Extending the Search Strategy in a Query Optimizer*

Rosana S. G. Lanzelotte[1], Patrick Valduriez
INRIA - Rocquencourt
78153 - Le Chesnay cedex - France

Abstract

In order to cope efficiently with simple or complex queries as well as different application requirements (e.g., ad-hoc versus repetitive queries), a query optimizer ought to support an extensible search strategy that can ideally reduce to enumerative, randomized or more recent genetic search algorithms. In this paper, we give a solution to the extensibility of the query optimizer search strategy. This solution is based on the object-oriented modeling of the query optimizer, where the search space and the search strategy are independently specified. It is illustrated by the application to different search strategies. This modeling facilitates the specification of assertions that enforce the successful termination of the search process.

1. Introduction

Query optimization refers to the process of producing an "optimal" *execution plan*, for a given query, where optimality is with respect to a cost function to be minimized. This is made difficult by the necessary trade-off between optimization cost and quality of the generated plans (the latter translates into query execution cost). A "high" optimization cost may be acceptable for a repetitive query since it can be amortized over multiple executions. However it is not practical for ad-hoc queries that are executed only once. The cost of optimizing a query is mainly incurred by the investigation of the solution space for alternative execution plans. Typically, these plans are abstracted in terms of processing trees [Krishnamurty86] to capture in a compact way the aspects that are essential for cost estimation and optimization.

As the solution space gets larger for complex queries, the *search strategy* that investigates alternative solutions is critical for the optimization cost. Traditional query optimization uses an *enumerative* search strategy which considers most of the points in the solution space, but tries to reduce the solution space by applying heuristics. The System R optimizer [Selinger79] exemplifies this approach by restricting the solution space to binary processing trees and using dynamic programming for searching. Enumerative strategies can lead to the best possible solution, but face a combinatorial explosion for complex queries (e.g., a join query with more than ten relations) [Ibaraki84]. In order to investigate larger spaces, *randomized* search strategies have been proposed to improve a start solution until obtaining a local optimum. Examples of such strategies are simulated-annealing [Ioannidis87] and iterative-improvement [Swami88]. With the same objective, *genetic* search strategies [Goldberg89] can be applied to query optimization, as a generalization of randomized ones [Eiben90]. Randomized or genetic strategies do not guarantee that the best solution is obtained, but avoid the high cost of optimization. As an optimizer might face different query types (simple vs. complex) with different requirements (ad-hoc vs. repetitive), it should be easy to adapt the search strategy to the problem, which implies some form of extensibility.

Extensibility in query optimization has been studied in the framework of extensible database systems [Graefe87, Lohman88]. Extensible query optimizers have primarily focused on adapting to extensions of the search space (e.g., new features of the database language or physical storage system). However, they have not stressed the extensibility of the search strategy. In particular, it is difficult, if not impossible, to implement randomized or genetic strategies in such extensible optimizers. The main reason is the adoption of a rule-based approach [Freytag87], which is appropriate for query rewriting (e.g., using algebraic restructuring rules) but inconvenient for specifying the search strategy, which is essentially procedural.

---

* This work was partially funded by the Esprit project EDS.

1 Visiting INRIA on leave from the Pontifícia Universidade Católica do Rio de Janeiro (PUC-RIO).

In this paper we give a solution to the extensibility of the search strategy in a query optimizer. This solution has three important aspects: the independence of the search strategies from the search space, the viewing of query optimization as a particular case of a search system, and the object-oriented modeling of query optimization search systems to gain extensibility of both the search space and the search strategy. This is illustrated by modeling within the same framework different enumerative, randomized and genetic search strategies. Furthermore, we show how the search strategies thus produced can be controlled in the sense that successful termination can be enforced by assertions. The isolation of the search strategies from the search space makes the solution compatible with that of [Valduriez89] and thus applicable to more general database programming languages which can be deductive or object-oriented [Lanzelotte90]. However, for simplicity and without loss of generality, we limit ourselves to relational queries.

The rest of the paper is organized as follows. In Section 2, we model the search space, which describes the query optimization problem and the associated cost model. In Section 3, we view query optimization as a generic search problem and introduce a class hierarchy to model search strategies. These two class hierarchies are the building blocks for the optimizer. In Section 4, we illustrate the use of the previous classes in specifying different search strategies. In Section 5, we show how the behavior of the generated search strategies can be controlled by means of assertions. Section 6 concludes and indicates the status of a prototype that implements this solution.

2. Modeling the Search Space

In this section, we introduce the optimization problem following the model of [Krishnamurty86] for relational query optimization. In particular, we model the search space independently of the optimization algorithms and related heuristics. Therefore, a query execution plan is modelled as a processing tree which captures all the optimization decisions for executing the query, e.g., join ordering, join algorithms, etc. In order to keep this section short, we limit ourselves to conjunctive select-project-join queries. After extending the definition of [Krishnamurty86], we present the operations to manipulate processing trees and incorporate them within a SearchSpace class hierarchy.

2.1. Processing Trees

A *processing tree* (PT) is a labelled binary tree where the leaf nodes are relations of the input query and each non-leaf node is a temporary relation. Different from [Krishnamurty86], we consider a temporary relation to be materialized only when explicitly indicated. Thus, we are able to model pipelined and non-pipelined joins with the same binary PT. This removes the need for n-ary nodes to model pipelined joins and facilitates the uniform specification of operations on PTs.

