Cherry Picking: A Semantic Query Processing Strategy for the Evaluation of Expensive Predicates

Fabio Porto[1*] Eduardo Sany Laber[2†] Patrick Valduriez[3]

[1]Dep. of Computer Engineering
IME-Rio, Brazil

[2]Departament of Informatics
PUC-Rio, Brazil

[3]INRIA and IRIN
Nantes, France

fporto@de9.ime.eb.br, laber@inf.puc-rio.br,patrick.valduriez@inria.fr

Abstract

A common requirement of many scientific applications is the ability to process queries involving expensive predicates corresponding to user programs. Optimizing such queries is hard because static cost predictions and statistical estimates are not applicable. In this paper, we propose a novel approach, called Cherry Picking (CP), based on the modeling of data dependencies among expensive predicate input values as a k-partite graph. We show how CP can be easily integrated into a cost-based query processor. We propose a CP Greedy algorithm that processes the graph by selecting the candidate values that minimize query execution cost. Based on performance simulation, we show that our algorithm yields executions up to 86% faster than statically chosen pipeline strategies.

1. Introduction

In several scientific applications [15] a common requirement is the ability to process data objects, which can be very large, by scientific user programs, which can be very long running. For instance, some objects could be satellite images and some user programs could perform image analysis and take a long time to complete. In a database environment, execution of these user programs can be simply modeled as expensive user-defined predicates [7] which can be included in SQL-like queries. In traditional query processing, predicates are processed as early as possible. When predicates are expensive, this approach may be quite inefficient since it may lead to multiple user program invocations.

As an example of an application with expensive predicates, consider the query in Example 1.1 that identifies regions with a given correlation among their pollution indexes and humidity factors. The input to the query has two data sources: the relational views *Meteo(region,map)* and *Pollution (region, city, pollindexes)*, where *Meteo* models the meteorological images per region and *Pollution* stores the pollution indexes for cities. Two scientific programs compute, respectively, a satellite image-based humidity factor and a pollution index from an array of collected pollution samples: $Humidity(blob) : float$ and $PollutionInd(blob) : float$. The scientific programs are registered in the database system as user defined functions [10] together with estimates of their per tuple execution cost (in seconds) and selectivity factor, as in [7].

*this author work was partialy funded by ABIN- Agência Brasileira de Inteligência

†this author work was partialy funded by FAPERJ- Projeto Jovem CIentista do Nosso Estado, E/26/150.175/2003 and CNPq- Conselho Nacional de Pesquisa

Example 1.1

Select *m.region,po.city*
From *Meteo m, Pollution po*
Where *m.region=po.region and*
Humidity(m.map) < 1.5 and
PollutionInd(po.pollindexes) > 0.6

1.1. Processing of Expensive Predicates

Let us discuss how this query can be executed. Let us assume that the average time for each invocation of the Humidity and Pollution programs are 3 and 1.5 minutes, respectively. A sequential query evaluation strategy that considers a Meteo relation with 10 distinct maps and a PollutionInd relation with 20 distinct pollution samples could take up to $3 \times 10 + 1.5 \times 20 = 60$ minutes to be concluded.

In such a scenario, an efficient QEP would place the expensive predicates on the results of Humidity (H) and PollutionInd (P) user programs after the *join* operation, in the hope of eliminating some tuples and, as a result, reducing the number of program invocations.

Let us now consider the ordering of expensive predicates. Assuming that some input values to predicates evaluate to *false*, different orders of evaluation produce unequal response times. Unfortunately, a static order is unable to adapt itself to fluctuations on data characteristics [1]. While such variations might be overlooked by traditional queries, their effect over the execution of expensive predicates is dramatic.

As an example, consider the relation instance R in Figure 1 representing the join result between Meteo and Pollution [1]. Let us also assume that the expensive predicates evaluations

Map	PollIndexes
Reg1	City1
Reg1	City2
Reg1	City3
Reg2	City4
Reg3	City4

Figure 1: Input relation and its bipartite graph

over the input values return the following results: Reg1 $\rightarrow$ false, Reg2 $\rightarrow$ true, Reg3$\rightarrow$ true, City1 $\rightarrow$ true, City2 $\rightarrow$ true, City3 $\rightarrow$ true, City4 $\rightarrow$ false. The most efficient evaluation would process tuples 1 to 3 by *H(map)* predicate and then process tuples 4 and 5 by expensive predicate *P(pollindexes)*. Considering that we do not repeatedly process duplicate values [8], the elapsed-time for this query would be 4.5 minutes.

On the other hand, a static order based strategy which processes H and then P would spend 10.5 minutes while the one which process P and then H would spend 9.0 minutes.

Thus a natural question is why static order based strategies may perform so poorly. We believe that efficient query execution algorithms that involve expensive predicates must take

[1]The values of *map* and *pollindexes* were replaced by Reg_i and $City_j$ for illustration purposes.

into account the data-dependency induced by the input relation. This is the key idea behind the CP approach proposed in this paper.

