Pipelining in Multi-Query Optimization

Nilesh N. Dalvi
Univ. of Washington, Seattle [1]
nilesh@cs.washington.edu

and

Sumit K Sanghai
Univ. of Washington, Seattle [1]
sanghai@cs.washington.edu

and

Prasan Roy
Bell Laboratories, Murray Hill, NJ [1]
prasan@research.bell-labs.com

and

S. Sudarshan
Indian Institute of Technology, Bombay
sudarsha@cse.iitb.ac.in

Database systems frequently have to execute a set of related queries, which share several common subexpressions. Multi-query optimization exploits this, by finding evaluation plans that share common results. Current approaches to multi-query optimization assume that common subexpressions are materialized. Significant performance benefits can be had if common subexpressions are pipelined to their uses, without being materialized. However, plans with pipelining may not always be realizable with limited buffer space, as we show. We present a general model for schedules with pipelining, and present a necessary and sufficient condition for determining validity of a schedule under our model. We show that finding a valid schedule with minimum cost is *NP*-hard. We present a greedy heuristic for finding good schedules. Finally, we present a performance study that shows the benefit of our algorithms on batches of queries from the TPCD benchmark.

[1] Work performed while at I.I.T., Bombay.

1. INTRODUCTION

Database systems are facing an ever increasing demand for high performance. They are often required to execute a batch of queries, which may contain several common subexpressions. Traditionally, query optimizers like [7] optimize queries one at a time and do not identify any commonalities in queries, resulting in repeated computations. As observed in [13, 17] exploiting common results can lead to significant performance gains. This is known as multi-query optimization.

Existing techniques for multi-query optimization assume that all intermediate results are materialized [4, 14, 19]. They assume that if a common subexpression is to be shared, it will be materialized and read whenever it is required subsequently. Current multi-query optimization techniques do not try to exploit pipelining of results to all the users of the common subexpression. Using pipelining can result in significant savings, as illustrated by the following example.

Example 1.1. Consider 2 queries, $Q_1 : (A \bowtie B) \bowtie C$ and $Q_2 : (A \bowtie B) \bowtie D$. Suppose we evaluate the 2 queries separately. In this case we pay the price of recomputing $A \bowtie B$. If we materialize the result of $A \bowtie B$, although we do not have to recompute the result, we have to bear the additional cost of writing and reading the result of the shared expression. Thus, results would be shared only if the cost of recomputation is higher than the cost of materialization and reading. While materialization of results in memory would have a zero (or low) materialization and read cost, it would not be possible to accomodate all shared results because of the limited size of memory, and in particular results that are larger than memory cannot be shared.

On the other hand, if we pipeline the results of $A \bowtie B$ to both the queries, we do not have to recompute the result of $A \bowtie B$ and we also save the costs of materializing and reading the common expression. ■

However, if all the operators are pipelined, then the schedule may not be realizable. We will formalize this concept later by defining *valid* schedules. The following example illustrates why every schedule may not be realizable.

Example 1.2. Consider the query execution plan shown in Figure 1. We assume nodes A and B produce results sorted on the join attributes of A and B and both joins are implemented using merge joins. Now, suppose all the operators are pipelined and a pull model of execution (Section 4.1) is used. Also suppose $MJ1$ has not got any tuples from A due to low selectivity of the select predicate $\sigma_{A.x=v1}$. Then, it may not pull any tuple from B. However, since $MJ2$ is getting tuples from A, it will keep pulling tuples from B. Since $MJ1$ is not consuming the tuples from B, B can not evict any tuple from its output buffer, which will become full. Now $MJ2$ cannot consume any more A tuples, so the output buffer of A will also become full. Once both output buffers are full, execution will deadlock. Hence, this schedule may not be realizable. The same problems would also arise with a push model for pipelining. ■

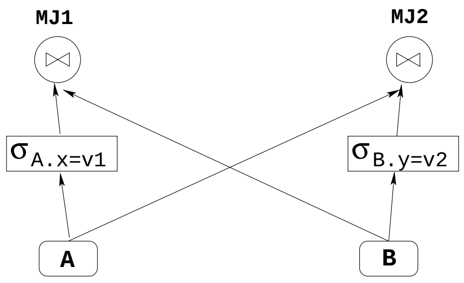

FIG. 1. Unrealizable Schedule

The main contributions of this paper are as follows.

• We present a general model for pipeline schedules, where multiple uses of a result can share a scan on the result of a subexpression; if all uses of an intermediate result can share a single scan, the result need not be materialized.

• We then present an easy-to-check necessary and sufficient condition for statically determining validity (realizability) of a schedule under our model.

• We show that given a plan that includes sharing of subexpressions, finding a valid schedule with minimum cost is *NP*-hard.

• We then present algorithms for finding valid pipelined schedules with low execution costs, for a given plan.

• Our overall approach to the query optimization process is then as follows: run a multi-query optimizer, disregarding the issue of pipelining in the first phase, and run our pipelining algorithms in the second phase. We have implemented our algorithms, and present a performance study that illustrates the practical benefits of our techniques, on a workload of queries taken from the TPCD benchmark.

The rest of the paper is organized as follows. Section 2 covers related work. Section 3 gives an overview of the problem and our approach to solving it. Section 4 gives a model for pipelining in a DAG, as well as necessary and sufficient condition for validity of a pipelined schedule. Section 5 shows that the problem of finding the least cost pipeline schedule for a given DAG structured query plan is *NP*-hard. We give heuristics for finding good pipeline schedules in Section 6. In Section 7, we give a detailed performance study of our heuristics. Section 8 gives some extensions and direction for future work and Section 9 concludes the paper.

2. RELATED WORK

Early work on multi-query optimization includes [5, 8, 12, 16, 17] and [18]. One of the earliest results in this area is by Hall [8], who uses a two-phase approach: a normal query optimizer is used to get an initial plan, common subexpressions in the plan are detected, and an iterative greedy heuristic is used to select which common subexpressions to materialize and share. At each iteration, the greedy heuristic selects the subexpression that, if materialized in addition to the subexpressions selected in the prior iterations, leads to the maximum decrease in the overall cost of the consolidated plan.

The work in [12, 17, 18] describes exhaustive search algorithms and heuristic search pruning techniques. However, these algorithms assume a simple model of queries having alternative plans, each with a set of tasks; the set of all plans of a query is extremely large, and explicitly enumerating and searching across this space makes the algorithms impractical.

More recently [14, 20] and [22] considered how to perform multi-query optimization by selecting subexpressions for transient materialization. [20] concentrates on finding expressions to share for a given plan. For the special case of OLAP queries (aggregation on a join of fact table with dimension tables) Zhao et al. [22] consider multiquery optimization to share scans and subexpressions. They do not consider materialization of shared results, which is required to handle the more general class of SQL queries, which we consider.

Roy et al. [14] study several approaches to multi-query optimization, and show that to get the best benefit, the choice of query plans must be integrated with the choice of what subexpressions are to be materialized and shared. The "post-pass" approach of [8] and [20] are not as effective since they miss several opportunities for sharing results. Roy et al. [14] also present implementation optimizations for a greedy heuristic, and showed that, even without the use of pipelining to share intermediate results, multiquery optimization using the greedy heuristic is practical and can give significant performance benefits at acceptable cost.

None of the papers listed above addressed the issue of pipelining of results, and the resultant problem of validity of schedules.

Chekuri et al. [1] and Hong [9] concentrated on finding pipeline schedules for query plans which are *trees*. These algorithms try to find parallel schedules for query plans and do not consider common subexpressions. Note that these algorithms cannot be used in the context of multi-query optimization, where the plans are DAGs.

Tan and Lu [21] try to exploit common subexpressions along with pipelining, but their technique applies only to a very specific query processing mechanism: join trees, broken into right deep segments where all the relations used in a segment fit in memory. Pipelined evaluation is used for each right deep segment. Their optimizations lie in how to schedule different segments so that relations loaded in memory for processing other segments can be reused, reducing overall cost. Database relations and shared intermediate results are assumed to fit in memory, which avoids the problems of realizability which we deal with, but the assumption is unlikely to hold for large databases. Further, they do not address general purpose pipeline plans for joins, or any operations other than joins.

Graefe [6] describes a problem of deadlocks in parallel sorting, where multiple producers working on partitions of a relation pipeline sorted results to multiple consumers; the consumers merge the results in their input streams. This problem is a special case of our problem: we can create a plan to model parallel sorting, and apply our techniques to detect if a pipeline schedule for the plan is valid.

Several database systems have long implemented shared scans on database relations, which allows multiple queries to share the output of a scan. These systems include Teradata and the RedBrick warehouse [2] (RedBrick is now a part of Informix, which is a part of IBM). Pipelining results of a common subexpression to multiple uses is a generalization of this idea.

Since intermediate results are not shared, the only problem of realizability in the context of shared scans of database relations arises when a database relation is used twice in the same query, or scans on more than one relation are shared by multiple queries. We are not aware of any work describing how this problem is handled in database systems . The techniques we describe in this paper can detect when scans can be shared without any problem of realizability, but to our knowledge our techniques have not been used earlier. A simple but restrictive solution for the special case of shared scans on database relations is as follows: allow at most one scan of a query to be a shared scan (This restriction prevents a query from sharing a scan even with itself.) Such a restriction may be natural in data warehouse settings with a star schema, where a query uses the fact table at most once, and only scans on the fact table are worth sharing; however it would be undesirable in a more general setting.

On the other hand, some systems, such as the RedBrick warehouse and the Teradata database, have an out-of-order delivery mechanism whereby a relational scan that is just started can use tuples being generated by an ongoing scan, and later fetch tuples already generated by the earlier scan. We do not consider such dynamic scheduling; our schedule is statically determined. (We discuss issues in dynamic materialization in Section 8.)

O'Gorman et al. [10, 11] describe a technique of scheduling queries such that queries that benefit from shared scans on database relations are scheduled at the same time, as a "team". Their technique works on query streams, but can equally well be applied to batches of queries. They perform tests on a commercial database system and show the benefits due to just scheduling, (without any other sharing of common subexpressions) can be very significant.

3. PROBLEM AND SOLUTION OVERVIEW

The main objective of this paper is to incorporate pipelining in multi-query optimization. We use a 2 phase optimization strategy. The first phase uses multi-query optimization to choose a plan for a given set of queries, ignoring pipelining optimizations, as done in [14]. The second phase, which we cover in this paper, addresses optimization of pipelining for a given plan. Single phase optimization, where the multi-query optimizer takes pipelining into account while choosing a plan, is very expensive, so we do not consider it here.

Multi-query optimizers generate query execution plans with common subexpressions used more than once, and thus nodes in the plan may have more than one parent. We therefore assume the input to our pipelining algorithms is a DAG structured query plan. We assume edges are directed from producers to consumers. Henceforth, we will refer to the plan as the *Plan-DAG*.

3.1. Annotation of Plan-DAG

Given a Plan-DAG, the first step is to identify the edges that are pipelinable, depending on the operator at each node. We say an edge is *pipelinable* if (a) the operator at the output of the edge can produce tuples as it consumes input from the edge, and (b) the operator reads its input only once. Otherwise the edge is materialized.

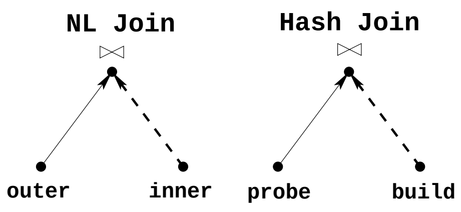

FIG. 2. Examples of Pipelinable Edges

The pipelinable edges for *nested loop join* and *hash join* operators are shown in Figure 2. Solid edges signify pipelining while dashed edges signify materialization.[3] Since the inner relation in *nested loop join* and the build relation in *hash join* have to be read more than once and we assume limited buffers, they have to be materialized. The inputs of *select* and *project* operators, without duplicate elimination, as well as both inputs of *merge join* are pipelinable. For *sort* the input is not pipelinable since the input has to be consumed completely before outputting any tuple. However, the merge sort operation can be split into run generation and merge phases, with the input pipelined to run generation, but the edges from run generation to merge being materialized.