A *join node* is a non-leaf node of a PT that captures the join between an outer join node and an inner join node. The *outer join node* corresponds to the operand relation from which tuples are retrieved first by the join algorithm. The *inner join node* corresponds to the operand relation from which tuples are retrieved next, possibly using join values of the outer relation. If there is no join predicate connecting the outer join node to the inner join node, the join reduces to a Cartesian product. The distinction between the two operand relations is important because some join cost formulas (e.g., nested loop join) are not symmetric with respect to the inner and outer relations [Selinger79]. We illustrate these definitions with the following query:

Select * From R1, R2, R3 **Where**
R1.A=R2.A **and** R2.B=R3.B **and** R1.C<100

Figure 1 shows two different PTs for the sample query. We always represent the outer relation as the left child of a join node and the inner as the right one.

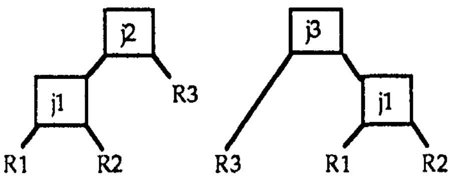

Figure 1: Two PTs for the sample query

The inner join node reduces to a base relation if the optimizer does not investigate bushy PTs (e.g., the second one in Figure 1). The ability of changing the type of inner to cope with bushy or non-bushy PTs, as well as with PTs involving Cartesian products or not is called *adaptability* of the search space in [Ono90].

2.2. Operations on PTs

The optimization process consists essentially in building and modifying PTs. We now describe the operations on PTs, that constitute the basic actions of a query optimizer which are controlled by its search strategy.

We call *PT generation* the process of successively building join nodes. A step in this process, called *expansion*, is to connect a PT to a new node by adding a relation (see Figure 2). A PT is *complete* when its root join node involves all the operand relations of the input query (e.g., j2 and j3). Conversely, a PT is incomplete when it does not capture at least one operand relation.

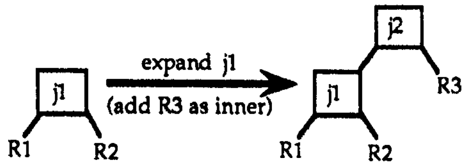

Figure 2: Expanding a join node

Randomized search strategies require the ability of applying *transformations* to complete PTs to generate *neighbor* PTs (which are also complete PTs). This phase is referred to as *PT modification*. Examples of transformations are the exchange of two relations inside a PT [Swami88, Ioannidis90]. Our model for PTs is appropriate for implementing transformations, because we can distinguish inside each join node the incremental part added by an expansion (i.e., the inner part). Then, when transforming a PT, it is not necessary to rebuild it completely.

Transformations can be specified with rules as in transformation-based optimizers [Freytag87]. The proposed definition of join node enables to use the join operator as a recursive functional symbol for describing PTs in a syntactical way. Thus, it is the basis for specifying rewrite rules. For example, join node j2 of Figure 1 is specified as join(join(R1,R2),R3). The left join exchange rule of [Ioannidis90], which is illustrated in Figure 3 ((a join b) join c → (a join c) join b), is written as join (join (a, b),c) → join (join (a, c), b).

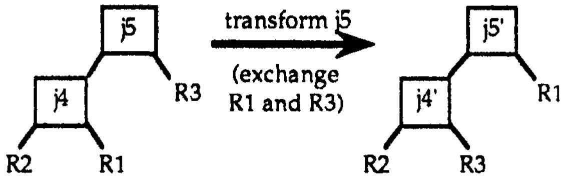

Figure 3: Transforming a PT

The basic actions in genetic strategies are crossover and mutation. A *crossover* consists in selecting two (the parents) from a population of complete PTs and generating two *offsprings* (the descendants) according to some principle (e.g., by merging characteristics of the parents). The individuals to be crossed are chosen at random, but the choice is biased by their *fitness*. The fitness is related to the function to be optimized (i.e., the cost function). Thus, new generations are expected to contain better individuals than the previous ones, because they are built from the features of selected parents. For example, consider a query involving 10 relations and two non-bushy PTs a and b represented by their sequences of relations:

a = R9 R8 R4 R5 R6 R7 R1 R3 R2 R10
b= R8 R7 R1 R2 R3 R10 R9 R5 R4 R6

The partially matched crossover operator (called PMX) is one possible crossover operator [Goldberg89]. It consists in choosing at random two points in the sequences corresponding to the individuals. Two descendants are generated such that the central sections (inside the two points) are exchanged and the other relations are exchanged accordingly. Then, PMX applied to individuals a and b produces two descendants a' and b'.

a'= R9 R8 R4 | R2 R3 R10 | R1 R6 R5 R7
b'= R8 R7 R1 | R5 R6 R7 | R9 R2 R4 R3

PMX or other crossover operators can be specified by means of functional syntactical transformation rules. *Mutations* may also come, when generating the new individuals, with a small probability (as in Nature). A mutation applies to a unique individual and has the same nature of transformations in randomized strategies. The incremental nature of PT nodes are also important for efficiently implementing crossover actions.

2.3. Search Space Class Hierarchy

The specification of the optimization search space is influenced by the input query and the nature of investigated PTs (i.e., bushy or not, involving Cartesian products or not). Figure 4 shows the SearchSpace class hierarchy. In the graphical representation of class hierarchies throughout this paper, the name of the class and its attributes are shown inside the ovals. The attached methods are shown outside. The type of an attribute or of the returned value from a method is denoted as *: type* (when the type is a set, it is denoted as {type-of-element}). Methods not in bold are *deferred*, i.e., they are actually implemented at a lower-level class (e.g., method **phytranslate** is deferred at the SearchSpace class level and actually implemented at the spjQuery class level). An arrow between two classes is for inheritance ("inherits from").

