technique that we call *iterative dynamic programming* or `IDP` for short. The main idea of `IDP` is to apply dynamic programming several times in the process of optimizing a query; either to optimize different parts of a plan separately or in different phases of the optimization process. As we will show, `IDP` has reasonable (i.e., polynomial) complexity and produces in most situations very good plans. In fact, our experiments show that `IDP` produces better plans than any other algorithm in situations in which dynamic programming is not viable because of its high (exponential) complexity. One particular advantage is that certain `IDP`-variants adapt to the optimization problem: if the query is simple, these `IDP`-variants produce an optimal plan like dynamic programming and in the same time as dynamic programming. If the query is too complex for dynamic programming, these `IDP`-variants produce sub-optimal plans, but these sub-optimal plans are significantly better than the plans produced by other algorithms that are applicable in those situations (e.g., randomized algorithms as proposed in [IK90]).

Another advantage of `IDP` is that all `IDP` variants can very easily be integrated into an existing query optimizer which is based on dynamic programming. As we will see, only a couple of lines of code need to be changed and complex components of the optimizer such as the cost model and the enumeration rules can be used in an `IDP`-enhanced optimizer just as well as in an existing dynamic-programming optimizer, without any adjustments. Using randomized algorithms, for example, practically involves re-building a completely new optimizer from scratch, if the existing optimizer is based on dynamic programming. Since most existing optimizers are indeed based on dynamic programming and database

[1] Throughout this paper, we call a plan *optimal* if it has the lowest cost of all plans according to the optimizer's cost model. Since the optimizer's cost model is not always accurate, an optimal plan, in our sense, may actually not be the best plan to execute a query.

vendors have invested greatly into their query optimizers, this is an important argument in favor of IDP.

This paper also presents the results of performance experiments that show how IDP fares in comparison to existing enumeration algorithms such as dynamic programming or randomized algorithms. These performance experiments are carried out using select-project-join (SPJ) queries in a distributed database environment. Therefore, the results are directly applicable to a wide range of database systems; e.g., standard centralized and distributed database systems (e.g., Oracle), client-server systems such as SAP R/3 [BEG96, KKM98], and heterogeneous database systems such as Garlic [HKWY97] or TSIMMIS [LYV+98]. We do not show the results of other types of complex queries which motivate the development of IDP; e.g., queries with expensive predicates [HS93, CS96], queries with group-by operators [CS94], or top N queries [CK97]. The presence of expensive predicates, group-bys, or top N increases the size of the search space in a similar way as a distributed environment.

The remainder of this paper is organized as follows. Section 2 gives an overview of related work. Section 3 describes dynamic programming; in particular, that section shows how dynamic programming can be extended to optimize queries for distributed databases and analyzes the complexity of dynamic programming in this case. Section 4 describes and analyzes greedy algorithms for query optimization. Section 5 presents *iterative dynamic programming* and a whole family of algorithms based on that principle. Section 5 also analyzes the complexity of the different IDP-variants and shows that all IDP-variants have polynomial time and space complexity. Section 6 discusses alternative *plan evaluation functions* which are special functions that are needed in order to implement IDP. Section 7 contains the results of our performance experiments. Section 8 concludes this paper with suggestions for future work.

2 Related Work

Due to its importance, a large number of different algorithms have already been developed for query optimization in database systems. All algorithms proposed so far fall into one of three different classes or are combinations of such basic algorithms [Swa89, Van98]. In the following, we will briefly discuss each class of algorithms; a more complete overview and comparison of many of the existing algorithms can be found in [SMK97].

Exhaustive Search: All published algorithms of this class have exponential time and space complexity and are guaranteed to find the best plan according to the optimizer's cost model. As stated in the introduction, the most prominent representative of this class of algorithms is (bottom-up) dynamic programming [SAC+79] which is currently used in most database systems. Efficient ways to implement dynamic programming have been proposed in [OL90, VM96]. Other representatives of this class are A^* search [KMP93] and transformation-based techniques (with top-down dynamic programming) such as those used in EXODUS, Volcano, and some commercial systems [GD87, GM93, PGLK97].

Heuristics: Typically, the algorithms of this class have polynomial time and space complexity, but they produce plans that are often orders of magnitude more ex-

pensive than a plan generated by an exhaustive search algorithm. Representatives of this class of algorithms are "Minimum Selectivity" and other greedy algorithms [Pal74, Swa89, SYT93, SMK97], the KBZ algorithm [KBZ86], and the AB algorithm [SI93]. We will specifically describe greedy algorithms in Section 4.

Randomized Algorithms: Various variants of randomized algorithms have been proposed in [IW87, SG88, IK90, LVZ93, GLPK94, SMK97]. The big advantage of randomized algorithms is that they have constant space overhead. The running time of most randomized algorithms cannot be predicted because these algorithms are indeterministic; typically, randomized algorithms are slower than heuristics and dynamic programming for simple queries and faster than both for very large queries. The best known randomized algorithm is called `2PO` and is a combination of applying *iterative improvement* and *simulated annealing* [IK90]. In many situations, `2PO` produces good plans. As we will see in Section 7, however, there are situations in which `2PO` produces plans that are orders of magnitude more expensive than an optimal plan.

`IDP` can be classified as a generalization of dynamic programming and greedy heuristics with the goal to combine the advantages of both: very good plans due to dynamic programming and acceptable overhead using heuristics, if necessary.

Independent from our work, Shekita and Young [SY98] devised one `IDP` variant that we will refer to as "$\texttt{IDP}_2$-standard-bestPlan" in this paper. Incidently, they also called their approach "iterative dynamic programming." As we will see, their variant does not produce as good plans as the other `IDP` variants.

As an alternative to any of the above-mentioned algorithms and `IDP`, one obvious way to process complex queries is the following: the optimizer first checks if the query can be optimized using dynamic programming. If yes, the optimizer carries out dynamic programming; if not, the optimizer applies heuristics. Such an approach has apparently been taken in the implementation of several commercial database systems. `IDP` is clearly better than such an approach: (1) As we will see in Section 7, `IDP` produces significantly better plans than any other heuristics we are aware of, if the query cannot be optimized using dynamic programming. (2) It is difficult, if not impossible, to predict for which queries and under which circumstances dynamic programming is viable. In a distributed system, for example, dynamic programming might be viable for certain 17-table queries but not for certain 10-table queries, depending on the locations at which copies of the tables are stored and other characteristics of the query and database. In either situation, `IDP` makes it possible to exploit dynamic programming as much as possible and adapt automatically if dynamic programming hits its limits.

Currently several new features are built into database systems which cause full-fledged dynamic programming to become impractical even for fairly simple queries. To take advantage of these features in systems that rely on dynamic programming, several special-purpose techniques have been proposed. For example, Hong proposed a 2-step approach to optimize queries for parallel databases [HS91], and such a 2-step approach was adopted for distributed query optimization in the Mariposa project [SAL+96]. Hellerstein (among others) proposed heuristics to optimize queries with expensive predicates [HS93, Hel94, CS96], and Chaudhuri and Shim proposed ways to reduce the complexity of dynamic

programming for aggregate queries [CS94]. None of these techniques guarantee optimal plans because they consider apriori only a small fraction of the search space of all possible plans to execute a query. `IDP` takes a different and more generic approach: to begin with, `IDP` always considers the entire search space, and only if the search space turns out to be too big, `IDP` switches and applies heuristics in order to find a good plan with acceptable effort.

3 Query Optimization by Dynamic Programming

In this section, we will describe the *classic* dynamic programming algorithm for query optimization. Specifically, we will give the basic algorithm, show how it can be extended to optimize queries in distributed database systems, and analyze its complexity. For the purpose of presentation, we will only consider select-project-join queries (SPJ queries) here and in the following sections. Dynamic programming can, however, be applied to optimize any kind of query; e.g., aggregate queries [CS94], queries with expensive predicates [CS96], *top N* queries [CK97], or queries in heterogeneous database systems [HKWY97]. Also, we will, like everybody else, define *plan* to be a tree of query operators such as *table scan, index scan, sort, nested-loop join*, etc.

3.1 Algorithm Description

Figure 1 gives the dynamic programming algorithm. As stated in previous sections, the algorithm is used in most database systems today, and it has already been described in several different articles; e.g., [GHK92]. The algorithm works in a bottom-up way as follows. First, dynamic programming generates so-called *access plans* for every table involved in the query (Lines 1 to 4 in Figure 1). Typically, such an *access plan* consists of one or two operators, and there are several different *access plans* for a table. Examples are *table_scan(Emp)* to directly read the *Emp* table or *idx_scan(Emp.salary)* to read the *Emp* table using an index on the *salary* field of the *Emp* table.

In the second phase (Lines 5 to 14 in Figure 1), dynamic programming considers all possible ways to join the tables. First, it considers all 2-way join plans by using the *access plans* of the tables as building blocks and calling the *joinPlans* function to build a join plan from these building blocks. From the 2-way join plans and the *access plans*, dynamic programming then produces 3-way join plans. After that, it generates 4-way join plans by considering all combinations of two 2-way join plans and all combinations of a 3-way join plan with an *access plan*. In the same way, dynamic programming continues to produce 5-way, 6-way join plans and so on up to n-way join plans. In the third phase (Lines 14 and 15), the n-way join plans are massaged by the *finalizePlans* function so that they become complete plans for the query; e.g., *project*, *sort*, or *group-by* operators are attached, if necessary (Line 14). Note that dynamic programming uses in every step of the second phase the same *joinPlans* function to produce more and more complex plans using simpler plans as building blocks. Just as there are usually several alternative *access plans*, there are usually several different ways to join two tables (e.g., nested loop joins, hash joins, etc.) and the *joinPlans* function will return a plan for every alternative join method.

Input: SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan for q

```
1:  for i = 1 to n do {
2:      optPlan({R_i}) = accessPlans(R_i)
3:      prunePlans(optPlan({R_i}))
4:  }
5:  for i = 2 to n do {
6:      for all S ⊆ {R_1, ..., R_n} such that |S| = i do {
7:          optPlan(S) = ∅
8:          for all O ⊂ S do {
9:              optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O), optPlan(S − O))
10:             prunePlans(optPlan(S))
11:         }
12:     }
13: }
14: finalizePlans(optPlan({R_1, ..., R_n}))
15: prunePlans(optPlan({R_1, ..., R_n}))
16: return optPlan({R_1, ..., R_n})
```

Figure 1: (Classic) Dynamic Programming Algorithm

The beauty of dynamic programming is that it discards inferior building blocks after every step. This approach is called *pruning* and is carried out by the *prunePlans* function (Lines 3, 10, and 15 of Figure 1). While enumerating 2-way join plans, for example, dynamic programming would consider an $A \bowtie B$ plan and a $B \bowtie A$ plan, but only the cheaper of the two plans would be retained in *optPlan*$(\{A, B\})$ so that only the cheaper of the two plans would be considered as a building block for 3-way, 4-way, ... join plans involving A and B. Pruning is possible because the $A \bowtie B$ plan and the $B \bowtie A$ plan do the sam work; if the $A \bowtie B$ plan is cheaper than the $B \bowtie A$ plan, then any complete plan for the whole query that has $A \bowtie B$ as a building block (e.g., $C \bowtie (A \bowtie B)$) will be cheaper than the same plan with $B \bowtie A$ as a building block (e.g., $C \bowtie (B \bowtie A)$). As a result of pruning, dynamic programming does not enumerate inferior plans such as $C \bowtie (B \bowtie A)$ and runs significantly faster than a naive exhaustive search.

It should be noted, however, that there are situations in which two, say, $A \bowtie B$ plans are *incomparable* and must both be retained in the *optPlan*$(\{A, B\})$ structure, even though one plan is more expensive than the other plan. For example, A *sort-merge-join* B and A *hash-join* B are incomparable if the *sort-merge-join* is more expensive than the *hash-join* and if the ordering of the results produced by the *sort-merge join* is *interesting* [SAC$^+$79, SSM96]; in this case the ordering of the results might help to reduce the cost of later operations (e.g., group-bys or joins with other tables). All the final plans are comparable so that only one plan will be retained after the final pruning in Line 15 of the algorithm and only a single plan will be returned as output of the algorithm.

Another point to notice is that the algorithm of Figure 1 enumerates all *bushy* plans. With slight modifications of Line 6 of the algorithm, however, the algorithm could also be used to enumerate only so-called *left-deep* plans, as originally proposed in [SAC$^+$79]. There has been a great deal of discussion about "left-deep vs. bushy" (see, e.g., [SMK97]), and we

use the "bushy" variant of dynamic programming in this work because most commercial query optimizers that are based on dynamic programming do the same thing [GLSW93].

3.2 Dynamic Programming for Distributed Databases

One of the nice properties of dynamic programming is that query optimizers that are built using dynamic programming can easily be extended. In the following, we will show how the algorithm can be extended in order to optimize queries for distributed database systems. Similar extensions for other types of queries have been proposed in [CS94, CS96, CK97, HKWY97]. Our extensions are along the lines of the extensions proposed for the query optimizer of the System R* project [Loh88]; however, we do present a novel pruning technique which significantly speeds up query optimization without sacrificing the quality of the generated plans.