1.2. Contributions

This paper makes three major contributions. First, we present the CP approach for processing expensive predicates based on the knowledge extracted from the associations among predicate input values. Such associations are modeled through a k-partite hypergraph (KHG). Second, we provide a cost based heuristic for integrating a CP algorithm into a query execution plan with expensive predicates. Finally, we propose a greedy algorithm that selects and processes the vertexes in the hypergraph following a greedy criteria. Our experiments demonstrate that for some very common scenarios, the greedy algorithm outperforms a static strategy by 86% on the average.

The rest of this paper is organized as follows. Section 2 presents the CP approach in more details. Section 3 shows how the CP approach can be integrated into a query processor. In Section 4, we present the greedy algorithm that processes the k-partite hypergraph. In Section 5, we evaluate the performance of the CP approach against static strategies. Section 6 discusses related work. Finally, Section 7 concludes.

2. The Cherry Picking (CP) Approach

We define the CP approach as the set of techniques and algorithms that take advantage of the data-dependency among input attribute values in order to evaluate a query with expensive predicates.

The approach name, Cherry Picking, stands from its main principle of directing the processing of expensive predicates towards the selection of values that minimize query execution time, similar to what one would do when picking cherries in a cherry tree.

In this Section, we formally introduce the CP approach. We begin by specifying the type of queries we focus on this paper. Next, we show how to build the KHG from expensive predicates input values. We illustrate the process with the query in Example 1.1. Having presented the KHG, we are able to formalize our optimization problem for which the CP-algorithms are employed.

2.1. Preliminaries

In this paper, we focus on the evaluation of conjunctive SPJ (select, project, join) queries with both simple and expensive predicates.

A predicate is expensive if it evaluates on the result of a user program that was registered in the database system as expensive. Different types of user programs have been identified in [10]. In this paper, we concentrate on programs of the user-defined simple function (UDSF) type, that take a tuple as input and produce a scalar value as output. For the sake of presentation, we assume that functions receive only one attribute as input. A consequence of focusing on UDSF functions is that we restrict the discussions in this paper to unary expensive predicates, i.e. we do not treat expensive joins.

We consider a query execution plan (QEP) structured as an operator tree where internal nodes are operators and leaf nodes are input relations. Different shapes of trees can be consid-

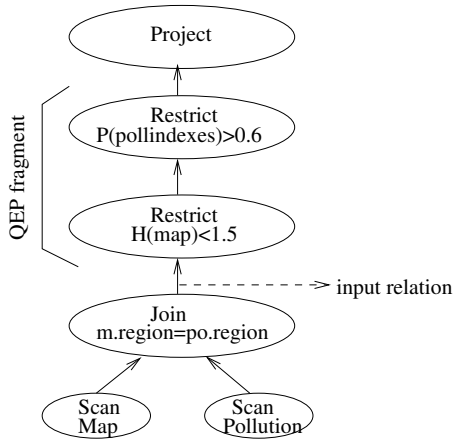

Figure 2: QEP for a query with a set of expensive predicates

ered: left-deep, right-deep or bushy. In this paper we make no restrictions regarding the shape of the operator tree.

A QEP is defined as $P = \{\rho, \Omega, \prec\}$, where ρ is a set of relations, Ω is a set of operators that includes: algebraic operators, control operators and modules (see brief presentation bellow) and $\prec$ is a set of ordering operators whose elements define an execution order between operators in Ω (which we will read as *preceds* in execution order). We say that $\omega_i \prec_1 \omega_j$ iff executing ω_i produces a result that is consumed by ω_j. We observe that $\prec$ is transitive, so if $\omega_i \prec_1 \omega_j$ *and* $\omega_j \prec_2 \omega_k$ then $\omega_i \prec_3 \omega_k$. We also say that one operation ω_i *immediately precedes* ($\ll$) another operation ω_j in Ω, if $\omega_i \prec \omega_j$ and *not* $\exists\, \omega_k \in \Omega$, such that $\omega_k \prec \omega_j$ and $\omega_i \prec \omega_k$.

We further define a *QEP fragment* $P' = \{\rho', \Omega', \prec'\}$, with $P' \subseteq P$, if, forall $\omega_i, \omega_j \in \Omega'$, $\omega_i \neq \omega_j$, either $\omega_i \prec \omega_j$ or $\omega_j \prec \omega_i$ in P'.

A *QEP fragment* P' is said to be a set of expensive predicates (SEP) if all operators in Ω' are expensive predicates.

In addition, we use the term *input relation* to a SEP S in a QEP P to denote the output tuples produced by an operation ω_i, with $\omega_i \in \Omega$ and $\omega_i \notin S$, and $\omega_i \ll \omega_j$, for all $\omega_j \in S$.

The plan in Figure 2 illustrates these definitions. A QEP fragment is indicated, comprising a SEP with two restriction (select) operators. The *input relation* to this SEP is produced by the preceding Join operation.

As a last definition regarding a QEP, we denominate a *Module*, a *QEP fragment* implementing a specific execution semantic. In particular, in Section 3.2, we present a CP module, whose semantics implement the CP approach.

