Memory Cognizant Query Optimization

Arvind Hulgeri*
Deptt. of Computer Sci. & Engg.
IIT-Bombay
aru@cse.iitb.ernet.in

S. Seshadri
Bell Labs
Murray Hill, NJ
seshadri@research.bell-labs.com

S. Sudarshan
Deptt. of Computer Sci. & Engg.
IIT-Bombay
sudarsha@cse.iitb.ernet.in

ABSTRACT

Complex queries make heavy use of join, aggregation and sorting operations and these operations are memory intensive. Typical optimizers assume all the memory to be available to each operator in the query tree. But while executing pipelines memory will get divided amongst all the operators running simultaneously in a pipeline. The cost of an operator generally depends on the available memory. If the memory allocated to an operator is less than what an optimizer assumes, cost estimated by the optimizer would be wrong. Thus the query optimization and memory distribution are interdependent and if done separately may not yield best results. The query optimizer should not only consider the total memory available but should also decide how to divide it optimally among the operators of the plan.

We show how to optimize a query given the *cost versus memory allocation* function for each operator. We have extended the Volcano optimizer to make it memory cognizant. Part of the job of the optimizer is to decide which edge to pipeline and which edge to block. A pipelinable edge can be broken (i.e. converted) into a blocking edge. But the decision to break a pipelinable edge depends upon whether the extra memory available to individual pipelinable trees thus formed can more than offset the extra disk IO of the intermediate results. This decision is integrated into our memory cognizant optimizer.

1. INTRODUCTION

Complex queries make heavy use of join, aggregation and sorting operations which are memory intensive. Conventional query optimizers have the drawback that they assume each of these operators can avail entire memory available to the query execution engine. This assumption is clearly not valid when executing a pipeline, where the available memory has to be divided among several concurrently executing operators in the pipeline and may lead to suboptimal plans.

Motivating Example: Consider a query $R \bowtie T \bowtie S$ with two join predicates: one between relations R and T and other between relations S and T. Cross products are not allowed. Relation sizes are $|R| = 60$, $|S| = 60$ and $|T| = 130$ disk blocks, memory available is 80 blocks. Cost is measured in terms of number of disk block accesses.

A traditional optimizer generates plan shown in Figure 1(a) and (b). Since the available memory is sufficient to execute either of the two nested loops join operators *in memory*, the two nested loops join operators in the plan are assumed to run in pipelined fashion in the best plan. The estimated cost of this plan is the cost of reading each relation once from the disk and equals $|R| + |S| + |T| = 240$. However, both nested loops join operators cannot be executed simultaneously in given memory. Thus the cost predicted by the optimizer is inaccurate; the actual cost is more than the estimated cost and depends upon scheduling of the operators in given memory. For example, as shown in Figure 1(a) a worst-case memory scheduling will allocate 79 blocks to one nested loops join and only one unit to other, thus making the cost approximately 7860 units which is significantly higher than the estimated cost.

Least cost would be incurred when the memory is divided equally between the operators as shown in Figure 1(b), Here the total cost incurred will be 610 units and is the optimal for the given query plan and the given memory.

However, if we employ hybrid hash[1] join instead of nested loops join for the two join operations as shown in Figure 1(c) and equally divide memory between the two joins, the total cost incurred will be 500 and the plan is optimal for given memory. □

The example illustrates two key issues: First, cost of a plan may change when division of memory is changed. Second, the choice of plan itself needs to involve consideration of memory allocation.

We propose two approaches to solve this problem: a *2-phase* approach and a *1-phase* approach.

In the 2-phase approach, we first optimize the query using a conventional optimizer to get a traditional optimal plan, and in the second phase we divide memory among operators in each pipeline of the plan so that each pipeline runs optimally in available memory. In the example above, Figure 1(b) shows application of the 2-phase approach.

In the 1-phase approach we modify the traditional query optimizer to make it memory cognizant. The modified optimizer takes into account division of memory amongst operators while choosing between equivalent plans. In the ex-

*Supported by Infosys Fellowship.

[1] assume left input to be a build input and right input to be a probe input

ADVANCES IN DATA MANAGEMENT 2000
Krithi Ramamritham, T.M. Vijayaraman (Editors)
Tata McGraw-Hill Publishing Company Ltd.

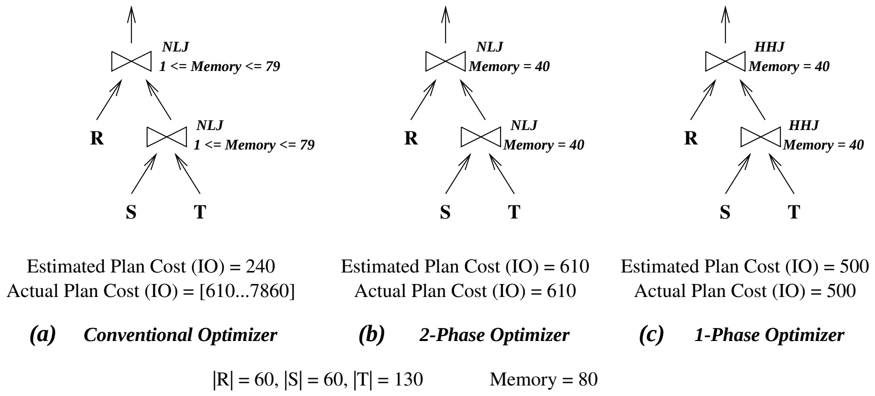


Figure 1: Motivating Example

ample above, Figure 1(c) shows application of the 2-phase approach.

Although the 2-phase approach is able to optimally divide the available memory amongst the operators of a given plan, it may not necessarily give the optimal execution time for a query since the plan being considered may itself be suboptimal for available memory. Therefore, the 1-phase approach seems to be a better alternative.

In this paper we address the problem of choosing an optimal, memory-aware execution plan for a query given the *cost versus memory allocation* function for each operator. We build *memory cognizant query optimizer* to solve this problem. Our contributions are as follows:

- We design efficient techniques to divide the available memory optimally among operators in a pipeline. If done naively, this process can take time quadratic in the available memory size, and is impractical. We show how to solve this problem in reasonable time by using piecewise linear approximation of cost-versus-memory functions of various operators.
- We show how some of the assumptions made while evaluating cost of various operators are not valid when cost is a function of available memory and we are dividing memory amongst operators. Based on these observations we define various memory cognizant execution algorithms/schemes for each operator.
- We show how to make a cost-based decision of breaking (i.e. converting) a pipelined edge into a blocking edge so that the child operator writes the intermediate result to the disk and the parent operator reads it back. The decision to break a pipelinable edge depends upon whether the extra memory available to individual pipelinable trees thus formed can more than offset the extra disk IO of the intermediate results. This decision is integrated into memory cognizant optimizer.
- It has been conjectured [7] that 1PO will perform no better than 2PO. But the paper gives no conclusive evidence of this claim. We evaluate 1PO against 2PO and study the results obtained. Performance results show that 2PO performs reasonably well as compared to 1PO for tests conducted.

Our discussion is in the context of Volcano query optimizer [3] but the techniques can be used with a System-R style optimizer [10] also.