In addition to deciding which access paths (i.e., indices), which join order, and which join methods to use, the query optimizer of a distributed system must decide *where* to carry out all the operations. To make this decision, the *accessPlans*, *joinPlans*, and *finalizePlans* functions must be extended as follows:

- if a table is replicated, the *accessPlans* function must generate different access plans for every site at which the table is replicated; e.g., *table_scan(Emp, Passau)*, *idx_scan(Emp.salary, Passau)*, or *table_scan(Emp, Maryland)* if the *Emp* table is replicated in Passau and Maryland, and there is an *Emp.salary* index in Passau.
- the *joinPlans* function must generate different join plans in order to specify that a join can be carried out at the site at which the outer table is produced, at the site at which the inner table is produced, and at all other *interesting sites*. (We will describe the concept of *interesting sites* below.)
- the *finalizePlans* functions must attach a *ship* operator if the top-level operator of a plan is not executed at the site at which the results of the query must be returned (i.e., the client).

Furthermore, we need to adjust the *prunePlans* function and be careful when we prune plans that produce their results at different sites. If, for example, the *Emp* table is stored in Passau and Maryland and the *Dept* table is stored in Passau only, then we may not always prune the *table_scan(Emp, Passau)* access plan in an $Emp \bowtie Dept$ query even if it is more expensive than the *table_scan(Emp, Maryland)* access plan because the *table_scan(Emp, Passau)* access plan might be a building block of the best overall plan that carries out the join in Passau. In general, we need to consider all *interesting sites* when we decide which plans may be pruned. The concept is similar to the concept of *interesting orders* [SAC+79, SSM96] and is defined as follows: for an access or join plan involving tables $R_{i_1}, \ldots, R_{i_k}$ of a query involving tables $R_1, \ldots, R_n$, all those sites are interesting that store a copy of R_j for all $j \notin \{i_1, \ldots, i_k\}$. In addition, the site at which the query results must be returned (i.e., the client) is interesting. Given this definition of *interesting sites*, an access or join plan P_1 may be pruned if there exists an access or join plan P_2 which must involve the same tables and the following criteria holds:

$$\forall i \in \mathit{interesting\_sites}(P_1) : \mathbf{cost}(\mathit{ship}(P_1, i)) \geq \mathbf{cost}(\mathit{ship}(P_2, i)) \quad (1)$$

The *ship* operator sends tuples from one site to another. Naturally, the cost of a *ship* operator is 0 if the source and target sites are identical; e.g.,

$$\textbf{cost}(ship(table\_scan(Emp, Passau), Passau) = \textbf{cost}(table\_scan(Emp, Passau)).$$

Equation (1) can be evaluated very easily if the network is homogeneous; i.e., if the cost to ship data between any two sites is identical. Another simple case that can be tested first is that P_1 can be pruned if the following equation holds:

$$\textbf{cost}(P_1) \geq \textbf{cost}(ship(P_2, x)) \tag{2}$$

if x is the site at which P_1 produces its results.

3.3 Complexity of Dynamic Programming

It has been shown in [OL90, VM96] that the time complexity of dynamic programming is $\mathcal{O}(3^n)$ and the space complexity is $\mathcal{O}(2^n)$ in a centralized system. In the following, we will show that in a distributed system the time complexity of dynamic programming is $\mathcal{O}(s^3 * 3^n)$ and the space complexity is $\mathcal{O}(s * 2^n + s^3)$, where s is the number of sites at which a copy of at least one of the tables involved in the query is stored plus the site at which the query results need to be returned. s, thus, is a variable whose value depends on the query and which might be smaller or larger than n, depending on the number of replicas of the tables used in the query.

Time Complexity of Dynamic Programming: The time complexity of dynamic programming is $\mathcal{O}(s^3 * 3^n)$ in a distributed database system.

Proof: The idea is to show that dynamic programming enumerates in every step at most s^3 times as many plans in a distributed as in a centralized system. To show this, we consider the worst case in which all tables involved in a query are fully replicated at s sites. As a result, s sites are *interesting* after every step of dynamic programming. If we, furthermore, assume that no pruning can be carried out using the condition of Equation (1), then we can conclude that every entry of the *optPlan* structure contains s times as many plans in a distributed system than in a centralized system. Now, let us look at the number of plans the *joinPlans* function generates whenever it is called (Line 9 of Figure 1). Furthermore, let us assume that the system supports only one join method and that there are no *interesting orders*. (Several join methods or the presence of interesting orders would introduce constants and variables which are irrelevant for this proof.) Under these circumstances, every entry of the *optPlan* structure contains exactly one plan and the *joinPlans* function generates exactly one plan in a centralized system. In a distributed system, every *optPlan* entry contains at most s plans and the *joinPlans* function generates at most s^3 plans because it combines (at most) s plans of $optPlan(O)$ with (at most) s plans of $optPlan(S - O)$ and it must consider all s interesting sites for the join. For $s = 2$, for example, the *joinPlans* function would enumerate the following eight plans:

$$O_1 \bowtie_1 I_1,\ O_1 \bowtie_1 I_2,\ O_2 \bowtie_1 I_1,\ O_2 \bowtie_1 I_2,\ O_1 \bowtie_2 I_1,\ O_1 \bowtie_2 I_2,\ O_2 \bowtie_2 I_1,\ O_2 \bowtie_2 I_2,$$

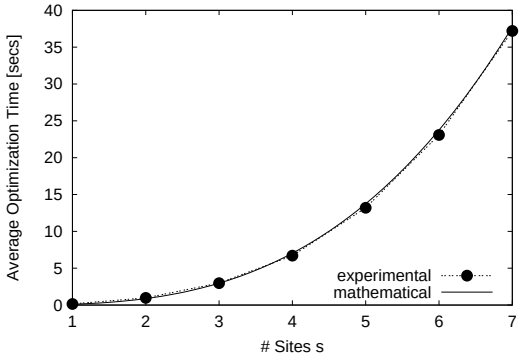


Figure 2: Running Time of (Classic) `DP`
10-way `STAR` Query

where O_i represents the ith plan of $optPlan(O)$, I_i the ith plan of $optPlan(S-O)$, and $\bowtie_x$ denotes that the join is executed at site x. Thus, as we intended to show, dynamic programming enumerates at most s^3 times as many plans in a distributed as in a centralized system. □

Space Complexity of Dynamic Programming: The space complexity of dynamic programming is $\mathcal{O}(s * 2^n + s^3)$ in a distributed database system.

Proof: We already observed that every *optPlan* entry contains at most s times as many plans in a distributed as in a centralized system. So, the whole *optPlan* data structure keeps $\mathcal{O}(s * 2^n)$ plans. Furthermore, dynamic programming needs space for $\mathcal{O}(s^3)$ plans for calls of the *joinPlans* function. In all, therefore, dynamic programming needs space for $\mathcal{O}(s * 2^n + s^3)$ plans. □

Figure 2 shows experimentally how the running time of dynamic programming grows with the number of sites. The query is a ten-way STAR join query and all the tables are replicated at all sites. It becomes clear that the running time of dynamic programming grows cubically with the number of sites, as predicted in the analysis. Furthermore, it becomes clear that a ten-way STAR join query can be optimized very fast in a centralized database system (for $s = 1$, the running time is 0.16 sec) whereas in a distributed system with many sites, the running time to optimize the same query can become prohibitively high. Thus, we do need alternative ways to optimize queries in a distributed database system.

4 Greedy Algorithms for Query Optimization

As an alternative to dynamic programming, greedy algorithms have been proposed [Pal74, Swa89, SYT93]. These greedy algorithms run much faster than dynamic programming, but they typically produce worse plans [SMK97]. In this section, we will describe the basic variant of such greedy algorithms. As we will show in the next section, `IDP` can be classified as a generalization of dynamic programming and this basic greedy algorithm with the goal to combine the advantages of both.

Input: SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan for q

1: **for** $i = 1$ **to** n **do** {
2: optPlan($\{R_i\}$) = accessPlans(R_i)
3: prunePlans(optPlan($\{R_i\}$))
4: }
5: toDo = $\{R_1, \ldots, R_n\}$
6: **for** $i = 1$ **to** $n - 1$ **do** {
7: **find** O, $I \in$ todo, $P \in$ joinPlans(optPlan(O), optPlan(I) **such that**
eval(P) = min{eval(P') | $P' \in$ joinPlans(optPlan(O'), optPlan(I')); $O, I \in$ todo}
8: **generate** new symbol: $\mathcal{T}$
9: optPlan($\{\mathcal{T}\}$) = $\{P\}$
10: todo = todo – $\{O, I\} \cup \{\mathcal{T}\}$
11: delete(optPlan(O)), delete(optPlan(I))
12: }
13: finalizePlans(optPlan($\{R_1, \ldots, R_n\}$))
14: prunePlans(optPlan($\{R_1, \ldots, R_n\}$))
15: **return** optPlan($\{R_1, \ldots, R_n\}$)

Figure 3: Greedy Algorithm

4.1 Algorithm Description

Figure 3 shows the basic greedy algorithm for query optimization. Just like dynamic programming, this greedy algorithm has three phases and constructs plans in a bottom-up way. This greedy algorithm also makes use of the same *accessPlans, joinPlans*, and *finalizePlans* functions in order to generate plans. In the second phase, however, this greedy algorithm carries out a very simple and rigorous selection of the join order. With every iteration of the *greedy loop* (Lines 5 to 11), this greedy algorithm applies a *plan evaluation function*, *eval*, in order to select the next best join. As an example for a five-way join query with tables A, B, C, D, and E, the plan evaluation function could determine that A and D should be joined first; the result of $A \bowtie D$ should be joined with C next; B and E should be joined next; finally, the results of $C \bowtie (A \bowtie D)$ and $B \bowtie E$ should be joined. Obviously, the quality of plans produced by this greedy algorithm strongly depends on the plan evaluation function. We will describe alternative plan evaluation functions in Section 6.

4.2 Complexity of the Greedy Algorithm

Time Complexity of the Greedy Algorithm The time complexity of the greedy algorithm is $\mathcal{O}(n^3)$ in a centralized system and $\mathcal{O}(n^3 * s^3)$ in a distributed system.

Proof: Finding the *best* O and I in every iteration of the greedy loop can be carried out in $\mathcal{O}(n^2)$ steps. Since the greedy loop is carried out $n - 1$ times, the whole algorithm is carried out in at most $\mathcal{O}(n^3)$ steps. As mentioned in Section 3.3, the *joinPlans* function generates at most $\mathcal{O}(s^3)$ plans for each pair of *optPlan* entries in a distributed system,

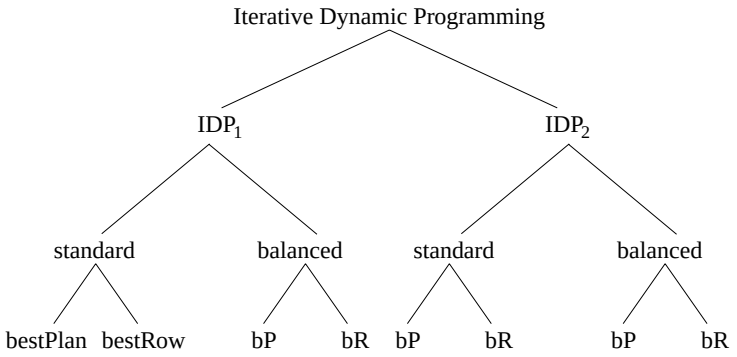


Figure 4: Iterative Dynamic Programming Variants

resulting in an overall complexity of $\mathcal{O}(n^3 * s^3)$ in a distributed system. □
For certain plan evaluation functions, it is possible to find the best O and I in constant time if the tables or predicates of the query are sorted in advance. This is possible, for instance, for the *Minimum Selectivity* plan evaluation function described in Section 6.1. In such cases, the greedy algorithm has a time complexity of $\mathcal{O}(n * \log n)$ for sorting.

Space Complexity of the Greedy Algorithm The space complexity of the greedy algorithm is $\mathcal{O}(n)$ in a centralized system and $\mathcal{O}(n * s)$ in a distributed system.

Proof: At the beginning, the greedy algorithm keeps at most $\mathcal{O}(s * n)$ access plans in the *optPlan* structure. As the algorithm progresses, the space requirements decrease.

5 Iterative Dynamic Programming

We are now ready to present IDP, a new class of query optimization algorithms that is based on iteratively applying dynamic programming and can be seen as a combination of dynamic programming and the greedy algorithm presented in the previous section. In all, we will describe eight different IDP variants that differ in three ways: (1) when an iteration takes place (IDP_1 vs. IDP_2), (2) the size of the building blocks generated in every iteration (*standard* vs. *balanced*), and (3) the number of building blocks produced in every iteration (*bestPlan* vs. *bestRow*). Figure 4 shows these eight different variants. As in the previous sections, we will concentrate on *select-project-join* queries (SPJ queries) in this section for ease of presentation and note that all IDP variants can be applied to optimize any query, just as dynamic programming or a greedy algorithm.