Thus finally we will have a set of pipelinable and materialized edges. We use the word pipelinable instead of pipelined because all the edges marked so are only potentially pipelinable. It may not be possible for all of them to be simultaneously pipelined, as explained below.

3.2. Problems in Pipelining

A schedule in which the edges are labeled purely on the basis of the algorithm used at that node may not be realizable using limited buffer space. Our basic assumption is that any result pipelined to more than one place has to be pipelined at the same rate to all uses. This is because of the limited buffer size. Any difference in the rates of pipelining will lead to accumulation in the buffer and either the buffer will eventually overflow, or the result would have to be materialized. We assume intermediate results will not fit in memory[4].

The following two examples illustrate schedules that may not be realizable with limited buffer space.

- Consider the first schedule in Figure 3. The solid edges show pipelining and dashed edges show materialization. The output of u is being pipelined to both m and n. Also note that the output of m is pipelined to v at the same time but the output of n is being materialized. Now v cannot consume its input coming from m till it sees tuples from n. Thus it cannot consume the output of m. Thus, either the buffer between v and m will overflow or the result of m will need to be materialized. Thus, this schedule cannot be realized.

[3]We follow this convention throughout the paper.

[4]If some, but not all intermediate results fit in memory, we would have to choose which to keep in memory. This choice is addressed by Tan and Lu [21] in their context, but is a topic of future work in our context.

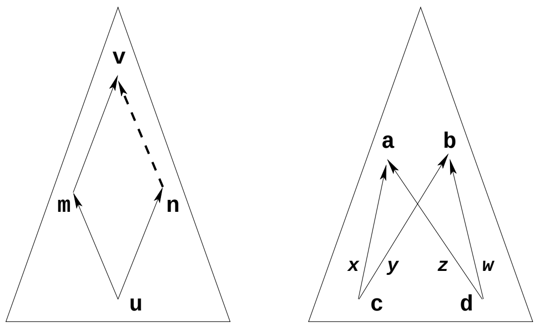

FIG. 3. Problems in Pipelining

• There is one more context in which problems can occur. Consider the second schedule in Figure 3. Suppose the operator at node a wants the rate of inputs in some ratio R_a. Similarly, the operator b wants input rates in ratio R_b. The rates of inputs in various edges are x,y,z and w as shown. However, as stated earlier, we require x to be same as y and z to be same as w. This forces R_a and R_b to be equal, which may not be always true. Moreover, the rates R_a and R_b may change dynamically depending on the data. Thus, there may be data instances that result in the buffers becoming full, preventing any further progress.

Thus, the above schedules cannot (always) be realized. We generalize these situations in Section 4.

3.3. Plan of Attack

To address the overall problem, in Section 4 we formally define the notion of "valid schedules", that is schedules that can be realized with limited buffer space, and provide easy-to-check necessary and sufficient conditions for validity of a given pipeline schedule. In Section 5 we define the problem of finding the least cost (valid) pipeline schedule for a given Plan-DAG, and show that it is NP-hard. In Section 6 we study heuristics for finding low cost pipeline schedules, and study their performance in Section 7. We then consider some extensions in Section 8.

4. PIPELINE SCHEDULES

We now define a model for formally describing valid pipeline schedules, that is, schedules that can be executed without materializing any edge marked as pipelined, and using limited buffer space. To do so, we first define a general execution model, and define the notion of bufferless schedules (a limiting case of schedules with limited buffer space). We then add conditions on materialized edges to the notion of bufferless schedules, to derive the notion of valid pipeline schedules. Later in the section we provide necessary and sufficient conditions for validity, which are easy to test.

Definition 4.1. *(Pipeline Schedule)* A *pipeline schedule* is a Plan-DAG with each edge labeled either pipelined or materialized. ■

4.1. Execution Model

Given a Plan-DAG, we exectute the plan in the following way. Each operator having at least one outgoing pipelined edge is assigned a part of the memory, called its output buffer, where it writes its output. If there is a materialized output edge from the operator, it writes it to disk as well. We say an operator o is in *ready* state if (i) each of the children of o that are connected to o by a materialized edge have completed exectution and have written the results to the disk. (ii) each of the pipelined children of o have, in their output buffer, the tuples required by o to produce the next tuple. (iii) the output buffer of o is not full.

If every pipelined parent of an operator has read a tuple from its output buffer, then the tuple is evicted from the buffer.

The execution of the plan is carried out as follows. Some operator that is in ready state is selected. It produces the next tuple and writes the tuple to its output buffer (if there is a pipelined outgoing edge) and to the disk (if there is a materialized outgoing edge). Then all of its pipelined children check if any tuple can be evicted from their output buffer.

An execution deadlocks if not all operators have finished and there is no operator in the ready state. A plan can complete if there is a sequence in which operators in ready state can be choosen so that each operator finishes execution.

As we will show later, if the schedule satisfies validity conditions that we define, the order of selection of ready nodes is not relevant; i.e., any order of selecting ready nodes will lead to the completion of the schedule.

The *pull model* is an instantiation of the general execution model. Under the pull model, operators are implemented as iterators [6]. Each operator supports the following functions: *open()*, which starts a scan on the result of the operator, *next()*, which fetches the next tuple in the result, and *close()*, which is called when the scan is complete. Consumers pull tuples from the producers whenever needed. Thus, each operator pulls tuples from its inputs. For instance, the *next()* operation on a select operation iterator would pull tuples from its input, until a tuple satisfying the selection condition is found, and return that tuple (or an end of data indicator if there are no more tuples). Operators such as nested loops join that scan an input more than once would close and re-open the scan. Some operators pass parameters to the *open()* call; for instance, the indexed nested loops join would specify a selection value as a parameter to the *open()* call on its indexed input.

If there are multiple roots in a Plan-DAG, the iterators for all the roots in the pull model start execution in parallel. For instance, in Example 1.1, the iterators for Q_1 and Q_2 would execute in parallel, both pulling tuples from $(A \bowtie B)$. A tuple can be evicted from the output buffer of $(A \bowtie B)$ only when it is consumed by both Q_1 and Q_2. If one of the queries is slower in pulling tuples, the other is forced to wait when the output buffer is full, and can proceed only when space is available in the output buffer. Thus the rates of the two queries get adjusted dynamically.

The *push model* can be similarly defined; in this case, each operator runs parallel with all others, generating results and pushing them to its consumers. It is even possible to combine push and pull within a schedule, where some results are pushed to their consumers, while others are pulled by the consumers.

4.2. Bufferless Schedules

Given a particular database, and a query plan, we can give sequence numbers to the tuples generated by each operator (including relation scan, at the lowest level). We assume that the order in which tuples are generated by a particular operation, is independent of the actual pipeline schedule used; this assumption is satisfied by all standard database operations.

Given a pipelined edge e, incoming to node n, the function $f(e, x)$ denotes the maximum sequence number amongst the tuples from edge e that the operator at node n needs to produce its x^{th} output tuple. The function $f(e, x)$ is independent of the actual pipeline schedule used.

We also define two functions whose value determines an actual execution of a pipelined schedule. We assume that time is broken into discrete units, and in each unit an operator may consume 0 or 1 tuple from each of its inputs, and may produce 0 or 1 output tuple. The function $P(e, t)$ denotes the sequence number of the last tuple that is pipelined through edge e at or before time t. Similarly $P(n, t)$ denotes the sequence number of the last tuple the operator at node n produces at or before time t. We also refer to the sequence number of the last tuple as the *max tuple.*

Definition 4.2. *(Bufferless Pipeline Schedule)* A pipeline schedule is said to be *bufferless*, if, given a function $f(e, x)$, defined for every pipelined edge e, there exists a function $P(e, t)$, non-decreasing w.r.t. t, such that for every node n, with outgoing edges $o_1, o_2, \cdots o_k$, and incoming edges $e_1, e_2, \cdots e_k$, the following conditions are satisfied.

(i) $P(o_1, t) = P(o_2, t) \cdots = P(o_k, t) = P(n, t)$

(ii) $P(e_i, t) = f(e_i, P(n, t)),\ \forall\ i.$

(iii) $\exists\ T$ such that $\forall\ n$, $\forall t \geq T$, $P(n, t) = CARD(n)$ where $CARD(n)$ is the size of the result produced by the operator at node n. ■

The first condition ensures that all the tuples generated at a node are passed immediately to each of its parents, thereby avoiding the need to store the tuples in a buffer. The second condition ensures that the tuple requirements of each operator is simultaneously satisfied. The third condition ensures that the execution gets completed.

4.3. Valid Pipeline Schedules

Definition 4.3. *(Valid Pipeline Schedule)* A pipeline schedule is said to be *valid* if it is bufferless and if each node n in the Plan-DAG can be given an integer *S(n)*, referred to as the *stage number*, satisfying the following property: If n is a node, with children $a_1, a_2, \cdots a_k$, and corresponding edges $e_1, e_2, \cdots e_k$ following conditions are satisfied:

(i) If e_i is labeled *materialized*, then $S(a_i) < S(n)$

(ii) If e_i is labeled *pipelined*, then $S(a_i) = S(n)$ ■

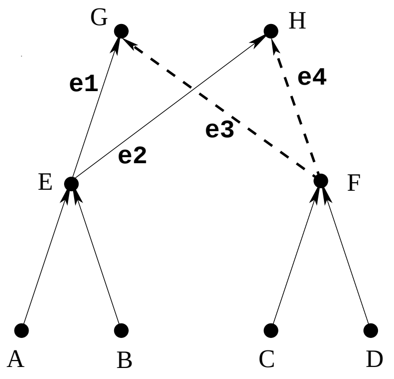

FIG. 4. The plan-DAG and the pipelining schedule for Example 4.1

The idea behind the stage number is that all the operators having the same stage number will be executed simultaneously. Also, all operators having stage number $i-1$ will get completed before execution of operators in stage i starts.

Note that the tuple requirements of the operators are dynamic and are not known a priori. But with limited buffers, the rates will get adjusted dynamically in an actual evaluation. Valid schedules will complete execution regardless of the order of selection of ready nodes; a detailed proof is given in Section 4.4. Invalid schedules, on the other hand, may deadlock with buffers getting full; execution can then proceed only if some tuples are materialized.

Example 4.1. Consider the Plan-DAG given in Figure 4. The dashed edges are materialized while the rest are pipelined. The pipeline schedule is valid, because we can have $S(C)$, $S(D)$ and $S(F)$ as 0, with the other stage numbers as 1, and functions can be assigned to all pipelined edges such that the conditions for the schedule to be bufferless are satisfied. At stage number 0, we would have computed C, D and F. At stage number 1, we would compute the results of the remaining nodes. Also note that the constraints on e_1 and e_3, placed by the operator at G, can be satisfied by reading the results of F and passing to G at the required rate. The rates of consumption of E at G and H would get adjusted dynamically: if the output buffer of E fills up, the faster of G or H will wait for the other to catch up. The case with e_2 and e_4 is similar. ■

4.4. Validity Criterion

As we have seen earlier, not all potentially pipelinable edges of the Plan-DAG can be simultaneously pipelined. We now give a necessary and sufficient condition for a schedule to be valid. But before that, we need to define some terminology.

Definition 4.4. *(C-cycle)* A set of edges in the Plan-DAG is said to form a *C-cycle*, if the edges in this set, when the Plan-DAG is considered as undirected, form a simple cycle. ■

DEFINITION 4.5. *(Opposite edges)* Two edges in a C-cycle are said to be *opposite*, if these edges, when traversing along the C-cycle, are traversed in opposite directions. ■

In the previous example, the edges e_1, e_2, e_3 and e_4 form a C-cycle. In it, e_1 and e_2 are opposite, so are e_1 and e_3, e_3 and e_4, and e_2 and e_4.

DEFINITION 4.6. *(Constraint DAG)* The equivalence relation $\sim$ on the vertex set of Plan-DAG is defined as follows: $v_1 \sim v_2$ if there exists vertices $v_1 = a_1, a_2, \cdots a_n = v_2$ such that there is a pipelined edge between a_i and a_{i+1} for each $1 \leq i < n$.

