Pipelining in Multi-Query Optimization[*]

Nilesh N. Dalvi [1] Sumit K Sanghai [1] Prasan Roy [2†] S. Sudarshan [1]

[1] Indian Institute of Technology, Bombay
{nilesh,bunny,sudarsha}@cse.iitb.ernet.in

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

ABSTRACT

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. 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 [6] optimize queries one at a time and do not identify any commonalities in queries, resulting in repeated computations. As observed in [10, 8] exploiting common results can lead to significant performance gains. This is known as multi-query optimization.

[*]Work 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.

[†]Work performed while this author was at I.I.T., Bombay.

Existing techniques for multi-query optimization assume that all intermediate results are materialized [9, 4, 11]. 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. Consider 2 queries, $Q_1 : \sigma_{A.x=5}(A \bowtie B)$ and $Q_2 : \sigma_{B.y=10}(A \bowtie B)$. 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. Now, if we pipeline the results of $A \bowtie B$ to both the selects, 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 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 is used. Also suppose $MJ1$ is getting very few 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. □

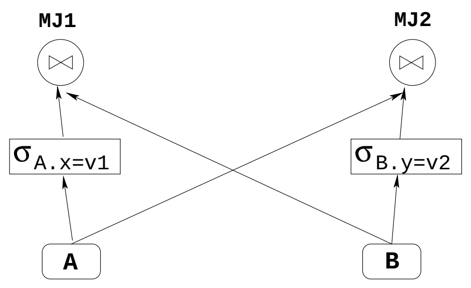

Figure 1: Unrealizable Schedule

Our Contributions: The following are the major contributions of this paper:

- We present a general model for pipeline schedules, where multiple uses of a result can share a scan on the result of a subexpression; as a special case, if all uses of the result can share a scan, the result need not be materialized.
- We then present a necessary and sufficient condition for determining validity (realizability) of a schedule under our model.
- We show that given a plan which includes sharing of subexpressions, finding a valid schedule with minimum cost is *NP*-hard.
- We then present algorithms for finding good pipelined schedules statically (dynamic materialization is briefly discussed in Section 6.5). 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. 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 gives our cost model and shows that the problem of finding the least cost pipeline schedule for a given DAG structured query plan is *NP*-hard. We give a greedy heuristic for finding good pipeline schedules in Section 6. In Section 7, we give a detailed performance study of our heuristics, and Section 8 concludes the paper.

2. RELATED WORK

Related works by Chekuri et al. and Hong [1, 7] have 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 [12] 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 rightdeep 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 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 [5] 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.

The RedBrick data warehouse (now part of Informix) has long implemented a shared scan operation on base relations, which allows multiple queries to share the output of a scan (see, e.g., [2]). 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 RedBrick arises when a database relation is used twice in the same query. A simple solution for this case would be to simply not share relation scans within a query, but we are not aware of any work describing how RedBrick handles this case. However, the RedBrick warehouse has 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 static.

3. INCORPORATING PIPELINING

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 multiquery optimization to choose a plan for a given set of queries, ignoring pipelining optimizations, as done in [9]. The second phase, which we cover in this paper, addresses optimization of pipelining for a given plan. Single phase optimization, where the multiquery 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*.

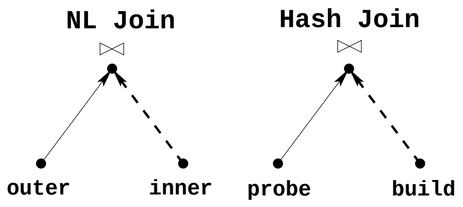

Figure 2: Examples of Pipelinable Edges

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.

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[1]. 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 it will eventually overflow or the result would have to be materialized. We assume intermediate results will not fit in memory[2].

For instance consider the following two examples.

[1]We will follow this convention throughout the paper

[2]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 [12] in their context, but is a topic of future work in our context.