The rest of the paper is organized as follows: We discuss the related work in Section 2. Section 3 gives brief overview of memory cognizant execution algorithms. In Section 4 we present our technique for choosing the optimal division of memory among a set of concurrently running operators. This technique forms an important building block of our algorithm for selecting the overall optimal plan. Section 5 describes the basic Volcano query optimizer, including the representation for the search space of all possible query plans. Section 6 describes how the Volcano optimizer is extended to include memory-cognizance and also describes how pipeline breaking decision is taken in cost-based manner. In Section 7 we present some experimental results, and finally conclude in Section 8.

2. RELATED WORK

Previous work on resource management mainly deals with scheduling resources efficiently for executing a given query plan. These resource scheduling decisions do not interact with the query optimizer and the two problems (query optimization and scheduling) are solved in separate phases: the query is optimized first to come up with a plan, and then the plan is scheduled.

Query schedulers can be broadly classified into two categories: *static* schedulers and *dynamic* schedulers. Static scheduling is applied after query optimization and before execution. The dynamic scheduling strategy is integrated with the query execution engine and makes the engine adaptive to fluctuations in resource availability.

To the best of our knowledge no previous work has tried to integrate scheduling decisions in query optimizer and our work is the first one to do this (1PO). Several papers have considered scheduling of query plans (2PO). Again, to the best of our knowledge no previous work has tried cost-based scheduling and all the previous scheduling strategies are heuristic based, the only exception being a strategy proposed by Nag and DeWitt[7] which assumes operator cost functions to be linear w.r.t to memory. They [7] conjecture that 1PO will perform no better than 2PO but give no conclusive evidence of this claim.

Yu and Cornell [11] consider an environment of concurrently running queries and study the problem of memory allocation to individual queries. They define a concept of *return-on-consumption (ROC)* to study overall reduction in response time due to additional memory allocation to a query. Each query is assumed to be a single-join query. A memory scheduling policy is proposed wherein more memory is allocated to queries which have high value of ROC.

Mehta and DeWitt[6] also consider the problem of memory scheduling in multi-query environment. They divide queries in different categories depending upon their memory requirement and provide several heuristics for memory allocation depending upon the classification. Again, only single-join queries are considered and only hash join operator is considered.

Bouganim et al.[1] propose various static and dynamic scheduling schemes for a query tree. They split a given query tree into a set of maximal pipeline chains, each called *pc_task* and scheduled separately. Under static scheduling they propose several heuristic-based strategies to divide memory among operators depending upon minimum (M^{op}_{min}) and maximum (M^{op}_{max}) memory requirements of each operator. The heuristics proposed include: divide memory equally amongst the operators, allocate maximum memory to the operator which has the smallest M^{op}_{max}, give each operator memory in proportion to its M^{op}_{max}, and give maximum amount of memory to an operator having the largest M^{op}_{max}. Under dynamic scheduling strategy they propose a memory adaptive execution engine, which dynamically allocates memory and changes query scheduling as and when required. If the memory requirement of a pc_task can not be satisfied, they split the pc_task into segments to be scheduled separately. The scheduling strategies defined consider only maximum and minimum memory requirement of the operators and the memory allocation decision is not cost-based.

Nag and DeWitt[7] propose several heuristic based static scheduling strategies which are more or less similar to those proposed by Bouganim et al.[1]. They divide a query tree into concurrently schedulable units called *shelves* and divide memory amongst operators in a shelf. They also propose a cost based strategy which assumes operator cost model to be linear. We consider more general cost model.

Davison and Graefe[2] and Zeller and Gray[12] show how to make a single hash join adaptive to memory fluctuations but do not consider scheduling of an entire query plan. Pang et.al.[8] examine the same problem in the context of real-time databases where the priority of a query needs to be considered and a query may need to be scheduled in absence of sufficient memory.

3. MEMORY COGNIZANT EXECUTION

Generally following assumptions are made while evaluating cost of an operator. These are not valid when cost is a function of available memory and memory is divided amongst operators:

- It is assumed that each operator in a pipeline utilizes all available memory and input is streamed into it in pipelined fashion. Thus the cost of an operator is decided by available memory and size of its input. This assumption is not valid as already described.
- It is assumed that for a join operator, the smaller of the two inputs is outer or left input and the larger one is inner or right input and this yields optimal cost. This assumption is not valid as described further.
- It is assumed that in multiphase sort or hash operation each merging or partitioning phase utilizes all available memory but in reality the first and the last phase need to share the available memory with child operator and parent operator resp. and only intermediate phases utilize all available memory.

Based on these observations we define various execution schemes for each operator which primarily dictate scheduling of various phases of an operator, memory utilization of these phases and which one of the two inputs acts as left input for join operation. We define following four schemes for executing a nested loops join (NLJ) operator. Similar schemes for rest of the operators can be found in [4]. Let M_{tot} be total available memory, M be memory available for the plan tree rooted at NLJ and L be size of left input.

- **Scheme 1:** Initially left input tree is completely processed to generate all left input tuples which are stored in-memory. Then right input tree is processed and right input tuples, as generated, are matched with left input tuples which are memory resident. Thus the left input is blocking though it is not written to disk, and the right input is pipelined.

  Of the memory, M units, allocated to the tree, L units is used by NLJ for storing left input. Thus remaining $(M - L)$ units is used for processing the left input tree. Once the left input tree is processed, the right input tree is executed and it also gets $(M - L)$ units of memory for its execution.

- **Scheme 2:** Here again the left input tree is blocking and the right input is pipelined but the left input is written to disk. The left input is completely processed before NLJ starts and all the left input tuples are stored on the disk. Then the right input is executed in pipelined fashion with NLJ. As the left input is written to disk, NLJ can use memory less than L units (unlike *scheme 1*) and read the left input multiple times from disk.

  As the left input is processed independently, it uses $(M_{tot} - 1)$ units of memory for its execution. One unit is used for buffering the tuples before writing them to the disk. Then the right input is executed in pipelined fashion with NLJ. So M units memory, allocated to the tree, will be divided optimally between NLJ and the right input, say M' units to NLJ and $M - M'$ units to the right input.

- **Scheme 3:** Same as *scheme 2* but with the roles of left and right inputs reversed.

In *scheme 2*, the cost of executing the tree rooted at NLJ in M units of memory would be the summation of cost of executing NLJ in M' units of memory, cost of the left input running in $M_{tot} - 1$ units of memory and cost of the right input running in $M - M'$ units of memory. It is easy to see that we can not decide which one of the two inputs should

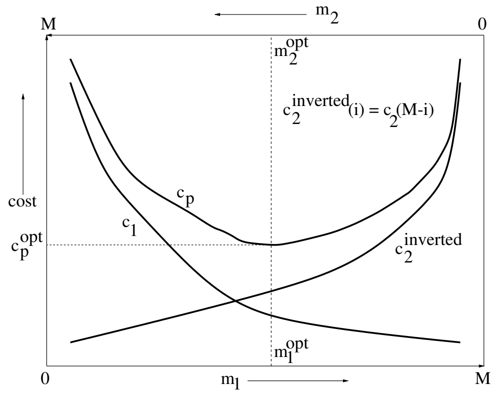


Figure 2: Optimal division of memory between o_1 and o_2

be the left input based only on their sizes and we need evaluate the cost for both the alternatives, hence the need for *scheme 3.*

4. OPTIMAL DIVISION OF MEMORY FOR A GIVEN QUERY PLAN

We present here a crucial building-block of our memory-cognizant optimizer: the technique used in optimally dividing memory among a set of operators running in a pipeline.