Let $E_q = C_1, C_2 \ldots C_k$ be the set of equivalence classes of $\sim$. We define a directed graph, referred to as the *Constraint DAG*, on E_q by the following rule: draw an edge from C_i to C_j if there exists vertices v_i and v_j such that $v_i \in C_i$, $v_j \in C_j$ and there is a path from v_i to v_j. ■

In the proof of Theorem 4.1, we show that the graph defined above is a DAG.

The following theorem provides a necessary and sufficient condition for determining the validity of a pipeline schedule.

THEOREM 4.1. *Given a Plan-DAG, a pipeline schedule is valid iff every C-cycle satisfies the following condition:* there exist two edges in the C-cycle both of which are labeled materialized, and are opposite.

Proof. We will prove that the criterion is necessary and sufficient in two parts.

Part (I): First we prove that if a pipeline schedule is valid, then any C-cycle will have at least two materialized edges which are opposite. On the contrary, assume that there exists a C-cycle such that all materialized edges are in the same direction. We consider two cases:

Case (i): There is at least one materialized edge in the C-cycle.
Let the C-cycle be $a_1, a_2, \cdots a_n$. Since, no two opposite edges in this C-cycle are both materialized, when we traverse through this cycle, all materialized edges are traversed in the same direction. Across pipelined edges $a_i a_j$, $S(a_i)$ and $S(a_j)$ values remain same, while across materialized edges from a_i to a_j, S values strictly increase. Hence we have,

$$S(a_1) \leq S(a_2) \leq S(a_3) \cdots \leq S(a_n) \leq S(a_1) \tag{1}$$

Since we know that at least one of the edges is materialized, one of the inequalities in equation 1 becomes strict and we get $S(a_1) < S(a_1)$, leading to a contradiction.

Case (ii): Now suppose there is a C-cycle C with no materialized edges.
Suppose the cycle is $A_1, A_2 \cdots A_n$. Without loss of generality, we can assume that the edge between A_1 and A_2 is from A_1 to A_2. Let $A_{i_1}, A_{i_2}, \cdots A_{i_{2k}}$ be the vertices of this C-cycle such that between A_{i_k} and $A_{i_{k+1}}$ all edges have the same direction and that the direction of edges changes across these vertices, as shown in Figure 5.

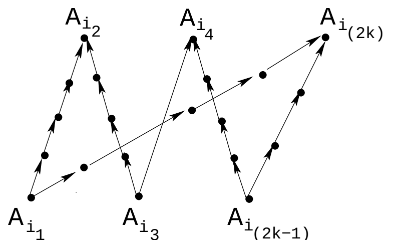

FIG. 5. C-cycle without materialized edges

Let f_j be the cascade of all functions $f(e, x)$ over all edges e in the path from $A_{i_{2j-1}}$ to $A_{i_{2j}}$, i.e., if $e_1, e_2 \ldots e_k$ are the edges in the path, then we have

$$f_j(x) = f(e_1, f(e_2, \cdots f(e_k, x)))$$

The function gives the max tuple the operator at node $A_{i_{2j}}$ needs from the operator at $A_{i_{2j-1}}$ to produce the x^{th} tuple. Similarly, let g_j be the cascade of all functions $f(e, x)$ over all edges e in the path from $A_{i_{2j-1}}$ to $A_{i_{2j-2}}$.

Then, we have the following set of equations

$$\begin{aligned}
f_1(P(A_{i_2}, t)) &= g_1(P(A_{i_{2k}}, t)) \\
f_2(P(A_{i_4}, t)) &= g_2(P(A_{i_2}, t)) \\
f_3(P(A_{i_6}, t)) &= g_3(P(A_{i_4}, t)) \\
&\cdots \\
f_k(P(A_{i_{2k}}, t)) &= g_k(P(A_{i_{2k-2}}, t))
\end{aligned}$$

Let $f^{-1}(e, x)$ denote the max tuple the operator at node n can produce given the xth tuple from the edge e, where edge e is an incoming edge into node n. Let g_j^{-1} be the cascade of the functions $f^{-1}(e, x)$ over all edges in the path from $A_{i_{2j-2}}$ to $A_{i_{2j-1}}$. It denotes the max tuple the operator at node $A_{i_{2j-2}}$ can produce given the tuple from $A_{i_{2j-1}}$. We see that $g_j^{-1} \circ g_j(P(A_{i_{2j-2}}, t)) = P(A_{i_{2j-2}}, t)$. This is because $P(x, t)$ is the max tuple that can be produced at time t by the operator at node x.

If we denote $g_j^{-1} \circ f_j$ by h_j, from the above equations we get $h_1 \circ h_2 \circ \cdots \circ h_k(P(A_{i_{2k}}, t)) = P(A_{i_{2k}}, t)$

We thus see that there is a constraint on these functions, and given an arbitrary set of functions $\{f_j\}$ and $\{g_j\}$, this constraint may not be satisfied. For instance, if we take $g_j(x) = x$ and $f_j(x) = 2x$ then we will get $P(A_{i_{2k}}, t) = 0$, which will violate the requirement that $P(n, t) = CARD(n)$ at some t. Hence the pipeline schedule is not bufferless, and hence not valid.

Thus, we have proved that if there is a valid pipeline schedule, then any C-cycle has at least two materialized edges which are opposite.

Part (II): Now, we prove that if any C-cycle has at least two materialized edges which are opposite then there exists a valid pipeline schedule.

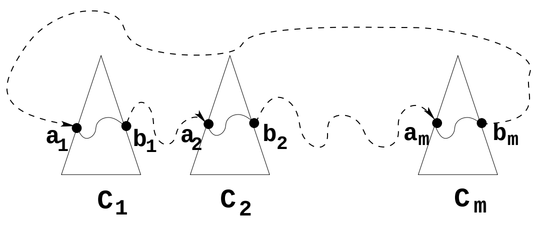

FIG. 6. The set of equivalence classes

Now let $E_q = C_1, C_2 \ldots C_k$ be the set of equivalence classes of $\sim$ defined in Definition 4.6. It can be shown that the subgraph induced by the vertices in C_i doesn't contain any materialized edge. On the contrary, assume that there is a materialized edge between two vertices. Since there exists a path between the 2 vertices consisting only of pipelined edges we see that there exists a C-cycle in the Plan-DAG which doesn't contain 2 materialized edges, which is a contradiction. Now, it is easy to see that none of the C_i contains any C-cycle. If there existed one, it would contain only pipelined edges which is not possible. Thus, each C_i is a tree.

Now, consider the graph on E_q as defined in Definition 4.6. We claim that it is a DAG. This is so because, if there is a cycle in this graph, say $C_1, C_2, \ldots, C_m, C_1$, then we will have vertices $a_1, b_1, a_2, b_2, \ldots a_m, b_m$ such that $a_i, b_i \in C_i$ and there will be paths (in directed sense) from b_i to a_{i+1} and b_m to a_1, because C_i is connected to C_{i+1} and C_m to C_1, and only these paths can have materialized edges. The graph is shown in Figure 6, where equivalence classes are represented as triangles. The solid lines represent that the path contains only pipelined edges where as dashed lines indicate the presence of materialized edges. Also there exist paths (in undirected sense) between a_i and b_i. Thus we will have a C-cycle from a_1 to b_1 to a_2 to $\ldots$ b_m and finally back to a_1 which contains materialized edges in only one direction. Hence there is a contradiction. Thus the graph is a DAG.

Now, let $C_{i_1}, C_{i_2}, \cdots C_{i_k}$ be a topological ordering on this DAG. Now for all vertices $v \in C_{i_j}$ we assign the stage label $S(v) = j$. To prove that the schedule is valid we have to show that for each C_i, given any set of function $\{f(e, x)\}$, each edge in C_i can be assigned valid function $P(e, t)$.

We construct the function P for each t serially. We will construct P in such a way that it will always satisfy the first two conditions needed for schedule to be bufferless. Also, we will make sure that at each stage, at least one operator is making progress, which will ensure that all the operators eventually complete execution. So suppose we have constructed $P(e, t)$ for each edge in C_i for $1 \leq t \leq T$. We then construct P for $t = T + 1$. We show that at least one operator can make progress, while the first two conditions are satisfied.

We say that an operator is *blocked* if it can neither consume nor produce any tuple. Further, an operator is said to be *blocked on its output* if it is able to produce a tuple but one or more of its parents are not able to consume it. The operator is *blocked on its input* if there is at least one child from which it needs to get a tuple but the child is itself blocked. Note that the first condition of Definition 4.2 ensures that if an operator produces a tuple it has to pass it to all the parents. So, even if one of the parents is not accepting tuples, the operator gets blocked on its output.

Also, if an operator does not get required tuples from its children in accordance with second condition of Definition 4.2, then operator gets blocked on its input.

Note that by definition, if an operator o_1 is blocked on its child o_2 then o_2 cannot be blocked on its output o_1. Let us associate the edge between o_1 and o_2 with o_1 if o_1 is blocked by o_2, or it is associated with o_2 if o_2 is blocked by o_1. Thus, every edge can be associated with atmost one blocked node. Also, every blocked node must have an edge associated with it. But since C_i is a tree, the number of nodes are greater than the number of edges. So, there must be at least one node which is not blocked. Hence, we can construct $P(e, T+1)$ so that the unblocked node progresses. Also, whenever an operator completes its execution, we can delete it from the tree and we get a set of smaller trees, on which we proceed similarly till every operator completes execution.

Thus each C_i is a bufferless pipeline schedule. Hence the whole schedule is a valid pipeline schedule. ■

Part (II) of the preceding proof leads directly to the following corrollary.

Corollary 4.1. *An execution of a valid schedule can be completed regardless of the order in which ready (unblocked) operators are chosen.* ■

Thus, if a schedule is valid, a pull execution, for example, will complete execution.

4.5. Testing for Validity

Now, we show that given a schedule we can test whether it is valid or not in polynomial time. First, we construct the equivalence classes $C_1, C_2, \cdots, C_m$ as described in the previous section. We then check that each of the subgraphs induced by the C_i is a tree, which is a necessary condition as shown in the proof of Theorem 4.1. Finally we construct the graph on these equivalence classes and check that it is a DAG, which is also a necessary condition as shown in the proof of Theorem 4.1. As shown in the same proof, if all the above conditions are satisfied then the schedule is valid, otherwise it isn't. All the above steps can be easily executed in polynomial time and hence we have the following theorem:

Theorem 4.2. *Validity of a pipeline schedule for a Plan-DAG can be checked in polynomial time.* ■

5. LEAST COST PIPELINE SCHEDULE

In the previous section we considered the problem of checking the validity of a pipeline schedule. Now, we come to the problem of finding the least cost pipeline schedule, given an input Plan-DAG. Before that we describe the cost model which forms the basis of the cost calculations.

5.1. Cost Formulation

The cost of a query execution plan can be broken up as the total of the execution costs of the operations in the schedule and the costs of reading data from and materializing (writing) results to disk. The execution costs of operations in a given

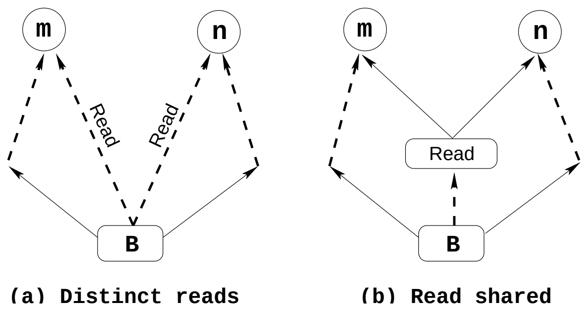

FIG. 7. Shared-read Optimization

schedule do not depend on which edges are pipelined, so we ignore them here; we only pay attention to the costs of materializing and reading data.

Given a pipeline schedule S, its materialization and reading cost $MC(S)$ is given by the following formula.

$$MC(S) = \sum_{n \in V(S)} (WC(n) + Matdeg(n) * RC(n))$$

where $V(S)$ is the set of all *materialized nodes* of S, i.e., all nodes having at least one outgoing materialized edge, $Matdeg(n)$ is the number of materialized edges coming out of n, and $WC(n), RC(n)$ are the read and the write costs of n.