5.1 IDP_1

5.1.1 Algorithm Description

We will begin and show how IDP_1 works. In fact, we will describe "IDP_1-standard-bestPlan" (cf. Figure 4) here and describe the other variants in Sections 5.3 and 5.4.
IDP_1 works essentially in the same way as dynamic programming with the only difference that IDP_1 respects that the resources (e.g., main memory) of a machine are limited or that

Input: SPJ query q on relations $R_1, \ldots, R_n$, maximum block size k
Output: A query plan for q

```
1:  for i = 1 to n do {
2:      optPlan({R_i}) = accessPlans(R_i)
3:      prunePlans(optPlan({R_i}))
4:  }
5:  toDo = {R_1, ..., R_n}
6:  while |toDo| > 1 do {
7:      k = min{k, |toDo|}
8:      for i = 2 to k do {
9:          for all S ⊆ toDo such that |S| = i do {
10:             optPlan(S) = ∅
11:             for all O ⊂ S do {
12:                 optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O), optPlan(S − O))
13:                 prunePlans(optPlan(S))
14:             }
15:         }
16:     }
17:     find P, V with P ∈ optPlan(V), V ⊆ toDo, |V| = k such that
            eval(P) = min{eval(P') | P' ∈ optPlan(W), W ⊆ toDo, |W| = k}
18:     generate new symbol: T
19:     optPlan({T}) = {P}
20:     toDo = toDo − V ∪ {T}
21:     for all O ⊆ V do delete(optPlan(O))
22: }
23: finalizePlans(optPlan(toDo))
24: prunePlans(optPlan(toDo))
25: return optPlan(toDo)
```

Figure 5: Iterative Dynamic Programming ($\texttt{IDP}_1$) with Block Size k

a user or application program might want to limit the time spent for query optimization. To see how $\texttt{IDP}_1$ does this, let us assume, as an example, that a machine has enough memory to keep all *access plans*, 2-way, 3-way, ..., k-way *join plans* (after pruning) for a query with n tables, but no more than that. Let us also assume that $n > k$. In such a situation, dynamic programming would crash or be the cause of severe paging of the operating system when it starts to consider $(k+1)$-way *join plans* because at this point the machine's memory is exhausted. $\texttt{IDP}_1$, on the other hand, would generate *access plans* and all 2-way, 3-way, ..., k-way *join plans* like dynamic programming, but rather than starting to generate $(k+1)$-way *join plans*, $\texttt{IDP}_1$ would *break* at this point, select one of the k-way *join plans*, discard all other *access* and *join plans* that involve one of the tables of the selected plan, and restart in order to build $(k+1)$-way, $(k+2)$-way, ... *join plans* using the selected plan as a building block. That is, just like the greedy algorithm breaks after two-way join plans have been enumerated, $\texttt{IDP}_1$ breaks after k-way join plans have been enumerate, the memory is full, or a time-out is hit. The complete $\texttt{IDP}_1$ algorithm is shown in Figure 5, and we will describe it more closely using the following example.

The example is a five-way join query and $k = 3$. Figure 6 shows the six steps that $\texttt{IDP}_1$ would take to optimize this query. In the first three steps, $\texttt{IDP}_1$ works just like classic dynamic programming; i.e., $\texttt{IDP}_1$ generates *access plans*, 2-way and 3-way *join plans*, keeping the best plans in the *optPlan* structure and discarding inferior plans. In this example, $\texttt{IDP}_1$ *breaks* in Step 4 after noticing that, e.g., the memory is exhausted. In this step, $\texttt{IDP}_1$ behaves like a greedy algorithm: it applies a "plan evaluation function," *eval*, to all 3-way *join plans*, selects the plan with the lowest value of *eval*, and discards all other plans that involve one or more tables considered in the selected plan. Specifically, $\texttt{IDP}_1$ selects the $\{A, B, D\}$ plan, and it discards the three *access plans* for A, B and D, the $A \bowtie B$, $C \bowtie A$, $E \bowtie D$ plans and all other 2-way *join plans* involving at least one of A, B or D, and it discards the $(A \bowtie B) \bowtie C$ and $(E \bowtie D) \bowtie C$ plans and all other 3-way *join plans* with A, B or D. Then, $\texttt{IDP}_1$ restarts dynamic programming for the 3-way join optimization problem with C, E, and $\mathcal{T}$, where $\mathcal{T}$ represents the (temporary) table produced by the selected $\{A, B, D\}$ plan. In Step 5, $\texttt{IDP}_1$ considers all 2-way *join plans* with $\{\mathcal{T}, C, E\}$ and in Step 6, $\texttt{IDP}_1$ considers all 3-way *join plans*. The 3-way *join plans* generated in Step 6 are actually 5-way join plans and after applying the *finalizePlans* function, the plan with lowest cost found in Step 6 is returned as the result of the optimization process. □

Before we continue, we would like to make a couple of comments:

1. If desired, k, the parameter of $\texttt{IDP}_1$, can be set by a user to limit the optimization time or as an optimization level, but k need not be set by a user or a system administrator. $\texttt{IDP}_1$ can find out itself when the memory is exhausted and, therefore, when to *break*. Impatitient users who halt the optimization process implicitly set k at the time they press the `STOP` button. Obviously, the higher k (more resources), the better are the plans generated by $\texttt{IDP}_1$ because $\texttt{IDP}_1$ becomes a better match to full-fledged (classic) dynamic programming.

2. It is possible that $\texttt{IDP}_1$ requires more than two iterations to optimize a query; if $n = 10$ and $k = 4$, for example, $\texttt{IDP}_1$ requires 3 iterations. (With every iteration, the optimization problem is reduced by $k - 1 = 3$ tables.) If $k \geq n$, $\texttt{IDP}_1$ needs only one iteration; in this case, $\texttt{IDP}_1$ behaves exactly like dynamic programming. If $k = 2$, $\texttt{IDP}_1$ behaves exactly like the greedy algorithm.

3. It should become clear from the example shown in Figure 6 that $\texttt{IDP}_1$ carries out the bulk of its work in its first iteration and that $\texttt{IDP}_1$ finds a plan very quickly after its first *break*. In particular, $\texttt{IDP}_1$ can reuse plans that were generated in the first iteration. In Figure 6, for instance, the *access plans* for C and E, and the 2-way *join plan* for $C \bowtie E$ were generated in the first iteration and need not be generated again in the second iteration. We will quantify this observation in Section 7.2.

4. Evidently, just like the greedy algorithm, the quality of plans returned by $\texttt{IDP}_1$ strongly depends on the plan evaluation function used to select plans. We will present alternative plan evaluation functions in Section 6. As we will see in Section 7.3, however, $\texttt{IDP}_1$ is able to produce good plans in most situations even with very simple plan evaluation function, if k is sufficiently large. This is not true for a greedy algorithm (i.e., $\texttt{IDP}_1$ with $k = 2$).

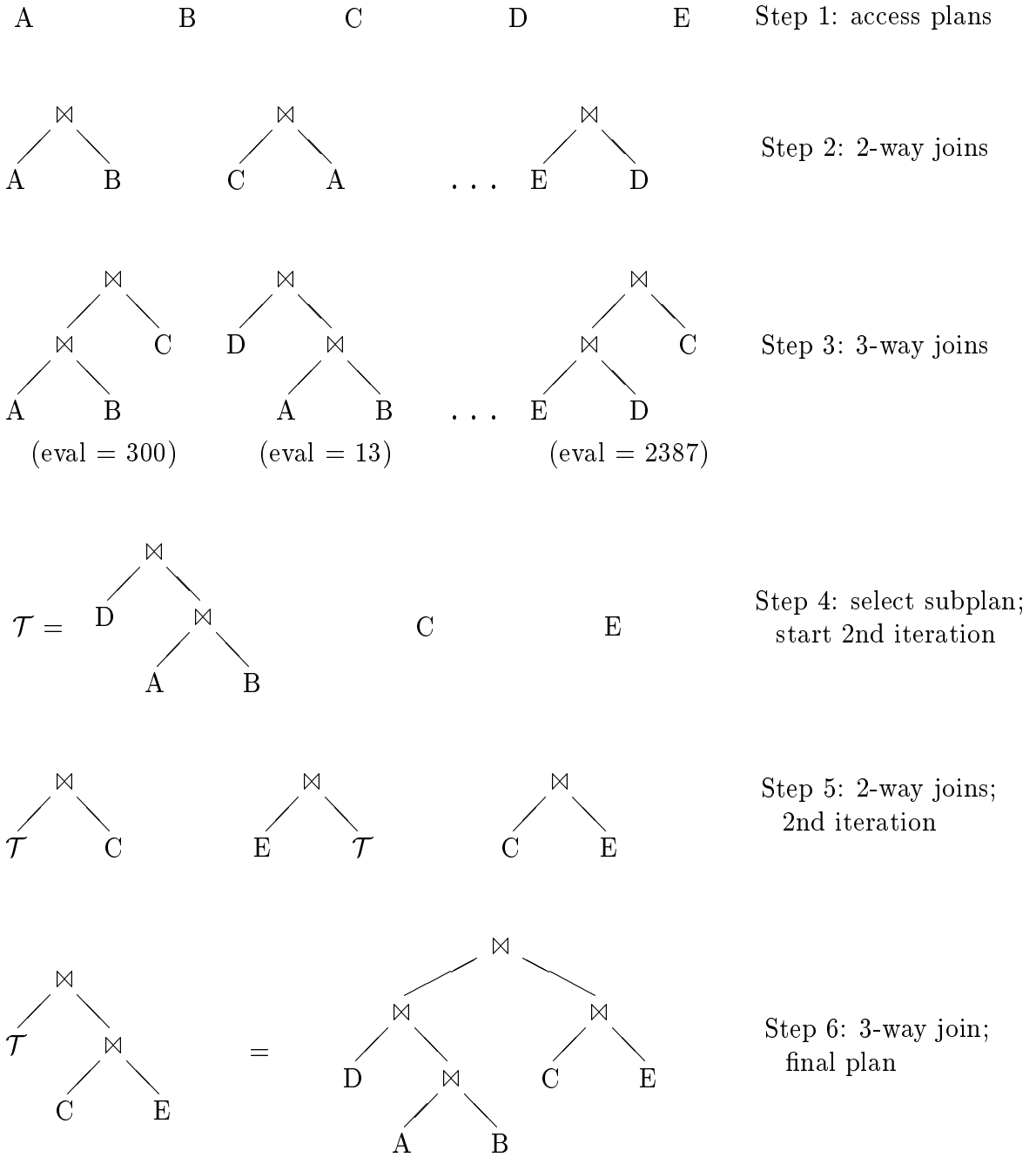


Figure 6: Optimizing a 5-Way Join Query with Block Size $k = 3$

5. Comparing Figures 1 and 5, we note that it is easy to extend an existing dynamic programming-based query optimizer to become an $\texttt{IDP}_1$-based optimizer: only a couple of lines of code need to be added or changed. The implementation of the *accessPlans*, *joinPlans*, *finalizePlans*, and *prunePlans* functions which are typically quite complex need not be changed at all. Also, $\texttt{IDP}_1$ can be applied in the same way as dynamic programming (and a greedy algorithm) in order to optimize queries in a distributed database system because all the necessary adjustments to deal with distributed databases are encapsulated in the *accessPlans*, *joinPlans*, *finalizePlans*, and *prunePlans* functions (Section 3.2).

5.1.2 Complexity

For $k = 2$, $\texttt{IDP}_1$ behaves exactly like the greedy algorithm analyzed in Section 4.2 and for $k = n$, $\texttt{IDP}_1$ behaves like dynamic programming (Section 3.3). In the following, we will therefore analyze the complexity of $\texttt{IDP}_1$ for $2 < k < n$. We will show that the $\texttt{IDP}_1$ algorithm of Figure 5 has polynomial time and space complexity of the order of $\mathcal{O}(s^3 * n^k)$. In this analysis, k (the size of the building blocks) is considered to be constant, and s (the number of sites) and n (the number of tables) are the variables which depend on the query to optimize.

Time Complexity of $\texttt{IDP}_1$ in a Centralized Database: In a centralized database system, the time complexity of the $\texttt{IDP}_1$ algorithm (Figure 5) is of the order of $\mathcal{O}(n^k)$ for $2 < k < n$.

Proof: The complete proof can be found in the Appendix of this paper. The basic idea is to show (1) that the first iteration of the $\texttt{IDP}_1$ algorithm can be carried out in $\mathcal{O}(n^k)$ steps and (2) that all the other iterations combined can also be carried out in $\mathcal{O}(n^k)$ steps. □

For $k = 2$ and $k = 3$, $\texttt{IDP}_1$ has the same, cubic complexity. In these cases, finding the plan with the minimum value (Line 17) has as high or higher complexity as enumerating subplans (Lines 8 to 15).

Time Complexity of $\texttt{IDP}_1$ in a Distributed Database: In a distributed database system, the time complexity of the $\texttt{IDP}_1$ algorithm is of the order of $\mathcal{O}(s^3 * n^k)$ for $2 < k < n$.

Proof: The proof goes along the lines of the proof for the time complexity of dynamic programming (Page 8). It can be shown that $\texttt{IDP}_1$ enumerates in every step at most s^3 times as many plans in a distributed system as in a centralized systems. □

Space Complexity of $\texttt{IDP}_1$: In a distributed database system, the space complexity of the $\texttt{IDP}_1$ algorithm is of the order of $\mathcal{O}(s * n^k + s^3)$ for $2 < k < n$.

Proof: This follows easily from the proof for the space overhead of dynamic programming (Page 9) and the proof for the time complexity of $\texttt{IDP}_1$ in a distributed database system. □

5.2 $\texttt{IDP}_2$

5.2.1 Algorithm Description

Figure 7 shows the $\texttt{IDP}_2$ algorithm. In fact, the figure shows the "standard-bestPlan" variant of this algorithm, and we defer the discussion of the other variants to Sections 5.3 and 5.4. This basic variant of the algorithm was originally proposed by Shekita and Young [SY98], and a similar idea to apply dynamic programming in order to re-optimize certain parts of a plan has also been proposed in form of the *bushhawk algorithm* by Vance [Van98].