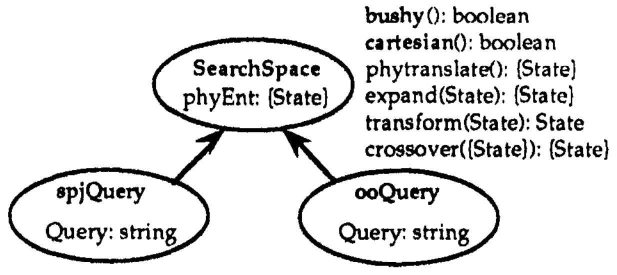

Figure 4: The Search Space class hierarchy

Class SearchSpace is specialized to conform to different types of input queries (e.g., relational or object-oriented ones). Some of the attached methods implement the basic operations on PTs (i.e., **expand, transform** and **crossover**). A method **phytranslate** is attached to any class that specializes the SearchSpace class. It implements the translation of the input query to the physical database schema, stored in attribute **phyEnt**, that is a set of subparts of the input query used when building PTs. A subpart is an object of the State class, whose hierarchy is shown in Figure 5.

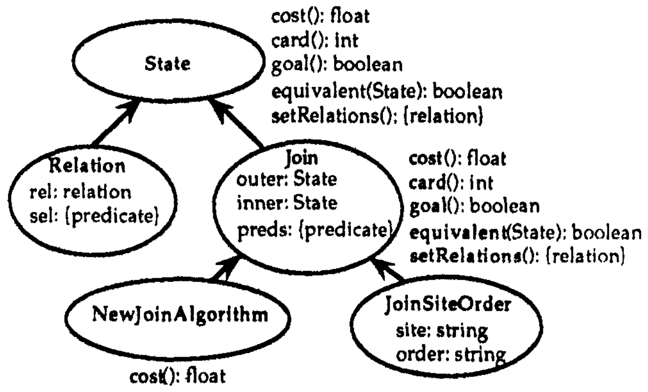

Figure 5: State class hierarchy

Class State is specialized by the Relation class, whose objects are the individual relations of the input query (together with selection predicates), and the Join class, whose objects are the generated PT join nodes. From the previous discussion, an instance of the Join class has attributes *outer* and *inner* of type State (if only inners of type Relation are allowed, bushy PTs are not investigated). Class Join can be further specialized to cope with several join algorithms (e.g., NewJoinAlgorithm), attributes specific to a given environment (e.g., JoinSiteOrder), the materialization or pipelining of intermediate results, etc.

In [Lanzelotte91b] we model the search space for object-oriented databases. There, the Join class corresponds to the *implicit* join (i.e., due to the connection between objects and their sub-objects), which is implied by the database schema. *Explicit* joins due to the occurrence of join predicates in the query are modelled as a specialized class.

For simplicity, in the rest of this paper, we adopt the definition of Join that corresponds to the Join class in Figure 5. It models a pipelined nested-loop join between the outer and the inner relations.

3. Query Optimization: a search problem

To establish the framework for modeling search strategies, we view the query optimization problem as a search problem in the most general sense. In this section, we propose an object-oriented modeling of search systems through a class hierarchy which can be easily extended to support various query optimization search strategies. We first introduce an enumerative search method which is further specialized to implement randomized and genetic strategies. Doing so we are able to identify common aspects of several search strategies and to specify them separately from other features of a query optimizer (e.g., the cost model). The resulting modeling is powerful enough to allow the easy implementation of different known optimizers within the same framework as well as the dynamic change of the search strategy, as suggested in [Ioannidis90].

3.1. Object-oriented Modeling of Search Systems

To formulate a search problem the following elements are required [Shapiro87]:

- *states*, which are configurations of the objects relevant to the problem; whether a state describes the problem totally or partially constitutes a design decision in a search problem; distinguished states are the *initial* state and *goal* states
- *actions* that, when applied to one state, generate a set of successor states

This framework applies to query optimization in two different ways. In PT generation, the *initial* state is constituted by the relations and predicates from the input query together with related schema information, *states* are join nodes, an *action* is an expand method and *goal* states are join nodes that correspond to complete PTs (e.g., j2 and j3 in Figure

1). In PT modification, which occurs in randomized and genetic strategies, states are complete PTs, an action is a **transform** or a **crossover** method and the goal description involves a stop condition based on specific parameters of the search strategies (e.g., time constraint in iterative-improvement, temperature in simulated-annealing or number of generations in genetic strategies).

3.2. A Class Hierarchy for Enumerative Search

In enumerative strategies, several states are successively inspected for the optimal solution (e.g., by breadth-first, best-first or depth-first search). The SearchStrategy class hierarchy shown in Figure 6 grasps the essence of enumerative strategies.