When a pipeline is executed, all operators in the pipeline run simultaneously in given memory. The cost of running such a pipeline depends upon how much memory each operator in the pipeline gets, and hence it is important to choose the optimal division of memory among these operators.

4.1 Cost Functions with Arbitrary Shape

In this section we consider optimal memory allocation for operators in a pipeline, where operator cost functions are of arbitrary shape. We describe our technique by first considering the problem of sharing available memory between two operators. We then consider three operators, and finally the general case with n operators.

Dividing memory between two operators: Consider a pipeline P composed of two operators o_1 and o_2 with cost functions c_1 and c_2 respectively. Let the available memory be M units. Here, unit refers to a unit of allocation. We can divide this memory between the two operators by giving m_1 units to operator o_1 and m_2 units to operator o_2 ($m_1 + m_2 = M$). Clearly, the cost of executing the pipeline is a function of this division. Let c_p be cost of the pipeline as a function of m_1, with $m_2 = M - m_1$. Thus $c_p(m_1) = c_1(m_1) + c_2(M - m_1)$.

This function can be computed by inverting the cost function c_2 along the memory axis and adding it to the cost function c_1. This is shown in Figure 2. We need to find value of m_1, say m_1^{opt} at which c_p is minimal, say c_p^{opt}, as shown in Figure 2. This is the optimal allocation to operator o_1. Correspondingly optimal allocation to operator o_2 will be $m_2^{opt} = M - m_1^{opt}$.

$$\forall i, 1 \leq i \leq M, \forall j, 1 \leq j \leq M :$$
$$i + j = M \Rightarrow c_1(i) + c_2(j) \geq c_1(m_1^{opt}) + c_2(m_2^{opt})$$

For cost functions with arbitrary shape we need to examine all possible division points and calculate c_p for each value of m_1 from 0 to M. This takes $O(M)$ time. Note that there is a tradeoff between time and granularity of the unit of memory.

Dividing memory amongst three operators: Consider a pipeline P composed of three operators o_1, o_2 and o_3 with cost functions c_1, c_2 and c_3 resp. The goal is to find optimal memory allocation, say m_1^{opt}, m_2^{opt} and m_3^{opt} units to operators o_1, and o_2 and o_3 resp. such that the execution cost of pipeline P is minimized and $m_1^{opt} + m_2^{opt} + m_3^{opt} = M$. Such a division is given by:

$$\forall i, 1 \leq i \leq M, \forall j, 1 \leq j \leq M, \forall k, 1 \leq k \leq M :$$
$$i + j + k = M \Rightarrow c_1(i) + c_2(j) + c_3(k) \geq$$
$$c_1(m_1^{opt}) + c_2(m_2^{opt}) + c_3(m_3^{opt})$$

A naive method would check all possible combinations of memory allocations and take $O(M^3)$ time. But we can do better by *merging* the cost functions (and thus the operators) incrementally, as follows:

First, we consider two operators o_1 and o_2 and find cost of a pipeline P' consisting these two operators as a function of memory used by the pipeline. Thus we need to calculate the optimal cost of running the pipeline P' with memory i, for each $i : 0 \leq i \leq M$. The cost function of P' stores the optimal division of memory along with the cost value for each memory point i (i.e. amount of memory j and k to be given to operators o_1 and o_2 resp., where $j + k = i$). With this the operators o_1 and o_2 are virtually *merged* into plan P' with a cost function defined for it. For further memory division operations, say between plan P' and some other operator or plan, the plan P' will be treated as a *super-operator.*

We can calculate optimal division for a given memory size i and hence cost of running the pipeline P' with memory size i in $O(M)$ time. To calculate this for all $i : 0 \leq i \leq M$ we need $O(M^2)$ time. Thus the cost function of the pipeline P' is calculated in $O(M^2)$ time.

Next, we find the optimal memory division between the plan P' and the operator o_3. Once we get this division, say m_3^{opt} and $m_{p'}^{opt}$, we can trace back optimal division of memory $m_{p'}^{opt}$ between operators o_1 and o_2.

Dividing memory amongst n operators: Consider a general case: a pipeline P composed of n operators o_1, o_2, ..., and o_n with cost functions c_1, c_2, ..., c_n respectively. To find the optimal memory division amongst these operators we extend the strategy used for the pipeline with 3 operators, and merge the cost functions/operators incrementally. We merge the cost functions of two operators to get their combined cost function. Then we merge this combined cost function with the cost function of the third operator to get the combined cost function of three operators. We continue in this manner and merge all operator cost functions to get the cost function of the pipeline P. Now, we can trace back each step and find, from the cost functions of the intermediate plans, the optimal memory allocation within each of these intermediate plans. This takes $O(n.M^2)$ time for cost functions with arbitrary shape.

$OptMerge$ **Procedure:** This procedure optimally merges two input cost functions running in pipeline and generates the combined optimal cost function along with the optimal memory division for all memory points. Time complexity of this procedure depends upon shape of the input cost functions. If the input cost functions are of arbitrary shape then the procedure examines all possible memory division alternatives for each memory point and the complexity is $O(M^2)$.

In the next section, we consider how to reduce the cost associated with the memory division operation using piecewise-linear approximations of the cost functions.

4.2 Piecewise Linear Approximation of Cost Functions

Consider a pipeline P with two operators o_1 and o_2 with cost functions c_1 and c_2 resp. Let cost function of the pipeline be c_p. If the cost functions c_1 and c_2 are linear, the procedure $OptMerge$ defined in previous subsection takes constant time. Assume that c_1 has slope $slope_1$ in range 0 to m_1 ($m_1 \leq M$) and slope 0 in range m_1 to M i.e. allocating memory beyond m_1 units yields no benefit. And c_2 has slope $slope_2$ in the range 0 to m_2 ($m_2 \leq M$) and slope 0 in range m_2 to M. Assuming that providing more memory will not increase the cost of an operator/plan, the cost functions are non-increasing and slopes are non-positive ($slope_1 \leq 0$ and $slope_2 \leq 0$).

The $OptMerge$ procedure simply allocates maximum possible memory to the operator which gains more per unit memory allocation (i.e., the cost function of which has less slope[2]) and allocates the remaining memory to the other operator. Let $slope_1 \leq slope_2$. The procedure would allocate m_1 units of memory to operator o_1 and c_p follows the slope $slope_1$ in the range 0 to m_1. Next, it allocates m_2 units ($m_2 <= M - m_1$) of memory to operator o_2, and c_p follows the slope $slope_2$ in the range m_1 to $m_1 + m_2$. For memory in the range $m_1 + m_2$ to M c_p has slope 0 and cost value same as that for $m_1 + m_2$.

To use this mechanism, we need to approximate cost functions of various operators to linear form. However, even after this approximation, cost functions derived by the procedure $OptMerge$ may not be linear: in fact the derived functions may be piecewise linear. Thus cost function of a plan can take piecewise linear form. Since input to $OptMerge$ procedure can be cost function of a plan, the procedure needs to handle piecewise linear cost functions.