Finally, a hypergraph $G = (V, E)$ is a mathematical structure, where $V = \{v_1, \ldots, v_n\}$ is the set of vertexes and $E = \{e_1, \ldots, e_m\}$ is the set of hyperedges. A hyperedge $e \in E$ is a subset of V. We define the degree d of a vertex v_i as the number of hyperedges containing v_i. Two vertexes, u and v, are adjacent if there is an hyperedge e such that $u \in e$ and $v \in e$. A hypergraph G is $k-$partite if V can be partitioned into k disjoint subsets $A_1, A_2, \ldots, A_k$ such that $|e \cap A_i| \leq 1, \forall\, e \in E,\ \ i = 1, \ldots, k$.

We also define the function $v_i.value()$ that returns the value associated to the vertex v_i and the boolean function $\delta_i(v_i.value())$ that returns the result of the evaluation of $v_i.value()$ by the expensive predicate δ_i.

2.2. Modeling the Input Relation through a k-partite Hypergraph

The CP approach promotes the efficient execution of a set of *k* expensive predicates in a query β by capturing the data dependency among expensive predicates input values through a k-partite hypergraph (KHG).

A KHG is an abstract representation of the *input relation*. Each partition of the hypergraph is associated to an expensive predicate in the SEP. The vertexes in a partition represent distinct values in an input relation attribute bound to the associated expensive predicate. Each hyperedge in the KHG represents a tuple of the *input relation*.

Input attributes to expensive predicates may come in both alpha-numeric (standard) and unstructured data types. Vertexes in the KHG corresponding to attributes of standard data types are associated to attribute values. Attributes of unstructured data types, like images, receive an Unified Resource Identifier (*URI*) for their values identification [2].

Given an hyperedge $e \in E$, corresponding to a tuple in *R*, we say that e is *true* if the evaluation of the set of expensive predicates on its vertexes values returns *true*. Otherwise, we say that e is false. As an example, consider the query plan in Figure 2 and the *input relation* $R(map, pollindexes)$, as in Figure 1. R attributes are bound to the set of expensive predicates $SEP = \{Humidity, PollutionInd\}$.

The corresponding KHG, illustrated in Figure 3, is defined as $G = \{A_1 \cup A_2, E\}$, where the partitions A_1 and A_2 contain distinct *URIs* for the values in attributes *map* and *pollindexes*, and the edges in E correspond to tuples in R. Then, $A_1 = \{Reg1, Reg2, Reg3\}$, $A_2 = \{City1, City2, City3, City4\}$ and $E = \{(Reg1, City1), (Reg1, City2), (Reg1, City3), (Reg2, City4), (Reg3, City4)\}$.

In this context, the bipartite (2-partite) graph *G* is equivalent to the *R input relation* regarding the evaluation of the expensive predicates. In order to produce a valid tuple from query β, the evaluation of vertexes values in the corresponding edge in G must be *true* for all the expensive predicates in β. On the other hand, whenever the evaluation of a vertex value returns false, all the edges which contain that vertex, and their respective tuples, are eliminated.

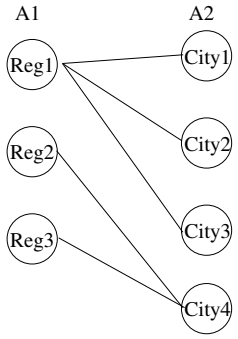

Figure 3: Bipartite graph G corresponding to the input relation R

2.3. Problem Formulation

Thus, given a query β with a set of expensive predicates, we want to devise a cost-based heuristic model to evaluate the adequacy of assigning a CP module for processing a SEP. Also, given

a k-partite hypergraph representation of an *input relation* attribute values to the SEP, we want to devise a CP algorithm that determines, as fast as possible, whether each hyperedge of the hypergraph is *true* or *false*. Finally, we want to integrate the CP approach into a QEP.

3. Integrating CP into Query Processors

In this Section, we discuss the integration of the CP approach into modern query processors. Query processing consists of two main phases [2]: query optimization and execution. During query optimization, an optimal QEP is produced based on the analysis of statistics collected from query objects and a cost model. Next, the QEP is submitted to a query execution engine that executes the operations in the QEP accordingly.

3.1. An Optimization Strategy for the CP Approach

The integration of the CP approach into query processing also consists of two phases. During the optimization phase, a query execution plan is created following the traditional dynamic programming strategy for plan enumeration adapted to the placement of expensive predicates according to a rank order [7]. In this strategy, each predicate δ_i is annotated with a rank computed as:$rank(\delta_i) = \frac{1-sel(\delta_i)}{pertuplecost(\delta_i)}$.

Figure 2 illustrates a QEP created for the query in Example 1.1, where ellipses represent query operators and directed edges represent the dataflow between operators. Observe that the expensive predicates are ordered according to their computed *rank* value.

Once the QEP has been produced, we exercise one extra pass through it to analyze the adequacy of adopting the CP approach. The process traverses the QEP bottom-up looking for set of expensive predicates, like the one indicated in Figure 2. When a SEP is found, a cost-based heuristic (see equation (1)) evaluates the benefit of assigning a CP module to process it.

Different cost models may be distinguished according to the specifics of an application. We opted for a conservative heuristic, which assigns a CP module to a QEP fragment with a SEP whenever the former estimated overhead cost represents a fraction of the registered per tuple invocation cost of the least expensive predicate in the SEP. More formally, the CP-module is employed if

$$overhead(CP\ module) \leq \rho * \delta_i.pertuplecost(), \tag{1}$$

where $overhead(CP\ module)$ models the extra cost incured when adopting the CP approach (see Section 4 for a concrete example), $\delta_i.pertuplecost()$ is the per tuple cost of the least expensive predicate in the analysed SEP and ρ is a constant [3].

