Query Optimization in the Presence of Limited Access Patterns

Daniela Florescu
INRIA Rocquencourt, France
Daniela.Florescu@inria.fr

Alon Levy
University of Washington, Seattle
alon@cs.washington.edu

Ioana Manolescu
INRIA Rocquencourt, France
Ioana.Manolescu@inria.fr

Dan Suciu
AT&T Labs – Research
suciu@research.att.com

Abstract

We consider the problem of query optimization in the presence of limitations on access patterns to the data (i.e., when one must provide values for one of the attributes of a relation in order to obtain tuples). We show that in the presence of limited access patterns we must search a space of *annotated query plans*, where the annotations describe the inputs that must be given to the plan. We describe a theoretical and experimental analysis of the resulting search space and a novel query optimization algorithm that is designed to perform well under the different conditions that may arise. The algorithm searches the set of annotated query plans, pruning invalid and non-viable plans as early as possible in the search space, and it also uses a best-first search strategy in order to produce a first complete plan early in the search. We describe experiments to illustrate the performance of our algorithm.

1 Introduction

The goal of a query optimizer of a database system is to translate a declarative query expressed on a logical schema into an imperative query execution plan that accesses the physical storage of the data, and applies a sequence of relational operators. In building query execution plans, traditional relational query optimizers try to find the most efficient method for accessing the necessary data. When possible, a query optimizer will use auxiliary data structures such as an index on a file in order to efficiently retrieve a certain set of tuples in a relation. However, when such structures do not exist or are not useful for the given query, the alternative of scanning the entire relation always exists. The existence of the fall back option to perform a complete scan is an important assumption in traditional query optimization.

Several recent query processing applications have the common characteristic that it is not always possible to perform complete scans on the data. Instead, the query optimization problem is complicated by the fact that there are only limited access patterns to the data. One such application is optimization in the presence of foreign and table functions [1, 3, 15]. In most cases, such functions require a set of given inputs in order to return a set of tuples.

Our work is mainly motivated by query processing for data integration and for semi-structured data. A data integration system needs to access autonomous remote sources in order to answer a query. Even if the data integration system is able to model the contents of the remote sources as relations, the sources may provide only limited access patterns to the data that they serve. This may happen for two main reasons (1) the underlying data may actually be stored in a structured file or legacy system hence the interface to the data is naturally limited, and (2) even if the data is stored in a traditional database system, the source may provide only limited access capabilities for reasons of security or performance.

The semi-structured data model of labeled directed graphs provides a flexible mechanism for integrating a larger class of data sources [4]. In this case, the graph provides a logical abstraction of the particular storage of the data. However, the actual storage of the graph poses natural limitations on the access patterns to the data. For example, it is often possible to allow following edges only in the forward direction, and the system may not allow to scan the entire set of objects in the graph or following edges backwards.

In both cases, the access pattern limitations to the data can be modeled using *binding patterns*. A binding pattern specifies which attributes of a relation must be given values when accessing a set of tuples. For example, the binding pattern $R(A^b, B^f)$ specifies that the only way of retrieving tuples of $R(A, B)$ is by providing values for the attribute A. In fact, binding patterns can be viewed as a method for abstracting the storage of the data for the query optimizer.

This paper considers the problem of query optimization in the presence of limitations on access patterns, described by binding patterns. Specifically, given a set of binding patterns describing the only ways of accessing tuples in a set of relations, and given a select-project-join query over these relations, our task is to efficiently find an optimal query execution plan for the query, if a plan exists.

1.1 Motivating Example

We begin by illustrating the problem with an example, drawn from an actual application of integration of scientific data. The goal of the application, conducted by the Marine Institute of Crete, is to study the levels of water pollution in the Mediterranean Sea. The application includes two sources of data each containing the results of sets of experiments. The first source stores the results of some experiments concerning water circulation, and the second source reports the results of experiments concerning the level of pollution in the water. Integrating the data from the two sources enables the scientists to predict water pollution levels for a wide range of times and locations. The data resulting from the experiments is stored in the three sources with the following schemas:

Source 1:	Source 2:
$Experim_1(key, date, depth)$	$Experim_2(key, date, depth)$
$Location_1(key, location)$	$Location_2(key, location)$
$Result_1(key, circulation)$	$Result_2(key, emission)$

Source 3:

$Coincides(location, location, similarity)$

In both of the sources the data is stored in a proprietary data store rather than a relational database. Accordingly, the possible operations on the data are limited. In source 1, it is possible to ask for the keys of all the experiments that have been done at a certain date (i.e., to select on a given date) or at a certain depth. Given an experiment key, it is possible to retrieve its location from relation Location1, which is a complex value encoding the geographical coordinates of a rectangle, or to retrieve the result of the experiment (from relation Result1) which is a complex picture describing the speed and direction of the water circulation. In this source it is not possible to perform a selection on a location or on the circulation. The situation is similar for Source 2.

In both sources, the location represents the geographical coordinates of a rectangle. However, the sharing of the sea surface in rectangles is not done in the same fashion across the two sources, i.e., the sources refer to different sets of rectangles. In order to facilitate the integration, the scientists use a third data source that answers queries about such rectangles. Given two input locations the source returns a number between 0 and 1 describing the similarity between the rectangles.

The typical operation that scientists need to perform on this data is to answer queries of the form: "retrieve the water circulation and the pollution emission on the 1/10/98 for locations matching with high degree of similarity (=0.9)". The query can be written as the following conjunctive query:

$$\begin{aligned} Query(w_1, w_2) : -&Experim_1(x_1, y_1, z_1), Location_1(x_1, t1), \\ &Result1(x_1, w_1), Experim_2(x_2, y_1, z_1), \\ &Location_2(x_2, t_2), Result_2(x_2, w_2), \\ &Coincides(t_1, t_2, s), y_1 = \text{“}1/10/98\text{”}, s = 0.9. \end{aligned}$$

Figure 1 describes two possible ways to evaluate this query, which are valid according to the limitations and one relational query execution plan which is not valid given the access limitations.

The evaluation strategy followed by the plan 1(a) uses the selection condition on date on both sources in order to retrieve the keys of the experiments performed on this date. Then, in each source, the keys are used to obtain the corresponding experiment locations. This operation is performed by a dependent join. The join (on the depth attribute) of those two temporary relations is sent to Source 3 which calculates the similarity degree for each pair of locations and selects those that satisfy the similarity predicate. Finally, the keys of the selected experiments are used again to retrieve the desired images (water circulation and pollution emission).

A different plan to evaluate this query, depicted in figure 1(b), is to start by retrieving from Source 1 the keys of the experiments performed on the "1/10/98", together with their corresponding depth. For each tuple in the result, the values of the depth and date could be used for the following complex computation (which is executed by the engine of the data integration system): using the depth, we can retrieve from the relation Experim2 the keys of the experiments performed in source 2 on this depth, then the result is filtered using the desired date; the key of the resulting experiments are sent to source 2 in order to retrieve their respective location, and then they are sent once more in order to retrieve the corresponding image. Finally, the location of the experiments of source 1 is retrieved, source 3 is tested for the similarity and, in the last step, the image from source 1 is retrieved.

Finally, we remark that the plan depicted in figure 1(c) cannot be executed given the limitations to the data sources that are given. The reason is that it is impossible to retrieve all the tuples of the relation Location in either of the two sources. The problem we consider in this paper is finding the optimal plan among all feasible plans for a query, given the access pattern limitations to the data.

1.2 Our Solution

The solution we propose in this paper is based on extending System-R cost-based optimization to incorporate limitations on binding patterns. The key idea underlying our solution is that the optimizer searches through the space of *partial annotated query plans*, where the annotation of a subplan describes which variables of the query must be given as input to the subplan. We study the effect of adding annotations on the size of the resulting search space, and describe an efficient algorithm for searching the space.

The idea of adding binding patterns as annotations to subqueries is not new. Such annotations were used in magic-set transformations [16, 14] and for exploring sideways information passing strategies. The focus of this paper is on incorporating such annotations into a cost-based optimizer.