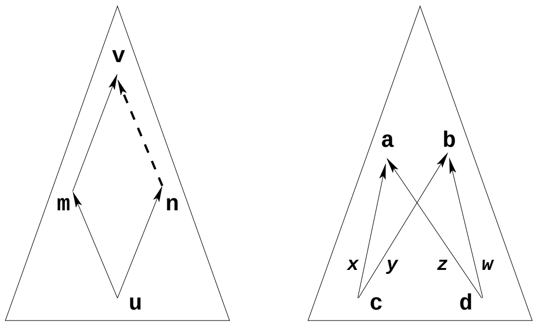

Figure 3: Problems in Pipelining

- 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.
- 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 will be changing dynamically depending on the data. So, this schedule cannot be realized.

In the next section we generalize the above situations which can give rise to schedules which cannot be realized.

4. PROBLEM FORMULATION

We now define a model for describing a valid pipeline schedule, i.e. the schedule can be executed without materializing any edge marked as pipelined and using limited buffer space.

4.1 Problem definition

DEFINITION 1. (Pipeline Schedule) A *pipeline schedule* is a Plan-DAG with each edge labeled either pipelined or materialized. □

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 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 $P(e, t)$, increasing 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 enforces that all the tuples generated at a node are passed immediately to each of its parents, thereby avoiding the need of them to be stored in buffer. The second condition enforces that the tuple requirements of each operator is simultaneously satisfied. The third condition enforces that the schedule gets completed.

Definition 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 $S(a_i) = S(n)$ □

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.

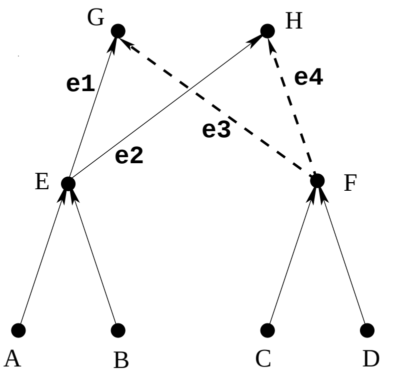

Figure 4: The plan-DAG and the pipelining schedule for Example 3

Note that the tuple requirements of the operators are dynamic and are not known a priori. But, using either push or pull, with limited buffers, the rates will get adjusted dynamically in an actual evaluation. Valid schedules will complete execution and invalid ones may deadlock unless results are materialized. A detailed proof is given in Section 4.2. For instance, using the pull model, operators can be implemented as iterators as described, for example, in [5]. If there are multiple roots in a Plan-DAG, the iterators for all the roots start execution in parallel. For instance, in Example 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.

Example 3. Consider the Plan-DAG given in Figure 4. The dashed edges are materialized while the rest are pipelined. Also, 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. Also, functions can be assigned to all pipelined edges so that all the conditions 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.□

4.2 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. (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 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 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 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 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.* □

The proof is given in the Appendix.

4.3 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 the subgraphs induced by each of the C_i is a tree, which is a necessary condition as shown in the proof of Theorem 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 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 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.

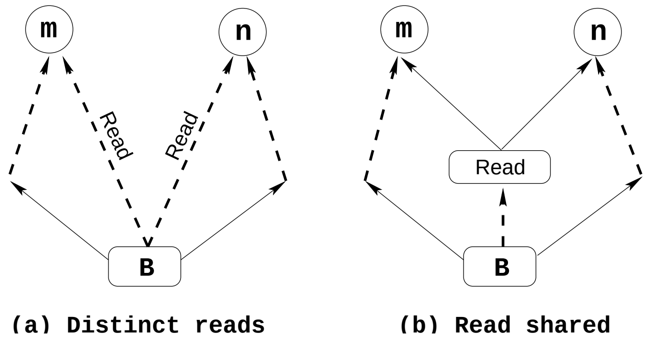

Figure 5: Shared-read Optimization

5.1 Cost Formulation

Given a pipeline schedule S, its materialization and reading cost $MC(S)$ is given by the following formula.[3]

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

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. We can further reduce costs by optimizing the multiple reads of materialized nodes. The following example illustrates this point.

