Deadlock Resolution in Pipelined Query Graphs

Vladislav Shkapenyuk **Ryan Williams**
Stavros Harizopoulos **Anastassia Ailamaki**

March 2005
CMU-CS-05-122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work was supported in part by IBM through an IBM faculty partnership award, and by the National Science Foundation, under grands CCR-0113660, IIS-0133686, and CCR-0205544, and also as part of the Aladdin Center (www.aladdin.cmu.edu) under grants ACI-0086093, CCR-0085982, and CCR-0122581.

Keywords: pipelining, deadlock resolution, dynamic buffer management, query plan, materialization

Abstract

Pipelining is a widely used technique that query execution engines employ to improve individual query execution times. In recently proposed settings, pipelining is used as an alternative to materialization to evaluate *query plan graphs*, where nodes in a query plan can have multiple parents. Such scenarios include shared table scans, runtime operator sharing, parallel sorting, and pipelined Multi-Query Optimization (MQO) plans. While pipelining in query graphs can increase performance, it can also lead to runtime deadlocks. The existing deadlock solutions focus on total deadlock prevention by statically deciding between pipelining and materialization, and apply only to MQO. The lack of runtime information leads to highly conservative decisions. Formally, this conservatism makes it NP-hard to find an optimal pipelining strategy that materializes a minimum cost set of nodes in a query graph.

We propose a novel dynamic scheme for detecting and resolving deadlocks in pipelined query graphs. Instead of statically determining what nodes to materialize, we pipeline every pipelinable operator in the query plan, resolving deadlock when it arises by materialization. At runtime, more information about producer-consumer behavior is available and as a result, the problem of detecting and resolving deadlock becomes polynomial time solvable, where each particular deadlock resolution is of minimum cost. Our techniques are applicable to any system that uses pipelined query graphs. An evaluation on TPC-H and Wisconsin benchmarks demonstrates cases where our approach eliminates all unnecessary materializations, and minimizes the overall query execution cost.

1 Introduction

Pipelining intermediate results when evaluating a query is a widely used method for improving execution costs. Typically, query plans are represented by operator trees, with tuples flowing from the leaf nodes towards the root of the tree. Each operator-node consumes tuples from its children and outputs produced tuples, using pipelining whenever possible, to its parent. Recent systems and research proposals apply pipelining to certain query evaluation strategies that are based on *query plan graphs* (more specifically, DAGs) instead of trees. The difference is that a single node can have multiple parents, which can either belong to the same or different queries. Such scenarios include sharing a table scan [5, 1], runtime operator sharing [8], parallel sorting [6], and pipelining of common operators in Multi-Query Optimization (MQO) [2]. Pipelined query graphs can speed up execution when compared to pipelined tree plans, by eliminating redundant computation and data accesses, but they may also lead to runtime execution deadlocks.

Given a DAG $G = (V, E)$, let us define *the directionless graph of* G to be the undirected graph

$$G_d = (V, \{\{u, v\} | ((u, v) \in E) \vee ((v, u) \in E)\}).$$

G_d is sometimes also called the *shadow* of G. Deadlocks may occur in a pipelined query plan graph every time there exists a cycle in the directionless graph, due to different tuple consuming/producing rates across nodes and the finite buffers between producers and consumers. In a pipelined plan, both the consuming and producing rate of a node may depend on the producing rate of the children nodes and the consuming rate of the parent node. For example, one such dependence arises when, in order for a node with two children to produce a tuple, that node must first wait until its left child node provides a tuple. Also, a producer may also have to wait on the consumption of a tuple if its current output buffer is full. In general, two nodes may indirectly affect each other's consuming/producing rate as long as there is some graph connection between them (a path between the two nodes in the directionless graph). Intuitively, a cycle in the directionless graph means that a node can be in a situation where a requirement for a tuple production/consumption places an additional requirement to that node. If those two requirements are conflicting, a deadlock may occur. For instance, a node can set the requirement "to produce a tuple, a tuple needs to be consumed," which may trigger a requirement on the same node of the form "to consume a tuple, a tuple needs to be produced."