Since each materialized node is written one time and read $Matdeg(n)$ times, we get the above expression for the cost.

5.2. Shared-read Optimization

The cost formulation assumes a read cost for every use of a materialized result along a materialized edge. Further, each scan of a database relation (i.e., a relation present in the database) has been assumed to pay a read cost. We can further reduce costs by optimizing the multiple reads of materialized nodes and database relations. The following example illustrates this point.

Example 5.1. Consider the query with a section of Plan-DAG given in Figure 7(a). Assume that the node B is either materialized or a database relation, and both the operators m and n have to read the node. The reading is shown by dashed lines. Now, we can execute the whole query by reading the node B just once, as shown in the Plan-DAG in Figure 7(b). ■

However, not all scans of a relation can be shared. For example, if the two nodes reading a relation are connected by a directed path containing a materialized edge, then they cannot share the read. This is because sharing a read will force both of them to be computed together, but the materialized edge in the directed path connecting them forces one to be completed before the other starts execution.

The criterion for checking the validity of a pipeline schedule can be used here for checking whether a set of reads of a materialized node can be shared. This can be done by transforming the Plan-DAG as shown in Figure 7. An extra node corresponding to a scan operator is added to the Plan-DAG, a materialized edge is added from the database relation/materialized node to the scan operator, and then pipelined edges are added from the scan node to each of the nodes sharing the read. The cost formula given earlier can be applied on this modified Plan-DAG, where sharing of reads is explicit.

5.3. NP-Completeness

In this section, we prove the *NP*-hardness of the problem of finding least cost schedules, as stated in Theorem 5.1. Clearly the corresponding decision problem belongs to the class NP, since by Theorem 4.2, the validity of a schedule can be checked in polynomial time.

THEOREM 5.1. *Given a Plan-DAG, the problem of finding the least cost set of materialized edges, such that in any C-cycle there exists two edges which are materialized and are opposite, is* NP*-hard.* ■

The proof of this theorem is given in the Appendix.

6. FINDING LEAST COST SCHEDULES

In this section, we present algorithms for finding the least cost pipeline schedule. We present an algorithm which performs an exhaustive search. We then describe a polynomial time greedy algorithm. Finally, we describe an extension for incorporating shared-read optimization. But before that, we describe a merge operation on the Plan-DAG, which is the basis for the algorithms.

6.1. Merge operation

Given a Plan-DAG, and two nodes n_1 and n_2 belonging to the Plan-DAG, we define $Merge(n_1, n_2)$ as follows: If there is no edge from n_1 to n_2, then $Merge$ is unsuccessful. If there is at least one edge, and after removing it, there is still a directed path from n_1 to n_2, again $Merge$ is unsuccessful. Otherwise, $Merge$ combines n_1 and n_2 into a single node. The $Merge$ operation on a Plan-DAG has some special properties, as described in the following theorem.

THEOREM 6.1. *If in any Plan-DAG, there is an edge e from n_1 to n_2, then the following hold:*

1.*Edge e can be pipelined in a valid schedule only if the operation $Merge(n_1, n_2)$ is successful.*

2.*A valid pipeline schedule of the Plan-DAG formed after merging, together with pipelining e, gives a valid pipeline schedule for the original Plan-DAG.*

3.*Any valid pipeline schedule for the original Plan-DAG can be achieved through a sequence of merge operations.*

Proof. (i) If $Merge$ is not successful, then there is a path P from n_1 to n_2, which together with e forms a C-cycle. In this C-cycle all edges in P are in one direction which is opposite to that of e. Since any pair of opposite edges in this C-cycle necessarily contains e, it must be materialized, and hence cannot be pipelined.

(ii) Now suppose this edge is merged, and consider any valid pipeline schedule in the new Plan-DAG. We have to show that this pipeline schedule, together with pipelined e, is valid. So consider any C-cycle K in the old Plan-DAG. If it does not contain e, it is also there in the new Plan-DAG, and hence must contain two materialized edges in opposite direction. If it contains e, then the C-cycle formed by collapsing e is present in the new Plan-DAG, and therefore contains two materialized edges which are opposite. Since they will still be opposite in K, the condition is satisfied, and hence the pipeline schedule is valid.

(iii) Given a valid pipeline schedule, collapse all the edges (by merging the required nodes) that are pipelined. If we are able to do so then we are through; otherwise, suppose we are not able to collapse some pipelined edge, e joining two nodes n_1 and n_2. This implies that there exists a path between these two vertices in the current Plan-DAG. Hence, a path must have been there between these two vertices in the original Plan-DAG, since a merge operation cannot induce a path between 2 disconnected components. The cycle containing e and the edges in this path then violate the validity condition. This contradicts the validity of pipeline schedule . Hence proved. ■

Example 6.1. Consider the Plan-DAG shown in Figure 4, and suppose all edges are initially labeled as materialized. We can first apply the merge step to each of edges AE, BE, CF, DF to get a graph with only nodes G, H, E (representing the merged EAB) and F (representing the merged FCD). We can then merge G with E. At this point we cannot merge FH since there would be another directed path with edges FE and FH. Similarly we cannot merge FG, but we can merge EH. This is exactly the pipeline schedule represented by Figure 4.

6.2. Exhaustive Algorithm

We saw that any valid pipeline schedule can be obtained from the Plan-DAG by a sequence of Merge operations. Therefore, we can get the optimal solution by considering all the possible sequences, and choosing the one with most benefit. Such a naive algorithm is however exponential in the number of edges. Note that we are working on a combined plan of a set of queries and hence the number of edges will depend on the number of queries, which may be quite large. Although the query optimization algorithms, such as System-R [15] and Volcano [7], also have an exponential cost for join order optimization, their time complexities are exponential in the size of a single query, which is generally assumed to be relatively small. But the exhaustive algorithm discussed above has a time complexity exponential in the sum of the sizes of all queries in a batch. For instance, a batch of 10 queries each with 5 relations would have an exponent value of 50, which is impractically large. Hence, we consider a lower cost greedy heuristic in the next section.

6.3. Greedy Merge Heuristic

Since the problem of finding the least cost pipeline schedule is *NP*-hard, we present a greedy heuristic, shown in Algorithm 1. In each iteration, the Greedy Merge heuristic chooses to *Merge* the edge that gives the maximum benefit.

Algorithm 1 Greedy Merge heuristic

```
$GreedyMerge(dag)$
begin
  $E \leftarrow$ set of all edges of $dag$
  $E_m \leftarrow \phi$
  for $e \in E$ do
    if $Merge(e)$ is possible then
      Add $e$ to set $E_m$
    end if
  end for
  if $E_m = \phi$ then
    return
  end if
  $e \leftarrow$ edge in $E_m$ with highest benefit
  output $e$ as pipelined
  $dag1 \leftarrow dag$ after Merge(e)
  call $GreedyMerge(dag1)$
end
```

We take the benefit of an edge to be its read cost, if it is materialized. This is done because if an edge is materialized it will incur a certain read cost, so we select the edge with the highest read cost to be pipelined because we will save the maximum read cost. Also, if the edge is the only edge originating from the node, (or all the remaining edges are already merged), then its benefit is taken to be the sum of read and write costs, because if such an edge becomes pipelined, we can save a read and a write cost.

At each iteration, the Greedy Merge heuristic calls Merge for each of the edges in the Plan-DAG. Each $Merge$ operation requires $O(m)$ time, and hence, each iteration takes $O(m^2)$ time, where m is the number of edges in the Plan-DAG.

Example 6.2. Consider again Example 6.1, using the Plan-DAG in Figure 4. Suppose the edges EG and EH had the highest benefit. Then these would be merged first, and would prevent the merging of FG and FH. However, if FG and FH had a higher benefit they would get merged first. The merging of AE, BE, CF and DF can be done successfuly since there are no paths that prevent the merging.

6.4. Shared-read Optimization

In Section 5.2, we discussed the shared read optimization to reduce the number of reads of materialized results. A specified sharing of reads can be represented by means of a scan operator and pipelining, as outlined in that section. However, we cannot represent the space of shared read alternatives in this fashion. We now consider how to choose which reads to share, in order to minimize cost.

We first consider, in Section 6.4.1, the case where the pipeline schedule has already been selected (say using the Greedy Merge heuristic), and the problem is

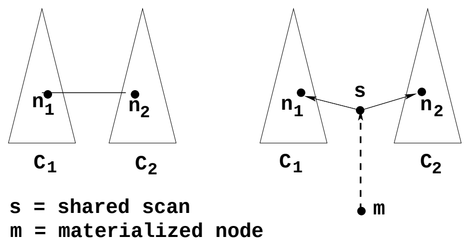

FIG. 8. Transformation of Plan-DAG for shared-read optimization

to choose which relations scans to share using the shared read optimization. Thus the shared read optimization runs as a post-pass to the Greedy Merge heuristic.

Sharing of scans by two operators (using the shared read optimization) has the same effect as pipelining results from one operator to another, in that the rates of execution of the two operators become interlinked. Indeed, the test applied by the Greedy Merge heuristic when deciding whether to pipeline an edge between two operators can be used unchanged when deciding whether to share a scan between two operators. We use this intuition, in Section 6.4.2, to show how to integrate the shared read optimization with the choice of edges to be pipelined. In our performance study (Section 7), we found that the integrated algorithm performs significantly better than the post-pass shared read optimization algorithm.

Before describing the algorithms, we note some necessary (but not sufficient) conditions for sharing reads:

1. Sharing of a read can occur only between nodes of different equivalence classes.
2. Two nodes belonging to different equivalence classes can share a scan only if they are not constrained to run at different stages due to materialization edges.
3. Two equivalence classes having the same stage number cannot share more than one read.

The proofs of these easily follow from the criterion for valid schedule given in Section 4.4 by applying the transformation described in Section 5.2.

6.4.1. Post-pass Shared Read Optimization

We now consider shared read optimization on a given pipeline schedule. We construct a graph with vertices as the set of equivalence classes. First, we add edges present in the *Constraint DAG*, defined in Section 4.4. These edges are all directed. Let this set of directed edges be denoted by E_d.

Next, for each pair of equivalence classes such that both read data via non-pipelined edges from some node, say n, we add an undirected edge between the two equivalence classes; the edge is labelled by the node name n. We set the weight of the edge to the read cost of node n. Note that there can be multiple edges between two classes, corresponding to different nodes. We call the edges as *sibling edges*, and call the set of sibling edges as E_u.

THEOREM 6.2. *Let S be any subset of E_u. Then, the reads denoted by the edges in S can be shared by the corresponding equivalence classes if and only if the subgraph formed by $S \cup E_d$ does not contain any cycle.*

Proof. A cycle in this graph corresponds to a C-cycle in the transformed Plan-DAG. This is because every undirected sibling edge in this graph will be replaced by two pipelined edges in the transformed Plan-DAG, as shown in Figure 8. Also the directed edges will appear as it is in the transformed Plan-DAG and will be in the same direction in the C-cycle. Thus if there is a cycle in this graph there will be a C-cycle in the Plan-DAG with all the materialized edges in same direction. Also if no cycle exists in this graph, then any C-cycle in the Plan-DAG will have materialized edges in opposite directions. The theorem then easily follows. ■

So, now the problem is to find the optimal set S, that is the set with largest total weight where no cycle is formed. The following greedy heuristic can be used for this problem.[5] The input is the initial Constraint DAG and the set of candidate sibling edges E_u.

1. Set the initial graph to the Constraint DAG
2. Sort all the sibling edges in E_u in decreasing order of weight
3. Step through the edges in decreasing order, and add an edge to S and to the graph, provided it does not result in a cycle in the graph.

Note that the graph is directed, so the addition of an sibling edge is actually implemented by adding two directed edges in opposite directions.

4. Return S

We call the above heuristic as the *Postpass Greedy Shared Read* heuristic.

The heuristic is modeled after Kruskal's minimum spanning tree algorithm [3], but unlike Kruskal's algorithm, it is not guaranteed to give the optimal set S. For instance, given a graph with directed edges $A \rightarrow B$ and $C \rightarrow D$, and the set E_u consisting of $D - B$ with weight 3, $A - C$ with weight 3 and $B - C$ with weight 5. Then the above heuristic would add only edge $B - C$ to S, giving a total weight of 5, whereas the optimal set is $\{A - C, D - B\}$ with total weight 6.