Moreover, the ability to handle piecewise linear cost functions means we can use piecewise-linear approximations for single-operator cost functions. This provides a better approximation than the linear approximation. It is easy to approximate cost functions of various database operators[3] to piecewise linear form.

[2]considering sign. If we consider absolute values then it would pick the one with higher slope.

[3]including multiphase sort and hash operators which typically have discontinuous cost functions w.r.t. memory

```
OptMerge(c1, c2)
    mergeCost = ∞
    for each change-over point (m, c) in c1 do
        c2' = c2 shifted by (m, c)
        mergeCost = MinMerge(mergeCost, c2')
    for each change-over point (m, c) in c2 do
        c1' = c1 shifted by (m, c)
        mergeCost = MinMerge(mergeCost, c1')
    return mergeCost
```

Figure 3: Pseudo Code for $OptMerge$ Procedure for Piecewise Linear cost Functions

The $OptMerge$ procedure dealing with piecewise linear cost functions is shown in Figure 3. The operation *shift by* used in the procedure shifts the cost function along memory and cost axis by resp amount. The routine $MinMerge$ used in the procedure compares input cost functions for entire memory range and at each memory point picks up the lower cost value.

If the input cost functions to this procedure have o_1 and o_2 segments (i.e. number of straight line segments in a piecewise linear function) resp. and $z = max(x, y)$ then the number of segments in the output cost function will be $\leq z^2$ and the time complexity of the procedure would be $O(z^2 \log z)$.

For each given point i, the algorithm essentially checks each possible memory division, say j units to the first cost function and k units to the second cost function ($i = j + k$), where at least one of j and k is at a change-over point (a point where the cost function changes slope) in the resp. cost functions. And following theorem establishes that the procedure correctly calculates the optimal cost function of the pipeline.

Theorem: *If a pipeline tree P composed of two operators o_1 and o_2 with cost functions c_1 and c_2 respectively is executed in memory i, then at least one of the (possibly many) optimal memory divisions, say j units to o_1 and k units to o_2 ($i = j+k$), is such that at least one of j and k is at a change-over point.*

Proof: Assume that there is no such optimal division. Consider one of the optimal divisions of the available memory i, say j units to o_1 and k units to o_2 ($i = j + k$). Neither of i and j is at changeover point. We come up with an alternate memory division, say j' units to o_1 and k' units to o_2 ($i = j' + k'$) such that at least one of j' and k' is at a change-over point and $c_1(j') + c_2(k') \leq c_1(j) + c_2(k)$.

Assume that the point j lies on segment s_1 in c_1 and k on segment s_2 on c_2. Assume further that the segment s_1 begins at memory point b_{s1} and ends at memory point e_{s1} and the segment s_2 begins at memory point b_{s2} and ends at memory point e_{s2}.

Let the slopes of the segments s_1 and s_2 be $slope_{s1}$ and $slope_{s2}$ resp. Assuming that providing more memory will not increase the cost of an operator/plan, the cost functions are non-increasing and slopes are non-positive ($slope_{s1} \leq 0$ and $slope_{s2} \leq 0$) though the algorithm does not depend on this assumption.

We consider three cases:

Case I: $slope_{s1} = slope_{s2}$
Consider two subcases:

Cast I.A: $j - b_{s1} \leq e_{s2} - k$
Let $j' = b_{s1}$ and $k' = k + j - b_{s1}$.
It is easy to see that
$c_1(j') + c_2(k') = c_1(j) + c_2(k)$ and $j' + k' = i$.

Cast I.B: $j - b_{s1} > e_{s2} - k$
Let $j' = j - e_{s2} + k$ and $k' = e_{s2}$.
It is easy to see that
$c_1(j') + c_2(k') = c_1(j) + c_2(k)$ and $j' + k' = i$.

Case II: $slope_{s1} > slope_{s2}$
Consider two subcases:

Cast II.A: $j - b_{s1} \leq e_{s2} - k$
Let $j' = b_{s1}$ and $k' = k + j - b_{s1}$.
It is easy to see that
$c_1(j') + c_2(k') < c_1(j) + c_2(k)$ and $j' + k' = i$.

Cast II.B: $j - b_{s1} > e_{s2} - k$
Let $j' = j - e_{s2} + k$ and $k' = e_{s2}$.
It is easy to see that
$c_1(j') + c_2(k') < c_1(j) + c_2(k)$ and $j' + k' = i$.

Case III: $slope_{s1} < slope_{s2}$
Symmetric to case II.

We see that in all the cases, either j' or k' is at changeover point and this alternative division incurs cost equal to or less than that incurred by the original division. This contradicts the assumption made in the beginning, hence the proof. □

If the piecewise linear approximation introduces maximum error of $\pm\delta$ at any memory point in each operator cost function in a plan with n operators then cost of the plan calculated using piecewise linear approximation will be within $\pm n\delta$ of the actual cost.

5. VOLCANO OPTIMIZER BACKGROUND

In this section we describe the basic Volcano query optimizer[3]. First we describe an AND-OR DAG representation for the search space of all possible query plans. Then we describe the volcano search algorithm.

5.1 The DAG Representation of Queries

An AND–OR DAG is a directed acyclic graph whose nodes can be divided into AND-nodes and OR-nodes; the AND-nodes have only OR-nodes as children and OR-nodes have only AND-nodes as children.

An AND-node in the AND-OR DAG corresponds to an algebraic operation, such as the join operation ($\bowtie$) or a select operation (σ). It represents the expression defined by the operation and its inputs. Hereafter, we refer to the AND-nodes as *operation nodes*. An OR-node in the AND-OR DAG represents a set of logical expressions that generate the same result set; the set of such expressions is defined by the children AND nodes of the OR node, and their inputs. We shall refer to the OR-nodes as *equivalence nodes* henceforth.

The given query tree is initially represented directly in the AND-OR DAG formulation. For example, the query tree of Figure 4(a) is initially represented in the AND-OR DAG formulation, as shown in Figure 4(b). Equivalence nodes (OR-nodes) are shown as boxes, while operation nodes (AND-nodes) are shown as circles.

The initial AND-OR DAG is then expanded by applying all possible transformations on every node of the initial query DAG representing the given set of queries. Suppose the only transformations possible are join associativity and commutativity. Then the plans $A \bowtie (B \bowtie C)$ and $(A \bowtie C) \bowtie B$, as well as several plans equivalent to these modulo commutativity can be obtained by transformations on the initial AND-OR-DAG of Figure 4(b). These are represented in the DAG shown in Figure 4(c). We shall refer to the DAG after all transformations have been applied as the *expanded DAG*. Note that the expanded DAG has exactly one equivalence node for every subset of $\{A, B, C\}$; the node represents all ways of computing the joins of the relations in that subset.

5.2 Physical AND-OR DAG

Properties of the results of an expression, such as sort order, that do not form part of the logical data model are called *physical properties*. Physical properties of intermediate results are important, since e.g. if an intermediate result is sorted on a join attribute, the join cost can potentially be reduced by using a merge join. It is straightforward to refine the above AND-OR DAG representation to represent physical properties and obtain a physical AND-OR DAG[4].