A proactive way to remove the possibility of deadlock in query plan graphs is to always materialize the results produced by shared query nodes. This is also the default strategy in Multi-Query Optimization. Ideally, we would like to use pipelining as much as possible and rely on selective materialization only when it is absolutely necessary. Recent work has studied the deadlock problem in pipelined MQO plans [2], and proposed to statically decide, at optimization time, the minimum cost set of shared nodes to materialize such that no deadlock is possible. This decision problem was shown to be NP-hard and a polynomial time greedy approximation algorithm was proposed (note, no guaranteed performance ratio was proven for this algorithm). Since no guaranteed statistics about operator selectivities are available during optimization, the problem suffers from a lack of information: even if we were to solve the NP-hard problem, its materialization decisions are not only pessimistic but also suboptimal, possibly performing more materialization than necessary. Moreover, such an approach is not applicable to systems employing runtime sharing of operators or table scans.

This paper first proposes a new model for defining deadlocks in query plan graphs based on the state of the buffers used for inter-operator tuple transfers. Then, we introduce a *reactive* approach to solving deadlocks in pipelined query plan graphs. Instead of statically deciding on the set of nodes to materialize, we initially pipeline every pipelinable operator in the query plan. Since we expect deadlocks to be infrequent during the lifetime of the query, we avoid unnecessary materialization costs. Based on our model,

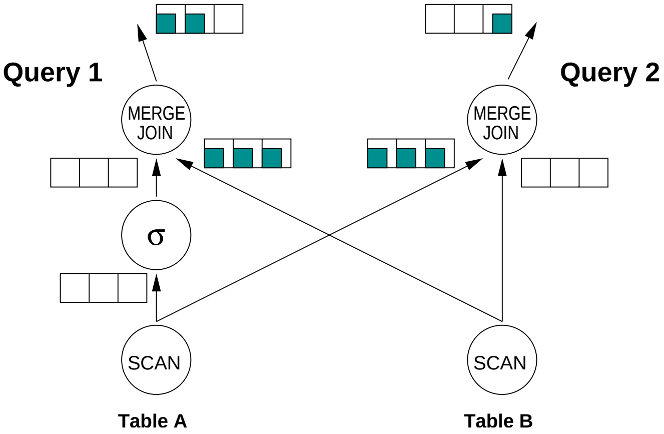

Figure 1: **Deadlocked query graph.**

we propose polynomial time algorithms to detect deadlocks at runtime and find an optimal set of nodes to materialize that resolves a given deadlock and minimizes the total execution cost. The proposed algorithm can be easily integrated to systems employing pipelined query graphs, providing a low-overhead, optimal solution to the deadlock problem. We evaluate our approach against the static materialization and materialize-everything approaches using queries from the TPC-H benchmark and the Wisconsin Benchmark.

The rest of the paper is organized as follows. In the next section we describe in more detail the deadlock problem in pipelined query graphs and discuss related work. Section 3 describes our dynamic model and its basic properties. Section 4 presents our dynamic approach to optimally removing deadlocks and describes how our techniques can be applied to execution engines that support pipelined query graphs. Section 5 carries out the experimentation, and finally, we conclude in Section 6.

2 Background and Related Work

Pipelined query graphs appear in scenarios where an operator may have its output pipelined to more than one parent nodes. The simplest case of a multi-output operator is scan sharing. The RedBrick warehouse [5] and Microsoft SQL Server [1] allow multiple queries to share an ongoing table scan by consuming its output as it becomes available. Queries arriving in the middle of the scan can still share the last part of the table and read the remaining tuples later, provided that the order tuples are read does not matter. A more aggressive form of sharing is to share the work performed by any type of relational operator, whenever two or more queries have an overlap in their plans that is detected at runtime [8]. In that case, the intermediate results produced by the shared operator are pipelined to all participating queries. In the context of parallel sorting [6], multiple producers sort partitions of a relation and pipeline their results to multiple consumers who perform a merge. Lastly, a recent proposal [2] applies pipelining in the context of Multi-Query Optimization [12]. Work done in MQO focuses on identifying common subexpressions in a query batch at optimization time, and provide globally-optimal plans. The shared nodes are typically materialized and subsequently read independently by each query. The work in [2] proposes to pipeline the results of some of the shared nodes.