6.4.2. Integrated Shared Read Selection

One problem with performing the shared read optimization after the choice of pipelined edges is that it is possible for edges with a small benefit to get pipelined, and as a result prevent sharing of reads on a large database relation that could have provided a much larger benefit. Thus, although we get the best sharing of reads for the given pipeline schedule, a different pipeline schedule could have resulted in a much lower cost with the shared read optimization.

In this section we describe how to integrate the choice of pipelining and shared reads into a single heuristic, instead of splitting it into two phases.

A simple heuristic, which we call the *Greedy-Integrated-Naive*, for integrating the choices is as follows:

1. Modify the Plan-DAG as follows: for each database relation with more than one read, replace all the reads by an edge from a new single scan operation node reading from the base relation; the scan operation thus passes the output to all the

[5]We conjecture that the problem is NP hard.

nodes that originally read the base relation, as shown in Figure 7. All the edges out of the scan operation are potentially pipelinable.

2. Run the Greedy Merge heuristic, which considers each of these edges for pipelining. As a result of greedy selection of edges for pipeling, reads of a large database relation would get selected for pipelining ahead of pipelining of small intermediate results. Unlike other operations that have to pay a materialization cost in case some outgoing edge is not pipelined, we set the materialization cost to 0 for this special scan operation, since the relation is already materialized on disk.

Note that the above heuristic does not allow situations such as the following: uses A and B share a read, and independently uses C and D share a read. The heuristic only permits some set of uses to share a single read, and all the other uses are forced to read the relation independently. Shared reads of intermediate materialized results are also not considered, but can be handled by running the Greedy Shared Read algorithm as a post-pass to the above algorithm.

A better option for selecting pipelined edges and shared reads in an integrated manner, which we call *Greedy-Integrated*, is as follows.

1. Introduce undirected edges in the Plan-DAG for every pair of nodes reading from the same database relation. As before, we refer to these edges as sibling edges, and the edges are labelled by the relation name. Note that this is done only for database relations, and *before* running the Greedy Merge heuristic.

The sibling edges above correspond to the sigling edges introduced between equivalence classes in the Constraint DAG (Section 6.4.1); the difference is that they are introduced in the Plan DAG, before pipelining decisions are taken.

2. Execute the Greedy Merge heuristic. The heuristic can choose to merge sibling edges in addition to pipelineable edges, based on their benefit. The test for whether Merge can be applied to an edge remains the same as before, with sibling edges being treated in the same way as pipelined edges. The benefit of pipeling a sibling edge is the read cost of the relation whose read is being shared.

Unlike the Greedy-Integrated-Naive algorithm, this algorithm allows the creation of multiple different shared reads on the same relation, and is therefore, superior. As in the case of the Greedy-Integrated-Naive heuristic, reads of intermediate materialized results are also not considered, but can be handled by running the Greedy Shared Read algorithm as a post-pass to the above algorithm. The correctness of this algorithm follows from Theorem 6.3.

THEOREM 6.3. *The Greedy-Integrated algorithm produces a valid pipeline schedule.*

Proof. Consider the final schedule produced by the algorithm, after applying the shared-read transformations. (i.e. for every set of nodes sharing a read on a base relation, we create a new node scanning the base relation and pipelining it to all the nodes in that set). We have to show that this pipeline schedule does not contain any C-cycle with all materialized edges in the same direction. Assume, on the contrary, that it does contain a such a C-cycle. We know that pipelined edges in the pipeline schedule are those edges which were merged (the two pipelined edges coming out from a shared scan node correspond to the merging of a single sibling node). Now, since this C-cycle have all materialized edges in same direction, it will correspond to a cycle in the merged graph. However, merge operations cannot result in the creation of a cycle, which leads to a contradiction. Thus, the resulting pipeline schedule is valid. ■

6.5. Generating a Good Initial Plan-DAG

Our overall algorithm is a 2-phase algorithm, with the first phase using any multi-query optimizer, and our heuristics for pipelining and shared read optimization forming the second phase. However, the best plan of the first phase may not result in the best Plan-DAG with pipelining. As a heuristic we consider the following two approaches for generating the initial Plan-DAG.

- **Pessimistic Approach:** In the pessimistic approach, the optimizer in the first phase assumes all materialized expressions will incur a *write cost* once, and a *read cost* whenever they are read.
- **Optimistic Approach:** In the optimistic approach the optimizer in the first phase is modified to assume that all the materialized expressions will get pipelined in the second phase and will not incur any materialization (read or write) cost.

The optimistic approach can give plans with schedules that are not realizable, but our pipelining technique is used to get realizable schedules. The resultant schedules may be better than pessimistic in some cases, but can potentially be worse than even not using multi-query optimization. Therefore it makes sense to run both optimistic and pessimistic, find the minimum cost realizable schedule in each case, and choose the cheaper one.

6.6. Optimization Alternatives

Multiquery optimization, pipelining and shared read optimization are three ways of optimizing a query, and it is possible to use different combinations of these. To study the benefits of these techniques, we consider the following alternatives.

1. **MQO without pipelining**: Multi-query optimization using the greedy MQO heuristic of [14], without pipelining and shared-read optimizations; however, each shared result is assumed to be pipelined to one of its uses. This alternative acts as a base case,
2. **GREEDY-BASIC**: This is the basic Greedy Merge heuristic without shared read optimization, applied on the result of MQO.
3. **SHARED-READ**: The greedy shared-read optimization technique is applied directly on the Plan-DAG, without applying the Greedy Merge heuristic; no pipelining is used. This can be applied on the results of pessimistic and optimistic multi-query optimization. We refer to this as MQO-SHARED-READ. The shared-read technique can even be applied to the result of plain query optimization, without multiquery optimization. We refer to this as NO-MQO+SHARED-READ.
4. **GREEDY-POSTPASS**: The Greedy Merge heuristic is used to get a pipeline schedule, and the post-pass greedy shared-read technique is then applied to the pipeline schedule.

5. **GREEDY-INTEGRATED**: This is the Greedy-Integrated heuristic described in Section 6.4.2, which integrates the selection of pipelined edges and shared reads.

Since MQO can be performed using either the pessimistic approach or the optimistic approach, each of the above alternatives actually has two versions, one with the pessimistic approach and one with the optimistic approach.

7. PERFORMANCE STUDY

We now present the results of a preliminary performance study of our algorithms. The algorithms described in the previous section were implemented by extending and modifying the existing Volcano-based multi-query optimizer described in [14].

Through the experiments, we analyze the performance of the different algorithm variants described in Section 6.6. We applied the alternatives on the Plan-DAGs generated by the pessimistic and the optimistic approaches.

7.1. Experimental Setup

For experimental purposes, we use the multi-query optimizer algorithm described in [14]. In all the experiments conducted, the time taken by the 2nd phase is only a few milliseconds and is negligible as compared to the 1st phase. So we do not report execution time details.

We present cost estimates instead of actual run times, since we currently do not have an evaluation engine where we can control pipelining. All the cost estimate calculations were with respect to the cost model described in Section 5.1 for materialization costs, in conjunction with the cost model from [14]. The cost model is fairly accurate as shown in [14].

We use the TPCD database at scale factor 0.1 (i.e., 0.1 GB total data). The block size was taken to be 4KB and the cost functions assume that 6MB is available to each operator during its execution. Standard techniques were used for estimating costs, using statistics about the base relations. The cost estimates contain an I/O component and a CPU cost, with seek time as 10 m-sec, transfer time of 2 m-sec/block for read (corresponding to a transfer rate of 2 MB/sec), 4 m-sec/block for write, and CPU cost of 0.2 m-sec/block of data processed. The materialization cost is the cost of writing out the result sequentially. We assume the system has a single disk.

We ran two sets of experiments. In the first set, we assumed that no indices are present. In the second set, we assumed clustered indices on the primary keys of all relations.

We generate the input plans for our algorithm using the optimistic and the pessimistic approach. In each case, we evaluate the performance of MQO without pipelining and shared-read optimizations, GREEDY-BASIC, GREEDY-POSTPASS, MQO-SHARED-READ and GREEDY-INTEGRATED.

Our workload consists of batched TPCD queries. It models a system where several TPCD queries are executed as a batch. The workload consists of subsequences of the queries Q_{10}, Q_3, Q_5, Q_7 and Q_9 from TPCD. (Some syntactic modifications were performed on the queries to make them acceptable to our optimizer, which handles only a subset of SQL.) These queries have common subexpressions be-

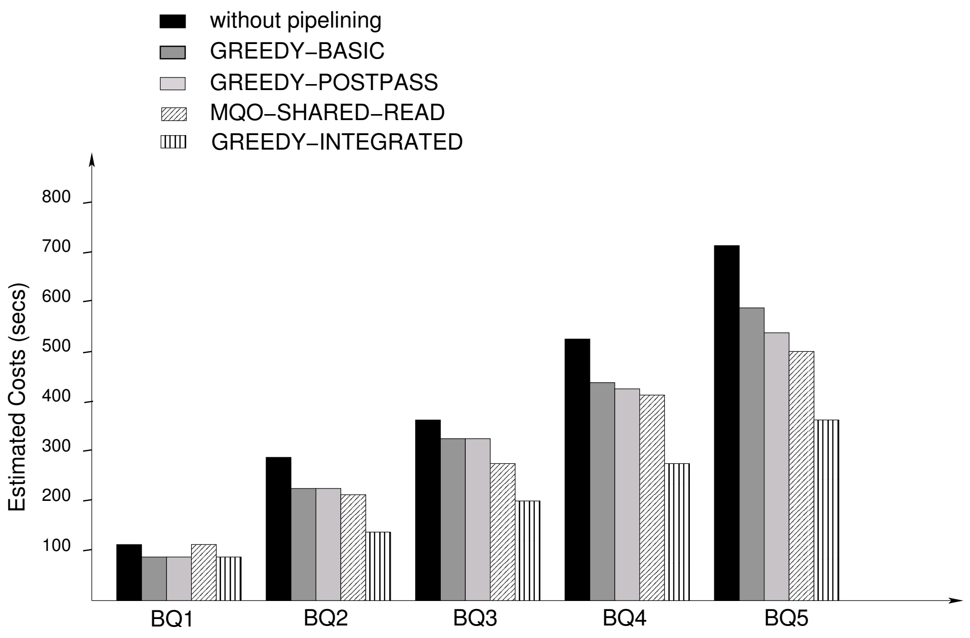

FIG. 9. Results on batched TPCD queries with Pessimistic approach (No Index)

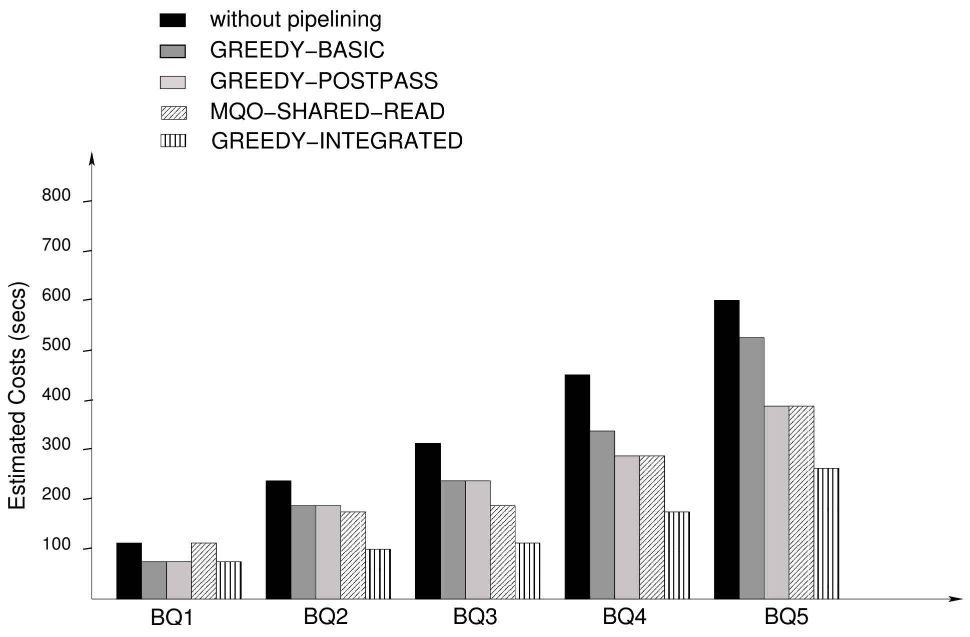

FIG. 10. Results on batched TPCD queries with Pessimistic approach (With Indices)

tween themselves. The batch query BQ_i contains the first i queries from the above sequence, together with a copy of each of them with different selection conditions.