The left side of equation (1) should be adapted to the type of CP algorithm chosen for evaluating a SEP. In addition, the ρ constant provides for a tuning mechanism to be modified according to application characteristics.

If the CP approach overhead is considered negligible compared to the cost of the expensive predicates evaluation, then the QEP fragment is replaced by a CP module containing: a procedure for building the KHG, a CP algorithm and a reformulation operator (see Figure 5). Once

[2]Parsing and preprocessing have been left out for simplicity.

[3]we expect to predict an ρ value according to statistics on executions history

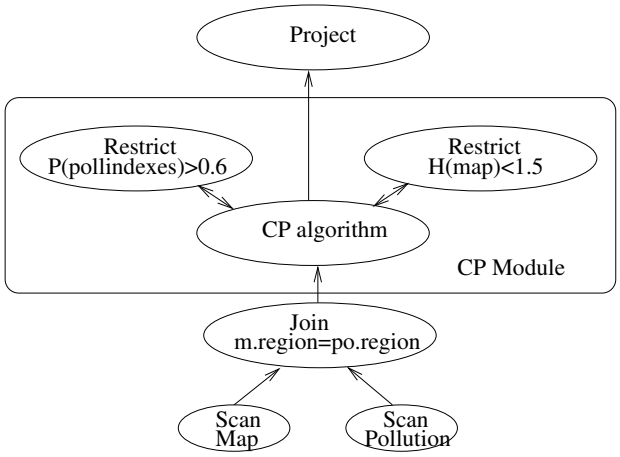

Figure 4: QEP for a query with a CP module

the traversal of the QEP is done, the second phase of query processing initiates where the new QEP is sent to the query engine for processing.

3.2. The CP Module in a QEP

A CP module implements the CP approach into a QEP. Its execution proceeds according to the iterator model [6] (interface open(), getnext() and close()), which makes its integration into a QEP transparent to the query engine. Its main components are the *Build KHG*, *the CP algorithm* and *the Reformulation* operators.

The *Build KHG* operator prepares the input to the CP algorithm in the form of a KHG. It receives an input relation and builds the corresponding KHG, according to the bindings to the expensive predicates in the SEP. The *CP Algorithm* operator evaluates the KHG against the expensive predicates in the SEP and produces valid tuples. Finally, the Reformulation operator obtains valid tuple-ids from the CP Algorithm and retrieves the corresponding materialized tuples.

The implementation of the iterator interface by the CP module provides for a simple execution model, as described next.

In order to easy the presentation, we will consider a *QEP fragment* composed of the operators in $\Omega = \{cp_i, \omega_i, \omega_j\}$, where cp_i corresponds to a CP module operator, and the precendence relations $\prec = \{\{\omega_i, \prec_1, cp_i\}, \{cp_i, \prec_2, \omega_j\}\}$.

The execution of the CP module operator cp_i is coordinated by its consumer operation ω_j, which issues synchronous calls to the former. The ω_j operation requests the CP module to take initialization procedures by issuing an *open()* request, which causes initialization procedures to take place through all module components (see Figure 5). The *open()* request is retransmited to the ω_i operation. The synchronous nature of calls in the iterator model requires a producer operator to answer a request before its consumer can progress. Once ω_j receives a positive answer to the *open()* request, it progresses by issuing a first call to the cp_i operator *getnext()* method. Figure 5 details the CP module components and their interaction during the execution of a first *getnext()* call received from ω_j. The blocking characteristics of this implementation for

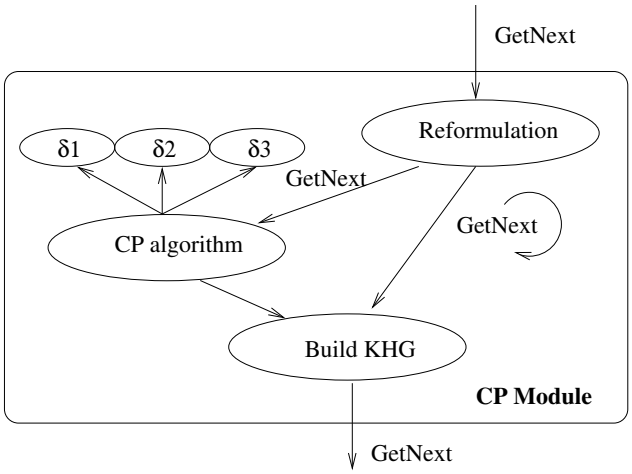

Figure 5: CP module

the CP approach is depicted in Figure 5 where the CP module consumes all incomming tuples from ω_i before producing an answer to the first *getnext()* request. During this phase, a KHG is built by the homonimous operator in the CP module.

Once the KHG has been built, the Reformulation operator issues a *getnext()* call on the CP algorithm operator, which motivates an evaluation of the KHG by the latter.