In all above-mentioned scenarios, the collective evaluation plan is a DAG (Plan-DAG) instead of a tree or a collection of trees. As also shown in [2] for MQO pipelined graphs, it is possible that deadlocks may occur while evaluating a Plan-DAG, because of the fact that operators have finite buffers. To illustrate this problem consider the Plan-DAG shown in Figure 1. Queries 1 and 2 each perform a merge-join on tables

A and B, with Query 1 applying a selection predicate on table A. Since both queries scan the same tables, the results from each scan are pipelined to both queries. Each operator in a query plan has a finite buffer in which incoming tuples are temporarily stored, before they are processed; the produced tuples are sent to the buffer of the parent operator. In Figure 1, next to each edge, we show a snapshot of the state of those buffers. In this specific scenario, where two of the buffers are full, the evaluation of the Plan-DAG cannot proceed. In order for the merge-join of Query 2 to proceed (by consuming a tuple from its left full buffer), it needs to receive a tuple from scan B. However, scan B cannot provide a tuple since it would overflow the right buffer of Query 1 merge join. Similarly, Query 1 cannot consume any tuples from its right full buffer, causing a deadlock in the evaluation of the Plan-DAG.

The authors in [2] propose to solve the deadlock problem in pipelined MQO plans by deciding at query optimization which shared nodes will be pipelined and which will be materialized, forming this way a *valid schedule* (deadlock-free evaluation of the Plan-DAG). They observe that whenever a cycle exists in the directionless graph (called a C-cycle) and certain conditions apply, then a schedule is not realizable (may deadlock at runtime). Once the deadlocking C-cycles are identified, they are broken by selective materialization of a subset of shared nodes. Due to the pessimistic nature of the described approach the, problem of finding minimal cost solution was shown to be NP-hard. Furthermore, there are no constant-time approximations due to the reduction used, so the authors rely on greedy heuristics with no provable guarantees of optimality. Under these pessimistic assumptions, a large number of safe query graphs will be flagged as potentially deadlocking and will lead to unnecessary materialization. Since the analysis of the query graph is done at query optimization time, the algorithm suffers from a lack of information about operator selectivities and real materialization costs. As a result, even if the graph is going to deadlock at the runtime, the algorithm can make suboptimal choices.

3 Formal Problem Definition

We now define a framework for manipulating pipelined query graphs dynamically that generalizes the Plan-DAGs of [2]. A Plan-DAG which is a directed acyclic graph (DAG) formed by merging query execution plans (trees) that share some subplans. We say that a DAG is *connected* if it is connected when its directionless graph is connected. All DAGs we will consider in this paper are connected; this is the interesting case, otherwise the corresponding Plan-DAG is executing disjoint queries. Each node in a Plan-DAG corresponds to either a base relation, the final result of some query, or an intermediate result: these nodes have indegree 0, outdegree 0, and arbitrary indegree and outdegree, respectively. We will refer to indegree-0 nodes as “initial producers”, and outdegree-0 nodes as “final consumers”.

Edges represent producer-consumer relationships: edge (u, v) means that tuples of u are used by v. Each edge of a Plan-DAG has a finite buffer associated with it. Over time, some buffers may become full and some may become empty; let us say the full-buffer edges are “heavy” and the empty-buffer edges are “light”. Heavy and light edges can slow the execution of the overall plan, and if enough edges are heavy or light, the execution will introduce deadlocking structures within the graph.

Worst-case deadlock prevention in Plan-DAGs attempts to materialize such that one avoids heavy and light edges altogether: if there exist situations that might lead to such deadlocking structures, the (possibly) offending nodes are materialized. Clearly, this is a pessimistic approach to the problem. Our dynamic model is *optimistic*, fixing deadlock when and if it arises. Our framework allows us to define the deadlock problem in terms of existing vocabulary for fault-tolerance, *i.e.* we use waits-for graphs to directly formalize what goes wrong in a deadlocked pipelined schedule (and how it goes wrong), as opposed to assuming ignorance of runtime information and “materializing away” every possible deadlock case. Moreover, our model is clean and simple to understand, while deadlock detection and (optimal) resolution can be done in polynomial time; that is, each deadlock that arises can be optimally resolved efficiently. The *only* runtime information we require for deadlock detection is knowledge of which buffers are full or empty, if any. Hence, our optimistic approach can potentially outperform the off-line (NP-hard) scenario, without solving an NP-hard problem to do it.

3.1 The Dynamic Model

In our model, we attempt to pipeline everything, only materializing nodes in the event of a deadlock. Deadlock detection and resolution (materialization choices) are done by monitoring the status of buffers in the query graph. To capture this, we define a Buffered Plan-DAG, a graph which represents the current state of buffers at some point during the query execution.