5.3 The Volcano Search Algorithm

The Volcano Search Engine follows a top-down, goal-driven approach. It generates the logical DAG and expands it into the physical DAG on the fly. We present below parts of the Volcano search algorithm, that are relevant to our optimization algorithms. Figure 6 in Appendix A shows a simplified version of the Volcano search algorithm.

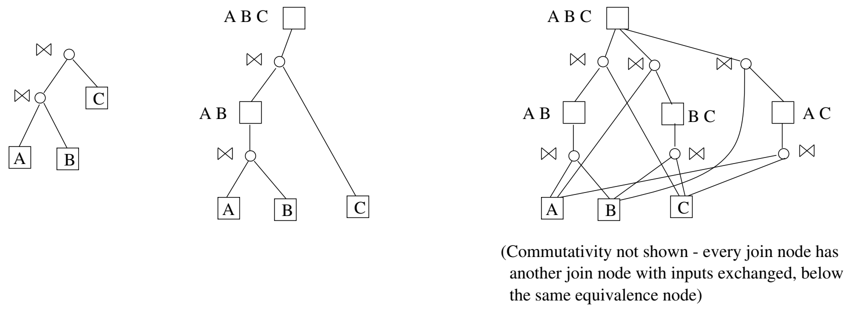
Before the Volcano search algorithm is called on a query, the initial query DAG corresponding to the given query is created. Next, the initial DAG is fully expanded by applying the transformations as described earlier, to get an expanded logical AND-OR DAG. The search procedure is then called on the root of the expanded logical AND-OR DAG. (Note: the description of Volcano in [3] does not make this separation explicit, but the actual implementation does follow this two phase approach for completeness of transformations [5].)

The input to the Volcano search procedure is a logical equivalence node, an initial physical property specification and an optional cost limit.

The search procedure tries alternative enforcers and algorithms for the operation nodes below the equivalence node, recursively calling itself to find the best plan for the inputs of the operation nodes. A cost limit is passed as a parameter to the search algorithm, and if the cumulative cost of an operation node and the costs of the best plans for its inputs chosen so far exceeds the limit, the operation node can be abandoned from consideration.

Once the best plan for an (*equivalence node*, *physical*

[4]For example, an equivalence node is refined to multiple physical equivalence nodes, one per required physical property, in the physical AND-OR DAG. Enforcer operation nodes, such as sort also get introduced.



(a) Initial Query (b) DAG representation of query (c) Expanded DAG after transformations

Figure 4: Initial Query and DAG Representations

property) pair is found, it is stored in case it needs to be reused. Therefore, in fact, the first thing to check before performing the above optimization for a given node and a given physical property is to check for potential reuse. If a plan matching the property specification is found among the plans stored at the equivalence node, and the plan satisfies the cost limit, the plan is returned; if a plan is found but does not satisfy the cost limit a failure indication is returned. If there is no plan for the expression and the property specification, then actual optimization (as described above) starts.

The best plan for a logical equivalence node, physical property pair (thus, a physical equivalence node) is compactly specified by merely noting the corresponding physical operation node, and its input physical equivalence nodes. The overall best plan is reconstructed when required by recursively looking up the best plan for the inputs.

6. MEMORY COGNIZANT OPTIMIZATION

In this section we present an overview of the extensions to make the Volcano optimizer algorithm memory cognizant. Details of the memory cognizant optimization algorithm are presented in Appendix B.

Our execution algorithms (described in Section 3) for query operators include memory-awareness and division of memory among concurrent operators. In this framework, we propose following extensions to make the optimization process memory-cognizant:

- While evaluating cost of an operator (AND node in Volcano DAG framework), evaluate cost functions for all the execution schemes (as defined in section 3) and for each memory size pick the one with the minimum cost. Note that the inputs to the operator are already optimized and we know their cost functions.

- While comparing alternative operators or plans, for evaluating an expression (OR node in Volcano DAG framework), compare their cost functions for each memory size. If one operator/plan is consistently better than other in the entire memory range, retain the operator/plan with less cost and discard the operator/plan with more cost. If one operator/plan is better at some memory range and other one is better at some other memory range, maintain both of them indicating which one in better in which range. The cost of the expression at a memory size m is the cost of the operator/plan which incurs minimum cost at that memory size.

- While evaluating cost of a plan P (AND node in Volcano DAG framework), given cost function of root operator O and that of sub-plan $P' = P \setminus O$, we consider following possibilities:

  - **Edge between O and P' is blocked:** P' runs fully before O starts executing. Thus full memory can be allocated to P'. This will result in minimum execution cost for P' and it will be $P'.CostFunction(MaxAvailMem)$. The memory to be allocated to operator O cannot be decided independently of what type of edge it will be connected to its parent by and what is the execution cost of its parent/ascendents. Thus cost function of the plan P will be the cost function of O with $P'.CostFunction(MaxAvailMem)$, a constant, added for each memory point.
    $\forall i, 1 \leq i \leq MaxAvailMem:$
    $$P.CostFunction(i) = O.CostFunction(i) + P'.CostFunction(MaxAvailMem)$$

  - **Edge between O and P' is pipelined:** O and P' run simultaneously in memory allocated to the plan P. The cost function of P is obtained by $OptMerge$-ing cost function of O and that of P'. Recall the definition of the procedure $OptMerge$ from Section 4.
    $$P.CostFunction = OptMerge(O.CostFunction, P'.CostFunction)$$

6.1 Breaking Pipelined Edges

Consider a pipelinable plan P and a pipeline edge E in it. If we break the plan P at edge E we get two independent subplans P_1 and P_2 and these plans can be scheduled separately. Let us assume that output of plan P_1 is fed to plan P_2 through edge E.

We have two options for evaluating plan P:

- Schedule whole plan P in given memory with the edge E behaving as a pipeline edge. Here all operators in the plan P execute simultaneously sharing available memory. There is no I/O incurred at edge E, as it is a pipeline edge.

- Schedule P_1 first, store its output on disk. Then schedule P_2 with its input being read from disk. Here, as P is divided into two parts and each part is scheduled separately, operators will have more memory for execution. However, we incur materialization I/O at edge E which now behaves as a blocking edge.

Clearly, there is a tradeoff between letting the edge E behave as a pipeline edge and breaking it to make it behave as a blocking edge. If it is a pipelined edge, no materialization IO is incurred but operators in P will get less memory for execution as all the operators in the plan execute simultaneously in the available memory. If the edge E is a blocking edge, materialization IO is incurred but as the operators in the plan are divided into two independent plans and scheduled separately, the operators will get more memory for execution.

We incorporate, into our memory cognizant optimizer, a cost-based technique for deciding when to break a pipelined edge and is described below.