The algorithm works as follows: in every iteration, the algorithm applies a variant of the greedy algorithm described in Section 4.1 in order to find k (or more tables) which should be joined early (Lines 7 to 16 of Figure 7). After that, the $\texttt{IDP}_2$ algorithm applies dynamic programming, as described in Section 3, in order to find a good plan for the tables selected by the greedy algorithm (Line 22). After that, the algorithm continues to optimize the processing of the temporary table (denoted $\mathcal{T}$) generated by the subplan produced by dynamic programming and all the other tables of the query by iteratively applying greedy heuristics and dynamic programming until a complete plan for the query is found. It should be noted that the $\texttt{IDP}_2$ algorithm actually uses the greedy algorithm to find more than k tables that should be joined in the same subplan and then breaks up this set of tables so that dynamic programming is applied to at most k tables (Lines 18 to 21). In this respect, other variants of the $\texttt{IDP}_2$ algorithm are conceivable; we chose this variant because it coincides with the proposal by [SY98].

Comparing $\texttt{IDP}_1$ and $\texttt{IDP}_2$, we observe that the mechanisms are essentially the same: both algorithms apply heuristics (i.e., plan evaluation functions) in order to select subplans, and both algorithms make use of dynamic programming. Also, both algorithms can (fairly) easily be integrated into an existing optimizer which is based on dynamic programming. The difference between the two algorithms is that $\texttt{IDP}_2$ makes heuristic decisions *a priori* and applies dynamic programming after that; $\texttt{IDP}_1$, on the other hand, starts with dynamic programming and makes heuristic decisions *a posteriori*, only when it is necessary. In other words, $\texttt{IDP}_1$ is adaptive and k is an optional parameter of the algorithm which may or may not be set by a user in order to limit the optimization time. For $\texttt{IDP}_2$, k must be set before the algorithm starts. As we will see in the next subsection, another difference is that $\texttt{IDP}_2$ has lower asymptotic complexity than $\texttt{IDP}_1$. We will study the quality of plans produced by $\texttt{IDP}_1$ and $\texttt{IDP}_2$ in Section 7.

5.2.2 Complexity

In the following, we will give upper bounds for the time and space complexity of the $\texttt{IDP}_2$ algorithm in a distributed database system. Again, we consider s and n as variables which depend on the query and k as a constant which must be set by a user or system administrator.

Input: SPJ query q on relations $R_1, \ldots, R_n$, maximum block size k
Output: A query plan for q

```
1:  for i = 1 to n do {
2:      optPlan({R_i}) = accessPlans(R_i)
3:      prunePlans(optPlan({R_i}))
4:  }
5:  toDo = {R_1, ..., R_n}
6:  while |toDo| > 1 do {
7:      // apply greedy algorithm to find a good building block
8:      bblocks = ∅
9:      for all V ∈ toDo do {
10:         find P ∈ optPlan(V) such that eval(P) = min{eval(P') | P' ∈ optPlan(V)}
11:         bblocks = bblocks ∪ {P}
12:     }
13:     do {
14:         find L, R ∈ bblocks, P ∈ joinPlans(L, R) such that
                eval(P) = min{eval(P') | L', R' ∈ bblocks, P' ∈ joinPlans(L', R')}
15:         bblocks = bblocks ∪ {P} − {L, R}
16:     } while P involves no more than k tables and |bblocks| > 1
17:     // reoptimize the bigger of L and R, selected in the last iteration of the greedy loop
18:     if (L involves more tables than R)
19:         reopTables = {tables involved in L}
20:     else
21:         reopTables = {tables involved in R}
22:     P = dynamic_programming(reopTables)
23:     generate new symbol: T
24:     optPlan({T}) = {P}
25:     toDo = toDo ∪ {T} − reopTables
26:     for all O ⊆ V do delete(optPlan(O))
27: }
28: finalizePlans(optPlan(toDo))
29: prunePlans(optPlan(toDo))
30: return optPlan(toDo)
```

Figure 7: Iterative Dynamic Programming ($\texttt{IDP}_2$) with Block Size k

Time Complexity of $\texttt{IDP}_2$: The time complexity of the $\texttt{IDP}_2$ algorithm is of the order of $\mathcal{O}(n * (n^2 + s^3 * 3^k))$.

Proof: First, we note that the $\texttt{IDP}_2$ algorithm carries out $\mathcal{O}(n)$ iterations. (In the best case, $\lceil \frac{n}{k-1} \rceil$ iterations are necessary because the number of tables in the toDo list are reduced by $k-1$ tables in the best case.) In every iteration, the *greedy loop* requires $\mathcal{O}(n^2)$ steps and dynamic programming enumerates $\mathcal{O}(s^3 * 3^k)$ plans, as shown in the proof for the time complexity of dynamic programming (Page 8). □
For large n, $\texttt{IDP}_2$ therefore, has a complexity of $\mathcal{O}(n^3)$ which is lower than $\mathcal{O}(n^k)$, the complexity of $\texttt{IDP}_1$, for $k > 3$.

Space Complexity of $\texttt{IDP}_2$: The space complexity of the $\texttt{IDP}_2$ algorithm is of the order of $\mathcal{O}(n + s * 2^k + s^3)$.

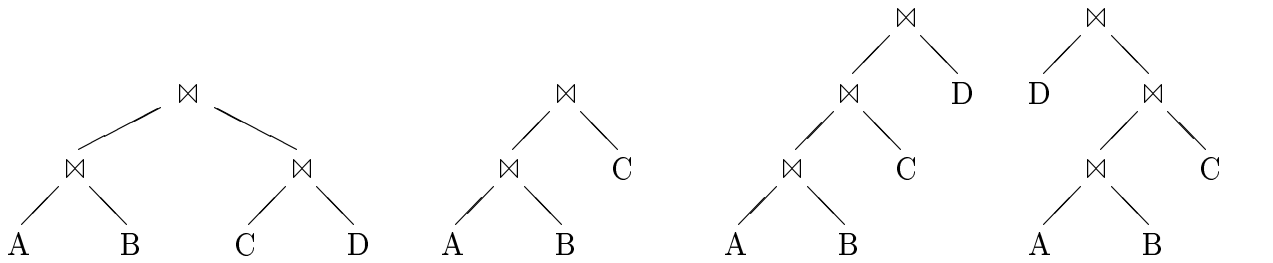
Proof: Inbetween iterations (i.e., for the *toDo* list) and for the greedy loop (i.e., for the *bblocks* structure), we need to keep $\mathcal{O}(n)$ plans. Furthermore, we need space for $\mathcal{O}(s * 2^k + s^3)$ plans to apply dynamic programming in every iteration, as shown in the proof for the space complexity of dynamic programming (Page 9). □
For large n, therefore, $\texttt{IDP}_2$ has linear space complexity which is better than $\mathcal{O}(n^k)$, the space complexity of $\texttt{IDP}_1$.

5.3 Standard vs. Balanced Iterative Dynamic Programming

In this section, we will describe two variants of $\texttt{IDP}_1$ and $\texttt{IDP}_2$ that we call *standard* and *balanced* and that differ in the restrictions they impose on the size of the selected building blocks. The first variant, *standard*, imposes no restrictions; in fact, this is the variant described in Figures 5 and 7. The second variant, *balanced*, restricts the size of the selected building blocks in two ways:

1. the number of tables involved in the selected subplans must be even
2. the number of tables involved in the selected subplans must be less or equal to $\lceil \frac{d}{2} \rceil$, where d is the number of tables in the *toDo* list.

So, for a 10-way join query and $k = 7$, the *balanced* variant of the $\texttt{IDP}_1$ algorithm would enumerate all 2-way, 3-way, ... and 7-way *join plans*, break, select a 4-way *join plan*, and construct plans for the whole query in its second iteration. Likewise, the *balanced* variant of the $\texttt{IDP}_2$ algorithm would make sure that the dynamic programming step is only applied to produce 2-way or 4-way *join plans* for a query that involves ten tables; that is, the algorithm would break up L or R which are selected in the greedy loop to meet the two restrictions.
The motivation to consider *balanced* variants of the $\texttt{IDP}_1$ and $\texttt{IDP}_2$ algorithms is demonstrated by Figure 8. Figure 8a shows the optimal plan for an example 4-way join query. Now let us assume that $k = 3$ and we use the $\texttt{IDP}_1$ algorithm (the same observations hold for the $\texttt{IDP}_2$ algorithm). The standard $\texttt{IDP}_1$ algorithm (as of Figure 5) enumerates all 3-way *join subplans*, breaks, and selects, say, the 3-way *join subplan* shown in Figure 8b.



(a) Optimal Plan (b) Selected Subplan (c) Complete Plans from Selected Subplan

Figure 8: Example: Restrict the Size of Selected Subplans

After restart, the $\texttt{IDP}_1$ algorithm constructs complete plans for the query using this subplan as a building block, but there is no way for the optimizer to construct the optimal plan from the selected 3-way *join subplan*. (All the complete plans that the optimizer is able to construct are shown in Figure 8c.) In fact, for $k = 3$ and $n = 4$, the standard $\texttt{IDP}_1$ algorithm is a-priori never able to produce a bushy plan like the one shown in Figure 8a, regardless of which 3-way *join plan* is selected when the algorithm breaks. The *balanced* variant of the $\texttt{IDP}_1$ algorithm would select a 2-way (rather than 3-way) *join plan* and is, thus, able to consider all bushy plans, including the bushy plan shown in Figure 8a. So, the *balanced* variants of $\texttt{IDP}_1$ and $\texttt{IDP}_2$ are likely to produce better plans than the *standard* variants in situations in which bushy plans are desirable. On the negative side, the *standard* variants might produce better plans in other situations because they consider bigger building blocks. We will quantify these observations in Section 7.2.

5.4 Selecting Plans or Rows

In this section, we will describe two further variants: *bestPlan* and *bestRow*. Both variants can be combined with *standard* $\texttt{IDP}_1$ and $\texttt{IDP}_2$ as well as with *balanced* $\texttt{IDP}_1$ and $\texttt{IDP}_2$ so that we have, in all, eight different IDP variants, as shown in Figure 4. *bestPlan* specifies that only a single plan is selected in every iteration. For $\texttt{IDP}_1$ this means that whenever the algorithm breaks, the *join plan* with the smallest value according to the plan evaluation function is retained and all other *join plans* that involve the same number of tables are discarded. For $\texttt{IDP}_2$, *bestPlan* specifies that a single plan is kept in every step of the *greedy* loop of the algorithm and a single plan is produced in the dynamic-programming step. *bestPlan* is the variant used in the algorithms of Figures 5 and 7.

bestRow extends *bestPlan* by retaining a whole entry of the *optPlan* structure; recall that every such entry contains several incomparable plans involving the same tables. To integrate this strategy into the $\texttt{IDP}_1$ algorithm, we simply need to change Line 19 of the $\texttt{IDP}_1$ algorithm shown in Figure 5 to

$$\text{optPlan}(\{\mathcal{T}\}) = \text{optPlan(V)}$$

That is, all the plans of the *optPlan* entry of Plan P are retained rather than just keeping Plan P which is the *join plan* with the minimum value of the plan evaluation function. For $\texttt{IDP}_2$, we need to change the call to dynamic programming so that dynamic programming produces a set of incomparable plans (Line 22 of the algorithm of Figure 7). In a distributed system, for example, we call dynamic programming in such a way that it

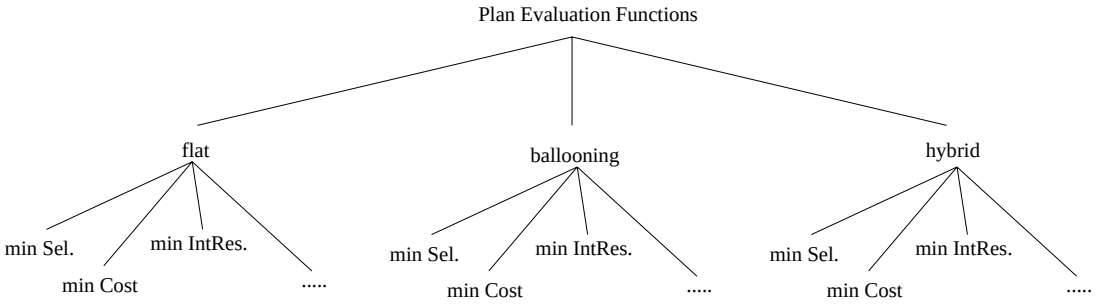


Figure 9: Alternative Plan Evaluation Functions

produces different plans for all *interesting sites* of the selected subplan. (*interesting sites* as defined in Section 3.2.)

The tradeoffs of the *bestPlan* and *bestRow* variants are fairly straightforward, and we will quantify them in Section 7.2. Obviously, the *bestRow* variants produce better plans than the *bestPlan* variants because they consider more plans. On the negative side, the running time and memory requirements of the *bestRow* variants are higher, just for the same reason.