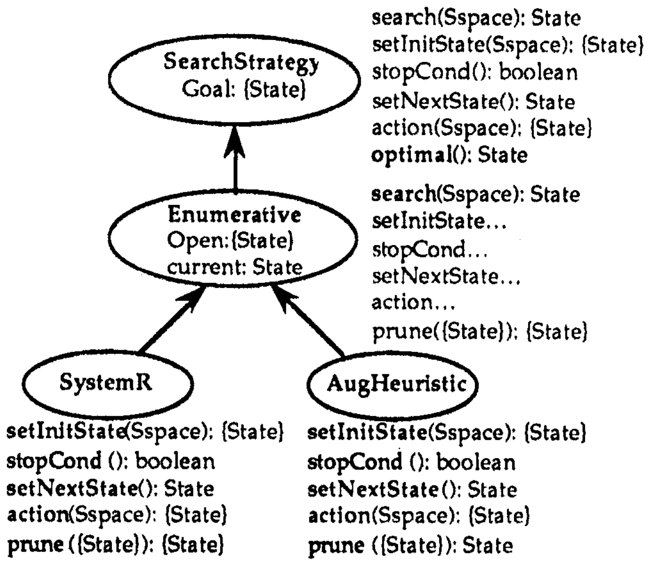

Figure 6: Search Strategy class hierarchy for enumerative strategies

Algorithm 1 implements the **search** method of the **enumerative** class, that performs the generation of PTs. It is based on a generic branch-and-bound search strategy [Papadimitriou82]. The other methods used within its body constitute the *extensibility primitives*. They capture the properties that, when modified, change the behavior of the search strategy. By overloading them, the same enumerative algorithm can be used for implementing different search strategies, as shown in Section 4. In the specification of the methods and extensibility primitives, we denote a method or attribute of an object or a set of objects by qualifying it with the corresponding variable name (e.g., current.goal). We use capital letters for beginning set-valued variable names and small letters for single-valued ones. The signature of a method determines the class to which it is attached and type of the returned value (e.g., class::method() : returntype).

An enumerative search strategy is first characterized by the choice of the next state to apply an action on, performed by the **setNextState** method, which determines in which way the states are investigated. If its implementation is such that the least recent state is chosen, then the search strategy is breadth-first. If it chooses the most recently generated state, then it implements depth-first search. The method **action** decides of a number of successors to be generated. Heuristics are used to discard *bad* states, which are recognized by comparison with equivalent ones and **pruned** from the set Succ of successor states. Pruning is a feature of the so-called *branch-and-bound* algorithm, which is a variant of the enumerative search one.

Algorithm 1: Enumerative (branch-and-bound)

```
enumerative::search (Sspace: SearchSpace): State
begin
   Open := setInitState (Sspace);
   while not stopCond ()
   begin
      current := setNextState ();
      Open := Open - {current};
      if  current.goal ()
      then  Goal := Goal ∪ {current}
      else  begin
         Succ := action (Sspace);
         Succ := prune (Succ);
         Open := Open ∪ Succ
         end
   end;
   return optimal ()
end
```

Figure 7 shows the states generated by Algorithm 1 implementing the breadth-first search strategy of System R [Selinger79] for the sample query.

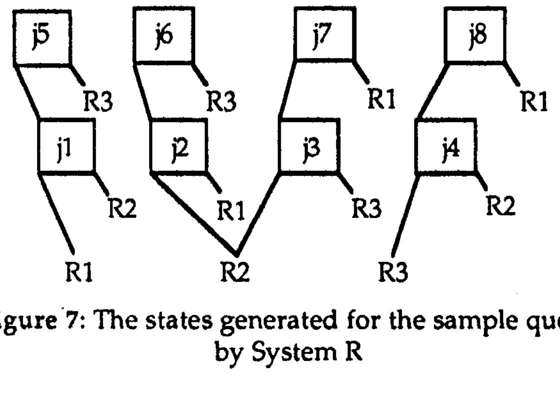

Figure 7: The states generated for the sample query by System R

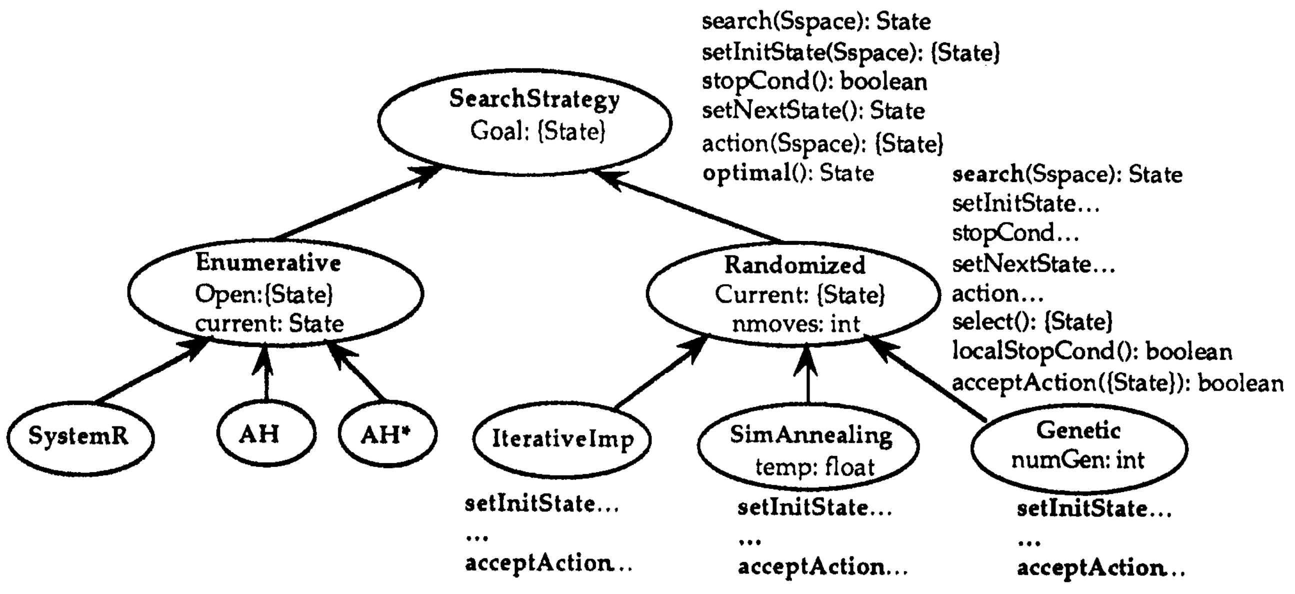

Figure 8: Class Hierarchy for implementing Randomized and Genetic Search Strategies