Example 4. Consider the query with a section of Plan-DAG given in Figure 5(a). Assume that the node B is materialized, 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 5(b). □

However, not all uses of a materialized node can be shared. For example, if the two nodes reading the materialized node have different stage numbers, then they cannot share the read. This is because sharing a read will force both of them to be computed together, but different stage numbers imply that one has to be completed before the other starts.

The criterion for checking the validity of a pipeline schedule can be used here for checking whether a set

[3]Do not confuse this with the plan cost. The actual plan cost includes the materialization cost and the algorithm cost at each node, but algorithm costs do not change as pipeline schedules change, so we ignore them here.

of reads of a materialized node can be shared. This can be done by transforming the Plan-DAG as shown in Figure 5. An extra node corresponding to a scan operator is added to the Plan-DAG, a materialized edge is added from the 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. Later, we present a polynomial time exact algorithm for finding the best way of sharing reads to maximize the benefit from shared-read optimization.

5.3 NP-Completeness

In this section, we prove the *NP*-hardness of the problem of finding least cost schedules, as stated in Theorem 3. Clearly the corresponding decision problem belongs to the class NP, since by Theorem 2, the validity of a schedule can be checked in polynomial time.

Theorem 3. 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, which may have multiple edges, 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 4. If in any Plan-DAG, there is an edge e from n_1 to n_2, then the following hold:

1. 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 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 C-cycle formed by collapsing e is there in 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)which are to be finally 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. Thus there is a contradiction. Hence proved. □

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. It is however exponential in the number of edges.

6.2 Greedy Algorithm

Since the problem of finding the least cost pipeline schedule is *NP*-hard, we present a greedy heuristic, shown in Algorithm 1. At each stage, the Greedy heuristic chooses that edge to $Merge$ which will give the maximum benefit.

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 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.

Algorithm 1 Greedy algorithm

```
$findBestGreedy(dag)$
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 $findBestGreedy(dag1)$
end
```

6.3 Algorithm for Shared-read Optimization

In section 5.2, we discussed the optimization which reduces the number of reads of materialized results. Before describing the algorithm, we note the following points:

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 read only if they have same stage numbers.
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 4.2 by applying the transformation described in Section 5.2.

We now construct a graph with vertices as the set of equivalence classes. Firstly, we add edges present in the *Constraint Dag*, defined in Section 4.2. These edges are all directed and are given weights equal to *infinity*. Let this set of directed edges be denoted by E_d. Also, for each pair of equivalence classes and for each materialized node, we add an undirected edge between the two equivalence classes if they both read from this materialized node. We put the weight of the edge as the read cost of node. Note that there can be multiple edges between two classes, corresponding to different materialized nodes. We call the set of undirected edges as E_u.

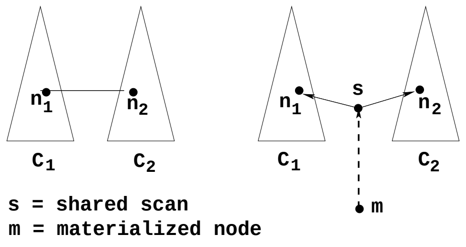

Figure 6: Transformation of Plan-DAG for shared-read optimization

THEOREM 5. Let S be any subset of E_u. Then, the materialized nodes denoted by the edges in S can be shared by 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 edge in this graph will be replaced by two pipelined edges in the transformed Plan-DAG, as shown in Figure 6. 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. The theorem then easily follows. □

So, now the problem is to find the set S with largest total weight such that no cycle is formed. This can be done in a greedy fashion by sorting all the undirected edges in decreasing order of weight and keep on adding those which do not result in a cycle. To prove that this algorithm will give the required S, we can use arguments similar to the Kruskal's algorithm [3] for finding minimal spanning trees.

6.4 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 Greedy heuristic for pipelining 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.

6.5 Discussion

It is possible to execute a schedule by dynamically materializing results when buffer overflow happens. But in a naive implementation of this approach overflows may occur at nodes which have high materialization cost and thus a naive implementation may result in a high price for materialization. On the other hand, our algorithm which gives a static schedule forces materialization. Combining the benefits of the static and dynamic approaches to materialization is a topic for future work.