We make the following contributions.

- We show how the presence of binding pattern limitations affects several fundamental properties of the search space, such as the need to consider different binding-pattern annotations on query execution plans, the need to explore the space of bushy trees, and the specialized handling of placing selections.
- We provide an analytical and empirical study of the effect of adding annotations on the size of the search space. The study considers different shapes of queries, bushy vs. left-linear trees, plans with or without cartesian products, and different numbers of binding patterns associated with each database relation. While the study shows that in some important cases the number of valid query execution plans is actually considerably smaller than the corresponding case without annotations, there are still important cases in which search spaces grows significantly compared to traditional System-R optimization.
- We describe a query optimization algorithm that is designed to perform efficiently under the properties exposed by our analysis. First, the algorithm considers only *valid*

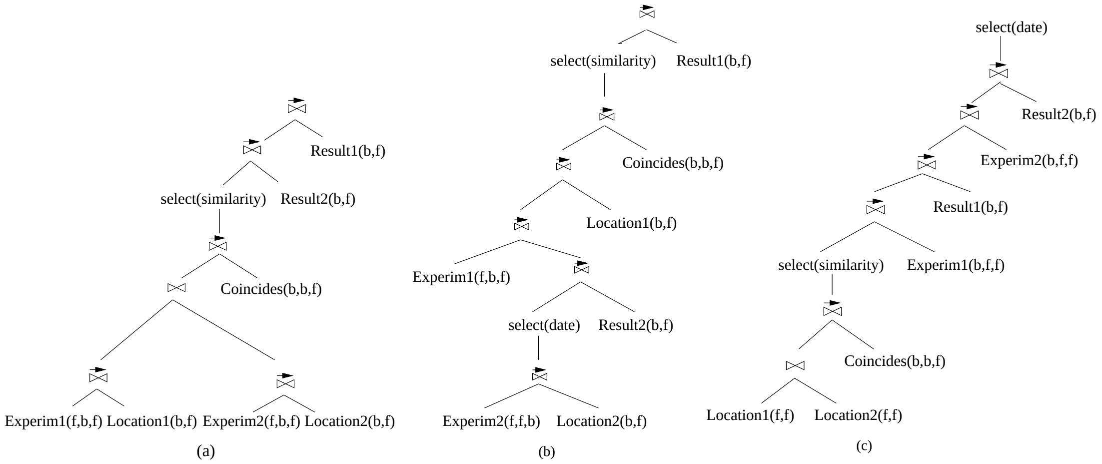

Figure 1: Three different execution plans for the example. (a) and (b) are valid plans, while (c) is not.

execution plans, i.e., which adhere to the given binding pattern. The algorithm also prunes early on plans that are not viable, i.e., cannot be part of any valid execution plan. Second, the algorithm uses a best-first search strategy in order produce the first complete query execution plan relatively fast. In contrast, System-R bottom-up optimization only produces a complete query towards the end of the optimization process. Such behavior would not be acceptable in cases in which the search space is significantly larger than in the traditional case. Finally, the algorithm uses a novel method to combine the join enumeration and selection placement.

- We describe an experimental study of our algorithm. The experiments consider the performance of our algorithm under different conditions (varying query size, shape and number of available binding patterns). The experiments show how our algorithm obtains the first query execution plans much faster than a pure dynamic programming approach. Furthermore, when we consider the time to perform exhaustive search of the space of plans, we show that the extra cost associated with employing a best-first search algorithm compared to dynamic programming only causes linear slowdown w.r.t. the size of the search space. Hence, we argue that a best-first search strategy is more appropriate for the optimization problem we consider.

The paper is organized as follows. Section 2 formally defines our problem. Section 3 describes the effects of binding patterns on various properties of the search space, and Section 4 describes an analysis of the size of the resulting search space. Section 5 describes our query optimization algorithm. Section 6 describes our implementation and Section 7 describes the results of our experiments. We end with a discussion and comparison to related work.

2 Problem Definition

In this section we formally define the problem setting we consider in this paper.

Queries: We consider the class of select-project-join queries, also known as conjunctive queries. We use the following notation of conjunctive queries. A query q is denoted by:

$$q(\bar{X}) : -e_1(\bar{X}_1), \ldots, e_n(\bar{X}_n), \mathcal{C}.$$

The predicates $e_1, \ldots, e_n$ denote database relations, and $\bar{X}, \bar{X}_1, \ldots, \bar{X}_n$ are tuples of variables. The atoms $e_1(\bar{X}_1), \ldots, e_n(\bar{X}_n)$ are the conjuncts (or subgoals) of the query, which together with $\mathcal{C}$ form the query's body. The atom $q(\bar{X})$ is the head of the query, and the variables in $\bar{X}$ are those that are selected in the result. We require the query to be safe, i.e., any variable that appears in the head must also appear in at least one of the $\bar{X}_i$'s. $\mathcal{C}$ is a set of atoms of the form $X_i = c_i$, where X_i appears in $\bar{X}_1 \cup \ldots \cup \bar{X}_n$, and c_i is a constant. The set of variables appearing in $\mathcal{C}$ are called the bound variables in the query, denoted by $bound(q)$.

Data access descriptions: With each database relation we associate a set of binding patterns, describing the possible access patterns to the tuples in the relation. Formally, a binding pattern for a relation R is a mapping from the arguments of R to the alphabet $\{b, f\}$. The meaning of a binding pattern bp for a relation R is that the attributes of R that bp maps to b must be given values when accessing the tuples of R. The traditional scan of a relation corresponds to the case where all attributes are mapped to f. A relation may have multiple binding patterns describing the different possible ways to obtain tuples from the relation. For example, the binding patterns for the Example in Section 1.1 are the following.

$Source$ 1:	$Source$ 2:
$Experim_1(key^f, date^b, depth^f)$	$Experim_2(key^f, date^b, depth^f)$
$Experim_1(key^f, date^f, depth^b)$	$Experim_2(key^f, date^f, depth^b)$
$Experim_1(key^f, date^b, depth^b)$	$Experim_2(key^f, date^b, depth^b)$
$Location_1(key^b, location^f)$	$Location_2(key^b, location^f)$
$Result_1(key^b, circulation^f)$	$Result_2(key^b, emission^f)$

$Source$ 3:
$Coincides(location^b, location^b, similarity^f)$

In addition, each binding pattern is also labeled with (1) the cost of accessing it once, and (2) the cardinality of the expected output per given input.

Query execution plans: a query execution plan for a query q is a tree whose leaves are labeled with relations in the query and whose internal nodes are algebraic operators. We refer to the leaves of a query execution plan as *atomic plans*. In our discussion we consider plans with join and selection operators. In this paper we consider only selections of the form $X_i = c_i$. To simplify our discussion we do not consider plans with projections, and assume they are introduced at a later stage.

We distinguish two kinds of join operators: regular joins and dependent joins. Both types of joins are binary operators and apply recursively on subtrees corresponding to query execution sub-plans. In the case of a regular join, the two input query execution sub-plans can be executed independently of each other, resulting in two tables that can be joined using any of the traditional join algorithms (e.g., hash-join, sort-merge join). In the second type of join, the right input subtree cannot be executed independently, because it requires bindings that are obtained from the result of the left subtree.

Among the algorithms developed for the join operator, only the nest-loop join is applicable to dependent joins. The efficient implementation of dependent joins are considered in [2] in the context of optimization for foreign functions, as well as in the context of evaluation of path expressions in object-oriented databases. Furthermore, several caching techniques to optimize the implementation of dependent joins are discussed in the context of the Montage system [3]. An ellaborated implementation of dependent joins, that combines hashing and caching, has been proposed in [6]. Finally, we note that that dependent joins have received several other names in the literature (e.g., functional join, implicit join, filter join, theta semi-join, bind join).

Annotated plans: As mentioned earlier, our algorithm is going to search the space of *annotated* query execution plans. The annotation describes which variables in the query must be given as input to the plan. Formally, each node n in a query execution plan tree is labeled by a pair $(conj(n), adorn(n))$, where $conj(n)$ is the set of conjuncts of q that is covered by n, and the adornment $adorn(n)$ describes which variables of q must be given, in order for the subtree n to be executable.