3.3. Class Hierarchy for Randomized Search

While enumerative search strategies consider the state space as a whole, randomized ones concentrate on searching a local optimal solution around some particular points. They consist of two steps. First, one or several start solutions are obtained by depth-first search, possibly using some heuristics. Second, the start solutions are improved until *local optimal solutions* are obtained. In this second phase, the search system framework is such that each state matches the goal description and an action is rather a *transformation* of a goal state into another goal state. Neighboring solutions are randomly obtained by applying transformations. A local optimal solution is the best among all the neighboring solutions. Randomized strategies involve the definition of several parameters (the number of transformations to apply, the criterion for accepting a transformation, the criterion for considering a solution to be a local optimal one,etc).

Genetic strategies start with a population of solutions, from which new generations are built by successively applying *crossovers* to individuals of the original population. A crossover generates two new individuals obtained by merging characteristics of the parents. If mutations are allowed, then an action in a genetic strategy is a crossover possibly followed by a mutation. Compared to other strategies, the genetic one is easily adaptable to the problem: by changing some parameters (e.g., the number of generations), the same genetic algorithm can be customized to get faster to an acceptable solution or to spend a longer time to get to a better solution. Randomized strategies can be modelled as particular cases of a genetic algorithm [Eiben90]. It is sufficient to reduce the size of the population to one and to produce new individuals only by mutations.

Figure 8 presents an extension of the class hierarchy for a search system that supports randomized and genetic strategies. Now, an action corresponds to the application of a transformation or a crossover to complete PTs.

Algorithm 2: Randomized (abstract genetic algorithm)

```
randomized::search(Sspace:SearchSpace) : State
begin
   setInitState (Sspace);
   while not stopCond ()
   begin
      nmoves := 0;
      while localStopCond ()
      begin
         Current := select ();
         Succ := action (Sspace);
         if acceptAction (Succ)
         then Goal := (Goal - Current) ∪ Succ;
         nmoves := nmoves + 1
      end;
      setNextState ()
   end;
   return optimal ()
end
```

Algorithm 2 implements the generic control strategy for randomized strategies. It is based on the Abstract Genetic Algorithm proposed in [Eiben90] for modeling at the same time the genetic and simulated-annealing strategies. The methods in bold are the extensibility primitives for them.

4. Customizing the Search Strategy

In this section we show how to customize the search strategy by overloading the extensibility primitives introduced by the SearchStrategy class hierarchy. They capture the common aspects of various known search strategies. The advantages of this approach are twofold. First, an already implemented search strategy can be easily tuned or modified. The motivation for this is that, even to well settled strategies (e.g., simulated annealing), several algorithms have been proposed [Ioannidis87, Swami88, Ioannidis90] and the differences between them go beyond the simple setting of parameter values. Second, new search strategies can be implemented with little effort.

4.1. Search Strategies for Generating PTs

To customize the search strategy for generating PTs, one must specify the methods attached to the **enumerative** class and its subclasses. These are: **setInitState**, **stopCond**, **setNextState**, **action** and **prune**. They have been used in the implementation of the **search** method shown in Algorithm 1. To specify them, the extensibility primitives associated to the instances of the Join class are used, which refer to the properties of join nodes. Examples of this is j.**setRelations**(), that returns the set of all the relations included in a join node j, and j.**equivalent**(k), which returns true if join node j is equivalent to k (according to some criterion, e.g., the set of contained relations).

Table 1 summarizes the specifications of the extensibility primitives for PT generation. Three cases are discussed here: the implementation of an enumerative branch-and-bound strategy, the one in [Selinger79] and the depth-first generation of a single or several PTs, which constitute the start solution for randomized or genetic strategies. In this table, we refer to the set of all relations of the input query by Relation.

	Branch-and-bound	**Augmentation Heuristic (one PT)**	**Augmentation Heuristic (a population of PTs)**
setInitState (Sspace)	return Sspace.phyEnt	j:=Sspace.phyEnt.**leastCard**() phyEnt := phyEnt - {j} return {j}	return Sspace.phyEnt
stopCond	(Open = {})	(Open = {})	(Open = {})
setNextState	j ∈ Open | j=Open.**leastRecent**()	j ∈ Open | j=Open.**mostRecent**()	j∈ Open | (j=Open.**mostRecent**() ∧ card(j.**setRelations**() > 1)) ∨ (j = Open.**leastCard**())
action(Ssp)	Ssp.**expand**(current)	Ssp.**expand**(current)	Ssp.**expand**(current)
prune (J)	{ j ∈ J | (∀j' ∈ J)(j'.**equivalent**(j) ∧ j.**cost**() < j'.**cost**()) }	j ∈ J | j=J.**leastCard**()	j ∈ J | j=J.**leastCard**()

Table 1: Extensibility primitives for PT generation

In the enumerative strategy of [Selinger79], as all the relations are used for starting PTs, the set Open is initialized with all of them. Search stops when no more open states exist . The extensibility primitive **stopCond** tests whether the Open set is empty. The search strategy proceeds by breadth-first. Thus, **setNextState** is specified using a method **leastRecent** on the Open set. Pruning eliminates expensive states that are equivalent to less expensive ones, where equivalence is related to the contents in terms of relations.