Once a valid tuple is identified by the CP algorithm operator, its tuple-id is returned to the Reformulation operator that joins [4] it to the corresponding materialized tuple, stored during the initialization phase. This ends the first call to the *getnext()* method, returning to the ω_j operation an output tuple. The execution continues by successive calls to the *getnext()* method, which motivates the production of a new valid tuple, if one exists.

Finally, a *close()* request executes cleaning procedures and is transmitted to the ω_i operator.

It is important to observe that, currently, the CP approach is a blocking technique, in the sense that it requires all input tuples to the SEP to be consumed by the corresponding CP module before producing a first output tuple.

4. A CP Algorithm

In this Section, we give a general introduction to the class of CP algorithms and present a greedy algorithm for the evaluation of k expensive predicates.

Given a KHG G, where the partition vertexes are associated to the bounded attribute values of an input relation to a SEP in a QEP, a CP algorithm determines a vertex selection policy that guides the evaluation of the expensive predicates on input relation values. In [11], it was demonstrated that any CP algorithm will process, at least, a cover of the KHG, that is, a set $C \subset V$ such that $C \cap e \neq \emptyset$ for every $e \in E$. That is because at least one vertex from each hyperedge must be evaluated in order to decide whether such an hyperedge is true or not. Nevertheless, considering differences in the evaluation costs of each expensive predicate, the set of vertexes processed by a CP algorithm must contain a minimum cost cover for a KHG.

[4] We implemented the Reformulation operator as a Hash Join operation.

Thus, in [11], two polynomial time CP algorithms are proposed that compute a minimum cost cover for a bipartite graph.

In this paper, we propose a CP algorithm that implements a greedy strategy for computing a minimum cost cover for a KHG. Since the minimum cover problem is NP-complete for $k-partite$ graphs, with $k \geq 2$ [5], one does not expect to find polynomial time algorithms, which justifies the design of heurisitcs, as the one proposed here. The strategy selects the vertexes to be evaluated according to their degree in the KHG and associated expensive predicate *rank*. We name this index the vertex *fanout* and compute it as: $F(a_{i,j}) = d_{a_{i,j}} * rank(\delta_i)$, where $a_{i,j}$ corresponds to the j vertex associated to i bound attribute, $d_{a_{i,j}}$ is the degree of the $a_{i,j}$ vertex and δ_i is the expensive predicate bounded to i input relation attribute.

Figure 6 presents the pseudo-code for the Greedy algorithm. In Step 1, the *fanout* is computed for each vertex in hypergraph *G*. The Step 2 consists of a loop. First, the vertex with highest fanout is selected for processing. Let $a_{i,j}$ be this vertex. If a predicate evaluation over $a_{i,j}$ value returns false, then all the hyperedges in graph *G* containing $a_{i,j}$ and the corresponding tuples, are eliminated. On the other hand, if the evaluation returns true, then $a_{i,j}$ is removed from every hyperedge which contains it. At the end of the loop every hyperedge with no more vertexes is written to the output and eliminated from G. Finally, $a_{i,j}$ is removed from V and the fanout of every vertex in V adjacent to $a_{i,j}$ is recalculated. This process continues until there are no more hyperedges left in G. The greedy CP-algorithm can be implemented in different ways.

Greedy($G, \beta, \delta_1, \delta_2, \cdots, \delta_k$)
Step 1: Compute $F(v)$, for every $v \in V$
Step 2: **While** (there are edges in E)
$a_{i,j} \leftarrow argmax_{v \in V}\{F(v)\}$
$N \leftarrow$ set of vertexes adjacent to $a_{i,j}$ at the current graph
If $\delta_i(a_{i,j}) = 0$ then
Remove from E every hyperedge which contains $a_{i,j}$.
else
Remove $a_{i,j}$ from every hyperedge e which contains it
For every $e \in E$, with $e = \emptyset$
Output e;
$E = E - \{e\}$;
Remove $a_{i,j}$ from V
Recompute $F(v)$, for every $v \in N$

Figure 6: Pseudo code for the greedy CP algorithm

An efficient implementation uses a priority queue data structure to store the vertexes of G. The priority queue is ordered by the vertexes fanouts. Each cell of the heap stores four fields: *id* , *value*, *fanout* and p, where *id* is the vertex identification, *value* is its associated value, *fanout* is its fanout and p is a pointer for a list that stores the hyperedges which contain this vertex. Furthermore, a vector V is used to store the hyperedges of the graph. The entry $V[e]$ holds three fields: a flag indicating if e has been deleted or not, an integer indicating the number of vertexes contained in e and a pointer for a list of vertexes in e. In addition, the vector index is used as the tuple-id for the associated tuple. It is possible to show that this data structure allows for an $O(mk \log mk)$ time implementation of greedy CP-algorithm with small hidden constants, where *m* is the estimated cardinality of the input relation and *k* is the number of expensive predicates in the QEP fragment. Assuming the usage of the data structure described so far, the over-

head cost for the greedy CP algorithm is computed as $overhead(CP\,module) = c{*}mk{*}logmk$, where c represents the average cost for one comparison operation.

5. Validation

In this Section, we report on experimental results obtained by evaluating queries with expensive predicates using the CP approach. The experiments compare results from the greedy CP algorithm with pipelined strategies [7]. We simulated both strategies and execute them over synthetically generated data.