Definition 3.1 *A* Buffered Plan-DAG $G' = (G, s)$ *is a Plan-DAG* $G = (V, E)$ *along with a buffer state function* $s : E \to \{\mathbf{F}, \mathbf{E}, \mathbf{N}\}$.

That is, each edge of a Buffered Plan-DAG is labeled either **F** (for a full buffer), **E** (for empty), or **N** (for neither). We will precisely characterize below the states for which deadlock arises. For a Buffered Plan-DAG, we can formulate a notion of "waiting" which captures producer and consumer interaction. Given any final consumer v, at some point in time v may be waiting on a tuple from another node u in order to process its output. (Formally, we would have $s(u, v) = \mathbf{E}$.) Analogously, for every initial producer v, v may be waiting to put something on the input buffer of one or more u. (Formally, $s(v, u) = \mathbf{F}$.) In general, a node with nonzero indegree and outdegree could be waiting on one of its predecessors or several of its successors. To model this for a Buffered Plan-DAG G', we define a corresponding *waits-for graph*:

Definition 3.2 *Let G' be a Buffered Plan-DAG* $G' = ((V, E), s)$. *Define* $\mathsf{Waits}(G')$ *as the directed graph* (V, E') *such that*

$$E' = \{(v, u) \mid [(u, v) \in E \wedge s(u, v) = \mathbf{E}] \\ \vee [(v, u) \in E \wedge s(v, u) = \mathbf{F}]\}.$$

We will first define what a deadlock scenario means in our situation, and then show that this is in a sense equivalent to the existence of a cycle in the above waits-for graph. When execution is deadlocked, no progress is possible, hence every node is waiting on some other node to progress. This motivates the following definition.

Definition 3.3 *Let* G *be a Buffered Plan-DAG.* $G = (V, E)$ *is* deadlocked *iff for all* $v \in V$, *there exists* $u \in V$ *such that* $(v, u) \in \mathsf{Waits}(G)$.

The intuition is simply that if no node v can make progress, then v is waiting on some other vertex u, either by having an output that cannot be sent to u, or by not receiving an input from u. To illustrate, let us state a few simple propositions concerning deadlock.

Lemma 3.1 *Let* G *be a Buffered Plan-DAG. If* G *is* deadlocked, *then for all vertices* v, *either* v *has an incoming edge labeled* **E** *or an outgoing edge labeled* **F**.

Proof. By definition of $\mathsf{Waits}(G)$ and of deadlock. □

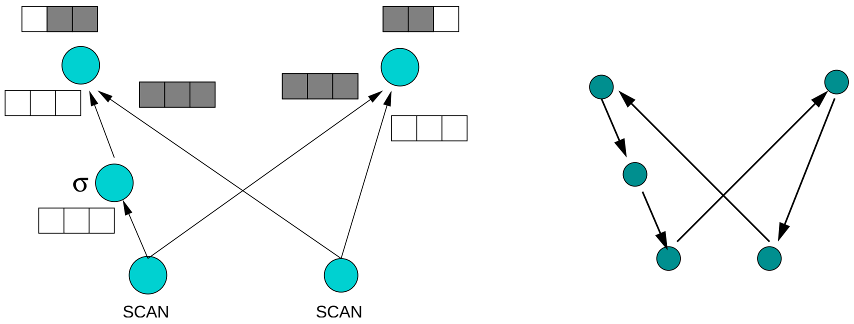

Figure 2: **Deadlocked Plan-DAG with corresponding waits-for graph.**

Corollary 3.1 *Let v be a node in a deadlocked Buffered Plan-DAG.*
1. If v is a final consumer, then at least incoming edge to v has label **E***.*
2. If v is an initial producer, then at least one outgoing edge from v has label **F***.*

Proof. Final consumers do not have outgoing edges, and initial producers do not have incoming edges. □

That is, in a deadlocked situation, all final consumers are waiting on one of its input buffers to contain something, and all initial producers are waiting on one of its output buffers to pull a tuple.

3.2 An example

To illustrate our concept, consider the example query Plan-DAG shown in Figure 2. The all-shaded rectangles represent full buffers, and the all-white rectangles are empty buffers. In the figure, there are two paths (modulo edge directions) from one SCAN node to the other. The intuitive argument from [2] for why some of the nodes here need to be materialized (prior to execution) is simply that the rates of consumers and producers along the two paths could possibly vary greatly, even if one side has a materialized node and the other one does not, leading to buffer overflow on one side. If this were to occur in our model, then the two paths between the SCAN nodes would have all **F** and E edges with the proper edge directions. This results in a cycle in the waits-for graph. Therefore, the intuitive arguments for deadlock prevention are made completely formal in our model.