We now show that the same set of extensibility primitives presented for a branch-and-bound enumerative search strategy can be overloaded in order to transform Algorithm 1 into an algorithm that generates only one solution by depth-first search. We model the Augmentation Heuristic [Swami89], in which the relation with the least cardinality is chosen for starting a PT. This relation is eliminated from the phyEnt set, so that further calls to **search** will choose other relations. Compared to the previous version of **setInitState**, only one join node is generated, that corresponds to the relation with least cardinality. A depth-first search strategy is characterized by choosing the most recent state as the next one to expand. Pruning reduces the successors to only one state. Several heuristics can be used for choosing the state to be kept (five such heuristics have been proposed in [Swami89]). One possible heuristic is to keep the state that corresponds to the most selective join. This means that the intermediate result is the one of least cardinality.

In the case of a depth-first search strategy for generating a population of PTs, as needed in genetic strategies, all the relations are used for starting PTs. Actions proceed by depth-first search until one complete PT starting with each different relation is generated.

4.2. Randomized Strategies for Transforming PTs

Table 2 shows the specifications of the extensibility primitives of Algorithm 2 to implement either Iterative Improvement, Simulated Annealing or a genetic strategy.

4.2.1. Iterative Improvement

Iterative Improvement is characterized by the choice of several start states, one for each *run* (the inner loop in Algorithm 2). Both **setInitState** and **setNextState** generate one start solution (i.e., a complete PT) by calling AH.**search**, which implements the generation of one PT by Augmentation Heuristic. Each time AH.search is called, a new PT is depth-first generated starting at a different relation. For each start state, which is the unique element of the Goal set, neighbor states are obtained (i.e., by applying **actions**, which correspond to transformations) until a *local mininum* is reached. The modified PT replaces the original one if the **acceptAction** method returns true (i.e., if the cost of the transformed PT is less than that of the original one). The method **stopCond** corresponds usually to a time constraint [Swami89]. A local minimum is defined as the least costly solution in the neighborhood of the current state. Then, to guarantee that a local minimum was reached, all the neighbors of the current state should be tested, which would be very expensive. We simplify the criterion, by setting to zero the counter of transformations, **nmoves**, every time the current state is replaced by a neighbor conditioned by **acceptAction** (i.e., the replacement of the current state implies that the neighborhood has changed). Then, the method **localStopCond** can be related to the number of neighbors of a state. This has been estimated as card(phyEnt) * k (factor k has been proposed to be equal to 1 in [Swami89] and to 16 in [Ioannidis90]).

	Iterative Improvement	Simulated Annealing	Genetic Algorithm
setInitState (Sspace)	Goal := AH.**search(Sspace)**	Goal := AH.**search(Sspace)** temp := 2 * s.**cost**()	Goal := AH*.**search(Sspace)** numGen := 0
stopCond	elapsedTime > maxTime	temp < 1 ∧ Goal unchanged for 4 stages	numGen ≥ maxGen
localStopCond	nmoves > card(phyEnt)*k	nmoves > card(phyEnt)*k	nmoves > card(Goal)/2
select	Goal	Goal	**Goal.2randomFitness()**
action(Ssp)	Ssp.**transform**(Current)	Ssp.**transform**(Current)	Ssp.**crossover**(Current)
acceptAction (Succ)	(s ∈ Current) (s'∈ Succ) (s'.cost() < s.cost()) if true then nmoves := 0	(s ∈ Current) (s'∈ Succ) (s'.cost() < s.cost()) ∨ (s'.cost() > s.cost() ∧ Prob (temp,s,s'))	true
setNextState	Goal := AH.**search(Sspace)**	temp:=0.95*temp	numGen:=numGen+1

Table 2: Extensibility Primitives for implementing randomized and genetic strategies

4.2.2. Simulated Annealing

Contrary to Iterative Improvement, all the stages in Simulated Annealing (the inner loop in Algorithm 2) are performed over the same start state. Besides **nmoves**, the system has a temperature property, **temp**, that is set by **setInitState** and reduced by **setNextState**. The method **stopCond**, which is the global stop condition, is related to the temperature and not to the elapsed time (it corresponds to the fact that the system has *frozen*). It is important to parameterize this constraint, because several authors provide different definitions for it [Swami88, Ioannidis90]. To precise "unchanged for 4 stages", some temporary variables are needed that are not shown here. The specifications of **localStopCond** and **action** are the same as in Iterative Improvement. The criterion for accepting a transformation is different, because transformed PTs with higher cost than the original PT are accepted with some probability. Then, the method

acceptAction uses Prob, which is a boolean function that returns true with a probability that depends on **temp** and the costs of the compared states, usually $e^{-(s'.cost-s.cost)/temp}$. Accepting *bad* moves corresponds to perform what is called a *hill climbing*: on the other side of the hill there may exist a better solution.

4.2.3. Genetic Algorithms

To implement a genetic algorithm, a population is first generated by calling AH*.**search**, which implements Augmentation Heuristic for generating a population of PTs. Unlike with randomized strategies, where the Goal set contains a unique PT, it contains several PTs. Primitive **stopCond** is related to a parameter maxGen, which specifies the number of generations to be produced. The number of generations numGen is increased each time a new generation is produced, which is performed by the inner loop of Algorithm 2. This stops when a new generation with the same number of individuals as the previous one has been produced. The selection of the parents for crossover is performed by **select**, usually by applying a random function biased by the fitness of individuals (i.e., the ratio between their costs and the total cost of all the individuals). An **action** in this case is a **crossover** performed by any available operator, which generates two individuals of the new generation from two selected parents from the precedent generation. The new generation always replaces the previous one.