It should be noted that a third variant that we call *bestRows* is conceivable for the $\texttt{IDP}_1$ algorithm. This variant would keep, say, 10 entries of the *optPlan* structure or 1% of all the entries of the *optPlan* structure whenever $\texttt{IDP}_1$ breaks. We will not study this variant in more detail in this paper for the following two reasons: (1) using this *bestRows* variant, the complexity of the $\texttt{IDP}_1$ algorithm increases sharply because only few plans can be discarded whenever the algorithm breaks; and (2) we could not find a way to produce better plans with such a *bestRows* variant than with the (plain) *bestRow* variant.

6 Selecting Good Subplans

We will now turn to the question of how to select a good subplan which is important for all IDP variants. Obviously, we are not shooting for perfect decisions (i.e., finding subplans which are guaranteed to be part of an optimal plan for the whole query), since making such perfect decisions is just as hard as the original problem of finding an optimal plan for the query. So, instead, we will try to select "promising" subplans which are likely to be part of a good plan for the query. In this section, we will present three classes of plan evaluation functions that can be used by all eight IDP variants for this purpose. The first class consists of very simple functions that are very easy to implement and can be evaluated in constant time; we refer to this class as *flat* plan evaluation functions. The functions of the second class are more difficult to implement and have higher complexity; we refer to the technique exploited by this class of functions as *ballooning*. The *third* class of plan evaluation functions, termed *hybrid*, combines the ideas of the two other classes. Within each class, we will look at three different plan evaluation functions; many more are conceivable. The whole spectrum of plan evaluation functions is depicted in Figure 9.

6.1 Flat Plan Evaluation Functions

Flat plan evaluation functions evaluate a subplan by looking at the properties of the subplan only. In this work, we consider the following three different flat plan evaluation functions:

Minimum Cost: This plan evaluation function chooses the subplan with the smallest (estimated) cost.

Minimum Intermediate Result: This plan evaluation function chooses the subplan with the smallest (estimated) result cardinality.

Minimum Selectivity: This plan evaluation function chooses the subplan by the *selectivity* of the predicates applied in the subplan; i.e.,

$$eval(\text{subplan}) = \frac{\text{result cardinality}}{|R_{s_1}| * |R_{s_2}| * \ldots |R_{s_k}|}$$

if the subplan involves tables $R_{s_1}, \ldots R_{s_k}$.

Of course, many other plan evaluation functions are conceivable; e.g., functions that combine the *cost* of a subplan with the *result cardinality*. We chose these plan evaluation functions because of their simplicity and because these plan evaluation functions have been used at different occasions in previous work. *Min. Selectivity* is, for example, used in [Pal74, SMK97]; *Min. Intermediate Result* is used in [SYT93]; and we know of one commercial database product that uses a *Min. Cost* plan evaluation function. We also experimented with other, new plan evaluation functions, but none of these new plan evaluation functions showed significantly better performance than the best of these three plan evaluation functions.

6.2 Ballooning

The idea of ballooning is to peek into the future in order to evaluate a subplan; that is, ballooning quickly generates a complete plan from the subplan and uses the cost of this complete plan as a metric to evaluate the subplan.[2] This way, `IDP` can choose a subplan and be fairly sure that a low-cost complete plan can be constructed from it.

Obviously, there are many ways to generate a complete plan from a subplan, but we want to emphasize again that we are interested in generating such a plan very quickly because all `IDP` variants need to evaluate a large number of subplans. To quickly generate a complete plan, we chose to apply the greedy algorithm described in Section 4.1. That is, given a subplan, we add join by join to the subplan until we have a complete plan, and with every step a flat plan evaluation function decides which join to add to the subplan. For the *bestPlan* `IDP` variants, we balloon every plan of an *optPlan* entry individually using exactly the same code as in the greedy loop of the $\texttt{IDP}_2$ algorithm and a flat plan evaluation function. For the *bestRow* variants, we balloon all the plans of an *optPlan*

[2]The name "ballooning" was adopted from [KMP93] where a technique was proposed to quickly construct a complete plan from a subplan. Our "ballooning" should, however, not be confused with their "ballooning"; the two are different techniques used for different purposes.

entry simultaneously and keep whole *optPlan* entries (i.e., set of incomparable plans) rather than individual plans in every step of the *greedy loop*. Again, we experimented with a *Min. Cost*, *Min. Intermediate Result*, and *Min. Selectivity* flat plan evaluation function for this purpose.

It should be noted that the use of ballooning increases the complexity of `IDP`. Ballooning with a *Min. Cost* or *Min. Intermediate Result* plan evaluation function can be carried out in $\mathcal{O}(n^3)$ steps. Ballooning with a *Min. Selectivity* plan evaluation function is cheaper and can be carried out in $\mathcal{O}(n)$ steps because the predicates of a query can be sorted at the beginning of the optimization process so that the next join during ballooning can be selected in constant time (Section 4.2). In all, therefore, the time complexity of the $\texttt{IDP}_1$ variants in a distributed database system increase to $\mathcal{O}(s^3 * n^{k+1})$ if ballooning with *Min. Selectivity* is used and to $\mathcal{O}(s^3 * n^{k+3})$ if ballooning with *Min. Cost* or *Min. Intermediate Result* is used. The complexity of the $\texttt{IDP}_2$ variants increase to $\mathcal{O}(n^4)$ (*Min. Selectivity*) or $\mathcal{O}(n^6)$ (*Min. Cost or Min. Intermediate Result*) because ballooning would be applied to every plan considered in the *greedy loop* in the $\texttt{IDP}_2$ algorithm. The space complexity of the $\texttt{IDP}_1$ and $\texttt{IDP}_2$ algorithms is not affected by ballooning because all subplans can be ballooned using the same (small) piece of memory.

To summarize: `IDP` with ballooning has certainly got a higher running time than flat `IDP`. Ballooning is also more difficult to implement, although most of the code used for ballooning can be reused. On the positive side, ballooning is likely to make better decisions than any flat plan evaluation function. Another advantage of ballooning is that the cost of the ballooned, complete plans can be used to prune subplans, at any time; this advantage might in many situations compensate for the additional overhead of ballooning. (The advantages of having complete plans early in the optimization process have been studied in more detail in [VM96].)

6.3 Hybrid Plan Evaluation Functions

Hybrid plan evaluation functions work in the following way:

1. use a flat plan evaluation function to select x subplans (for *bestPlan* variants) or x *optPlan* entries (for *bestRow* variants); here, x is a tuning parameter which must be set by a user or system administrator

2. apply ballooning to these x subplans (or *optPlan* entries) in order to select a single subplan (or *optPlan* entry, respectively); during ballooning the same or a different (flat) plan evaluation function as in the first step can be used

Obviously, hybrid plan evaluation functions have higher complexity than flat plan evaluation functions and lower complexity than full-fledged ballooning. At the same time, it should be expected that they produce better plans than flat plan evaluation functions and worse plans than full-fledged ballooning.

7 Performance Experiments and Results

In this section, we will discuss the results of experiments that assess the running times and the quality of the plans produced by the different `IDP` variants, dynamic programming

```
SELECT *
FROM   R0, R1, ..., R9
WHERE  R0.a0 = R1.i ∧ R1.i = R2.a1 ∧
       R2.a2 = R3.i ∧ R3.i = R4.a4 ∧
       R4.a5 = R5.i ∧ R5.i = R6.a5 ∧
       R6.a6 = R7.a7 ∧ ...∧ R8.a9 = R9.a
```

Figure 10: 10-way Join Chain Query

```
SELECT *
FROM   R0, R1, ..., R9
WHERE  R0.i = R1.i ∧
       R0.i = R2.i ∧
       R0.i = R3.i ∧
       R0.a4 = R4.a ∧ ...∧ R0.a9 = R9.a
```

Figure 11: 10-way Join Star Query

(as described in Section 3), greedy algorithms (i.e., IDP-bestPlan with $k = 2$), and a randomized algorithm called 2PO which is today the best known algorithm to process very complex queries. An extensive set of experiments with other query optimization algorithms is reported in [SMK97]. That paper shows that 2PO outperforms these other algorithms; so we will use 2PO as a baseline for our comparisons in this work.

7.1 Experimental Environment

To assess the various IDP variants, dynamic programming, and 2PO, we used the two 10-way join queries shown in Figures 10 and 11. The first query is a CHAIN query, and the second query is a STAR query. In both queries, three of the nine joins are carried out with a *shared* join column (denoted as i) while the other six joins are carried out with separate join columns. The presence of a *shared* join column favors the use of sort-merge joins since R_0 needs to be sorted only once to join it with R_1, R_2, and R_3. The presence of *shared* join columns also makes it more complicated to optimize these queries because $R_0.i$ is an *interesting order* for all subplans that involve R_0, but do not involve all of R_1, R_2, R_3; as a result, sort-merge join and hash-join plans might be incomparable and neither of them may be pruned (see Section 3.1). We also carried out experiments with 20-way CHAIN and STAR queries; in those queries, six of the nineteen joins were carried out with *shared* and *interesting* columns.

It should be noted that most modern query processors would rewrite queries with *shared* join columns by adding predicates such as "$R_1.i = R_3.i$" (for the STAR query) to the WHERE clause in order to have more flexibility during query optimization [GLSW93]. This rewrite changes the join topology so that the queries no longer are pure CHAIN or STAR queries. We experimented with such rewritten queries, but we saw the same effects as in the experiments with the pure (un-rewritten) CHAIN and STAR queries, so we will only show the results obtained using those pure CHAIN and STAR queries here. Of course, we also experimented with other join topologies (e.g., CYCLE+3 or CLIQUE [VM96]) and queries without *shared* join columns since such experiments are part of the standard repertoire to validate a query optimizer. Again, we will not show the results of these experiments here because they only confirmed the findings.

We studied all queries with 100 different settings for the cardinality of the base tables and the selectivity of the join predicates. These settings were made randomly following the approach proposed in [SMK97]. Using this approach, we were able to study a large range of different scenarios: queries with small and large base tables, low selectivity queries that produce many results, high selectivity queries that produce few results, and everything in between. Extending the approach proposed in [SMK97], we also carried

out experiments with distributed database configurations in which some of the tables were replicated. A distributed environment significantly increases the size of the search space and the complexity of the algorithms in a similar way as the presence of expensive predicates, group-bys or top N. If not stated otherwise, we measured a distributed database configuration with three sites in which three (random) tables were replicated at all sites and the other tables were not replicated and distributed round-robin among the three sites. Again, replicating and distributing the tables made the query optimization problem more complex because *interesting sites* made alternative plans incomparable, as described in Section 3.2. We also carried out other experiments in which we varied the number of sites, and we will present those experiments as well. We also carried out experiments in which we varied the degree of replication, but we will not present the results of those experiments in this paper because these experiments did not reveal any new effects.

To validate the quality of plans, we used the optimizer's cost model rather than a real query engine. This way, we were able to isolate the merits and deficiencies of the individual algorithms. The cost model we used is an extension of the cost model used in [SMK97]: the cost formulae for the different join methods, assumptions on buffer management, etc. are identical, and we added a simple cost formula (as of [ML86]) to model the costs of shipping tables and intermediate results from one site to another. That is, the cost of communication is estimated as *bytes sent / bandwidth*. The cost model parameters for CPU and disk IO costs were set in the same way as proposed by [SMK97]; the cost model parameter for the network costs (i.e., bandwidth) was set to model a wide-area network with ISDN communication links. We also experimented with other parameter settings that model fast local-area networks, but we will not present the results of those experiments here because these experiments showed the same effects. The plans we generated consisted of *scan*, *sort*, *ship*, *nested-loop join*, *hash join*, and *merge join* operators. Throughout this section, we will report on the *average scaled cost* of the plans produced by the algorithms. For the 10-way join queries, we applied dynamic programming to find the 100 optimal plans for every query and the 100 different base cardinality and selectivity settings, and then we applied the different `IDP` variants and `2PO` in order to find out how much, on an average, their plans were more expensive than the optimal plans. To compute the *average scaled cost* for the 20-way join queries, the *scaled cost* of every plan was computed using the best plan produced by `2PO` or any `IDP` variant for the same parameter settings as a baseline because the 20-way join queries could not be optimized using dynamic programming. Since the *average scaled cost* results alone are often misleading because they tend to be biased to a couple of extremely bad cases, we will also report how often `IDP` and `2PO` produced *good, acceptable*, and *bad* plans using the definitions of [Swa91]; i.e., a plan is *good* if its scaled cost is less than 2, a plan is *acceptable* if its scaled cost is greater than 2 but less than 10, and a plan is *bad* if its scaled cost is greater than 10. In addition to the quality of the plans, we measured the (average) running time of dynamic programming (if applicable), all `IDP` variants, and `2PO` using UNIX's *getrusage* system call on a Sun Ultra with a 167 MHz SPARC processor.

An important point to note is that we implemented dynamic programming, the eight `IDP` variants, and `2PO` in such a way that no plans with Cartesian products are enumerated. We did this because most commercial optimizers today do the same thing. Considering Cartesian products would have increased the running times of the optimizer and, in a few cases, it would have also impacted the quality of the plans produced by the optimizer.