3.3 Modelling Deadlock Effects Over Time

In order for us to prove an equivalence between the above notion of deadlock (which implies no progress at initial producers or final consumers is possible) and cycles in a waits-for graph, we need to more explicitly model the temporal effects of buffer behavior in a Buffered Plan-DAG. In a real system, once a small portion of a pipelined query execution becomes deadlocked, this effect “trickles” throughout the graph: more and more buffers fill up, while others empty out completely. Here we briefly show how this epidemic nature of deadlock can be neatly inserted into our framework.

Definition 3.4 *A vertex v in a Buffered Plan-DAG G with edge set E is in* deadlock-wait on vertex *w if:*
(a) (v, w) is in a cycle of $\mathsf{Waits}(G)$*,*
(b) Either $(v, w) \in E$ or $(w, v) \in E$, and w is in deadlock-wait on some vertex $x \neq v$.

This definition has a strong intuitive interpretation: clearly every node in a cycle is in deadlock and waiting, any node producing tuples for some deadlock-waiting w eventually deadlock-waits (its output buffer will become full or one of its other neighbors will deadlock-wait), and any node consuming tuples from a deadlocked w eventually deadlock-waits (by an empty input buffer, or one of its other neighbors deadlock-waiting).

Let G be a Buffered Plan-DAG. Suppose $\mathsf{Waits}(G)$ has a cycle. If we model deadlock propagation by extending $\mathsf{Waits}(G)$ to include edges for deadlock-waiting, we immediately get the following equivalence between deadlocking and waits-for cycles.

Proposition 3.1 *Suppose we repeatedly add (u, v) to $\mathsf{Waits}(G)$ if u is in deadlock-wait on v. After this process converges, G is deadlocked if and only if G is connected and there is a cycle in $\mathsf{Waits}(G)$.*

Proof. If G is deadlocked, then by definition, every vertex of $\mathsf{Waits}(G)$ has an outgoing edge to some other vertex. Starting from an arbitrary vertex in $\mathsf{Waits}(G)$, follow the outgoing edge of the current node. After $n + 1$ iterations of following edges, at least one node must have reached twice, so $\mathsf{Waits}(G)$ has a cycle. For the other direction, remove the directions from edges in G. One can easily prove that all vertices are deadlock-waiting by induction on the length of the shortest path from a vertex v to a cycle C in $\mathsf{Waits}(G)$. (This suffices since G is connected.) $\square$

4 Detecting and Breaking Deadlock Dynamically

As mentioned earlier, deadlock is typically prevented in pipelining scenarios by materializing nodes in every configuration where deadlock could arise. Three nice properties of our dynamic approach are:

1. Our use of waits-for graphs potentially allows us to employ much existing deadlock literature to our problem.

2. Finding a waits-for cycle in our model is equivalent to detecting the precise moment at which a particular buffer configuration will eventually stall computation. That is, we can detect and optimally resolve the problem immediately as it arises, when the last buffer in the cycle empties or becomes full. We do *not* have to wait until all progress is blocked.

3. The optimal materialization problem becomes much simpler in the dynamic setting. In particular, the NP-hardness is eliminated! That is, each deadlock resolution is “optimal” in that we materialize the minimum amount of nodes necessary to resolve a given deadlock, and each one is resolved in polynomial time. Now, over time, if many deadlocks were to repeatedly arise throughout the plan DAG, our approach could potentially fare poorly, as we are only making quick decisions about what to materialize on-the-fly. However, when there is little deadlock potential overall, our online method appears to be the clear winner.

We assume that the waits-for graph $\mathsf{Waits}(G)$ has restricted access: anytime a buffer becomes full or empty, or goes from full or empty to neither, it acquires a lock on $\mathsf{Wait}(G)$, updates it accordingly, then releases the lock. Our model of this formally will be that $\mathsf{Waits}(G)$ gets updated over time by a single addition (or deletion) of a node from it. Technically, one could also model this by adding or deleting edges. However, note the node addition/deletion model is only stronger, since an edge addition/deletion can be modelled by two node addition/deletions, but not the other way around.