Consider a plan P feeding its output to parent C in pipelined fashion. Let the pipelined cost function of P be PPC (with its output edge pipelined and no IO incurred at it). And let read/write cost be IO at its output edge if it is blocked. The problem is to decide at each memory point, say i:

- Let the plan P and its parent C execute in pipelined fashion. The cost of the plan P is $PPC(i)$.
- Let the plan P execute independent of its parent C utilizing all available memory, say $MaxAvailMem$[5] and write intermediate result to disk which will be read by the parent C. The cost of the plan P is $PPC(MaxAvailMem - 1) + IO$. Note that it is independent of the available memory i.[6]

Let blocking cost function of plan P be BPC (with its output edge blocked and IO incurred at it). It is given by:

$$\forall i, 1 \leq i \leq MaxAvailMem : \\ BPC(i) = P(MaxAvailMem - 1) + IO$$

The optimal cost function for plan P with the blocking decision incorporated within is given by:

$$MinMerge(PPC, BPC)$$

The routine $MinMerge$ compares two input cost functions for the entire memory range and at each memory point picks up the lower cost value. Thus applying $MinMerge$ chooses better of the options: blocking and pipelining the edge. Figure 5 shows the operation graphically. If an operator or a plan is made to execute in memory less than certain threshold m_{cut_off}, it will, instead, utilize full memory and write its output to disk. For a cost function with arbitrary shape the time complexity of this decision is $O(M)$, whereas for a piecewise linear cost function with x linear segments it is $O(x)$.

[5]Actually, child will get $(MaxAvailMem - 1)$ for its execution as one unit of memory will be used for holding the intermediate tuples as they are written to disk.

[6]Actually, we need $1 \leq i$ since at least one buffer is needed to read back the intermediate result from disk and feed the parent.

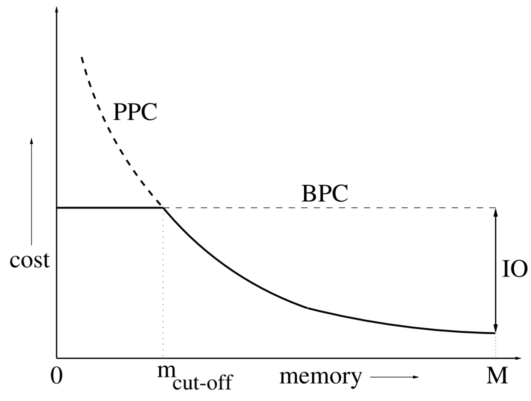


Figure 5: Breaking a pipelined edge

7. EXPERIMENTAL EVALUATION

In this section we describe our experimental setup and the results obtained.

The 1PO and 2PO algorithms are based on Volcano query optimizer. The first phase of 2PO (which uses the basic volcano optimizer to optimize the query in conventional manner) assumes that each operator in the plan uses all available memory.

The memory block size is taken as 4K. Standard techniques are used for estimating costs, using statistics about relations. The cost estimates contain an I/O component and a CPU component. The metric used to compare the goodness of the optimization algorithms is the estimated cost of the optimal plan produced by the optimizer; all our cost numbers are estimates from the optimizer.

The tests are performed on a Sun workstation with UltraSparc 10 333Mhz processor, 256MB main memory, running Solaris 5.7.

Test Queries

We tested our algorithms with around 20,000 randomly generated queries on a TPCD-based star schema similar to the one proposed by [9]. The schema has a central *orders* fact table, and four dimension tables *part, supplier, customer* and *time*. The size of each of these tables is same as that in the TPCD-1 database. This corresponds to base data size of approximately 1 GB. Each generated query is of the form:

```
select sum(quantity)
from orders, supplier, part, customer, time
where join-list and select-list
group by groupby-list;
```

The *join-list* enforces equality between attributes of the order fact table and primary keys of the dimension tables. The *select-list* i.e., the predicates for the selects are generated by selecting some attributes at random from the join result, and creating random equality or inequality predicates on the attributes. The *groupby-list* is generated by picking a subset of {*custkey, suppkey, partkey, custkey, month, year*} at random.

We randomly choose, between 10 blocks to 10,000 blocks, the total memory available to execution engine and this forms part of the input to the optimizer.

Experimental Results

We tested total 23,603 randomly generated queries and performance benefit of 1PO over 2PO is reported below[7]:

Cost Reduction of 1PO over 2PO	#Queries	%Queries
00-10 %	22682	96.097
10-20 %	57	0.241
20-30 %	527	2.232
30-40 %	238	1.001
40-50 %	99	0.419

The maximum cost reduction reported by 1PO over 2PO in our experiments is 50%. For around 96% of the queries reduction is between 0% to 10%, and for only 4% of the queries reduction is between 10% to 50%. Thus, for the class of queries we considered, 1PO gives benefits, but generally 2PO performs about as well as 1PO.

The average optimization time taken by 2PO and 1PO is shown in the table below:

Algorithm	Optimization Time (msec)
2PO	150
1PO	1110

The cost based pruning feature of Volcano is not implemented in 1PO algorithm and 1PO explores full search space. Whereas, 2PO uses standard volcano implementation in its first phase and hence includes cost based pruning.

8. CONCLUSION AND FUTURE WORK

We have designed efficient techniques to divide available memory optimally among operators in a pipeline. If done naively, this process is impractical. We showed how to improve optimization time by using piecewise linear approximation for the cost-versus-memory functions of various operators and this made evaluation of 1PO feasible.

It has been conjectured that 1PO will perform no better than 2PO, but there has been no published evidence of this claim. We designed a practical cost-based algorithm for 1PO and compared it against 2PO. For the class of queries we considered, 1PO gives benefits, but generally 2PO performs about as well as 1PO. Thus, the preliminary results indicate that using 1PO for query optimization may not be beneficial. This is a good news in general as the optimizer remains simpler and faster.

The techniques developed here are of independent interest and can very well be applied to other problems.

The cost of a query plan depends on many parameters and available memory is just one of them. We see a natural connection between memory cognizant optimization and parametric query optimization and consider applying techniques developed in context of memory cognizant optimization to solve general parametric query optimization problem.

Unlike simple predicates and expressions, in Object-Relational Database (ORDB), expensive predicates operate on large complex data types and consume significant memory. This resource usage should be taken into consideration during optimization. Thus the techniques developed for memory cognizant query optimization can be easily applied to query optimization in ORDB. We wish to explore the possibility of integrating memory allocation decisions with the ORDB optimizer.

We propose to modify some query execution engine to make it memory cognizant. Such an execution engine will take the optimized plan from the optimizer along with the memory allocation number for each operator, pipeline or blocking decision for each edge, sequencing decisions of the pipelinable segments in the plan and execute the plan accordingly.

[7] Since we are using strictly cost-based exhaustive exploration of the search space, 1PO will never miss a 2PO plan, and hence is atleast as cheap as 2PO.

Acknowledgment

We are grateful to Krithi Ramamritham and Prasan Roy for comments on final draft of the paper. Prasan Roy provided code of basic Volcano Query Optimizer prototype.

9. REFERENCES

[1] L. Bouganim, O. Kapitskaia, and P. Valduriez. Memory-adaptive scheduling for large query execution. In *Proc. of the 7th CIKM Conf.*, pages 105–115, Bethesda, USA, 1998.

[2] D. L. Davison and G. Graefe. Memory-contention responsive hash joins. In *Proc. of the Int'l Conf. on VLDB*, pages 379–390, Santiago, Chile, 1994.

[3] G. Graefe and W. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proc. of the IEEE Conf. on Data Engg.*, Vienna, 1993.

[4] A. Hulgeri, S. Seshadri, and S. Sudarshan. Memory cognizant query optimization. Technical report, Indian Institute of Technology, Bombay, Sept 2000.

[5] B. McKenna. Personal communication.