5. Enforcing Successful Termination

When the search strategy is extensible, it is essential to assure that the optimizer behaves as expected and that the process will end. The extensibility primitives that we proposed can also be used for specifying *assertions* that provide a form of monitoring the behavior of the optimizer. The introduction of some form of metacontrol by assertions depends on the implementation environment of the optimizer. In any case, it is important to be able to make explicit the conditions that guarantee the success of the optimization process.

To exemplify, we state the assertion for successful termination. *Successful termination* is attained if, when the search stops, at least one goal state has been obtained. The control of the optimization process is implemented by the search algorithms. In the two of them, a step forward is performed by the **action** method, that is responsible for generating successor states from the current state. Successful termination is, then, characterized by one of two conditions: either when stopCond is met there exists at least one state that matches the goal description (i.e., a complete PT) or there exists an open state that is still liable to an **action**. We state formally the assertion for successful termination:

$(\mathbf{stopCond}() \wedge Goal \neq \{\}) \vee (\mathbf{action}(Ssp) \neq \{\})$

We discuss separately the assertion in the case of each one of the algorithms.

5.1. Enumerative Strategies

In PT generation by Algorithm 1, the assertion can be rewritten as:

$(Open=\{\} \wedge Goal \neq \{\}) \vee (Ssp.\mathbf{expand}(current) \neq \{\})$

The methods attached to the Search Space class hierarchy can also be specified in a high-level way, for example:

Definition 1: A PT is generated by successively applying **expand** which is defined as
$Sspace.\mathbf{expand}(s) = \{ j \in Join \mid j.outer = s \wedge j.inner \in Sspace.\mathbf{setInners}(s)\}$

Recall that the definition of **setInners**, that specifies the set of possible inners for a join node, is referred to as *adaptability* of the search space [Ono90]. In the following definition, we assume that Relation is the set of all relations referenced in the input query, Predicate is the set of join predicates of the input query and an instance of Predicate has an attribute *relations* that is the set of relations referenced in the predicate.

Definition 2: Search space NX, in which bushy PTs and PTs with Cartesian products are not investigated, is characterized by the following specification for **setInners**

$$NX.\mathbf{setInners}(s) = \{ r \in Relation \mid (s.\mathbf{setRelations}() \cup \{r\}) \subseteq Relation \wedge (s.\mathbf{setRelations}() \cap \{r\}) = \{\} \wedge (\exists p \in Predicate)(\{r\}=p.relations-s.\mathbf{setRelations}())\}$$

Definition 2 includes the disjointness criterion of [Ono90], i.e., (s.**setRelations**() ∩ {r} = {}) besides the requirements of a join predicate and a bound on **setRelations** (i.e., (s.**setRelations**() ∪ {r}) ⊆ Relation). In our implementation, as any set of relations (i.e., of a State or of a predicate) is implemented through a bit string, it is very efficient to determine **setInners**.

The formalized assertions and the specifications of the extensibility primitives are the basis for proving the successful termination of the optimization process. The proofs are not shown here for space reasons (see [Lanzelotte91a] for more details).

5.2. Randomized Strategies

In randomized or genetic strategies, Goal ≠ {} is always true. A PT obtained by an **action**, which corresponds to the **transform** or **crossover** methods, should satisfy the same constraints that were posed when generating PTs (this is referred to as *valid* transformations in [Swami88]). Analogously to the **expand** method, the specifications of **transform** and **crossover** require the **setInners** definition.

5.3. Controlling the Behavior of the Optimizer

The ability to specify assertions for controlling the behavior of the optimizer using the extensibility primitives illustrates well their power of abstracting the optimization problem. Assertions can be used as a basis for an exception mechanism, which is worth in two ways. Either an exception means an error condition or it enables the dynamic change of the behavior of a program, which is useful in our context. For example, the specifications of some extensibility primitives can be changed during the optimization process to conform to some unexpected configuration of the input problem. An example is the possibility of moving from a search space that does not admit PTs with Cartesian products (NX) to one that does by changing the definition of **setInners**. Another example is to change the **goal** condition (usually s.**setRelations**() = Relation) to be also met in case of non-applicability of the **expand** method during PT generation:

s.**setRelations**() = Relation ∨ Ssp.**expand**(s) = {}
    ⇒ s.**goal**()

This **goal** condition prevents the optimizer from investigating PTs with Cartesian products. The search stops when the incomplete PTs which do not involve Cartesian products have been generated. Of course, the incomplete PTs must then be put together to form the complete PTs, but this task does not require any more search, unless the optimizer considers features that influence the execution time when commuting the operands of a Cartesian product (e.g., the sequence order or the site of the results).

The proposed object-oriented approach matches well the idea of using the violation of assertions as exceptions. The dynamic change of the extensibility primitives can be performed through late binding. New subclasses can be specified where the extensibility primitives are overloaded and the dynamic change of behavior is obtained by moving an object from one class to a subclass.

6. Conclusion

In this paper, we gave a solution to the extensibility of the query optimizer search strategy. This solution is based on the clear separation between the search space and the search strategies for which we provided an object-oriented design in order to gain extensibility. Therefore, we maintain high independence of the search space from the optimization algorithms and related heuristics. By viewing query optimization as a particular case of a search system, we were able to capture the extensibility primitives which can be customized to generate various search strategies. We illustrated our solution in specifying enumerative search strategies with different heuristics and three randomized strategies (Iterative Improvement, Simulated Annealing and genetic).