7.2 The Right IDP Variants

7.2.1 Running Time

Before we compare IDP with dynamic programming and 2PO, we would like to know which of the eight IDP variants is the best one. To this end, we will start and study the running times of the eight IDP variants. Figures 12 and 13 show the running times of the four $\texttt{IDP}_1$ and the four $\texttt{IDP}_2$ variants, respectively, varying k and using our default database configuration with three sites and three tables replicated at all three sites. In these and the following experiments of this subsection we use a flat *Minimum Intermediate Result* plan evaluation function (see Section 6.1). The reason for using this particular plan evaluation function will become clear in Section 7.3, but the choice of the plan evaluation function does not play a significant role for the observations we are going to make in this subsection. Looking at Figures 12 and 13, we see that the running time of all IDP variants increases with k for all variants. For $k = 10$, all $\texttt{IDP}_1$ variants have the same running time as dynamic programming and all $\texttt{IDP}_2$ variants have a slightly higher running time because of the extra *greedy loop*. Recall that the basic greedy algorithm described in Section 4 is identical with $\texttt{IDP}_1$-standard-bestPlan with $k = 2$. Looking more closely at the figures, we can make the following observations:

1. In all cases, the *standard* variants have as high or higher running times than the *balanced* variants. The reason for this is that, in effect, the *balanced* variants limit the size of k in certain cases. For example, *balanced* $\texttt{IDP}_1$ with $k = 5$ behaves exactly like *balanced* $\texttt{IDP}_1$ with $k = 4$ and is therefore cheaper than *standard* $\texttt{IDP}_1$ with $k = 5$.

2. The *bestRow* variants are usually slightly more expensive than the *bestPlan* variants, but they are never significantly more expensive. Likewise, the bestRow variants have only slightly higher space requirements (not shown). The gap between the *bestRow* and *bestPlan* variants is so small because the *bestRow* variants do very little extra work: rather than enumerating more plans, *bestRow* can be characterized as discarding less plans than *bestPlan*.

3. For large k and $k < 10$, *standard* $\texttt{IDP}_2$ is much faster than *standard* $\texttt{IDP}_1$; the *balanced* $\texttt{IDP}_1$ and $\texttt{IDP}_2$ variants, however, have fairly much the same running time. Recall from Sections 5.1 and 5.2 that we expect $\texttt{IDP}_2$ to have lower running times than $\texttt{IDP}_1$ for large queries with large k; for a 10-way join query using *balanced* $\texttt{IDP}_1$ and $\texttt{IDP}_2$, this advantage of $\texttt{IDP}_2$ just does not become apparent. We will study the running times of $\texttt{IDP}_1$ and $\texttt{IDP}_2$ in more detail in Section 7.4.

One effect which is not shown in Figures 12 and 13 is that the standard deviation of the running times is quite high for all the $\texttt{IDP}_2$ variants whereas the standard deviation is almost 0 for all the $\texttt{IDP}_1$ variants. (The figures only show the *average* running times of the IDP variants.) The running times of the $\texttt{IDP}_2$ variants depend on the shape of the produced plan: if the plan is *deep* the running time will be higher than if the plan is *bushy* because $\texttt{IDP}_2$ will call dynamic programming with larger building blocks for *deep* plans. The shape of the produced plans strongly depends on the cardinality of the tables and the selectivity of the join predicates and cannot be predicted in advance.

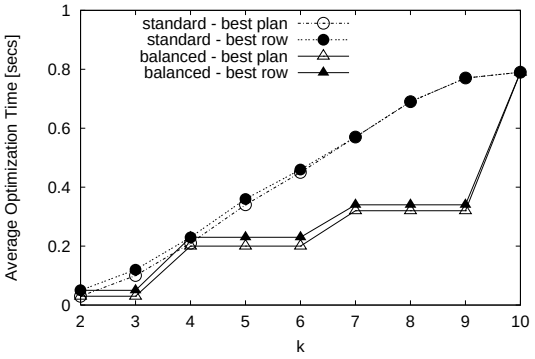


Figure 12: Running Time of $\texttt{IDP}_1$ vary k, 10-way `CHAIN` Query

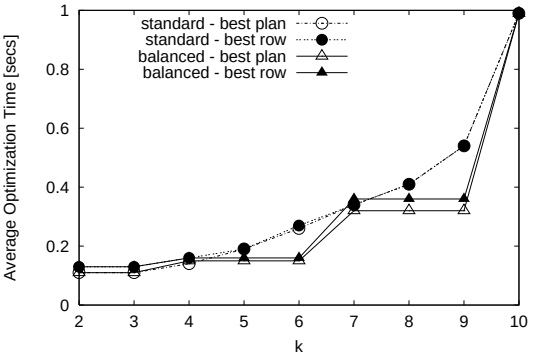


Figure 13: Running Time of $\texttt{IDP}_2$ Vary k, 10-way `CHAIN` Query

k	*Total*	*1st Iteration*	*All Other Iterations*
2	0.05sec	40%	60%
3	0.12sec	64%	36%
4	0.23sec	74%	26%
5	0.36sec	95%	5%
6	0.46sec	98%	2%
7	0.57sec	99%	1%
8	0.69sec	100%	0%
9	0.77sec	100%	0%
10	0.79sec	100%	0%

Figure 14: Running Time of First vs. All Other Iterations
$\texttt{IDP}_1$-standard-bestRow, Vary k, 10-way `CHAIN` Query

One of the interesting properties of the $\texttt{IDP}_1$ variants is that those variants find a complete plan for a query very quickly after their first iteration. This effect is demonstrated in Figure 14. It becomes clear that for $k \geq 5$ the portion of time spent for the 2nd and all other iterations is negligible compared to the effort spent in the first iteration. Furthermore it becomes clear that for all k, $\texttt{IDP}_1$ spends no more than a fraction of a second to complete a plan after the first iteration. In practice, this behavior of $\texttt{IDP}_1$ is important because it means that it is possible to press the `STOP` button at any time and $\texttt{IDP}_1$ will break and virtually immediately return a complete plan for the query.
Of course, we also studied the running times of the eight `IDP` variants for the 10-way join `STAR` query. We do not show the results in detail here because they demonstrate the same effects. The difference is that `STAR` queries are significantly more difficult to optimize than `CHAIN` queries. For example, it would have taken more than six seconds rather than 0.8 seconds to optimize the `STAR` query using dynamic programming or $\texttt{IDP}_1$ with $k = 10$.

7.2.2 Quality of Plans

Figures 15 and 16 show the average scaled cost (in logscale) of the plans produced by the four $\texttt{IDP}_1$ variants for the 10-way join `STAR` and `CHAIN` queries, varying k and using again

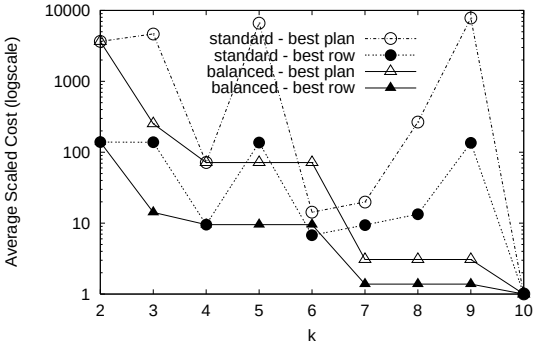


Figure 15: Quality of Plans, $\texttt{IDP}_1$
Vary k, 10-way `STAR` Query

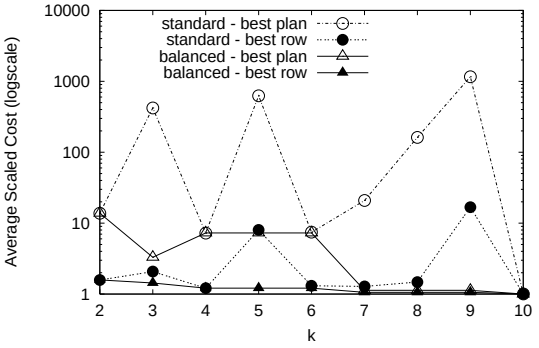


Figure 16: Quality of Plans, $\texttt{IDP}_1$
Vary k, 10-way `CHAIN` Query

our default distributed database configuration. We can see that the *balanced-bestRow* variant is the clear winner. For the `STAR` query, it produces very good plans for $k \geq 7$, and for the `CHAIN` query, it produces very good plans for all k. The other variants produce in many cases plans which are orders of magnitudes more costly than the plans produced by the *balanced-bestRow* variant. Recall again that the basic greedy algorithm of Section 4 corresponds to $\texttt{IDP}_1$-standard-bestPlan with $k = 2$. In general, we can make the following two observations:

- the *bestRow* variants produce significantly better plans than the *bestPlan* variants
- the *balanced* variants produce significantly better plans than the *standard* variants; this observation demonstrates the importance to consider all bushy plans during query optimization.

One particular feature of the *balanced* variants is that these variants produce better and better plans, the larger k gets; that is, these variants guarantee that the plans get better and better the more resources are available or the more resources a user is willing to invest into the optimization process. The *standard* variants have not got this property.
Figures 17 and 18 assess the quality of plans produced by the four $\texttt{IDP}_2$ variants. We observe essentially the same effects: (1) the *bestRow* variants produce better plans than the *bestPlan* variants, and (2) the *balanced* variants produce better plans than the *standard* variants. A little surprisingly, the difference between the *bestRow* and *bestPlan* variants are particularly pronounced for $\texttt{IDP}_2$ whereas the gap between *balanced* and *standard* is negligible in most cases. In all, however, we can safely conclude that *balanced-bestRow* is the best variant for $\texttt{IDP}_1$ and $\texttt{IDP}_2$ because it has low running time and produces the best plans. Note that the algorithm proposed by Shekita and Young [SY98] corresponds to the $\texttt{IDP}_2$-standard-bestPlan variant which is the overall worst variant of $\texttt{IDP}_2$.

7.3 The Best Plan Evaluation Functions

Having studied the eight `IDP` variants (including the basic greedy algorithm), let us now see what the best plan evaluation function is. Figures 19 and 20 show the quality of plans produced by the $\texttt{IDP}_1$-balanced-bestRow variant for `STAR` and `CHAIN` queries using the three alternative flat plan evaluation functions described in Section 6.1. We observe that

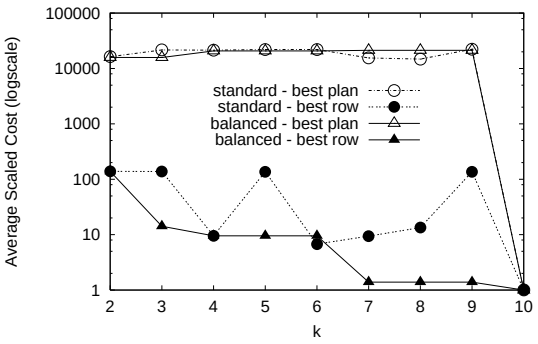


Figure 17: Quality of Plans, $\texttt{IDP}_2$
Vary k, 10-way `STAR` Query

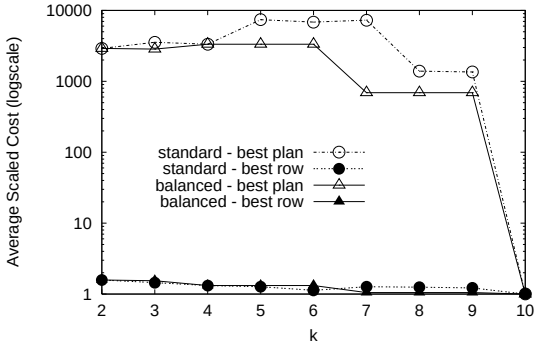


Figure 18: Quality of Plans, $\texttt{IDP}_2$
Vary k, 10-way `CHAIN` Query

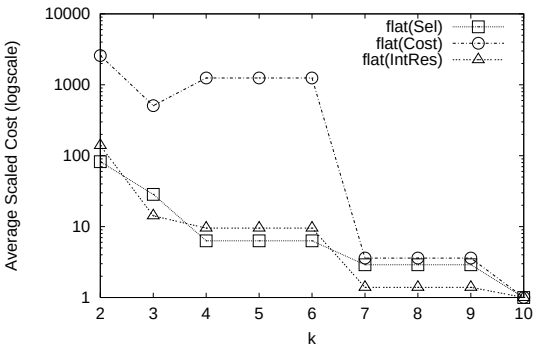


Figure 19: Quality of Plans, $\texttt{IDP}_1$
Vary k, 10-way `STAR` Query

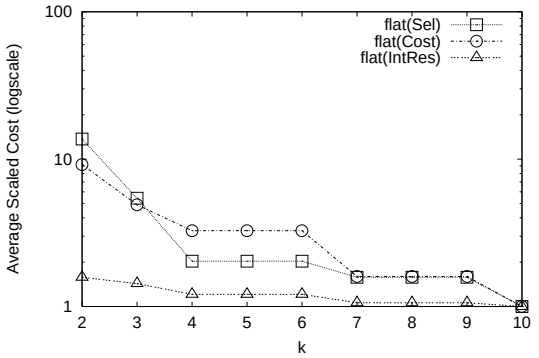


Figure 20: Quality of Plans, $\texttt{IDP}_1$
Vary k, 10-way `CHAIN` Query

$\texttt{IDP}_1$ generates the overall best plans if *Min. Intermediate Result* is used as a criterion. We experimented with many different plan evaluation functions, combined the plan evaluation functions with all `IDP` variants and looked at different queries and distributed database configurations, and in all cases *Min. Intermediate Result* was the winner or not much worse than the winner. *Min. Selectivity* is only competitive for `STAR` queries and *Min. Cost* is never good.