An adornment is a mapping from the variables in q to the set { b,f,_ }. The meaning of the adornment is the following: (a) if a variable is mapped to b, then its value is necessary for the execution of the subplan, (b) if a variable is mapped to _, then it does not appear in the sub-plan, and (c) if a variable X is mapped to f, then by executing that sub-plan, we obtain values for X. In our examples $adorn(n)$ is shown as a subscript of $conj(n)$, and shows exactly the set of variables mapped to b. For example, $[R_1(x_0, x_1), R_2(x_1, x_2)]_{\{x_0\}}$ denotes the adornment that maps x_0 to b, x_1, x_2 to f, and the rest of the variables in the query to _.

Clearly, annotations on the nodes in a query execution plan are not arbitrary. Given a set of data access descriptions to the database relations and a query execution plan, the annotations in the plan must satisfy the following conditions. If n is a leaf accessing a relation R, then n should specify which of the available access patterns to R should be used. The adornment of n is correct if it is obtained by extending one of the binding patterns associated with R by mapping each variable not appearing in n to _. If n is a selection node whose child is n_1, and its selection variables are $\bar{Y}$, then the adornment of n is obtained from the adornment of n_1 by changing the mapping for the variables in $\bar{Y}$ from f to b. If n is a join node whose children are n_1 (left) and n_2 (right), then the adornment of n satisfies the following constraints: (a) a variable in $adorn(n)$ is mapped to _ only if it is mapped to _ in both $adorn(n_1)$ and $adorn(n_2)$, (b) a variable is mapped to b in $adorn(n)$ if it is mapped to b in $adorn(n_1)$ or if it is mapped to b in $adorn(n_2)$ and mapped to _ in $adorn(n_1)$, (c) the rest of the variables are bound to f. The set of variables whose values are passed from the left subtree to the right one are those that are mapped to b in $adorn(n_2)$ and mapped to f in $adorn(n)$. If this set is non-empty, then the join is a dependent join, otherwise it is just a regular join.

Cost Model Each of the query execution plans p has an associated cost, denoted by $cost(p)$. Our optimizer includes a component which takes a query execution plan, and chooses a physical implementation for each of the relational operators. Therefore, the cost of a query execution plan is the cost of the best physical query execution plan implementing it. The cost of atomic query execution plans are directly deduced from the cost associated with the data access descriptions, and the cost of the non-atomic query execution plans is an estimate based on the cost of the operator and the cost of the subplans. The particular cost function we use is orthogonal to the search strategy that our algorithm employs, though it can, in some cases, influence the effectiveness of our pruning methods. We make the monotonicity assumption about our cost model: if the plan P' is obtained from the plan P by replacing a subplan P_1 of P by a cheaper and equivalent subplan P_2, then P' is cheaper than P. This property is required in order to ensure that the algorithms will not miss the optimal plan in the presence of pruning.

Problem Definition A query execution plan that covers all the conjuncts in the query and whose adornment maps precisely the bound variables in the query to b is called a *complete* query execution plan. The problem we address in this paper is: given a set of data access descriptions and a query q, our goal is to find a complete query execution plan for q whose cost is minimal.

Before ending this section we introduce several terms that will be convenient in our discussion. Two query execution plans are considered *equivalent* if they are labeled with the same set of conjuncts from the query and have identical adornments. A query execution plan is *viable* if it can be part of a complete query execution plan. An adornment bp_1 is said to be *weaker* than an adornment bp_2, denoted by $bp_1 < bp_2$ if every variable that is mapped to b in bp_1 is also mapped to b in bp_2, and the two adornments map the same set of variables to _. A query execution plan p_1 *covers* a query execution p_2 if they are labeled with the same set of conjuncts from the query, and the adornment of p_1 is weaker than the adornment of p_2.

Intuitively, two equivalent query execution plans solve the same subquery. Obviously, the equivalence relation partitions the set of query execution plans into classes. All the plans in the same equivalence class are labeled with the same set of conjuncts and the same adornments. In our discussion we refer to coverage and viability of classes of plans, with the following meanings. If a plan is viable, then all the plans in the same equivalence class are viable. If a plan p_1 covers a plan p_2 then any plan p'_1 equivalent to p_1 will cover any plan p'_2 equivalent to p_2.

Example 2.1 Consider the following chain query whose bound variables are x_0 and x_4:

$$R_1(x_0, x_1), R_2(x_1, x_2), R_3(x_2, x_3), R_4(x_3, x_4), x_0 = c_1, x_4 = c_2$$

where each relation symbol R_i has two adornments: bf and fb. Consider the following equivalence classes:

- The equivalence classes $[R_1(x_0, x_1), R_2(x_1, x_2)]_{\{x_0\}}$ and $[R_1(x_0, x_1), R_2(x_1, x_2)]_{\{x_2\}}$ are valid and viable.
- The equivalence class $[R_1(x_0, x_1), R_3(x_2, x_3), R_4(x_3, x_4)]_{\{x_1, x_2\}}$ is valid but not viable, because a plan that includes only R_2 cannot produce bindings for both required inputs x_1 and x_2.
- The equivalence class $[R_1(x_0, x_1)]_{\{\}}$ is not valid.
- The equivalence class $[R_1(x_0, x_1), R_2(x_1, x_2)]_{\{x_0\}}$ covers the equivalence class $[R_1(x_0, x_1), R_2(x_1, x_2)]_{\{x_0, x_2\}}$.

3 Optimization with Binding Patterns

In this section we show that some of the basic properties underlying System-R style optimization need to be reconsidered in the presence of binding-pattern limitations. In fact, we will compare not with System-R, but with the Garlic data integration system [5] which partially handles binding pattern limitations within the framework of System-R. We begin by arguing that it is essential to consider query execution plans annotated by their input variables. This will be the key feature distinguishing our algorithm from that in Garlic.

"Open" partial query execution plans: in order to compare our approach with that of Garlic, we distinguish two classes of partial query execution plans. An *open* partial query execution plan is a non-atomic plan which cannot be executed only with the constants available in the query. Of course, in order for an open subplan to be part of a complete plan, it must receive bindings from some other parts of the plan. A *closed* subplan is one that can be executed given the constants from the query.

Garlic's search strategy only considers closed partial query execution plans.[1] We now argue that optimal plans may include open subplans, and therefore, it is important to consider them in the search. Subsequently we show that looking at this larger search space has several important ramifications to the properties of our search space.

Example 3.1 Consider the following query $[R(x, y), S(y, z), T(z, w)]_{\{\}}$ and suppose that the only access patterns allowed to the relations R, S and T are: $R(x^f, y^f)$, $S(y^b, z^f)$, and $T(z^b, w^f)$. Garlic will find a single execution plan: $(R(x^f, y^f) \overrightarrow{\bowtie}_y S(y^b, z^f)) \overrightarrow{\bowtie}_z T(z^b, w^f)$, and will miss the second plan: $R(x^f, y^f) \overrightarrow{\bowtie}_y (S(y^b, z^f) \overrightarrow{\bowtie}_z T(z^b, w^f))$ becuase the subplan $S(y^b, z^f) \overrightarrow{\bowtie}_z T(z^b, w^f)$ is open.

The search strategy employed by Garlic, which is implemented using dynamic programming, has two important properties. First, it can be shown that if there exists a plan for answering the query, then Garlic will find one, even if it is not the optimal one. Second, if all dependent join operators are implemented in a limited fashion (with nested loops), then any complete plan with open subplans results in the same execution as some plan without open subplans: in the example above, the two query plans have equivalent executions under the nested loop implementation of dependent joins. However, more efficient implementations have been proposed [3, 6], for which this property no longer holds. For example consider an implementation using caching techniques where every time the expression $(S(y^b, z^f) \overrightarrow{\bowtie}_z T(z^b, w^f))$ is executed for a given value of y, the output tuples are cached (with the input y): then the two plans in the example above result in distinct executions. Thus, Garlic will miss a potentially more efficient plan, and this can be especially detrimental in the context of data integration, where the cost of accessing each source is usually high.