[6] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple query workload. In *Proc. of the Int'l Conf. on VLDB*, pages 354–367, Dublin, Ireland, 1993.

[7] B. Nag and D. J. DeWitt. Memory allocation strategies for complex decision support queries. In *Proc. of the 7th CIKM Conf.*, pages 116–123, Bethesda, USA, 1998.

[8] H. Pang, M. J. Carey, and M. Livny. Managing memory for real-time queries. In *Proc. of the SIGMOD Conf. on Management of Data*, pages 221–232, Minneapolis, USA, 1994.

[9] P. Scheuermann, J. Shim, and R. Vingralek. Dynamic caching of query results for decision support systems. In *Intl. Conf. on Scientific and Statistical Database Management*, 1999.

[10] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the SIGMOD Conf. on Management of Data*, pages 23–34, 1979.

[11] P. S. Yu and D. W. Cornell. Buffer management based on return on consumption in a multi-query environment. *VLDB Journal*, 2(1):1–37, 1993.

[12] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environment. In *Proc. of the Int'l Conf. on VLDB*, 1990.

APPENDIX

A. VOLCANO OPTIMIZER ALGORITHM

Figure 6 presents pseudo code for basic volcano query optimization algorithm.

B. MEMORY COGNIZANT VOLCANO

B.1 Extended Cost Function and Plan

We present here some definitions and extensions that are used in memory cognizant optimization algorithm.

A conventional optimizer has a single value as cost for an operator or a query plan and its corresponding ($LogExp$, $PhysProp$) pair. Here we have a cost associated with each memory size, i.e. we define cost as a function of memory size. We refer to a function of cost versus memory size as $CostFunction(m)$, and we have one such cost function for each operator and query plan with a $(LogExp, PhysProp)$ pair. $Plan.CostFunction(m)$ denotes cost function of a plan, $AlgorithmCostFunction(m)$ denotes cost function of an algorithm and $EnforcerCostFunction(m)$ denotes cost function of an enforcer.

For each operator, as we are considering a range of memory, more than one memory cognizant execution schemes (as described in section 3) may be optimal, each being optimal in a particular memory range.

Further, for a $(LogExp, PhysProp)$ pair, as we are considering the range of memory, more than one physical plan may be optimal, each being optimal in a particular memory range[8]. The optimal $Plan$ for a $(LogExp, PhysProp)$ pair will contain a list of pairs. Each pair will contain a range of memory size and an optimal physical plan in that range. A physical plan P specified with a memory range (x,y) implies that the optimal way of evaluating the $(LogExp, PhysProp)$ pair, given that the memory size is in the range (x,y), is by using the plan P.

A conventional optimizer has a single value as a cost limit. Here we have a cost limit for each memory point. We use $CostLimitFunction(m)$ to denote a function giving the value of the cost limit for memory m.

Additionally, in a cost function for a $(LogExp, PhysProp)$ pair, an algorithm or an enforcer, we may have a segment where we have no optimal plan or execution scheme but a failure indication. In this segment, the cost function actually indicates a cost limit on the plan. The cost of the plan will be more than the cost function at each memory point in this range.

B.2 Operations on Cost Functions

Let $MaxAvailMem$ be the available memory. We define following operators on $CostFunction$:

- $\leq_{all}$: $costFunction_x \leq_{all} costFunction_y$ means, $\forall i, 1 \leq i \leq MaxAvailMem$:
  $$costFunction_x(i) \leq costFunction_y(i)$$
- $>_{all}$: $costFunction_x >_{all} costFunction_y$ means, $\forall i, 1 \leq i \leq MaxAvailMem$:
  $$costFunction_x(i) > costFunction_y(i)$$

[8]This may increase the search space

- $AddCostFunctions$: It takes two $CostFunctions$ as arguments, and creates a new $CostFunction$ by adding the input $CostFunctions$ at each memory point.
- $SubtractCostFunction$: it takes two $CostFunctions$ as arguments, and creates a new $CostFunction$ by subtracting second input $CostFunction$ from the first one.
- $MinMerge$: It compares two cost functions for the entire memory range and at each memory point picks up the lower cost value.
- $OptMerge$: It optimally combines the $CostFunctions$ of the two operators/plans which run simultaneously. Given two $CostFunctions$ corresponding to two plans it divides the memory available between the two plans such that the combined execution cost is minimized and does this for all memory points from 0 and $MaxAvailMem$. This procedure has been described in Section 4.

B.3 Detailed Algorithm

Figure 7 shows the Memory Cognizant Volcano Search Algorithm $FindBestPlan$.

The function $FindBestPlan$ returns:

- $SUCCESS$: when the optimized plan $optPlan$ for the $(LogExp, PhysProp)$ pair to be optimized is s.t. $optPlan.CostFunction() \leq_{all} CostLimitFunction$.
- $FAILURE$: when the optimized plan $optPlan$ for $(LogExp, PhysProp)$ pair to be optimized is s.t. $optPlan.CostFunction() >_{all} CostLimitFunction$.
- $PARTIAL\_SUCCESS$: when the optimized plan $optPlan$ for the $(LogExp, PhysProp)$ pair to be optimized is s.t. $\exists m, 1 \leq i \leq MaxAvailMem$: $optPlan.CostFunction()(m) \leq CostLimitFunction(m)$.

To optimize a given $(LogExp, PhysProp)$ pair within a given $costLimitFunction$, if there has been no previous attempt to optimize this $(LogExp, PhysProp)$ pair, the search algorithm proceeds as follows: It first applies transformation on the given logical expression to generate all equivalent logical expressions. Figure 8 shows pseudo code for application of transformations. Then it recursively optimizes transformed $(LogExp, PhysProp)$ pairs by applying each applicable operator (algorithm or enforcer) with the specified cost limit.

If there has been a previous attempt to optimize this $(LogExp, PhysProp)$ pair then we have a plan and cost function available. The $(LogExp, PhysProp)$ pair may have successful plan in some memory ranges and failures w.r.t the previous cost limit in some other memory ranges.

If the plan returned by the previous attempt has at least one point with failure indication and cost less than the cost limit then we need to reoptimize the the $(LogExp, PhysProp)$ pair.

Else, if for each memory point we have a successful plan and the cost limit is more than or equal to the cost of the plan at each point we return $SUCCESS$ along with the plan. Else, if the plan has cost less than or equal to the cost limit at some point we return $PARTIAL\_FAILURE$ along

```
Procedure FindBestPlan(LogExp, PhysProp, Limit)
    if the pair LogExpr and PhysProp is in the lookup table
        if the cost in the lookup table < Limit
            return Plan and Cost
        else return failure

    else /* Optimization required */
        create the set of possible "moves" from
          applicable transformations,
          algorithms that give the required PhysProp and
          enforcers for the required PhysProp

        for each move in the move set
            if the move uses a transformation
                apply the transformation creating NewLogExpr
                call FindBestPlan(NewLogExpr, PhysProp, Limit)
            else if the move uses an algorithm
                Limit = Limit - cost of the algorithm
                for each input I of the algorithm while Limit >= 0
                    determine required physical properties PP for I
                    Cost = FindBestPlan(I, PP, Limit)
                    Limit = Limit - Cost
            else    /* Move uses an enforcer */
                Limit = Limit - cost of enforcer
                modify PhysProp for enforced property
                call FindBestPlan for LogExpr with modified PhysProp

        /* Maintain the lookup table of explored facts */
        if LogExpr is not in the lookup table
            insert LogExpr into the lookup table
        insert PhysProp and best plan found into lookup table
```

Figure 6: Volcano Search Algorithm

with the plan. Else the plan has, at each memory point, cost greater than the cost limit and hence we return $FAILURE$.