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 an existing Volcano-based multi query optimizer described in [9].

Through the experiments, we try to analyze the performance of our algorithm as compared to the multi-query optimizer algorithm without applying our pipelining optimizations. We applied our pipelining algorithm on Plan-DAG generated by both pessimistic and optimistic approaches. We also show an optimistic lower bound, which assumes that all the shared expressions are pipelined.

For experimental purposes, we use the multi-query optimizer algorithm described in [9]. 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 evaluating 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 [9]. The cost model is fairly accurate as shown in [9].

We use the TPCD database at scale factor 1 (i.e., 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 10m-sec, transfer time of 2m-sec/block for read and 4m-sec/block for write, and CPU cost of 0.2m-sec/block of data processed. The materialization cost is the cost of writing out the result sequentially.

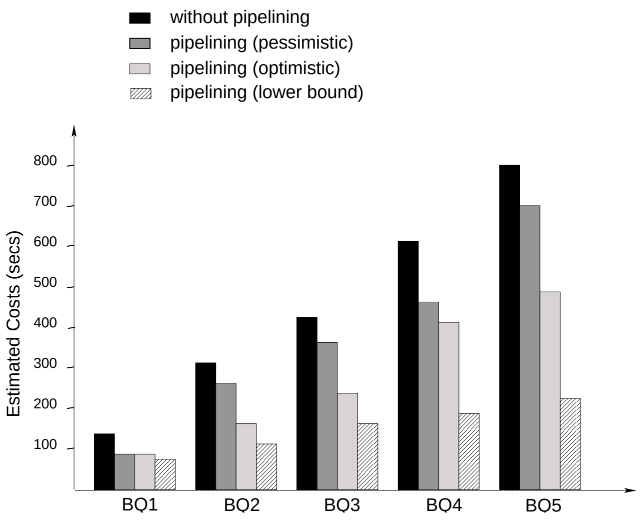

Figure 7: Results on batched TPCD queries

The batched TPCD workload models a system where several TPCD queries are executed as a batch. The workload consists of subsequences of the queries Q_1, Q_3, Q_5, Q_7 and Q_9 from TPCD. These queries have common subexpressions between themselves. The batch query BQ_i contains the first i queries from the above sequence. Thus BQ_2 contains Q_1 and Q_3, while BQ_5 contains all of the above queries.

The results of the workload are shown in Figure 7. The plot contains a set of four values on each query. The first bar shows the cost of the query plan generated by the multi-query optimizer without pipelining. The second one is the cost of the plan after applying greedy pipelining algorithm on the plan generated using pessimistic approach. The third bar is the cost of the plan after applying greedy algorithm on the plan generated using optimistic approach. The fourth bar shows the cost of the optimistic plan, assuming everything is pipelined. Pipelining everything may not result in a valid pipeline schedule, but it gives a lower bound on the cost of any valid pipeline schedule.

From the graph, we can infer that the percentage gain over basic multi-query optimizer varies from 13% to 35% for pessimistic approach, while for optimistic approach it varies from 32% to 50%. Thus, significant gains can be achieved through the optimistic approach without any noticeable increase in the optimization time.

We also see that the optimistic lower bound is significantly less than the cost of the pipelined approach. This indicates that there is a significant number of shared subexpressions and also that not all of these subexpressions can be pipelined. We found that the main reason subexpressions not getting pipelined is that there are several subexpressions having shared-scans on the same set of base relations. This results in the formation of several C-cycles which have to be broken. These

C-cycles are both inter-query and intra-query.

8. CONCLUSIONS

In this paper, we studied the issue of pipelining 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.

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 in multi-query optimization can lead to significant performance gains.

In conclusion, we can say that a good choice of a pipeline schedule gives significant benefits in multi-query optimization.

Acknowledgement

We wish to thank A. A. Diwan and Dilys Thomas 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.