7.2. MQO with Pessimistic Approach

The results for the batched TPCD workload (without indices) with pessimistic plans are shown in Figure 9. The corresponding results for the case with indices are shown in Figure 10. The figure shows five bars for each query. The first bar shows the cost of the query plan generated by the multi-query optimizer without pipelining (however, one use of each shared result is assumed to be pipelined). The other bars show the cost using GREEDY-BASIC, GREEDY-POSTPASS, MQO-SHARED-READ and GREEDY-INTEGRATED, in that order.

We see that in on query sets BQ1, BQ2 and BQ3, GREEDY-BASIC and GREEDY-POSTPASS perform roughly the same, meaning that there is no significant benefit of shared-read optimization when applied after running the Greedy Merge algorithm. The reason is as follows: in all the queries in these query sets, expressions that involving scans on the large relations (*LINEITEM* and *ORDERS*) were detected as common subexpressions are shared, resulting in these two relations being scanned only once in the resultant plan (with one exception: *LINEITEM* had two scans).

The *LINEITEM* relation was scanned twice, but the Greedy Merge algorithm, which was run before the shared read optimization, produced a plan with several internal edges pipelined, which prevented the sharing of of the reads on *LINEITEM* (sharing the read would have violated the conditions of Theorem 4.1). Thus, for these queries GREEDY-POSTPASS, which performs shared read optimization in a post-pass, performed no better than GREEDY-BASIC, which does not perform shared read optimization.

In many cases common subexpressions were created by subsumption derivations. For example, given two different selections $A = 7$ and $A = 10$ on the same relation, MQO may create an intermediate result with the selection $(A = 7) \vee (A = 10)$, and each of the original selections would be obtained by a selection on the intermediate result; as a result, only one scan needs to be performed on the database relation. Thus, the use of subsumption derivations with MQO provides an effect similar to shared reads.

The graphs show that MQO-SHARED-READ performs quite well. This can be explained by the fact that in almost all cases in our benchmark, the base relations are the nodes with largest cardinality in the whole Plan-DAG. Also, base relations cannot be pipelined, but they can have shared reads, which can produce large benefits. MQO-SHARED-READ exploits these benefits.

However, as the graph shows, GREEDY-INTEGRATED performs the best; this is because it exploits both shared reads on database relations, and common subexpressions, and chooses what reads to share and what results to pipeline in an integrated manner. For example, unlike GREEDY-POSTPASS, shared reads on *LINEITEM* were chosen preferentially to pipelining of some other smaller common subexpressions which would have prevented the shared reads, reducing the overall cost.

7.3. MQO with Optimistic Approach

The results for the batched TPCD workload with optimistic plans are shown in Figure 11 (without indices) and Figure 12 (with indices). The plot contains the same set of five values. However, notice that the bar labelled as "optimistic plan" is not necessarily a valid plan, since it assumes that all shared expressions are pipelined; this may not be possible since some of the subexpressions may have to be materialized, increasing the cost. Thus the cost of the optimistic plan serves as

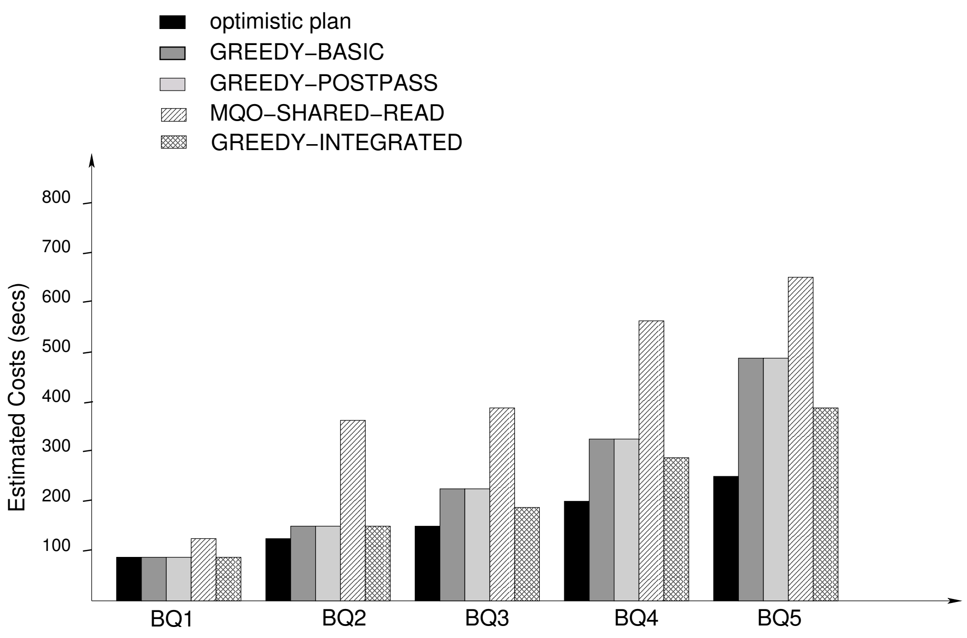

FIG. 11. Results on batched TPCD queries with Optimistic approach (No Index)

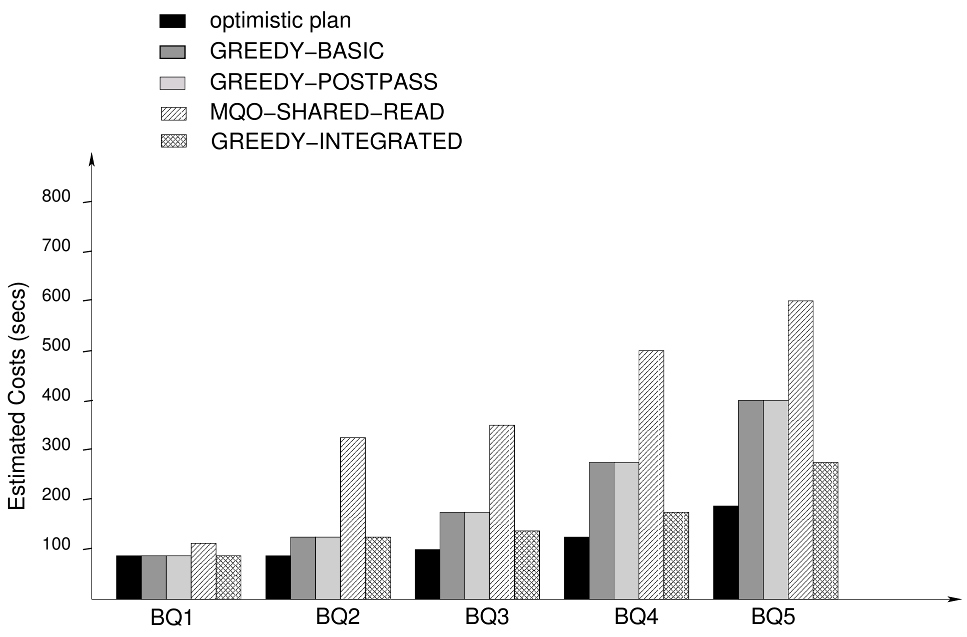

FIG. 12. Results on batched TPCD queries with Optimistic approach (With Indices)

an absolute lower bound for any pipelining algorithm (however, this lower bound does not take shared reads into account).

We can see that across all queries in the optimistic case, GREEDY-POSTPASS performed no better than GREEDY-BASIC. As explained in the comparison of the two for the pessimistic case in Section 7.2, this is partly due to the elimination of shared reads and partly because pipelining of some common subexpressions prevents the use of shared reads. The latter effect is more marked in the optimistic case, since more subexpressions are shared.

Similar to the case of pessimistic plans, GREEDY-INTEGRATED performs the best, since it makes the shared read and pipelining decisions in an integrated fashion.

MQO-SHARED-READ performs significantly worse than GREEDY-BASIC in the optimistic approach. Here, the first phase (plan generation using MQO) assumes that all shared subexpressions will be pipelined, resulting in significantly more subexpressions being shared. But with MQO-SHARED-READ, which does not attempt to do any pipelining of shared expressions, these subexpressions do not get pipelined at all; hence MQO-SHARED-READ performs poorly with the optimistic approach.

7.4. Overall Comparison

To find the overall best approach, we need to consider the best pipelining/shared-read technique for plans generated using the optimistic approach and the pessimistic approach, and compare them with plans without using multi-query optimization. For plans without multi-query optimization (NO-MQO), running our pipelining algorithm does not make sense, as in the absense of sharing, everything that can be pipelined is pipelined, which the cost model assumes anyway. However, these plans can share the reads of base relations, and hence the shared-read technique can be applied on them, shown in the bars labeled as NO-MQO+SHARED-READ. For pessimistic and optimistic plans, GREEDY-INTEGRATED is the best candidate. Thus, we compare GREEDY-INTEGRATED for the optimistic and pessimistic cases with NO-MQO and NO-MQO+SHARED-READ. Figures 13 and 14 show the comparision without indices, and with indices, respectively.

From the graphs we can see that for each query, one of the two variants of GREEDY-INTEGRATED gives the best performance. For queries BQ1 and BQ4, both give same results. For BQ2 and BQ5, pessimistic GREEDY-INTEGRATED is better while for BQ3, optimistic is better. So there is no clear winner and it may be a good idea to run the pipelining algorithm for both cases and take the best plan.

The graphs also show that just applying shared reads without MQO also performs very well. The main reason for this surprising effectiveness of NO-MQO+SHARED-READ is that the cost of the plans is dominated by the cost of reading relations from disk; the CPU component of the cost is relatively small in our cost model. MQO achieves an effect similar to shared scans by means of subsumption derivations, as mentioned in Section 7.2. Shared reads without MQO allow a single scan to be used for both selections, providing the same benefits as subsumption derivations (avoiding a second scan), but without the overheads of creating intermediate relations. GREEDY-INTEGRATED still performs better since there are other opportunities

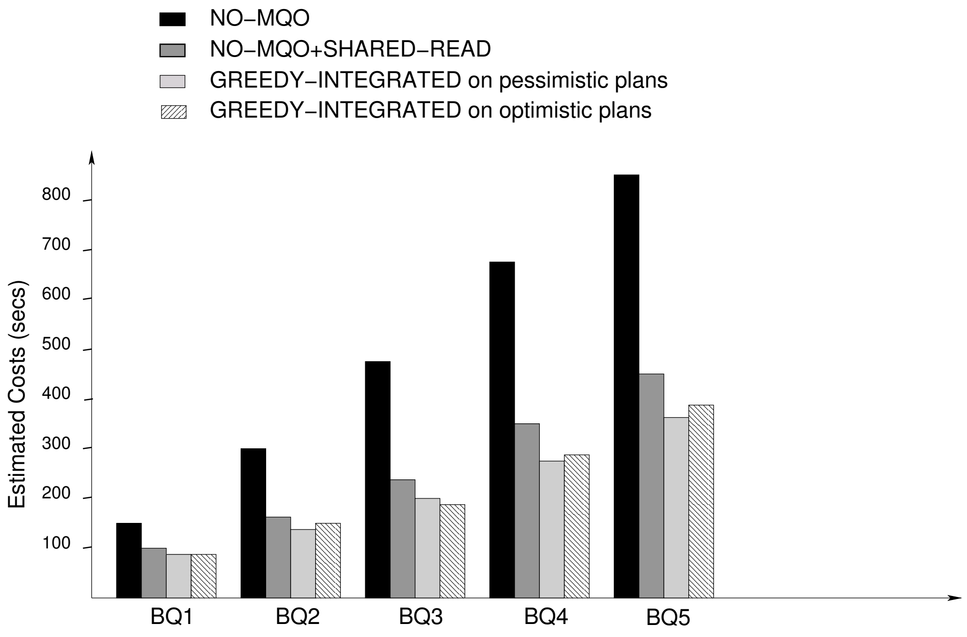

FIG. 13. Comparision of Different Techniques (Without Indices)