This approach can be useful in many ways. Overall, we can use it to build an optimizer with several search strategies, each one being best for a particular class of queries. Thus, the search strategy can be dynamically selected to achieve the desirable trade-off between optimization cost and execution cost for a given query. In the case of an ad-hoc query for which a randomized strategy is probably best, an "optimization budget" could be assigned to the query (somehow by the "user" or the compiler) in order to provide an upper bound for optimization cost.

An important result was that the extensibility primitives also provide a means for specifying assertions that enforce the successful termination of the optimization process. Unlike rule-based optimizers, our approach insists on the procedural specification of the control in search strategies.

To simplify our presentation, we limited ourselves to relational queries, in fact, conjunctive select-project-join queries. Thus, we were able to reuse the model of processing trees [Krishnamurty86] for specifying the search space. To deal with more general queries such as object-oriented or deductive queries, the same approach holds providing that the search space is changed (in [Lanzelotte91b] we extended this approach to object-oriented queries).

The proposed solution has been validated at INRIA by the implementation in C^{++} of an optimizer prototype, as part of the EDS database compiler [Bergstein91]. The code corresponding to the search space class hierarchy for PTs as defined in this paper consists of about 800 lines of code. Another implementation of the search space class hierarchy for coping with an object-oriented database model and language was implemented where 400 of those were replaced by 600 lines of code. Two search strategies implementing the branch-and-bound algorithm and Iterative Improvement incurred only 60 additional lines of code each. These numbers are quite encouraging. The prototype will be enhanced with other strategies and extended to cope with more general cost models (e.g., considering the cost of evaluating complex predicates). Overall, we plan to use it as an experimental vehicle to measure the effectiveness of randomized search strategies in optimizing more complex (deductive, object-oriented) queries.

7. Acknowledgements

The authors wish to thank Georges Gardarin, Eric Simon, Mikal Ziane and the members of the EDS project for their useful comments. They also want to thank Mohamed Zait and Alexandre Ribenboim for their contribution in the implementation of the optimizer prototype.

8. References

[Bergstein91] Bergstein B., Couprie M. and Valduriez P., "Prototyping DBS3, a Shared-Memory Parallel Database System", submitted for publication, 1991.

[Eiben90] Eiben A.E., Aarts E.H.L. and Van Hee K.M., "Global Convergence of Genetic Algorithms: an Infinite Markov Chain Analysis", Proc. First Int. Workshop on Parallel Problem Solving from Nature, Dortmund, 1990.

[Freytag87] Freytag J.C., "A Rule-Based View of Query Optimization", Proc. ACM SIGMOD Conf., 1987.

[Goldberg89] Goldberg D.E., "Genetic Algorithms in Search, Optimization and Machine Learning", Addison-Wesley, 1989.

[Graefe87] Graefe G. and DeWitt D.J., "The EXODUS Optimizer Generator", Proc. ACM SIGMOD Conf., 1987.

[Ibaraki84] Ibaraki T. and Kameda T., "Optimal Nesting for computing N-relational joins", ACM TODS, vol. 9, n. 3, September 1984.

[Ioannidis87] Ioannidis Y.E. and Wong E., "Query Optimization by Simulated Annealing", Proc. ACM SIGMOD Conf., 1987.

[Ioannidis90] Ioannidis Y. and Cha Kang Y., "Randomized Algorithms for Optimizing large join queries", Proc. ACM SIGMOD Conf., 1990.

[Krishnamurty86] Krishnamurty R., Boral H. and Zaniolo C., "Optimization of Nonrecursive Queries", Proc. 12th VLDB Conf., Kyoto, 1986.

[Lanzelotte90] Lanzelotte R.S.G., "OPUS: an extensible OPtimizer for Up-to-date database Systems", Ph. D. Thesis, Computer Science, PUC-RIO, 1990, available at INRIA, Rocquencourt, n° TU-127.

[Lanzelotte91a] Lanzelotte R.S.G. and Valduriez P., "An Object-oriented Approach for Extensible Query Optimization", Proc. VII Journées Bases de Données Avancées, Lyon, 1991.

[Lanzelotte91b] Lanzelotte R.S.G., Valduriez P., Ziane M. and Cheiney J.-P., "Optimization of Nonrecursive Queries in OODBs", submitted for publication.

[Lohman88] Lohman G., "Grammar-like Functional Rules for Representing Query Optimization Alternatives", Proc. ACM SIGMOD Conf., 1988.

[Ono90] Ono K. and Lohman G., "Measuring the Complexity of Join Enumeration in Query Optimization", Proc. 16th VLDB Conf., 1990.

[Papadimitriou82] Papadimitriou C. H. and Steiglitz K., "Combinatorial Optimization: Algorithms and Complexity", Prentice-Hall Inc., New Jersey, 1982.

[Selinger79] Selinger P.G. et al, "Access path selection in a relational database management system", Proc. ACM SIGMOD Conf., Boston, May 1979.

[Shapiro87] Shapiro S.C., "Encyclopedia of Artificial Intelligence", Wiley-Interscience, New York, 1987.

[Swami88] Swami A. and Gupta A., "Optimization of Large Join Queries", Proc. ACM SIGMOD Conf., 1988.

[Swami89] Swami A., "Optimization of Large Join Queries: combining Heuristics and Combinatorial Techniques", Proc. ACM SIGMOD Conf., 1989.

[Valduriez89] Valduriez P. and Danforth S., "Query Optimization in Database Programming Languages", Proc. Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, 1989.