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

[6] Goetz Graefe and William J. McKenna. Extensibility and Search Efficiency in the Volcano Optimizer Generator. In *Intl. Conf. on Data Engineering*, 1993.

[7] Wei Hong. Exploiting inter-operation parallelism in XPRS. In *SIGMOD Intl. Conf. on Management of Data*, pages 19–28, 1992.

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

[9] 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.

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

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

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

APPENDIX

Proof of Theorem 1

THEOREM 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,

$$S(a_1) \leq S(a_2) \leq S(a_3) \cdots \leq S(a_n) \leq S(a_1) \qquad (1)$$

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

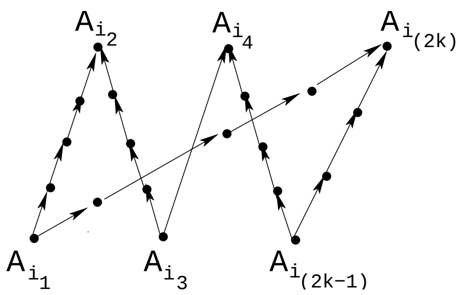

Figure 8: C-cycle without materialized edges

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 8.

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.

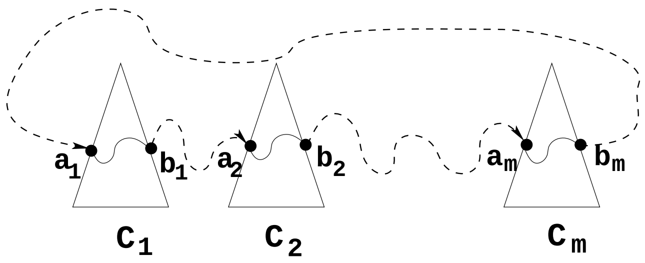

Figure 9: The set of equivalence classes

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.

Now let $E_q = C_1, C_2 \ldots C_k$ be the set of equivalence classes of $\sim$ defined in Definition 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 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 9, 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, atleast 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 atleast 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 atleast one child from which it needs to get a tuple but the child is itself blocked. Note that the first condition of Definition 2 enforces that if an operator produces a tuple it has to pass 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 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's 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 atleast 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. □

Proof of Theorem 3

THEOREM 3. 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 6 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 6. 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 4.2. We have seen that it produces equivalence classes $C_1, C_2 \ldots C_k$ 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,

$$n = \sum_{1 \leq i \leq k} |C_i| \tag{2}$$

$$\begin{aligned} p &= \sum_{1 \leq i \leq k} (|C_i| - 1) \\ &= n - k \end{aligned} \tag{3}$$

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 minimum number of equivalence classes. In other words, we have to divide the Plan-DAG into minimum number of trees so that all the edges between any two trees are in the same direction and the graph made on these trees do 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 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 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 $S_1, S_2 \ldots S_n$ and the elements be $a_1, a_2 \ldots a_n$. 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 $a_i \in S_j$

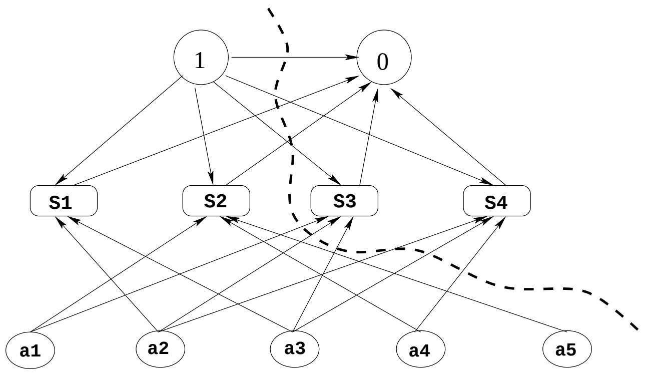

Figure 10: The graph G for Example 5 along with partition.

Refer ahead to Example 5.

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

First, suppose there is a partition of the above graph into two trees so that the edges between the two trees are in the same direction. We will show that 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 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 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 $\{S_j | S_j \in Cover\}$, 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 1.

EXAMPLE 5. 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 10. □