More interesting is the comparison of flat vs. ballooning vs. hybrid plan evaluation functions. Figure 21 shows the running time of `IDP` (more specifically, $\texttt{IDP}_1$-balanced-bestRow) using different evaluation functions along this dimension. As expected, full-fledged ballooning has the highest running time, flat has the lowest running time, and hybrid is somewhere in between. Looking closer, we can see that full-fledged ballooning has an almost as high running time as dynamic programming (`IDP` with $k = 10$) for $k \geq 4$; in fact, `IDP` with full-fledged ballooning can become even more expensive than dynamic programming so that this variant is not very attractive. We can, however, also see that a hybrid plan evaluation function that balloons 5% of the *optPlan* entries incurs only a slight running time overhead (less than 20%) compared to a flat plan evaluation function, so that a hybrid plan evaluation function is very well affordable in terms of running time. Turning to the quality of plans produced by the different variants (Figure 22), we see that a hybrid plan evaluation function is also very good to help `IDP` select good subplans. Obviously, the best plans are produced with full-fledged ballooning, but hybrid plan evaluation functions produce just as good plans for $k \geq 4$. Flat plan evaluation functions

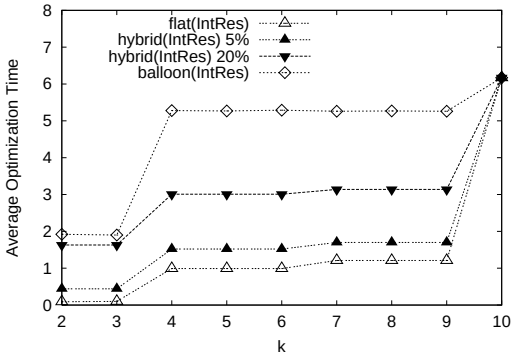


Figure 21: Running Time, $\texttt{IDP}_1$
Vary k, 10-way STAR Query

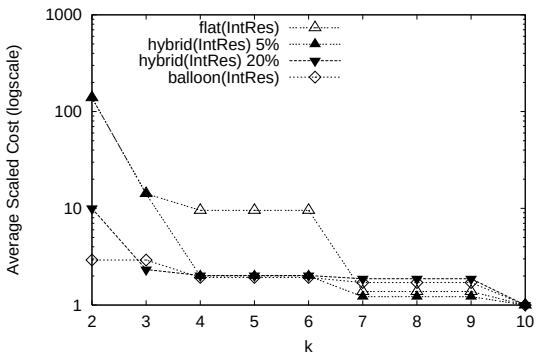


Figure 22: Quality of Plans, $\texttt{IDP}_1$
Vary k, 10-way STAR Query

only produce good plans for large k ($k \geq 7$ in this case).

7.4 IDP vs. Dynamic Programming vs. 2PO

We are now ready to compare IDP with dynamic programming and 2PO, the best known algorithm to optimize very complex queries for which dynamic programming is not viable [SMK97]. Specifically, we will study the $\texttt{IDP}_1$-balanced-bestRow and $\texttt{IDP}_2$-balanced-bestRow variants and use a flat *Min. Intermediate Result* plan evaluation function. To implement 2PO, we used the neighbor functions (plus a neighbor function for "site selection" [FJK96]) and parameter settings proposed in [IK90]. We also experimented with different parameter settings, but we were not able to improve the results of 2PO with different parameter settings. We will first present the results we obtained with 10-way join queries and then the results we obtained with 20-way join queries.

7.4.1 10-Way Join Queries

Running Time Let us first look at the running times of the different algorithms for 10-way CHAIN queries, varying the number of sites at which copies of the tables are stored (Figure 23). We already showed in Section 3.3 that the running time of dynamic programming grows cubically with the number of sites and the same observation holds for both IDP variants; the curves for the IDP variants, however, are significantly flatter than for dynamic programming. The running time of 2PO is almost independent of the number of sites. It should be noted, however, that 2PO has the highest running time for a centralized system. In this case, the running time of 2PO is even higher than that of dynamic programming. In terms of running time, the virtues of 2PO only become apparent in a distributed system and for queries that potentially involve many different sites or if n is larger than 10.

Figure 24 lists the running times of the different algorithms in more detail, including the running times for STAR queries and the running times of $\texttt{IDP}_1$ and $\texttt{IDP}_2$ for different k. It becomes clear that dynamic programming is prohibitively expensive for STAR queries in distributed systems, and that the differences in running time between $\texttt{IDP}_2$ and $\texttt{IDP}_1$ are particularly pronounced for STAR queries.

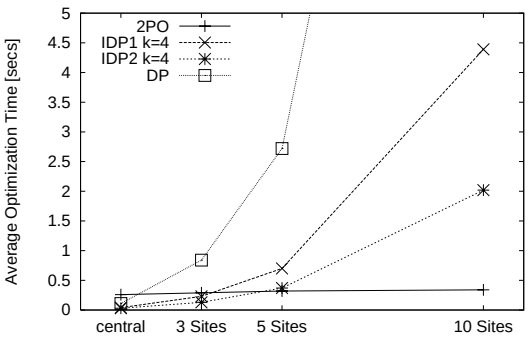


Figure 23: Running Time, 2PO, IDP, DP Vary s, 10-way CHAIN Query

Sites s	Chain $s=1$	Chain $s=10$	Star $s=1$	Star $s=10$
2PO	0.26	0.33	0.32	0.35
IDP_1, $k=2$	0.01	0.63	0.02	1.24
IDP_1, $k=4$	0.04	4.39	0.13	19.1
IDP_1, $k=7$	0.06	7.20	0.16	21.2
IDP_2, $k=2$	0.03	1.38	0.04	2.24
IDP_2, $k=4$	0.03	2.02	0.04	2.45
IDP_2, $k=7$	0.04	6.16	0.08	7.04
Dyn. Prog.	0.11	19.1	0.67	107.8

Figure 24: Running Time, 2PO, IDP, DP Vary s, 10-way Join Queries

Quality of Plans Figures 25 to 28 assess the quality of plans produced by the different algorithms for the 10-way CHAIN and STAR queries, varying the number of sites and k for IDP_1 and IDP_2. The plots show the *average scaled costs* and the tables list the number of *good, acceptable,* and *bad plans* produced by the algorithms. As a general trend, the plans produced by 2PO become worse with an increasing number of sites: while at least 70% of the plans produced by 2PO are good in a centralized system, only about every third plan can be classified as *good* in a distributed system. In other words, the plans produced by 2PO get worse the more difficult it is to optimize a query. In all, these experiments show that 2PO is not an attractive algorithm for query optimization: in a centralized system, 2PO often produces good plans, but it is slow (in some cases, even slower than dynamic programming); in a distributed system, 2PO is fast, but it only rarely produces good plans. Both IDP_1 and IDP_2 produce reasonably good plans for $k \geq 4$ in these experiments. For $k = 7$, the plans are almost perfect. Comparing IDP_1 and IDP_2, IDP_1 is typically as good or slightly better than IDP_2 for CHAIN queries and both are almost identical for STAR queries. Of course, dynamic programming is always perfect (scaled cost of 1 and 100% good plans).

Graceful Degradation One of the arguments in favor of randomized algorithms such as 2PO is that such algorithms degrade gracefully. That is, 2PO produces plans very quickly and it finds better and better plans the longer it runs. Dynamic programming and IDP_2 do not have this property: these two algorithms do not produce plans until the very end, making it impossible for a user to "press a STOP button" in the middle of the optimization process. As shown in Section 7.2, however, IDP_1 also degrades gracefully: at any time during the optimization process, it is possible to press the STOP button and IDP_1 will return a plan within fractions of a second. Also, the plans produced by IDP_1 will become better and better the later the STOP button is pressed, if a *balanced* IDP_1 variant is used. This is the main argument in favor of IDP_1 in comparison to IDP_2.

Figures 29 and 30 show how quickly 2PO and IDP_1 produce good plans. To carry out this experiment, we ran these two algorithms for the 100 different STAR and CHAIN queries, pressed the STOP button at different times, and computed the average scaled cost. The two figures show that IDP_1 always wins, regardless of when the STOP button is pressed. In

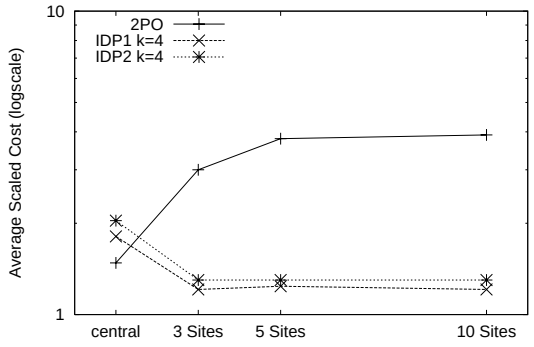


Figure 25: Average Scaled Cost (logscale) Vary s, 10-way CHAIN Query

Sites s	$s=1$			$s=3$			$s=10$		
	G	A	B	G	A	B	G	A	B
2PO	92	7	1	57	41	2	31	64	5
IDP_1, $k=2$	80	15	5	86	13	1	84	15	1
IDP_1, $k=4$	91	7	2	93	7	0	94	6	0
IDP_1, $k=7$	99	1	0	98	2	0	98	2	0
IDP_2, $k=2$	80	15	5	86	13	1	84	15	1
IDP_2, $k=4$	89	7	4	94	5	1	92	8	0
IDP_2, $k=7$	99	1	0	99	1	0	99	1	0
Dyn. Prog.	100	0	0	100	0	0	100	0	0

Figure 26: Good, Acceptable, Bad Plans Vary s, 10-way CHAIN Query

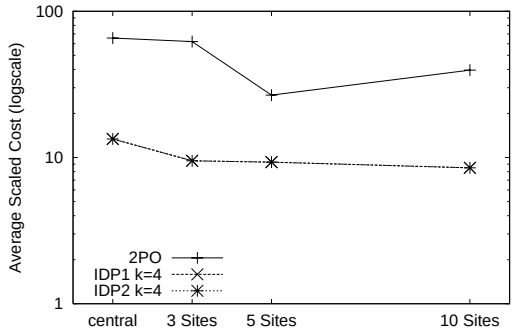


Figure 27: Average Scaled Cost (logscale) Vary s, 10-way STAR Query

Sites s	$s=1$			$s=3$			$s=10$		
	G	A	B	G	A	B	G	A	B
2PO	73	9	18	31	54	15	34	45	21
IDP_1, $k=2$	56	18	26	45	41	14	67	19	14
IDP_1, $k=4$	70	17	13	79	15	6	80	14	6
IDP_1, $k=7$	96	3	1	96	3	1	96	3	1
IDP_2, $k=2$	56	18	26	45	41	14	67	19	14
IDP_2, $k=4$	70	17	13	79	15	6	80	14	6
IDP_2, $k=7$	96	3	1	96	3	1	96	3	1
Dyn. Prog.	100	0	0	100	0	0	100	0	0

Figure 28: Good, Acceptable, Bad Plans Vary s, 10-way STAR Query

fact, IDP_1 produces very good plans in this experiment after at most one second whereas for some STAR queries, 2PO never produces acceptable plans (the average scaled cost is 60 or more).

7.4.2 20-Way Join Queries

We now turn to the results of the experiments with the 20-way join STAR and CHAIN queries. These queries could not be optimized using dynamic programming so we can only compare 2PO and IDP. The results essentially confirm that 2PO is not competitive. Figures 32 and 34 show that 2PO produces significantly worse plans than IDP_1 and IDP_2 in all cases, even for $k = 2$. These figures also show that the quality of plans produced by both IDP variants improves with growing k, but that the improvements are fairly small beyond $k = 5$ (less than a factor of 5). Figures 31 and 33 show that 2PO has a higher running time than IDP for small k and even for fairly large k the running time of IDP_2 and 2PO are comparable. These experiments also confirm that IDP_1 has higher running time than IDP_2 and that the quality of plans produced by IDP_1 and IDP_2 is almost the same. (IDP_1 is, again, slightly better for CHAIN queries.) We also ran experiments with even more joins (up to $n = 30$), but could not get any new insights from these experiments. In

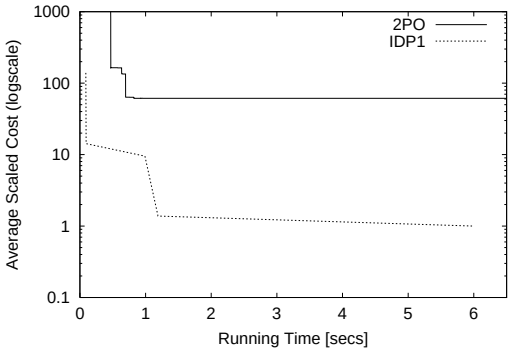


Figure 29: Quality by Running Time
$s = 3$, 10-way STAR Query

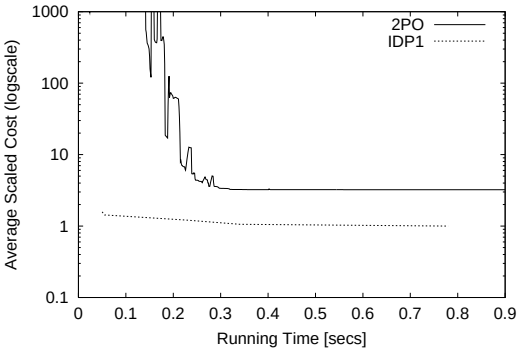


Figure 30: Quality by Running Time
$s = 3$, 10-way CHAIN Query

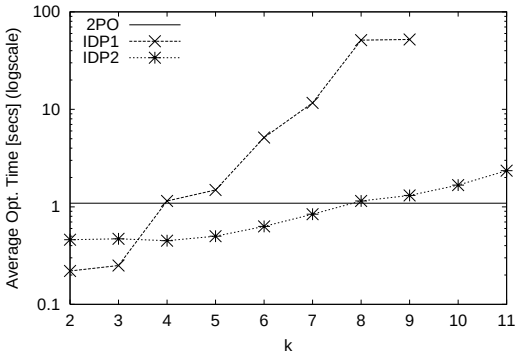


Figure 31: Running Time, 2PO, IDP
$s = 3$, 20-way CHAIN Query

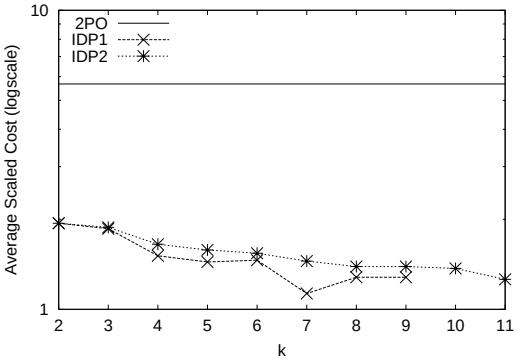


Figure 32: Quality of Plans, 2PO, IDP
$s = 3$, 20-way CHAIN Query

Figures 31 to 34, we do not show any results for which the optimization time was larger than 100 seconds (e.g., IDP_1 for $k > 6$ and STAR queries).