[1] For atomic plans (plans on a single relation) Garlic considers one plan for every viable binding pattern.

In what follows we describe several other important properties of the search space in the presence of binding pattern limitations.

Refined equivalence classes: as an immediate consequence of the above discussion is that we need to refine our notion of equivalence classes during our join-order enumeration. System-R style optimizers keep the cheapest query execution plan for every equivalence class, where two plans belong to the same equivalence class if they cover the same set of base relations in the query. In our new context, it is necessary to annotate every plan also with the set of variables that are required as inputs in addition to the set of base relations that are covered. From now on, the *equivalence class* of a plan is determined by the combination of the subquery solved by a plan and its required input variables.

Valid and viable plans: an immediate consequence of the fact that we must consider refined equivalence classes is that the number of plans kept during the generation phase (i.e., one per equivalence class) grows significantly, hence leading to a more expensive search problem. Fortunately, two classes of plans can be pruned early in the search: plans belonging to invalid or non-viable equivalence classes. In Section 4 we show that pruning these two classes of plans has a dramatic effect on the size of the search space, and in Section 5 we show that validity can be checked as part of the enumeration algorithm, and viability can be checked efficiently.

Two additional properties of our search space are important to understand before we can design an appropriate algorithm: the need to explore bushy trees and the special treatment of selections.

The need for bushy trees: as the following example shows, in the presence of limited access patterns, there are cases where the set of left-linear trees includes only plans with cartesian products, while the set of bushy trees does contain a query execution plan without cartesian products. Hence, if we want to avoid plans with cartesian products, we must search the space of bushy trees.

Example 3.2 Consider the following query $[P(x, y), R(y, z), S(t, w), T(w, z)]_{\{\}}$, and suppose that the only access patterns allowed to the relations P, R, S and T are: $P(x^f, y^f)$, $R(y^b, z^f)$ $S(t^f, w^f)$ and $T(w^b, z^f)$. It is easy to note that all the linear query execution plans will include a cartesian product. However, the following bushy-tree does not include a cartesian product: $(P(x^f, y^f) \overrightarrow{\bowtie}_y R(y^b, z^f)) \bowtie_z (S(t^f, w^f) \overrightarrow{\bowtie}_w T(w^b, z^f))$

Recall that in the traditional setting, if the query graph is connected, then the space of left-linear trees necessarily includes a plan without cartesian products. Hence, in that context, the query optimizer can limit its search to left-linear trees without having to use cartesian products.

Placement of selections: in the context of System-R it is possible to decouple the decision on join ordering from the decision on placement of selections. The placement could either be made heuristically by pushing selections as far down as possible in the query tree, in a cost-based fashion in a separate phase [7], or mixing the decision on the join order with the decision on the placement of expensive predicates in a dynamic programming style optimization like in [3]. In our context, since we are considering query execution plans that are annotated by variables that must be given as inputs to the plan, the interaction between the placement of the selection and the join ordering is much more subtle.

Example 3.3 Consider the query $[R(x,y), S(y,z), T(z,w)]_{\{\}}$, and suppose that the only access patterns allowed to the relations R, S and T are: $R(x^f, y^f)$, $S(y^b, z^f)$ $S(y^f, z^b)$ and $T(z^f, w^f)$. By combining the pattern $R(x^f, y^f)$ with the pattern $S(y^b, z^f)$ via a dependent join operator, we obtain a plan p_1 for the equivalence class $[R(x,y), S(y,z)]_{\{\}}$. By combining the pattern $R(x^f, y^f)$ with the pattern $S(y^f, z^b)$ via a join operator, we obtain a plan p_2 for the equivalence class $[R(x,y), S(y,z)]_{\{z\}}$. It is easy to see that the plan p_1 covers the plan p_2, i.e., by applying a selection on the variable z in the plan p_1, we obtain a plan p_3 which is equivalent to p_2. Hence, a plan without any selections turns out to be equivalent to a plan with selection, and in our case, it may turn out that p_3 is cheaper than p_2.

In standard System-R optimization the equivalence class $[R(x,y), S(y,z)]_{\{z\}}$ would have not be considered at all since z is not bound in the original query. In our setting, as shown previously, we have to keep one plan per set of conjuncts and set of bound variables (even if they are not bound in the original query). As a consequence, if we ignore selections, we will be in the situation where we do consider a plan for the class $[R(x,y), S(y,z)]_{\{z\}}$, but the plan we think is optimal for this class (i.e., the one obtained from the join) may not be the real optimal one.

In summary, in order to perform optimization in the presence of access pattern limitations, the optimizer must search the space of annotated query plans. The algorithm should avoid invalid plans and prune non-viable plans as early as possible. In order to avoid cartesian products, the algorithm needs to consider bushy trees and not only left-deep trees. Furthermore, special care must be given to placing selections and to detect multiple query plans that result in identical executions. In the next section we analyze the size of the search space sanctioned by the conclusions of this section.

4 The Size of the Search Space

The size of the space that needs to be searched by a query optimizer employing dynamic programming is relatively well understood [8, 11, 19]. In this section we study the effect on the size of the search space in the presence of access pattern limitations, and the associated need to search the space of annotated query execution plans. We present both an analytical and empirical study. The results of this study also justifies the choices we made in designing our query optimization algorithm described in Section 5.

4.1 Theoretical Study

Our study examines the size of two measures: the number of valid complete query execution plans and the number of viable partial query execution plans considered by a dynamic-programming style optimizer. Of course, while the number of query execution plans is usually very large (and that of partial plans even larger), dynamic programming only considers a small fraction of the partial plans. For example, for the case of chain queries with n relations, the number of plans without cartesian products is known to be $\binom{2(n-1)}{n-1}\frac{1}{n}$, while the number of bushy partial plans without cartesian products considered by the dynamic programming algorithm is known to be only $\frac{n^3-n}{6}$ [8]. We note that even though we consider the number of plans explored by a dynamic programming optimizer, the results are of interest even if we were to employ a different paradigm, since dynamic programming is sometimes used as a yardstick for the others. For example, [12] show that a classical rule-based optimizer considers, in general, a strictly higher sized search space than dynamic programming, and present an improved rule-based optimization algorithm whose complexity matches that of dynamic programming.

The results of our analysis are shown in Table 2. The table shows the maximal number of complete query execution plans and viable partial query execution plans generated by the dynamic programming algorithm for the cases in which all bushy trees are considered (columns 1 and 3) and for the case in which only left-linear trees are considered (columns 2 and 4). The formulas include query execution plans that have cartesian products.

We focus our theoretical study on chain queries with binary relations:

$$q : -R_1(x_0, x_1), R_2(x_1, x_2), \ldots, R_n(x_{n-1}, x_n), \mathcal{C}$$

and consider different combinations of access patterns and different sets of bound variables. Thus, in the first line of the table, all relations have access pattern, $R_i(u^f, v^f)$, for $i = 1, n$, and $bound(q) = \emptyset$. This line represents the classical case, with no access patterns and no selections, and is for comparison purposes (all entries are taken from the references). Note that since we are counting plans with cartesian products, the numbers in the first row apply to any query shape, not just chain queries. In the second line all relations have the access pattern, $R_i(u^b, v^f)$, $i = 1, n$, and $bound(q) = \{x_0\}$. The third line analyses the transition from line 1 to line 2, by letting the number m of relations with binding pattern ff vary from 1 to n (the other $n - m$ relations have binding pattern bf). We assume here that R_1 is always among these m, i.e., we have the access pattern $R_1(u^f, v^f)$: this guarantees that there always exists a query execution plan, although $bound(q) = \emptyset$. In line four each of the relations has two binding patterns, $R_i(u^b, v^f)$ and $R_i(u^f, v^b)$, and there are two bound variables: $bound(q) = \{x_0, x_n\}$ (i.e., we can start either from the left or from the right). Finally, in the last line each relation has both binding patterns bf and ff.