5.1. Experimental Setup

The experiments were executed on a single 2.0GHz processor Dell machine running Linux kernel version-2.4.18-2, with 532MB of RAM. The simulation program employed GCC v. 3.1 and the data generator is a java program using jdk1.3.1.

We consider one input relation R with 2 attributes, each bound to one expensive predicate. The data generator builds relations with randomly distributed values according to the parameters in Table 1. Values in each column are independently generated. Initially, for each column, we fill $distA_i$ slots, of a total of $card$ slots, with a value between 1 and $distA_i$, randomly selected using java Math.random method, normalized for the range of valid tuple numbers. Next, we fill each of the remaining slots with a randomly selected value between *1* and $distA_i$. For each attribute, an extra column indicates the result of evaluating its corresponding expensive predicates over each of its values. The assignment of *true* values are randomly distributed along the tuples according to the specified selectivity factor of the correlated expensive predicate. The simulation consists in: picking a tuple for processing by an expensive predicate, adding

Parameter	Meaning	Values
card	Number of tuples	100-100.000
k	Number of expensive predicates	2
$distA_i$	Number of distinct values in column $R[A_i]$	1-100.000
$costP_i$	Unitary invocation cost of an user program P_i	1-10
$sel\delta_i$	selectivity of expensive predicate δ_i	0.01-1

Table 1: Simulation run parameters

to a counter a value corresponding to its estimated unitary cost and checking its *evaluation* column value for the result. False evaluations induce the elimination of the corresponding tuples, whereas *true* evaluations either keep the tuple for futher predicate evaluations or send the corresponding tuples to the output. The simulation also imposes, during pipeline processing, that duplicate values are not considered, as if a caching mechanism prevented unnecessary program invocations.

A run gives the results obtained for a given set of parameter values, as specified in Table 1. Each result value is obtained by averaging the results of 20 executions for the same set of parameters values. At each run, we register the vertexes evaluated by each expensive predicate and compute the query total cost for the CP and the pipeline strategies.

5.2. Performance Results

To obtain meaningful performance results, we ran three different experiments. Our experiments considered query Q_1 in 5.1, where *R(A,B)* is an input relation to the expensive predicates s_1 and s_2.

Example 5.1

Select $*$
From *R*
Where $s_1(A)$ *and* $s_2(B)$

The first experiment analyzes the influence of the distribution of column distinct values on query execution. We considered a run with the following set of parameter values: $S = \{(card, 1000), (dist_A, 700), (sel_{s1}, 0.3), (sel_{s2}, 0.35), (cost_{s1}, 3), (cost_{s2}, 3)\}$. We registered 5 runs in which we varied the number of distinct input values on column B from 100 to 900, by steps of 100.

Figure 7 a) shows the results of executing the CP greedy algorithm and two pipeline strategies: s_1s_2 (s_1 followed by s_2) and s_2s_1 (s_2 followed by s_1). We observe that, for 100 distinct values, CP outperforms s_1s_2 by 86%, while it matches the results of s_2s_1. The important fact behind this experiment is that s_1s_2 would be the choice of a rank order based strategy [7]. Thus, although a nice execution can be obtained from a pipeline strategy, it is not possible to predict it if only estimated cost and selectivity statistics are taken into account.

As the number of distinct values grows towards 900, the pipeline lines in Figure 7 cross themselves, in a point near 700 distinct values, causing the s_1s_2 sequence to become the best choice for processing query Q_1. Note that statically computing the number of distinct values in columns of an intermediary relation, as the one produced by previous joins, can be very hard [9]. In addition, one can see the five runs as a single run on a relation of 5000 tuples where the distribution of distinct values varies for each 1000 tuples. In such a scenario, no pipeline order can produce an efficient query execution. On the other hand, the CP approach adapts nicely to the fluctuations on distinct input values. In fact, it constantly produces the best evaluation order for predicates in query Q_1. This is a direct consequence of considering the degree of each input value in the KHG, in addition to the predicate rank. Since most values in column A have no duplicates, a single evaluation of a vertex B, with average degree greater than 1 and selectivity 0.3, will very often eliminate multiple tuples, which saves s_1 from processing them.

Our second experiment considers the impact of variations of the selectivity factor on query execution. We ran again query Q_1 following the value set in $S = \{(card, 1000), (dist_A, 200), (dist_B, 900), (sel_{s1}, 0.5), (cost_{s1}, 6), (cost_{s2}, 3)\}$. We computed 5 runs, in which the selectivity factor of predicate s_2 varied from 10% to 90%, in steps of 10%. Here, we only compare the CP execution with the pipeline order s_2s_1, which should give the best rank order for all runs but the last.