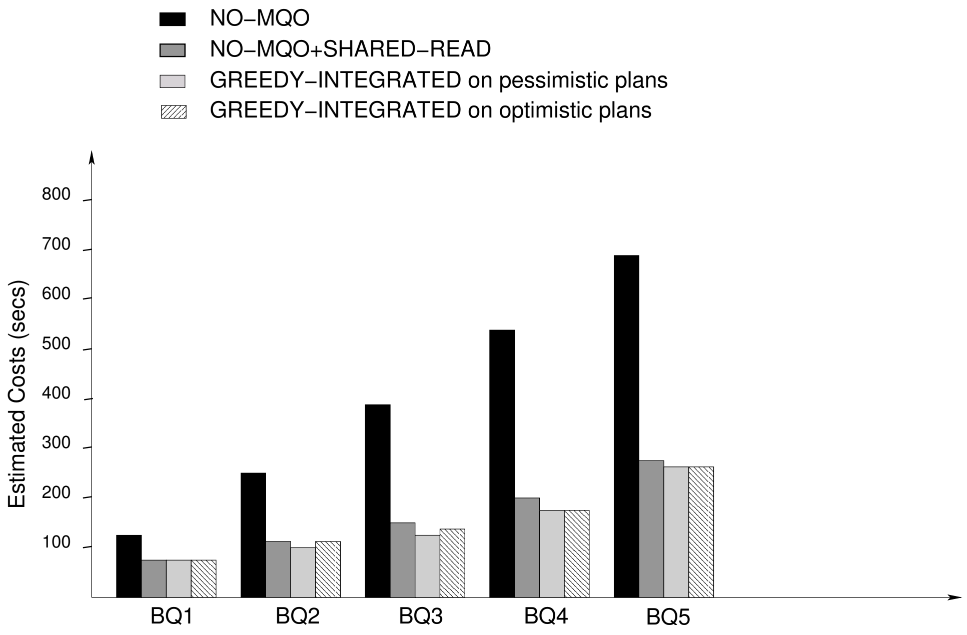

FIG. 14. Comparision of Different Techniques (With Indices)

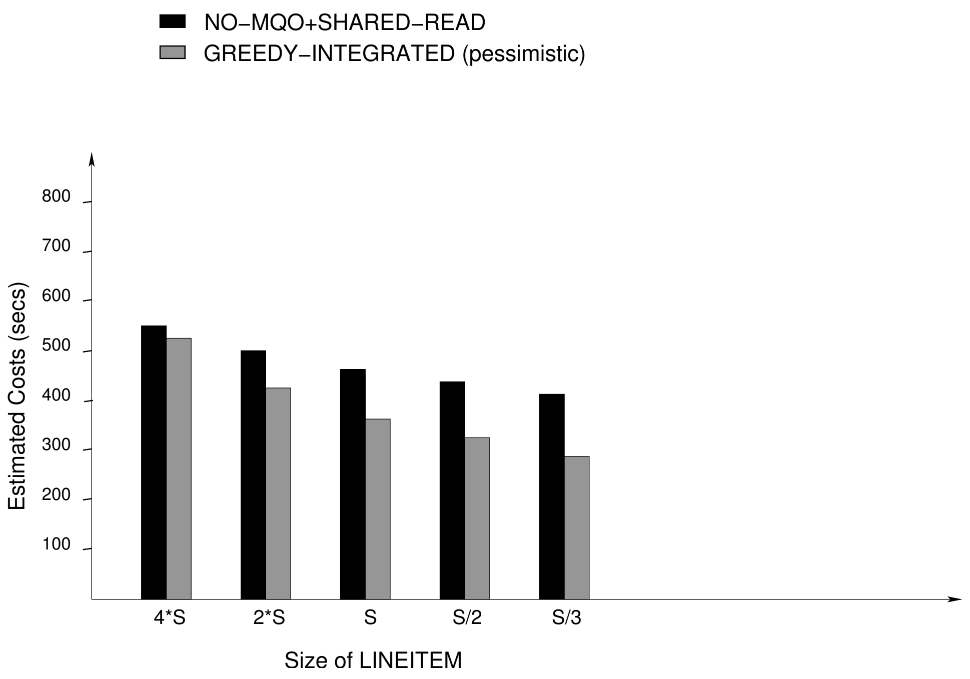

FIG. 15. Comparision of Techniques (Without Indices) with Varying Sizes of *LINEITEM*

for sharing common subexpressions. (MQO-SHARED-READ, not shown in these graphs but shown in earlier graphs does even worse since it does not allow the intermediate result to be pipelined to its uses.)

One noticeable difference between the graphs in Figures 13 and 14 is that the difference in performance between NO-MQO+SHARED-READ and GREEDY-INTEGRATED is considerably less in the presence of indices (Figure 14). The plans with indices actually use the fact that a clustered index ensures that the relation is sorted on the indexing attribute, and thereby perform merge-join without extra sorting. In the absence of indices, a (fairly expensive) sorting step is required, and MQO allows the sorted result to be shared, which just shared reads cannot achieve.

In the presence of indices, this sorting step is redundant. MQO is able to share several join results, which NO-MQO+SHARED-READ cannot exploit, but the benefits of sharing these join results is only the CPU cost, since the disk reads are effectively free since they are shared. Thus the gap between NO-MQO+SHARED-READ and GREEDY-INTEGRATED is greatly reduced in this case.

The *LINEITEM* relation is many times larger than the next largest database or intermediate relation, and significant benefits can be obtained by sharing scans on this relation. To check the effect of reducing the cost of reading data from disk, we ran some experiments with varying sizes of the *LINEITEM* relation. Figure 15 shows how the shared read technique and the greedy integrated (with pessimistic approach) technique compare with different sizes for the largest database relation, *LINEITEM.* The sizes shown are relative to the size (S) of *LINEITEM* as defined in the TPC-D benchmark. When the size of *LINEITEM* is increased substantially, shared reads provide very large gains, and little additional gain is to be had from

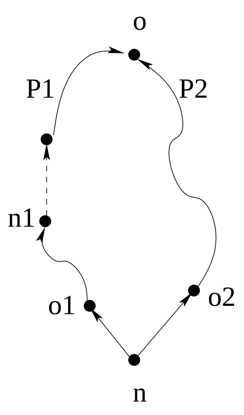

FIG. 16. Dynamic Materialization

MQO and pipelining. If the size of *LINEITEM* is reduced, this is no longer the case and GREEDY-INTEGRATED (MQO with pipelining and shared reads) gives good benefits over just shared reads.

In a recent work, O'Gorman et al. [10, 11] propose a technique of scheduling queries in a query stream; in their queries that can benefit from a shared scan are scheduled at the same time, as a "team". They perform tests on a commercial database system and show the benefits due to just scheduling, (without any other sharing of common subexpressions) can be very significant, and can be greater than the benefits reported for multi-query optimization without shared scans.

Our results are in line with their observations, and indicate that in cases where computation is highly I/O bound, such as in the TPC-D benchmark (without disk striping), shared reads play a major role in reducing query cost. Where computation is more CPU bound, multi-query optimization with pipelining is likely to have a more significant effect. In particular, data transfer rates from disks have been increasing rapidly, and disk striping using RAID organizations increase transfer rates even more, so the transfer (read) time per block is likely to be considerably less than 2 m-sec/block which we have assumed. The increase in transfer rate would have an effect similar to that of reducing the size of the *LINEITEM* table, shown in Figure 15.

In summary, MQO with pipelining, and even just shared reads without MQO, provide very significant benefits independently; using multiquery optimization combined with pipelining and the shared read optimization together gives benefits over using either independently, and the benefits can be quite significant.

8. EXTENSIONS AND FUTURE WORK

We now describe a dynamic materalization optimization that can be applied in a limited special case, and discuss issues in incorporating an out-of-order shared read optimization that is supported by some database systems.

8.1. Dynamic Materialization

It is possible to execute a schedule by dynamically materializing results when buffer overflow happens. For example, consider the plan in Figure 16.

Statically, we have to materialize a node each in paths P_1 and P_2. Dynamically, we could materialize tuples whenever a buffer overflow occurs, and no progress can be made. In other words, we use the disk to implement a buffer that will never overflow (assuming sufficient disk space is available). Dynamic materialization can provide significant benefits if the tuple requirements from the base operators by both the paths are nearly the same, in which case the plan can complete execution without the materializion of any tuples, or with materialization of a small number of tuples.

However, in a naive implementation of dynamic materialization, it is possible that overflows occur at nodes that have a high materialization cost, and this may result in a high price for materialization. Thus, there are cases where static materialization may be cheaper, and cases where some form of dynamic materialization may be cheaper.

Developing a technique that guarantees the best of both worlds under all situations is a topic of future work. However, we present a dynamic materialization technique that provides such a guarantee for a special case: when the plan is a single C-cycle of the form given in Figure 16. The intuition is that in such a case we do not need materialized edges in both the paths. We only need to have a materialized edge in the path in which the rate at which the tuples are read is slower. The execution starts with no materialized node. When the rate of tuples required by one of the operators, say o_1, becomes slower than the other, we select the node n_1 in P_1 which has the least materialization cost to be materialized. The operator o_1 will continue to be pipelined to n_1.

When the output buffer of the operator at n_1 is full, it outputs new tuples that it generates to disk. When these are required by the parent (s) of n_1, they are read back from disk.

Here, the operator does not wait for its child to be completely materialized before starting execution. This is because at that time, the rate at which the tuples are read is slower than the rate at which the tuples are written to the disk. Thus, the gap between the read and write pointers is increasing. So, the operator is guaranteed that whatever it wants to read is already present on the disk.

However, it may happen that after some time, the rate of tuples required along P_1 increases. Now the rate at which tuples are required by o_2 would be less than the rate of o_1. Then the gap between read and write heads will start reducing. If both the heads meet, then n_1 stops writing to the disk and we make its output pipelined. Also, we select a node n_2 along the other path with least materialization cost and materialize it. Thus, we shift the node to be materialized dynamically.

Suppose N_1 is the size of relation which n_1 produces and N_2 is the size of relation of n_2. In the static case, we necessarily materialize $N_1 + N_2$ tuples. In the dynamic case, we materialize part of n_1 and part of n_2. In the best case we may not have to materialize anything at all. Even in the worst case, we end up materializing only $max(N_1, N_2)$ tuples.

Implementing this optimization and studying its benefits are topics of future work.

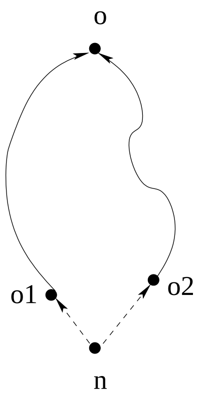

FIG. 17. Out-of-Order Read Optimization

8.2. Out-Of-Order Read Optimization

In this section, we consider another optimization which can be incorporated while executing the plan. As discussed in Section 2, the RedBrick warehouse supports out-of-order shared read execution, in which a relational scan that has just started can use tuples being generated by an earlier ongoing scan of the same relation, temporarily omitting tuples that have been generated already by the earlier scan, and after the earlier scan finishes, the later scan can fetch tuples that it missed. The same mechanism can also be applied to our case. For example, consider a portion of a plan shown in Figure 17.

We see that the node n is materialized (may also be a base relation). There are two operators o_1 and o_2 which are trying to read from this node. Note that we cannot apply the shared-read optimization here, as a C-cycle is formed which does not have any materialized edges on applying the transformation. Thus, normally o_1 and o_2 will not be able to share a read. Assume that the order in which the tuples are read from n by o_1 and o_2 is not important, for example the tuples in n are not required to be in a sorted order.