Lines two and three illustrate an example where the complexity of join ordering decreases because of the limited access patterns. Line two represents an extreme case, with a single left-linear solution, $(\ldots((R_1 \ltimes\!\!\!\rtimes R_2) \ltimes\!\!\!\rtimes R_3) \ldots \ltimes\!\!\!\rtimes R_n)$. There are several solutions with bushy trees, basically all ways to parenthesize this expression, but still less than in the classical case (line 1) where, in addition, one could

Graph	Qep's	LL Qep's	Pqep's in d.p.	LL Pqep's in d.p.
$R_i(u^f, v^f), i = 1, n, bound(q) = \emptyset$	$\binom{2(n-1)}{n-1}(n-1)!$ [11]	$n!$ [11]	$3^n - 2^{n+1} + 1$ [8, 19]	$n(2^{n-1} - 1)$ [8, 19]
$R_i(u^b, v^f), i = 1, n, bound(q) = \{x_0\}$	$\binom{2(n-1)}{n-1}\frac{1}{n}$	1	$\frac{n^3-n}{6}$ [8, 19]	$n - 1$
$R_i(u^f, v^f), i \in \{i_1, \ldots, i_m\}$, $R_i(u^b, v^f), i \notin \{i_1, \ldots, i_m\}$ $bound(q) = \emptyset$	$\leq \binom{2(n-1)}{n-1}\frac{(n-1)!}{((\frac{n}{m})!)^m}$	$\frac{n!}{((\frac{n}{m})!)^m}$	$\leq \frac{((\frac{n}{m})^3+6(\frac{n}{m})^2+5(\frac{n}{m})+6)^m}{6^m}$	$\leq m(\frac{n}{m} + 1)^m$
$R_i(u^b, v^f), R_i(u^f, v^b)$, $i = 1, n, bound(q) = \{x_0, x_n\}$	$\binom{2(n-1)}{n-1}\frac{2^n}{n}$	2^n	$O(n^6)$	$O(n^4)$
$R_i(u^b, v^f), R_i(u^f, v^f)$, $i = 1, n, bound(q) = \emptyset$	$\binom{2(n-1)}{n-1}n^n$	n^{n+1}	$O(5.36^n)$	$O(n3.73^n)$

Figure 2: Bounds on the size of the search space for a dynamic-programming optimizer in the presence of binding patterns.

take all permutations. The number of plans considered by the dynamic programming algorithm also decreases dramatically from line one to line two. We remark that the number of bushy trees considered here is the same as that considered in the classical case for plans without Cartesian products[8, 19]. The next line refines the analysis by allowing a number m of relations R_i to be ff, the rest being bf. Of course, the exact formulas in each entry depend on which m relation one chooses: the table only shows their maximum values, obtained precisely when the m are chosen equidistantly (i.e., $R_1, R_{\frac{n}{m}+1}, R_{2\frac{n}{m}+1}, \ldots$). It is interesting to observe that the formulas in this line coincide with those in line 1 for $m = n$, and with those in line 2 for $m = 1$.

Line four considers an interesting particular case when we can "start at both ends". All left linear plans are obtained by shuffling a join from the left $R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_k$ with one from the right, $R_n \bowtie R_{n-1} \bowtie R_{n-2} \ldots \bowtie R_{k+1}$) (e.g. like in $R_1 \bowtie R_2 \bowtie R_n \bowtie R_{n-1} \bowtie R_4 \bowtie R_{n-2} \bowtie \ldots$): there are 2^n ways to do that. The complexity here is higher than in line two, but still far less than the case without limited access patterns.

Finally, line five illustrates a case when the complexity increases because of the additional access patterns (both ff and bf). The increase however is still within the same general complexity: it increases from one exponential to a higher exponential, and not to, say, a double exponential. For example (comparing lines 1 and 5) the partial query execution plans considered by the dynamic programming algorithm increased from $O(3^n)$ to $O(5.36^n)$.

The key conclusion we draw from the table is that for some query shapes the presence of limited access patterns significantly reduces the number of valid plans; since the dynamic programming algorithm will discard invalid plans, it will be able to explore a significantly smaller space in these cases. At the same time, in other cases, the complexity actually increases, but the increase stays within the general complexity of join ordering (i.e., exponential).

4.2 Experimental study

The analytical study provides only upper bounds on the size of the search space, in cases where mathematical analysis is possible. We now describe a series of experiments designed to measure the impact of the presence of binding patterns on the size of the search space for more general cases.

We study the effects of several factors on the size of the space: size of the query, number of variables, shape of the query graph and the type and structure of binding patterns. In our study we consider three measures: (1) the number of complete query execution plans, (2) the number of viable partial query execution plans, and (3) the number of valid but possibly non-viable query execution plans. The real size of the search space is (1), but the complexity of our algorithm is not proportional to this number, but to (2). The complexity of our algorithm without the viability test would be proportional to (3). We measure those numbers for both left-linear trees, as well as for bushy trees. Finally, we measure the effect of considering plans with Cartesian products.

To facilitate the experiments we implemented a random query generator which takes as input: (a) the number of relations, (b) the number of variables and (c) the desired shape of the query graph, and produces as output a conjunctive query with the required properties. We support 4 kinds of query shapes: chain queries, star queries, complete queries and randomly constructed. The first three types of queries are well known in the literature [11]. The fourth type of query is randomly generated such that a few (about one-sixth) of the attributes participate in three-ways joins and one third participate in two-ways joins. Cardinalities of the relations are generated randomly from 1000 to 10000 tuples. The selectivities were randomly chosen between 0.00001 to 1.0.

Perturbations on the binding patterns: a simple analysis shows that the size of the search space can be affected by two contradictory factors: (1) when binding pattern limitations become more restrictive, the size of the space decreases, and (2) when new binding patterns are added, the size of the space increases. In order to analyze those two contradictory factors and their respective effect, we apply the following strategy. In both cases, we start from the simple case when all the relations have $fff \cdots f$ access. We iteratively apply one of the following two transformations on this set of original input binding patterns:

- *bind*: take a binding pattern from the current set, transform one of the f's into a b, and put it back to the current set.
- *addBind*: do as in *bind* but do not remove the original binding pattern.

In the figures each point has been obtained from the results of 30 queries generated randomly with the same parameters. We show the average ratios between the number of plans after the transformations and the number of plans for the $ffff$ binding patterns.

The effect of the *bind* transformation: in figure 3 we show how the number of complete query execution plans

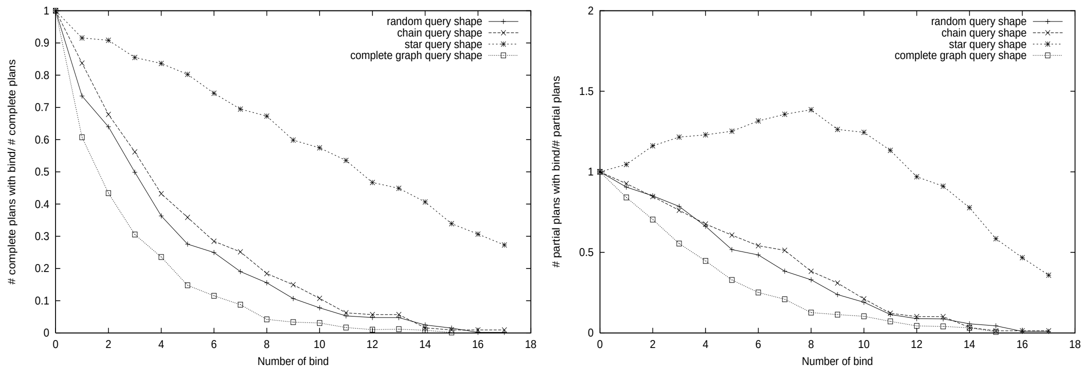

Figure 3: The evolution of the valid partial query execution plans and the complete query execution plans depending on the number of *bind* transformations.

and the number of partial viable query execution plans vary with the number of applications of the *bind* transformation. The queries have 6 relations, 35 variables, and 12 variables are bound in the query. We show the results for bushy trees, with Cartesian products, and for the four types of queries shapes. We can observe that the size of the search space is decreasing very quickly for all types of queries, as soon as binding patterns are introduced. For example, after 15 applications of the transformation, none of the queries have plans. The number of viable partial query execution plans is globally decreasing, even if sometimes, for star queries, it first slowly increases.