The rank value for predicate s_1 is 0.083. If we consider an average degree of 4 for vertexes in A and of 1 for vertexes in B, we can conclude that only for very low selectivity factors the *fanout* of B vertexes would be prevalent over those of A vertexes. This analysis is confirmed in Figure 7 b). Only when the selectivity factor of s_2 is 10% the rank order pipeline execution s_2s_1 becomes close (24% above) to that of the CP approach. This is reasonable as long as s_2 is fast and very selective, which demonstrates the relevance of the number of distinct values over query execution performance. Even being selective and fast, s_2 cannot be as effective as s_1 as

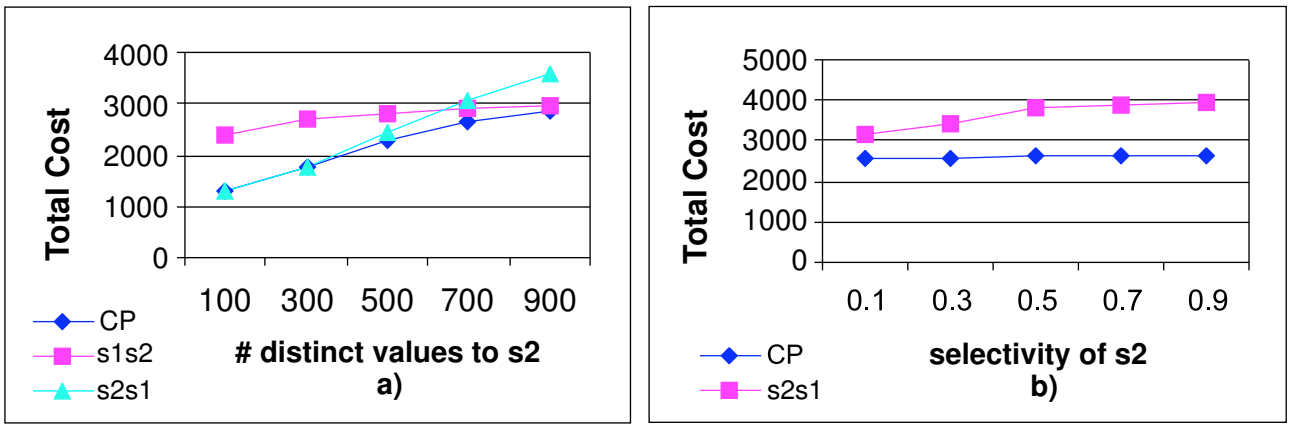

Figure 7: CP versus pipeline

a result of a large difference between A and B in vertex degrees. When the selectivity factor of s_2 has a maximum value of 90%, CP performs best in these runs, around 50% faster than $s_2 s_1$.

A further analysis of this experiment suggests the use of the CP approach for scenarios where selectivity factors are hard to predict. Under such hypotheses, estimating a 50% selectivity factor is a reasonable guess. The CP approach would compensate its lack of statistics with runtime knowledge of vertexes fanout yielding very good query execution performance (see Figure 7 b)).

Our last experiment processes query Q_1 over R where the fanouts of vertexes in A and B are symmetric with respect to half the number of tuples. We experimented with a value set $S = \{(card, 1000), (dist_A, 450), (dist_B, 450), (sel_{s1}, 0.5), (sel_2, 0.3), (cost_{s1}, 6), (cost_{s2}, 3)\}$. R tuples have an average of 50 distinct A values and 400 distinct B values from tuple 1 to 500. Then, the last 500 tuples show an opposite distribution of values. This is a case where no static order could lead to an optimal query execution time as the impact of an average vertex degree of 10 on estimated rank shifts the best choice for the first predicate to consume a tuple from one predicate to the other during R processing. The experimental results show that, in average, the CP outperforms the pipeline orders $s_2 s_1$ and $s_1 s_2$ by 65.4% and 63.3%, respectively.

6. Comparison with Related Work

The optimization of queries with expensive predicates has been the subject of extensive research [3, 4, 7, 13]. In some of these works, the traditional *dynamic programming optimization strategy* [14] is adapted to the evaluation of expensive predicates, i.e. predicates that range over the results of user programs. In particular, [7] proposes a *predicate migration* strategy, based on a polynomial time algorithm, in which predicates are ordered according to a *rank* value computed as a factor of their estimated selectivity factor and per tuple evaluation costs. [3] proposes the extension of the search space for potential QEPs by analyzing all possible orders of expensive predicates evaluation within each dynamic programming iteration, keeping the minimal cost QEP. In [4, 13], user programs are modeled as relations allowing for techniques based on traditional query optimization strategies.

These techniques, however, may yield sub-optimal execution times for queries such as in Example 1.1, for the following reasons. First, it is very difficult to predict the selectivity factors

for the expensive predicates, as data do not really exist (they are generated by program evaluation). Even historical samples give very poor information about what may come in the future, considering that input data for published programs can come from very diverse data-sources within the Internet. Second, the query execution tree based on static predicate order is not sensitive to variations on distinct input value distributions applied to user programs. In this case, as shown in [2], physical operators implementing expensive predicates placed in the query tree higher nodes alternate between idleness periods, waiting for tuples to arrive from previous expensive predicates, with overload periods, with a queue of distinct input values that were paired with duplicate ones, in attributes bounded to previous expensive predicates[5].

In order to adapt to execution time conditions, some dynamic strategies have been proposed. [2] presents an strategy for evaluating expensive predicates in distributed queries. The strategy adaptively reacts to variances on estimated selectivity and cost of expensive predicates as well as on the effect of non-uniformity on data distribution. The query tree generated presents branches with expensive predicates in different pipeline orders.