Now, we can apply the out-of-order read optimization. We read the tuples from n and pass it to both the operators o_1 and o_2. If the rates at which the tuples are required by o_1 and o_2 is same, we go on passing at the required rate. When the rate of tuples required by one of the operators, say o_1, becomes less than that of the other, the output buffer of n will start growing since it has to keep tuples in buffer for o_1 to read. All the tuples in the buffer will be those which o_2 has read but o_1 is yet to read. If the buffer becomes full, flush all the tuples from the buffer. Thus, some tuples will be lost for o_1. Keep track of the blocks which o_1 lost. Finally o_1 and o_2 can separately read the blocks which they did not read before. Implementing this optimization and studying its benefits are topics of future work.

9. CONCLUSIONS

In this paper, we studied the issue of pipelining and shared reads in DAG structured query plans generated by multi-query optimization. We began by motivating the need for pipelining and presented a model for a pipeline schedule in a Plan-DAG. We outlined key properties of pipelining in a Plan-DAG and showed *NP*-completeness of the problem of finding minimum cost pipeline schedules. We developed a greedy algorithm for scheduling the execution of a query DAG to reduce the cost of reading and writing the intermediate results to the disk. We extended the algorithm to handle the shared read optimization.

The implementation of our algorithm demonstrated that pipelining can be added to existing multi-query optimizers without any increase in time complexity. Our performance study, based on the TPCD benchmark, shows that pipelining and shared reads can individually provide significant performance gains, and can provide even better gains when used together.

There are several avenues for future work. We have assumed that executing multiple operations in a pipeline does not affect the time taken for each operation. Such operations run concurrently, and must compete for resources such as memory and disk I/O. The amount of memory allocated to an operator may affect its cost significantly, or if memory availability is very restricted, it may be impossible to execute certain sets of operations concurrently. Our algorithms need to be extended to handle such situations. Experiments with different benchmarks and with larger sets of queries need to be performed. Optimized queries need to be executed on on a query evaluation system that supports pipelining with multiple consumers, to measure real benefits.

ACKNOWLEDGEMENT

This work was partly supported by a Govt. of India, Department of Science and Technology grant. The work of Prasan Roy was supported by an IBM Research Fellowship.

We wish to thank A. A. Diwan, Dilys Thomas and Arvind Hulgeri for their constructive feedback on the paper, and Ashish Gupta for discussions during his Bachelor's thesis on the topic of this paper. We would also like to thank all the referees for their insightful reviews which helped us in improving the quality of the paper.

REFERENCES

1. Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. Scheduling problems in parallel query optimization. In *ACM Symp. on Principles of Database Systems*, pages 255–265, 1995.
2. Latha Colby, Richard L. Cole, Edward Haslam, Nasi Jazayeri, Galt Johnson, William J. McKenna, Lee Schumacher, and David Wilhite. Redbrick Vista: Aggregate computation and management. In *Intl. Conf. on Data Engineering*, 1998.
3. Cormen, Lieserson, and Rivest. *Introduction to Algorithms.* Prentice-Hall, 1990.
4. Ahmet Cosar, Ee-Peng Lim, and Jaideep Srivastava. Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. In *Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 433–438, 1993.
5. S. Finkelstein. Common expression analysis in database applications. In *SIGMOD Intl. Conf. on Management of Data*, pages 235–245, Orlando,FL, 1982.
6. Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
7. Goetz Graefe and William J. McKenna. Extensibility and Search Efficiency in the Volcano Optimizer Generator. In *Intl. Conf. on Data Engineering*, 1993.
8. P. A. V. Hall. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20(3), May 1976.
9. Wei Hong. Exploiting inter-operation parallelism in XPRS. In *SIGMOD Intl. Conf. on Management of Data*, pages 19–28, 1992.

10. Kevin O'Gorman. *On Tuning and Optimization for Multiple Queries in Databases.* PhD thesis, Univ. California, Santa Barbara, September 2002.

11. Kevin O'Gorman, Divyakant Agrawal, and Amr El Abbadi. Multiple query optimization by cache-aware middleware using query teamwork. In *Intl. Conf. on Data Engineering*, 2002. (poster paper).

12. Jooseok Park and Arie Segev. Using common sub-expressions to optimize multiple queries. In *Intl. Conf. on Data Engineering*, February 1988.

13. Arnon Rosenthal and Upen S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Intl. Conf. Very Large Databases*, pages 230–239, 1988.

14. Prasan Roy, S. Seshadri, S. Sudarshan, and S. Bhobhe. Efficient and extensible algorithms for multi-query optimization. In *SIGMOD Intl. Conf. on Management of Data*, 2000.

15. P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, pages 23–34, 1979.

16. T. Sellis and S. Ghosh. On the multi query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, pages 262–266, June 1990.

17. Timos K. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

18. Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data and Knowledge Engineering*, 12:197–222, 1994.

19. Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In *Intl. Conf. Very Large Databases*, pages 488–499, 1998.

20. Subbu N. Subramanian and Shivakumar Venkataraman. Cost based optimization of decision support queries using transient views. In *SIGMOD Intl. Conf. on Management of Data*, Seattle, WA, 1998.

21. K. Tan and H. Lu. Workload scheduling for multiple query processing. In *Information Processing Letters*, volume 55, pages 251–257, 1995.

22. Y. Zhao, Prasad Deshpande, Jefrrey F. Naughton, and Amit Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *SIGMOD Intl. Conf. on Management of Data*, Seattle, WA, 1998.

APPENDIX: PROOF OF THEOREM 5.1

THEOREM 5.1. Given a Plan-DAG, the problem of finding the least cost set of materialized edges, such that in any C-cycle there exists two edges which are materialized and are opposite, is *NP*-hard.

Proof. In order to prove the theorem we show in Theorem A.1 that a special case of the problem is *NP*-hard, in which all the edges have equal weight and all the edges are potentially pipelinable. ■

THEOREM A.1. *Given a Plan-DAG, the problem of finding the least number of edges to be materialized such that any C-cycle contains at least two materialized edges which are opposite, is* NP*-hard.*

Proof. Consider the equivalence relation $\sim$ on the vertex set defined in the Section 4.4. We have seen that it produces equivalence classes $C_1, C_2 \dots C_k$ such that each equivalence class, when considered as a subgraph, is a tree and contains no materialized edge. Also, between vertices of two distinct equivalence classes, there can only be materialized edges and that too, in the same direction.

Since each C_i is a tree, the number of edges in the subgraph formed by the vertices of C_i will be $|C_i| - 1$, where $|C_i|$ is the cardinality of C_i.

Now, let n be the number of vertices in the Plan-DAG and e be the number of edges, p be the number of pipelined edges and m be the number of materialized edges. Then,

$$\begin{aligned} n &= \sum_{1\leq i\leq k} |C_i| && \text{(A.1)} \\ p &= \sum_{1\leq i\leq k} (|C_i|-1) \\ &= n-k && \text{(A.2)} \end{aligned}$$

From the above equations, we get

$$m = e - p = e - n + k$$

Thus, to materialize minimum number of edges we have to minimize k, i.e., divide the Plan-DAG into a minimum number of equivalence classes. In other words, we have to divide the Plan-DAG into a minimum number of trees such that all the edges between any two trees are in the same direction and the graph defined by condensing each tree into a single node does not contain any cycle. If the Plan-DAG itself is a tree, then only one equivalence class is acceptable. Otherwise, at least two trees will be required. Lemma A.1 proves that the problem of deciding whether the division can be done with two trees is itself *NP*-hard. Hence proving the *NP*-hardness of finding the minimum number of trees. ■

LEMMA A.1. *The problem of deciding whether a directed acyclic graph can be broken into two trees with all the cross edges, from one tree to the other, in the same direction is* NP*-hard.*

Proof. We will prove the *NP*-hardness by giving a polynomial time reduction of *Exact Cover* problem to this problem. The exact cover problem is as follows: *given a set S and a collection C of subsets do there exist sets in C which partition S.*

We will assume that each element is present in at least two sets belonging to C. Suppose there is an element which is there in only one set. Then we will have to take that set in the cover and hence we can incrementally go on removing such elements, and some sets will be forced into the cover and some will be forced out of the cover. Finally, only elements which are present in at least 2 sets will remain. Let the remaining subsets be $S_1, S_2 \dots S_n$ and the elements be $a_1, a_2 \dots a_n$. We consider the following graph $G(V, E)$ on the above sets and elements.

(i) $V = 0, 1 \cup \{a_i\} \cup \{S_j\}$
(ii) $(1, 0) \in E$
(iii) $(S_i, 0), (1, S_i) \in E$
(iv) $(a_i, S_j) \in E$, if $a_i \in S_j$

Refer ahead to Example A.1.

We now show that an exact cover exists if and only if G can be divided into two trees with all the cross edges in the same direction.

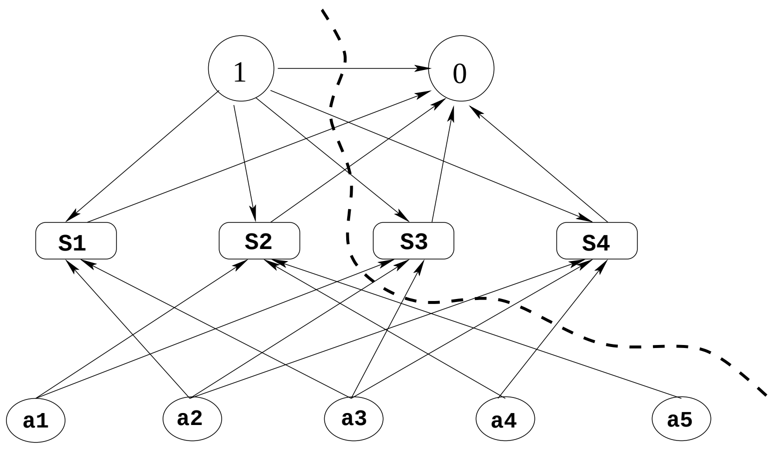

FIG. 1. The graph G for Example A.1 along with partition.

First, suppose there is a partition of the above graph into two trees such that the edges between the two trees are in the same direction. We will show that an exact cover exists.

We see that 0 and 1 cannot be in the same tree because if so, then if some S_i is there in the other tree then there will be 2 edges between the trees in opposite directions and if some S_i is there in the same tree then there will be a cycle 0-1-S_i-0 (in the undirected sense) and hence it won't be a tree. Thus 0 and 1 are in separate trees.

Now let T_0 and T_1 be the trees containing 0 and 1 respectively. We show that there cannot be any node corresponding to some element a_i in T_0. On the contrary, suppose there is an element e in T_0. Every element is there in at least two sets. Also e is connected to at least one set node in T_0. These together imply that either there exists S_i, S_j in T_0 such that both are adjacent to e or there exists S_i in T_0 and S_j in T_1 such that both are adjacent to e. In the first case there will be a cycle (in the undirected sense) $e, S_i, 0, S_j, e$ in T_0 and in the second case there will be 2 edges $(1, 0)$ and (e, S_j) in opposite directions, both of which are not possible. Hence all the element nodes are in T_1.

Now consider any element in T_1. It cannot be adjacent to two vertices S_i, S_j. Also it should be adjacent to at least one vertex. Thus, if we take all the S_i in T_1, they will, as a whole, cover each element once and exactly once. Hence if the graph satisfies the property stated in the lemma, we see that there exists an exact cover.

It is also easy to see that if there is an exact cover then we can partition the graph into two trees such that the required property is satisfied. Let T_1 contain the subgraph on the vertices 1, $\{a_i\}$ and $\{S_j | S_j \in \mathit{Cover}\}$, and let T_0 contain the remaining vertices and the subgraph formed. It is easy to see that both T_0 and T_1 are trees and all edges between T_0 and T_1 are from T_0 to T_1.

This completes the proof of the lemma. ■

The following example illustrates the reduction used in the proof of Lemma A.1.

Example A.1. *Consider $S = \{a_1, a_2, a_3, a_4, a_5\}$. Let C consist of S_1, S_2, S_3 and S_4, where $S_1 = \{a_1, a_4, a_5\}$, $S_2 = \{a_2, a_3\}$, $S_3 = \{a_1, a_2, a_3\}$ and $\{a_2, a_4, a_5\}$. The corresponding graph G is given in Figure 1.* ■