We ran the same tests when varying the other parameters of interest. We observed that the number of complete query execution plans depends on the shape of the query (it decreases faster for complete queries and much slower for star queries). On the other hand, the size of the query, the shape of the query execution plans (i.e., bushy vs. left-linear) and the consideration of plans with Cartesian products does not seem to have a strong effect on the relative average.

The effect of the *addBind* transformation: in Figure 4 we show how the number of complete query execution plan and the number of partial viable query execution plans vary with the number of applications of the *addBind* transformation (with the same parameters as in the previous case). The four types of queries manifested an exponential growth of the search space depending on the number of the *addBind* transformations. The number of viable partial query execution plans grows accordingly.

The number of non-viable plans: one of the important claims of our paper is that applying a viability test is essential. Figure 5 shows how the total (including non-viable) number of partial plans that can be obtained by a generative algorithm grows when non-viable plans are also considered. As shown by the two bottom curves in the figure, the number of viable partial plans and the number of complete plans are rapidly decreasing. However, the top curve shows that the total number of partial plans increases before it decreases. Hence, this underscores the importance of checking viable plans.

4.3 Discussion

The main problem raised by our analysis is that it is hard to predict how the size of the space will be affected: in some

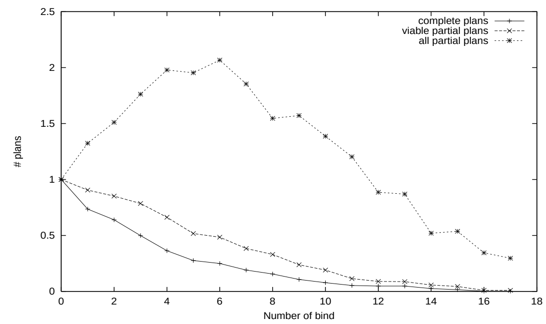

Figure 5: The number of partial plans increases with the number of *bind* transformations, but fewer plans are viable when we perform more *bind* transformations.

cases, it may be smaller than the traditional case, while in others it can be significantly larger. Hence, in order for an optimization algorithm to be effective in all cases, it must be able to handle large search spaces. The main problem with System-R style optimization when the search space is large is that the first plan is produced only towards the end of the optimization. Hence, the approach that we pursue in the next section is to employ a best-first search strategy whose main advantage is to produce a first plan relatively quickly, and improve it as the optimization proceeds.

5 Query optimization algorithm

In this section we describe our query optimization algorithm in detail. We begin by describing the key principles underlying the algorithm, and then focus on some of its important aspects.

5.1 Basic Principles

Our algorithm chooses the optimal plan in the search space characterized by the following properties: (a) bushy trees, (b) plans that include Cartesian products, and (c) all the possible placement of selections.

The algorithm is an extension of System-R style optimization, with the following principles:

- At every point in the optimization, the algorithm maintains a set of partial query execution plans, $\mathcal{S}$. Each plan $p \in \mathcal{S}$ is labeled with the equivalence class to which it belongs and its cost. The equivalence class is specified by

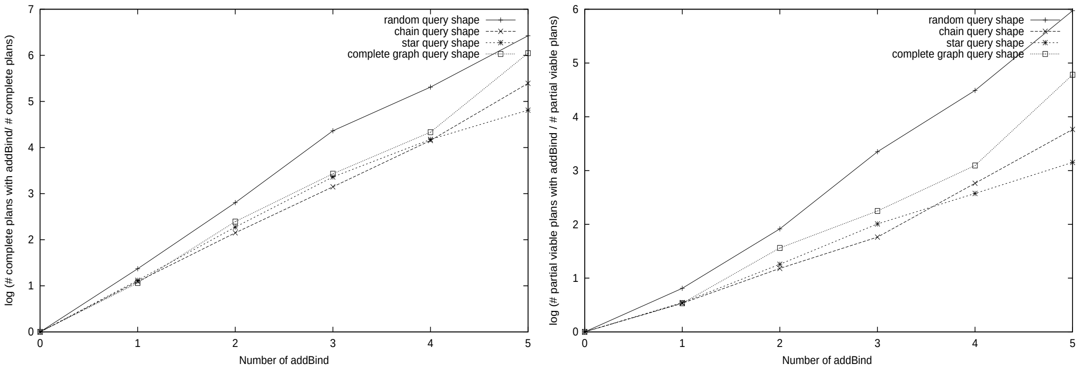

Figure 4: The evolution of the viable partial query execution plans and the complete query execution plans depending on the number of *addBind* transformations.

the set of conjuncts covered by p and its adornment.

- Initially, the set S contains atomic plans, i.e., plans for accessing a single relation. For a relation R, S contains an atomic query execution plan for every binding pattern describing an access pattern to the tuples of R.
- In the iterative step of the algorithm, we add new plans to S by combining existing plans in S using selection and join operations. We create one resulting plan for every adornment that satisfies the conditions on adornments described in Section 2, hence, creating only valid plans.
- At every point, S contains at most one plan for every equivalence class of query execution plans, which is the cheapest one found thus far.
- The choice of the partial query execution plans to be combined is based on a utility measure. This is significantly different from System-R, where equivalence classes are considered strictly in order of the number of conjuncts they cover.
- In the combination step we prune non-viable plans.

In the following subsections we discuss the main points in which our algorithm differs from System-R: (1) our search strategy, (2) our treatment of the placement of selections, and (3) the detection of useless equivalence classes. The algorithm is shown in Figure 6.

let $\mathcal{S}$ be the set of input binding patterns,
  extended to all variables in the query.
if the query has no execution plan (validity test)
then stop.
while new plans can be created **do**
  choose $p_1 \in \mathcal{S}$ maximizing the utility measure
  let $\mathcal{S}'$ be the set of plans that can be combined with p_1
  for each $p_2 \in \mathcal{S}'$ (in the order of their utility measure) **do**
    let p_3 be a (dependent) join product of p_1 and p_2
    if p_3 is not viable, **then** ignore p_3
    if $\exists p_4 \in \mathcal{S}$ s.t. p_4 covers p_3 and $cost(\sigma(p_4)) < cost(p_3)$
    then do $p_3 = \sigma(p_4)$
    if $\exists p_5 \in \mathcal{S}$ s.t. p_5 equiv. to p_3
    then if $cost(p_3) > cost(p_5)$
      then ignore p_3
      else in each $p_6 \in \mathcal{S}$ using p_5
        replace p_5 by p_3 and recalc. cost of p_5
    let $\mathcal{S} = \mathcal{S} \cup \{p_3\}$
    if $\exists p_7 \in \mathcal{S}$ s.t. p_3 covers p_7 and $cost(\sigma(p_3)) < cost(p_7)$
    then replace everywhere p_7 by $\sigma(p_3)$, recalc. cost

if $\mathcal{S}$ contains the equivalence class of the query
then return the optimal plan from $\mathcal{S}$
else
  let $\mathcal{S}''$ be the set of plans in $\mathcal{S}$ covering the query
  for each $p_8 \in \mathcal{S}''$ **do**
    generate all the possible placements for the selections
      not already included in p_8
    choose the optimal among those plans

Figure 6: Query optimization algorithm

5.2 Best First Search

System-R builds query execution plans by considering one equivalence class at a time. The equivalence classes are considered in increasing order of the number of conjuncts they cover. Therefore, the best query execution plan of a class and its cost are determined at one point and are not changed later. The disadvantage of this strategy is that the first complete query execution plan is obtained only at the last phase of the optimization. As the analysis in Section 4 showed, such behavior will not be acceptable in our context.