More recently, Madden and Hellerstein [1, 12] propose an interesting adaptive query execution framework, called Eddies. Rather than following a rigid QEP, in Eddies tuples are routed towards query operators following a flexible scheduling policy. A monitor module registers the consume/production rate of each operator. Whenever a synchronization point is detected, a *lottery* is ran between query operators. The most efficient query operator, according to the lottery policy, is scheduled for processing and is given the next tuple.

The flexibility of Eddies allows for different plans to be evaluated during one single query execution. Although not specifically designed for dealing with expensive predicates, the strategy naturally fits expensive predicates within its adaptive operators schedule strategy. CP is an adaptive strategy in the way that query execution conforms itself to a scheduling policy based on the relationship among expensive predicates input attribute values. As a matter of fact, one may use the CP approach as a scheduling policy for the Eddy framework for queries exclusively composed of unary predicates.

7. Conclusion

Optimizing queries involving expensive predicates corresponding to user programs has been hard because static cost predictions and statistical estimates are no longer useful. In this paper, we proposed a novel approach for the evaluation of expensive predicates in a query, called Cherry Picking (CP), based on the modeling of data dependencies among expensive predicate input values.

This paper made three major contributions. First, we formally defined the CP approach for processing expensive predicates based on the knowledge extracted from the associations among predicate input values. Such associations are modeled through a k-partite hypergraph (KHG), where vertexes correspond to the predicates' input values and edges correspond to tuples linking those values.

Second, we gave a cost based heuristic for integrating a CP algorithm into a query execution plan with expensive predicates. This makes it possible to integrate seamlessly the CP approach into modern query processors. We described the architecture of a CP module that fits in a query processor using the ubiquitous iterator model [6].

[5]Caching [8] avoids processing programs over duplicate input values.

Finally, we proposed a simple greedy algorithm that implements the CP approach. The algorithm selects and processes vertexes in the hypergraph based on their fanout following a greedy criteria. Our experiments demonstrate that for some very common scenarios, the greedy algorithm outperforms a static pipelined strategy by 86% on the average. These experiments assumed accurate estimates for the static strategy which is impossible to achieve in practice. Furthermore, we did not count the overhead of managing statistics for the static strategy. Therefore, the greedy algorithm is much more efficient and simpler.

In future work, we will extend the approach to dynamically react to variations on execution time conditions, like evaluation cost and selectivity factor. We will also extend it to deal with distributed data in the context of mediator systems. Finally, we plan to experiment with real data, which we will try to obtain from scientific applications.

References

[1] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 261–272, 2000.

[2] L. Bouganim, F. Fabret, F. Porto, and P. Valduriez. Processing queries with expensive functions and large objects in distributed mediator systems. In *Proc. 17th Intl. Conf. on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 91–98, 2001.

[3] Surajit Chaudhuri and Kyuseok Shim. Query optimization in the presence of foreign functions. In *Proc. 19th Intl. Conf. on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland*, pages 529–542, 1993.

[4] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Towards on open architecture for LDL. In *Proc. 15th Intl. Conf. on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, pages 195–203, 1989.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$–Completeness*. W. H. Freeman and Company, New York, NY, 1979.

[6] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[7] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. *TODS*, 23(2):113–157, 1998.

[8] Joseph M. Hellerstein and Jeffrey F. Naughton. Query execution techniques for caching expensive methods. In *Proc. 1996 ACM SIGMOD Intl. Conf. on Management of Data, June 4-6, 1996, Montreal, Quebec, Canada*, pages 423–434, 1996.

[9] Yannis E. Ioannidis and S.Christodoulakis. On the propagation of errors in size of join results. In *Proc. 1991 ACM SIGMOD Intl. Conf. on Management of Data, Denver, Colorado, May 29-31, 1991*, pages 268 – 277, 1991.

[10] M. Jaedicke and B. Mitschang. User-defined table operators: Enhancing extensibility for ORDBMS. In *Proc. 25th Intl. Conf. on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 494–505, 1999.

[11] E. Laber, F. Porto, P. Valduriez, and R. Guarino. Cherry picking: A data dependency-driven strategy for the distributed evaluation of expensive predicates. In *Technical Report, PUC-Rio, Informatics Department, PUC-Rio.Inf.MCC01/02, 2002*, 2002.

[12] S. Madden, M. Shah, Joseph M. Hellerstein, and V. Raman. Constinuously adaptive continuos queries over streams. In *Proc. of 2000 ACM SIGMOD Intl. Conf. on Management of Data, June 4-6, 2002, Madison, Wisconsin, USA*, pages 49–60, 2002.

[13] Tobias Mayr and Praveen Seshadri. Client-site query extensions. In *Proc. 1999 ACM SIGMOD Intl. Conf. on Management of Data, June 1-3,1999, Philadelphia, Pennsylvania, USA*, pages 347–358, 1999.

[14] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. Lorie, and T. G. Price. Access path selection in a relational data base management system. In *Proc. Intl. Conf. on Management of Data, May 30 - June 1, 1979, Boston, Massachusetts,USA*, pages 23–34, 1979.

[15] A. Tanaka, P. Valduriez, and et al. The ecobase environmental information system:applications, architecture and open issues. *ACM SIGMOD Record*, 3(5-6), 2000.