8 Conclusion

In this paper we presented a new class of query optimization algorithms which are based on a principle that we call iterative dynamic programming (IDP). We believe that existing optimizers should be extended to employ IDP and that new optimizers should be based on IDP. First of all, IDP is clearly better than dynamic programming which is used in most database systems today: IDP is able to produce as good plans as dynamic programming if there are enough resources available, and IDP is, in addition, able to adapt in cases where there are not enough resources available or the query is too complex for dynamic programming. Furthermore, we showed that IDP is better than a randomized approach and other heuristic approaches for query optimization. We compared IDP to 2PO which is the best known randomized algorithm and was shown to outperform other heuristics [SMK97]. 2PO is both slower and produces worse plans than simple IDP variants with a setting of $k = 2$. Furthermore, the 2PO algorithm is very difficult to integrate into an existing optimizer which is based on dynamic programming. Maybe, randomized algorithms such as 2PO are attractive to optimize very complex queries that involve dozens of tables and for which

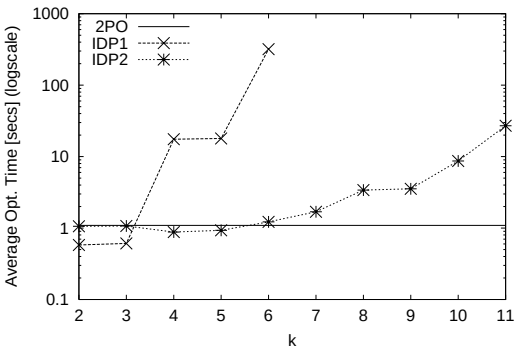


Figure 33: Running Time, 2PO, IDP
$s = 3$, 20-way STAR Query

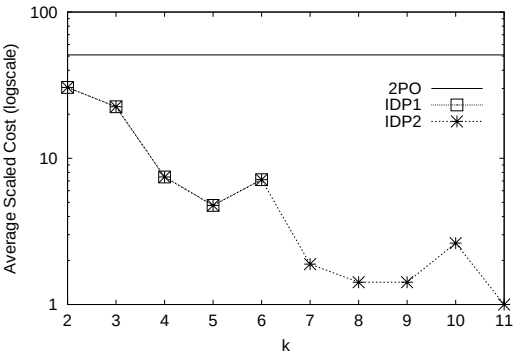


Figure 34: Quality of Plans, 2PO, IDP
$s = 3$, 20-way STAR Query

even polynomial time algorithms like IDP are too expensive; however, we did not experiment with such queries because such queries do not occur in any database applications we are aware of.

We identified eight different IDP variants. Our experiments showed clearly that what we call "balanced" IDP with "bestRow" should be used. No clear winner could be identified between the basic algorithm variants IDP_1 and IDP_2. The overall picture is that IDP_2 is faster than IDP_1 and produces as good plans as IDP_1. On the negative side, however, IDP_2 requires a-priori tuning by a user or system administrator (i.e., setting of the k parameter) whereas IDP_1 is adaptive. Our conclusion is that both IDP_1 and IDP_2 should be combined. That is, the optimizer should use IDP_2 with some default value of k in its main loop (e.g., $k = 15$), and the optimizer should employ IDP_1 (rather than dynamic programming) whenever it optimizes a building block. This way, the optimizer will always safely generate plans because IDP_1 is adaptive, and users can overwrite the default value of k in order to use IDP_2 to speed-up the optimization process.

As future work, we plan to look for ways to tune the implementation of IDP. We will, for example, devise models to predict the running time and memory requirements of IDP in order to avoid enumerating, say, all 7-way join subplans of a query and then falling back to select a 4-way join subplan as described in Section 5.3 for IDP_1. Another point of future work is to compare our IDP variants with, say, a 2-step optimization process as proposed in [SAL+96] for distributed databases or to compare IDP with the query optimization heuristics of [HS93, Hel94, CS96] for the optimization of queries with expensive predicates. We would also like to see how IDP can be used to better control the optimization process; for instance, spend more time to optimize queries with a high estimated execution time and when it is worth it. Finally, we are curious to see how well IDP works for other, non-database optimization problems.

Acknowledgments

We would like to thank Johann Christoph Freytag, Alfons Kemper, Guy Lohman, and Bennet Vance for many helpful comments on this work.

References

[BEG96] R. Buck-Emden and J. Galimow. *SAP R/3 System, A Client/Server Technology.* Addison-Wesley, Reading, MA, USA, 1996.

[CK97] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 219–230, Tucson, AZ, USA, May 1997.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 354–366, Santiago, Chile, September 1994.

[CS96] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 87–98, Bombay, India, September 1996.

[DHKK97] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database performance in the real world: TPC-D and SAP R/3. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 123–134, Tucson, AZ, USA, May 1997.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 149–160, Montreal, Canada, June 1996.

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, USA, May 1987.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 9–18, San Diego, USA, June 1992.

[GLPK94] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, randomized join-order selection—why use transformations? In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 85–95, Santiago, Chile, September 1994.

[GLSW93] P. Gassner, G. M. Lohman, K. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. *IEEE Data Engineering Bulletin*, 16(3):4–18, September 1993.

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. IEEE Conf. on Data Engineering*, pages 209–218, Vienna, Austria, April 1993.

[Hel94] J. M. Hellerstein. Practical predicate placement. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 325–335, Minneapolis, MI, USA, May 1994.

[HKWY97] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 276–285, Athens, Greece, August 1997.

[HS91] W. Hong and M. Stonebraker. Optimization of parallel execution plans in xprs. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, pages 218–225, Miami, Fl, USA, December 1991.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–276, Washington, DC, USA, May 1993.

[IK84] T. Ibaraki and T. Kameda. Optimal nesting for computing N-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.

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

[IW87] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 9–22, San Francisco, USA, May 1987.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 128–137, Kyoto, Japan, 1986.

[KKM98] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: a database application system. Tutorial handouts for the ACM SIGMOD Conference, Seattle, WA, USA, June 1998.

[KMP93] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 543–554, Dublin, Ireland, 1993.

[Kos98] Donald Kossmann. The state of the art in distributed query processing. 1998. Submitted for publication.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, IL, USA, May 1988.

[LVZ93] R. Lanzelotte, P. Valduriez, and M. Zait. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 493–504, Dublin, Ireland, 1993.

[LYV+98] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, and M. Valiveti. Capability based mediation in TSIMMIS. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 564–566, Seattle, WA, USA, June 1998.

[ML86] L. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 149–159, Kyoto, Japan, 1986.

[OL90] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 314–325, Brisbane, Australia, 1990.

[Pal74] F. Palermo. A database search problem. In *Information Systems*, pages 67–101. Plenum Publ., New York, NY, 1974.

[PGLK97] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 306–315, Athens, Greece, August 1997.

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

[SAL+96] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *The VLDB Journal*, 5(1):48–63, Jan 1996.

[SG88] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 8–17, Chicago, IL, USA, May 1988.

[SI93] A. Swami and B. Iyer. A polynomial time algorithm for optimizing join queries. In *Proc. IEEE Conf. on Data Engineering*, pages 345–354, Vienna, Austria, April 1993.

[SM97] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 238–248, Tucson, AZ, USA, May 1997.

[SMK97] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

[SSM96] D. Simmen, E. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 57–67, Montreal, Canada, June 1996.

[Swa89] A. Swami. Optimization of large join queries: Combining heuristics and combinational techniques. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 367–376, Portland, OR, USA, May 1989.

[Swa91] A. Swami. Distributions of query plan costs for large join queries. Technical Report RJ 72891, IBM Research Division, IBM Almaden Reseach Center, San Jose, CA, January 1991.

[SY98] E. Shekita and H. Young. Iterative dynamic programming. IBM Technical Report, 1998.

[SYT93] E. Shekita, H. Young, and K.-L. Tan. Multi-join optimization for symmetric multiprocessors. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 479–492, Dublin, Ireland, 1993.

[Van98] B. Vance. *Join-order Optimization with Cartesian Products.* PhD thesis, Oregon Graduate Institute, 1998. in preparation.

[VM96] B. Vance and D. Maier. Rapid bushy join-order optimization with Cartesian products. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, Montreal, Canada, June 1996.

Appendix: Time Complexity of $\texttt{IDP}_1$ in a Centralized Database $(2 < k < n)$

We carry out the proof (Page 15) in two steps. In the first step, we show that the first iteration of the basic $\texttt{IDP}_1$ algorithm can be carried out in $\mathcal{O}(n^k)$ steps. In the second step, we show that all the other iterations combined can be carried out in $\mathcal{O}(n^k)$ steps. In this proof, k is considered as a constant, and n is the variable.

Step 1: To prove that no more than $\mathcal{O}(n^k)$ steps are carried out in the first iteration of the $\texttt{IDP}_1$ algorithm, we will generously overestimate the cost of $\texttt{IDP}_1$ and count the number of subplans enumerated by $\texttt{IDP}_1$ in the first iteration assuming that *no* pruning is carried out. To do this counting, we first observe that there are $\frac{(2*m-2)!}{(m-1)!}$ bushy plans for a query with m tables [GHK92]. As a result, at most $\frac{(2*m-2)!}{(m-1)!}$ subplans need to be enumerated to determine the *optPlan(P)* entry if P involves m tables. (With pruning, significantly fewer plans would be enumerated.) Now, we observe that we have $\binom{n}{2}$ *optPlan* entries for all 2-way join subplans, $\binom{n}{3}$ entries for all 3-way join subplans, ..., $\binom{n}{k}$ entries for all k-way join subplans. In all, $\texttt{IDP}_1$, therefore, enumerates

$$A = n + \sum_{m=2}^{k} \binom{n}{m} * \frac{(2*m-2)!}{(m-1)!} \tag{3}$$

subplans in its first iteration (the first n subplans are the *access plans*). Now, we conclude

$$A \leq n + \frac{(2*k-2)!}{(k-1)!} * \sum_{m=2}^{k} \binom{n}{m} \in \mathcal{O}(n^k)$$

because

$$\binom{n}{m} = \frac{n*(n-1)*\ldots*(n-m)}{m!} \in \mathcal{O}(n^m)$$

and

$$\sum_{m=2}^{k} n^m \in \mathcal{O}(n^k)$$

Step 2: We now prove that all other iterations (except the first) combined require less than $\mathcal{O}(n^k)$ steps. First, we observe that $\texttt{IDP}_1$ carries out $\lceil \frac{n-k}{k-1} \rceil$ iterations after the first iteration, since the optimization problem is reduced by $k-1$ tables with every iteration. Next, we observe that in the ith iteration of these, the optimization problem has $n - i*(k-1)$ tables. As a result,

$$\sum_{m=2}^{k} \binom{n-i*(k-1)}{m} * \frac{(2*m-2)!}{(m-1)!}$$

join plans need to be considered in the ith iteration (just as in Equation (3)). Now, we observe that in every of these iterations only the join plans involving the *new* temporary table that is produced by the selected subplan of the previous iteration need to be newly

generated; or putting it differently, all join plans involving the $n - i * (k-1) - 1$ *old* tables already exist and need not be generated again. As a result,

$$\sum_{m=2}^{k} \binom{n - i * (k-1)}{m} * \frac{(2 * m - 2)!}{(m-1)!} \quad - \quad \sum_{m=2}^{k} \binom{n - i * (k-1) - 1}{m} * \frac{(2 * m - 2)!}{(m-1)!}$$

join plans need to be newly generated in the ith iteration. With some tricks, we can show that this equation is in $\mathcal{O}(n^{(k-1)})$, and since we need $\mathcal{O}(n)$ iterations after the first iteration, all these iterations combined can be carried out in $\mathcal{O}(n^k)$ steps. □