To address the problem of large search spaces, we employ a best-first search algorithm which interleaves the exploration of different equivalence classes. Specifically, we associate a *utility measure* with each partial execution plan. At each step of the search we choose the partial plan with the greatest utility measure, and try to combine it with plans that cover a disjoint set of conjuncts in the query.

The advantage of the best-first search algorithm is that we can tune the utility function to produce a complete plan relatively fast. Its main disadvantage is the cost of the extra bookkeeping needed to track changes to the costs of plans. Specifically, since we do not consider each equivalence class in isolation, the cost of the best plan for an equivalence class (and the cost of plans using it) may decrease over time, and hence the extra bookkeeping. We show in Section 7 that the tradeoff between the two factors is in our favor.

5.3 Placement of selections

As shown in the Example 3.3, it is possible that a plan with selections may be in the same equivalence class as a plan that does not contain any selection. Hence, if we completely ignore selections during the generation phase, we could miss the optimal plan. The goal of our algorithm is to consider selections in the combination phase only to the extent that it is required in order not to miss optimal plans.

The algorithm considers selections in the following fashion. Suppose we have created a new plan p which is the cheapest one found so far for its equivalence class. Before proceeding, the algorithm checks if it is possible to obtain an equivalent plan to p by applying a selection to a plan that already exists in $\mathcal{S}$. Specifically, we check if there exists a plan $p' \in \mathcal{S}$, such that p' covers p (i.e., p is equivalent to a selection applied to p'), and the cost of applying the selection to p' is less than the cost of p. In this case, the plan with a selection on p' is added to $\mathcal{S}$ instead of p.

Furthermore, the algorithm whether applying a selection on p enables to improve the best plan of another existing equivalent class. Specifically, the algorithm checks whether there exists $p' \in \mathcal{S}$ such that a selection on p is equivalent to p', and the cost of the selection on p is less than the cost of p'. In this case, the algorithm replaces the plan p' in $\mathcal{S}$ by the plan with a selection on p.

The effect of the two steps described above is that the set of equivalence classes maintained by $\mathcal{S}$ can be characterized as follows: if a class P is in $\mathcal{S}$, then it is there exists at least one query execution plan in P that uses only joins and no selections. In a sense, this property entails that the algorithm maintains a minimal number of equivalence classes, while still obtaining the optimal plan for the query.

As a result of the above property, the equivalence class corresponding to the original query may not be in $\mathcal{S}$ at the end of the generation phase. In this case, it is easy to check that the optimal plan for the entire query can be obtained by introducing selections in the optimal plans of the equivalence classes covering the query. Hence, the algorithm applies a second phase, which exhaustively enumerates all the possible placement of selections, but only in the optimal plans of the equivalence classes covering the query.

5.4 Detection of valid and viable equivalence classes

As stated earlier, our algorithm considers only valid plans and viable query execution plans. The validity of the query execution plans resulting in the combination step is guaranteed by the way we combine plans and generate adornments for the resulting plan.

An algorithm for testing the viability of a partial plan p while optimizing a query Q for can be obtained as follows. Let R be a fresh relation symbol. Construct a query Q' by removing from Q all subgoals that appear in p, and replacing them with $R(\bar{X})$, where $\bar{X}$ are Q's variables in p. Associate a single binding pattern with R, namely the adornment of p. It is easy to verify that the plan p is viable if and only if Q' has a query execution plan. Checking whether a given query has a query execution plan can be done by a simple greedy algorithm [10].

6 Implementation

We implemented our algorithm as well as a variant of the dynamic programming algorithm. In order to obtain a fair comparison, we extended the dynamic programming algorithm with a viability test. We use the same data structures (as described shortly) for the two algorithms, and we were careful to ensure that the optimizations made in the data structures to efficiently support best-first search do not bias the running times against dynamic programming. The implementation has been done in Java, using JDK 1.0.

A crucial issue that was considered in the implementation is developing a data structure for storing the set of partial plans that have been constructed (denoted by S). An optimal such structure would need to efficiently support the following accesses to the set of plans: (a) for a plan p, find all plans q in S, such that p and q have disjoint sets of conjuncts (i.e., the join candidates for p); (b) for a plan p, find an equivalent plan p' in S; (c) for a plan p, find all the plans q that cover p qnd (d) for a plan p, find all the plans q that are covered by p.

Given these requirements and the observed frequencies of the different accesses, we decided to adapt the following indexing structure for S. Plans are clustered by the set of conjuncts that compose them; note that the join candidates are the same for all the elements of a cluster. In order to avoid repetitive computation of the joinable clusters, the link between joinable clusters is established and materialized when the cluster is given its first member. In addition, the plans in each cluster are indexed by their adornments. It should be emphasized that since equivalent and covering plans belong to the same cluster, and the size of the clusters is relatively small, optimal performance was achieved by not adding structures for indexing equivalent and covering plans. Finally, in order to support cost recalculation due to best-first search, every plan contains a link to the plans using it.

In our experiments we considered a relatively simple cost model. The cost is derived from the cost of the leaf data accesses and standard formulas for computing the cost of joins. Costs of selections are assumed to be negligible, even though they affect the cardinality of the results. As long as the cost model respects the monotonicity property, the choice of the model is irrelevant to the results we show in the experiments.

A best-first search algorithm is based on a utility function for choosing the next plan to expand. In our experiments we considered several measures, including (1) the number of conjuncts covered by a plan (2) cost of the plans, (3) number of free variables, and several combinations of the 1–3. Considering only measure (1) resulted in better performance (e.g., up to a factor of 4) in terms of total time and time to first solution, even though the quality of the plans produced early on were not as good as in several of the more complex measures. Considering complex utility measures produces better plans early on in the search but the overhead of the search is significant. Careful tuning of the utility measure is a subject of ongoing research. Results presented further use a simple depth-first search.

7 Experiments

Experiments were run on a SUN 4 SPARC, under Solaris, using JDK with 100Mb of memory. Clearly, the use of JAVA affects absolute running times of both algorithms. Every point in the graphs is obtained by averaging over 20 queries generated randomly with the same parameters. All the experiments are done with queries including 10 relations and 50 variables. The following two sections quantify the gain in terms of finding the first solution and the price for total optimization time.

7.1 Time to First Solutions

Figure 7 (left) shows the time taken to obtain the first solution for our algorithm and the dynamic programming one, while increasing the size of the search space. We observe that the time to first solution for our algorithm is almost constant as the size of the search space increases, while dynamic programming degrades considerably.

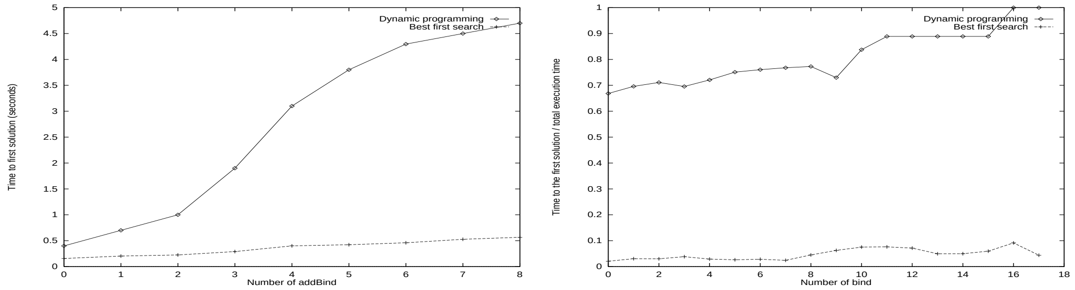

Figure 7: The left graph shows the influence of the *addBind* transformation on the absolute time taken to find the first solution The graph on the right shows the ratio between the time to first solution and the time for exhaustive search, in the presence of the *bind* transformation.