Figure 9 shows application of an algorithm. Application of an enforcer is similar and can be found in [4]. Application of an operator is done as follows: First we evaluate cost function of the operator. We need to optimize its children and for this we need to evaluate child cost limit. As of now, we do not know exactly how much memory the tree rooted at this operator is going to take. For calculating the child cost limit, we assume that the tree will execute in memory $MaxAvailMem$ units.

If an edge between the operator and the child is pipelined and the operator takes $MaxAvailMem - i$ units of memory, the child will take i units of memory. Thus the child cost limit for memory i, with pipelined edge, is calculated by subtracting the cost of the operator at memory point $MaxAvailMem - i$ from the cost limit of the plan at memory point $MaxAvailMem$.

If an edge between the operator and the child is blocked the child cost limit is calculated by subtracting the cost of the operator running in memory $MaxAvailMem$ and materialization cost at the edge.

The cost limit passed to the child is the maximum of the two child cost limits described above at each memory point. After optimizing each child we merge the cost of the child with that of the plan cost and this plan cost is used as the operator cost to calculate the cost limit of the next child to be optimized.

If the child optimization returns $FAILURE$ (i.e. within the given cost limit, the optimizer could not find a plan even for a single memory point) then for the given $(LogExp, PhysProp)$ pair there exists no plan within the given cost limit for any memory points, and the optimizer returns $FAILURE$. Instead, if the child optimization returns a plan for even a single memory point with the given cost limit, the optimization continues.

Finally when the optimization is over, if the optimizer could find plans within the given cost limit for all memory points, it returns $SUCCESS$. If it could find plans within the given cost limit for some memory points but not for all of them, then it returns $PARTIAL\_SUCCESS$. The cost function will have multiple segments. If for a memory range we get a successful plan then within that range the cost function will indicate success along with the plan. If no successful plan is found in a memory range failure will be indicated in that range along with the cost limit. If no plan is found within the given cost limit at any of the points in the given memory range then it returns $FAILURE$.

```
FindBestPlan(LogExp, PhysProp, CostLimitFunction)
    if the pair LogExpr and PhysProp is in the lookup table with Plan as the optimal plan
        /* optimized already, attempting reuse */
        if there exists a memory point i at which Plan is failed and its cost is < CostLimitFunction(i)
            goto Label X /* reoptimization required */
        else /* no reoptimization required */
            if for all memory points i, Plan is successful and its cost ≤ CostLimitFunction(i)
                return (SUCCESS, Plan)
            else if there exists a memory point i where Plan is successful and its cost ≤ CostLimitFunction(i)
                return (PARTIAL_SUCCESS, Plan)
            else return FAILURE /* for all memory points i, Plan cost > CostLimitFunction(i)

    else      /* Optimization required */
    Label X:
        (Result, Plan) = ApplyTransformations(LogExpr, PhysProp, CostLimitFunction)
        /* if plan returned, merge it with the planGroup optPlan */
        if Result ≠ FAILURE then
            optPlan = MinMerge(optPlan, Plan)
            CostLimitFunction = optPlan.CostFunction()

        (Result, Plan) = ApplyAlgorithms(LogExpr, PhysProp, CostLimitFunction)
        /* if plan returned, merge it with the planGroup optPlan */
        if Result ≠ FAILURE then
            optPlan = MinMerge(optPlan, Plan)
            CostLimitFunction = optPlan.CostFunction()

        (Result, Plan) = ApplyEnforcers(LogExpr, PhysProp, CostLimitFunction)
        /* if plan returned, merge it with the planGroup optPlan */
        if Result ≠ FAILURE then
            optPlan = MinMerge(optPlan, Plan)
            CostLimitFunction = optPlan.CostFunction()

        /* Maintain the lookup table of explored (expression, physical property) pairs */
        if LogExpr is not in the lookup table
            insert LogExpr into the lookup table
        insert (LogExp, PhysProp, optPlan) into lookup table
```

Figure 7: Memory Cognizant Volcano Search: $FindBestPlan$ **Algorithm**

```
ApplyTransformations(LogExp, PhysProp, CostLimitFunction)
    for each applicable transformation
        create NewLogExpr by applying the transformation
        (Result, Plan) =
            FindBestPlan(NewLogExpr, PhysProp, CostLimitFunction)
        /* if plan returned, merge it with the planGroup optPlan */
        if Result ≠ FAILURE then
            optPlan = Plan
            CostLimitFunction = Plan.CostFunction()

    if optPlan is successful for all memory points
        return (SUCCESS, optPlan)
    if optPlan is successful for no memory point
        return (FAILURE)
    return (PARTIAL_SUCCESS, optPlan)
```

Figure 8: Memory Cognizant Volcano Search: $ApplyTransformations$ **Algorithm**

```
ApplyAlgorithms(LogExp, PhysProp, CostLimitFunction)
    for each applicable Algorithm do
        if AlgorithmCostFunction >_all CostLimitFunction
            continue /* cost of the operator is more than the cost limit */

        AlgoPlan.CostFunction = AlgorithmCostFunction

        for each input I of the algorithm

            for memory i = 1 to MaxAvailMem
                ChildCostLimit(i) = CostLimitFunction(MaxAvailMem)−
                        Min(AlgorithmCostFunction(MaxAvailMem − i),
                            AlgorithmCostFunction(MaxAvailMem) + CostOfResultIO(I))

            if ∀i : ChildCostLimit(i) < 0
                break /* no plan within the given cost limit */

            determine required physical properties PP for I
            (Result, Plan) = FindBestPlan(I, PP, ChildCostLimit)

            if result = FAILURE
                break /* no plan within the given cost limit */

            for memory i = 1 to MaxAvailMem
                CostWithChildBlocked(i) = AlgoPlan.CostFunction(i)
                                +Plan.CostFunction(MaxAvailMem)
                                +CostOfResultIO(I)

            if an edge between the algorithm and the child I is pipeline edge
                /* divide memory optimally between the operator and its child */
                CostWithChildPipelined =
                        OptMerge(AlgoPlan.CostFunction, Plan.CostFunction)
                /* consider breaking the pipelined edge */
                AlgoPlan.CostFunction =
                    MinMerge(CostWithChildPipelined, CostWithChildBlocked)

            else /* blocking edge, full memory is available to the input */
                AlgoPlan.CostFunction = CostWithChildBlocked

        /* Merge the operator plan with the planGroup optPlan */
        optPlan = MinMerge(optPlan, AlgoPlan)
        CostLimitFunction = optPlan.CostFunction()

    if optPlan is successful for all memory points
        return (SUCCESS, optPlan)
    if optPlan is successful for no memory point
        return (FAILURE)
    return (PARTIAL_SUCCESS, optPlan)
```

Figure 9: Memory Cognizant Volcano Search: *ApplyAlgorithms* **Algorithm**