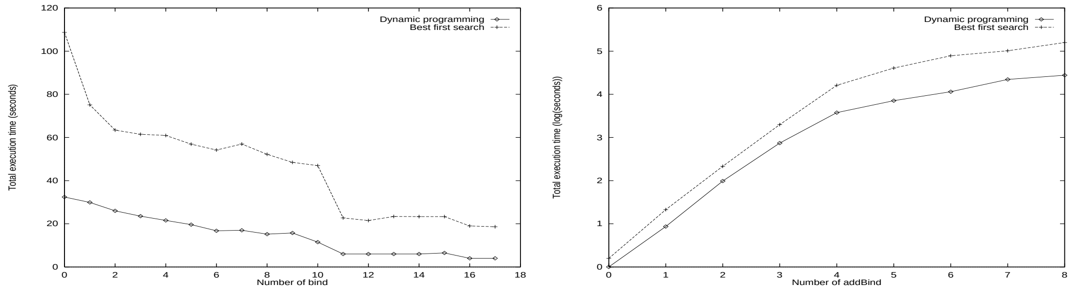

Figure 8: The graphs show the time for exhaustive search in the case of varying the number of *bind* transformations (left) and varying the number of *addBind* transformations (right).

Figure 7 (right) shows the ratio between the time to first solution and the total optimization time for both algorithms, when the size of the search space decreases. We observe that the ratio for our algorithm is relatively constant, while it grows for dynamic programming. Finally, it is important to emphasize that our algorithm produces solutions in a relatively steady pace. Hence, we are more likely to obtain a good solution even before dynamic programming produces its first.

7.2 Time for Exhaustive Search

Figure 8 compares the running times for exhaustive search for the two algorithms, as we vary the number of *bind* transformations (left) and as we vary the number of *addBind* transformations (right). We observe that in both cases dynamic programming has a better running time. In the case of *bind* transformations (when the size of the search space decreases) our algorithm takes more than double time than dynamic programming in the worst case. As the number of *bind* transformation increases, and hence the size of the space decreases, the differences between the running times are negligible. In the case of *addBind* transformations, the running time of both algorithms grows exponentially (note that the Y axis is on a logarithmic scale). Even though our algorithm performs worse, the general growth tendency is the same as for dynamic programming.

In conclusion, we have shown that our algorithm produces first answers considerably faster than dynamic programming. In cases when the search space is relatively small, the additional price paid by our algorithm is not significant. Finally, we argue that in the cases where we do much worse than dynamic programming are anyway cases in which dynamic programming is not a viable strategy and a non-exhaustive search algorithm is needed.

8 Discussion and Related Work

We described a query optimization algorithm which extends System-R style optimization to accommodate limited access patterns to the data. Our algorithm has several important features that are necessitated by the results of our analysis of the properties of the search space arising in the presence of limited access patterns, and a theoretical and experimental study of the size of the space. In particular, our algorithm searches the space of annotated query plans, and prunes as early as possible in the search plans that are invalid or are not viable. Furthermore, to perform well when the search space is large, the algorithm employs a best-first search strategy to produce a complete plan early in the optimization process. The algorithm also handles the placement of selections in a way that is tailored to this new context.

A natural question to ask is whether one of the other query optimization paradigms such as the transformational or randomized approach would be more appropriate. For example, in a transformation-based approach [16] the optimizer would start with some initial complete plan, and apply transformations to it in order to find an optimal plan. However, this approach requires a set of transformation rules that take one valid plan into another. In our context, the classical transformation rules such as associativity and commutativity of joins do not have this property, and therefore applying this approach is quite a bit more subtle. The situation is even worse for a randomized approach, because we cannot be guaranteed to cover only valid plans. Finally, the analysis of the search problem that we provided in this paper can facilitate future attempts to apply different search paradigms.

In most cases, the key reason for the existence of limited access patterns is the mismatch between the logical and physical views of the data. In our work, binding patterns were used to describe such mismatches. Tsatalos et al. [17] describe GMAPs which are also a mechanism for describing different storage patterns of the data. Using GMAPs, one can describe storage structures in which the stored data is a result of projections, selections and joins on the logical schema of the data, e.g. secondary indexes, path indexes and field replication. GMAPs and binding patterns characterize disjoint sets of mismatches between the logical and physical views of the data. To combine the two families of mismatches, we need to extend the algorithm of in [17] in several ways. First, as we did here, we need to consider annotated query execution plans. Note that in [17] the execution plans manipulate GMAPs (which can be thought of as materialized views) rather than database relations. Second, the join enumeration algorithm needs to consider plans of larger size. It follows from [13] that in the combined context of binding patterns and GMAPs, the query execution plan may require more joins than the number of relations in the query, and hence a relation mentioned once in the query may appear in more than one leaf in the query execution plan.

As we noted the problem of limited access to stored data also arises in the context of data integration. Hence, the problem of building query execution plans when only limited access patterns are available has been considered in work on data integration [10, 13, 9]. However, in that work they addressed the question of whether there exists some ordering of accesses to the data sources such that an answer to the query can be obtained. The question of finding an optimal order was not considered. A related issue is query optimization when the capabilities of the data sources are varying. Haas et al. [5] consider query optimization in the context of the Garlic system, where each data source may have different capabilities for performing joins internally. Vassalos and Papakonstantinou describe a powerful language for describing source capabilities [18].

The cost-based query optimization problem in the presence of binding patterns has been considered in [21]. The authors propose two heuristic-based algorithms, a greedy (inflationary) one, and a cluster-based algorithm. Moreover, they show that for the specific cost model they consider, the optimal plan is in the space of left-deep plans, and they show that the proposed algorithms have interesting properties (optimal or n-competitive for few conjuncts). In contrast, our algorithm is guaranteed to find the optimal plan, and is independent of the cost model.

Rather than dynamically find a query execution plan for an arbitrary given query, the work in [20] tries to statically compute the family of answerable queries, given a set of binding patterns. The authors consider a more ellaborate set of adornments, but do not address the problem of finding an optimal plan for the answerable queries.

This work is part of a bigger effort to build a query optimizer for contexts in which there is a mismatch between the logical and physical views of the data. As we already mentioned, one future direction is building an optimizer that can support both GMAPs and binding patterns. A second direction is to extend our optimizer to explore query execution plans that are directed acyclic graphs rather than trees. It has already been noted that such plans are useful even in the traditional optimization context, but this is even more so in the presence of limitations on access patterns.

References

[1] S. Chaudhuri, U. Dayal, and T. Yan. Join queries with external text sources: Execution and optimization techniques. In *Proc. of ACM SIGMOD*, 1995.

[2] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. of VLDB*, 1993.

[3] S. Chaudhuri and K. Shim. optimization in the presence of user-defined predicates. In *Proc. of VLDB*, 1996.

[4] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[5] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of VLDB*, Athens, Greece, 1997.

[6] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *Proc. of ACM SIGMOD*, 1996.

[7] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of ACM SIGMOD*, pages 267–276, 1993.

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

[9] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, Bombay, India, 1996.

[10] K. A. Morris. An algorithm for ordering subgoals in NAIL! In *Proc. of ACM PODS*, Chicago, Illinois, 1988.

[11] M.Steinbrunn, G.Moerkotte, and A.Kemper. Heuristic and randomized optimization for the join. *VLDB Journal*, 6(3), 1997.

[12] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of tranformation-based join enumeration. In *Proc. of VLDB*, 1998.

[13] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *Proc. of ACM PODS*, San Jose, CA, 1995.

[14] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *JLP*, 23(2), 1995.

[15] B. Reinwald and H. Pirahesh. Sql open heterogeneous data access. In *Proc. of ACM SIGMOD*, 1998.

[16] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proc. of ACM SIGMOD*, 1996.

[17] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2), 1996.

[18] V. Vassalos and Y. Papakonstantinou. Describing and using the query capabilities of heterogeneous sources. In *Proc. of VLDB*, Athens, Greece, 1997.

[19] W.Scheufele and G.Moerkotte. Efficient dynamic programming algorithms for ordering expensive joins and selections. In *Proc. of EDBT*, 1998.

[20] R. Yerneni, C. Li, H.Garcia-Molina, and J.Ullman. Computing capabilities of mediators. In *Proc. of ACM SIGMOD*, Philadelphia, 1999.

[21] R. Yerneni, C. Li, H.Garcia-Molina, and J.Ullman. Optimizing large join queries in mediation systems. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Jerusalem, Israel, 1999.