4.1 Deadlock Detection

Essentially, we will reduce the problem to *dynamic connectivity in directed acyclic graphs*. The reader should note that in general, dynamic directed connectivity is a notoriously hard problem; the best known theoretical solutions are still quite horrible [11] and take linear time per update, like ours. We note that while it is *theoretically* possible to perform detection faster [3], the algorithms to do this use fast matrix multiplication as a subroutine, which is unfortunately not yet practical.

In the below, let n be the number of nodes and m be the number of edges, as is standard. Since we will always resolve any deadlock (cycle in the waits-for graph) immediately, deadlock detection amounts to determining if, after adding the most recent node, the waits-for graph ceases to be a DAG. As it is typical in query plan graphs found in practice that the degrees of individual nodes are quite small, we focus on a dynamic method that exploits this fact. Our machine model will be a pointer machine capable of constant-time array accesses.[1]

We dynamically maintain n *undirected* connectivity graphs, one for each vertex v. Each graph contains all edges that can be traversed starting at v. We maintain an array A of pointers to these graphs, and each graph will have an n-length array B associated with it, where $B[i]$ points to where the ith node v_i is located in the graph (or nil, if v_i is not present).

When a new node v is added to an existing graph of n nodes, we perform node updates for each appropriate connectivity graph. Suppose we want to add a node u of total degree d, with incoming edges from $w_1, \ldots, w_k$ and outgoing edges $v_1, \ldots, v_\ell$. We add the edge $\{w_i, u\}$ to all connectivity graphs containing w_i. Assuming each graph keeps an n bit vector saying which nodes are in it and pointers to them, this takes $O(n)$ time, roughly constant time per graph. Then we build the connectivity graph for u with each $\{u, v_i\}$, appending copies of the connectivity graphs for each v_i at the end of each edge. This takes $O(\ell \cdot m)$ time. (Note, if we have constant degree per node, this is $O(n)$.)

To delete a node u of total degree d, we just remove u's connectivity graph from the list, and locate u in every connectivity graph, removing its edges. This takes $O(n)$ time, using the above pointer machine assumptions.

Notice if we ever add a node v to its own connectivity graph, this is equivalent to a cycle in $\mathsf{Waits}(G)$. Hence we will report deadlock and immediately resolve it, using the algorithm in the following subsection. Therefore, deadlock detection is solvable in $O(n)$ time by simply checking each node's vector. The below summarizes this subsection.

Theorem 4.1 *Using the above data structures,* $\mathsf{Waits}(G)$ *can be checked for cycles in* $O(n)$ *time. Assuming* $\mathsf{Waits}(G)$ *is a DAG, it can be maintained in* $O(dm)$ *worst-case time per update, where* d *is the total degree of the node being inserted/deleted.*

When the maximum total degree of G is constant (or even $d \in n^{o(1)}$), the update time of $O(m)$ is the fastest known. An alternative solution to deadlock detection uses a tool only recently developed in the theory community and uses only slightly more time. In [11], it is shown how to maintain strongly connected components in a directed graph with $O(m\alpha(m, n))$ *amortized* time per node update (where α is related to an inverse Ackermann function), and $O(1)$ worst-case time per query. This immediately leads to the following.

Theorem 4.2 *Assuming* $\mathsf{Waits}(G)$ *is a DAG, it can be maintained in* $O(m\alpha(m, n))$ *time per update until a cycle forms.*

[1] If the reader does not believe in constant-time array accesses, then he or she should mentally apply a $\log n$ factor, or simply replace O in all of the above with $\tilde{O}$.

Proof. After adding a set of edges $\{(u_i, v_i)\}$ incident to a node, immediately query if one of $\{u_i, v_i\}$ are strongly connected, for all i. The query is *yes* if and only if a cycle has formed in $\mathsf{Waits}(G)$. □

However, notice that in the case of constant degree, our method is still faster than this one.

4.2 Deadlock Resolution Via Vertex Cuts

As mentioned earlier, one resolves deadlocks in pipelined query graphs by materializing problematic nodes. Every materialization incurs a possibly significant cost; hence we must be careful how we choose what to materialize. Performed off-line (with no buffer information), we have seen that the problem of materializing a minimum set of nodes is NP-hard. The key idea of our dynamic materialization strategy is simply that when a cycle arises in $\mathsf{Waits}(G)$ after vertex v is added, every cycle in the graph must contain v. This sufficiently restricts the optimal resolution problem so that it may be efficiently solved, even when materialization costs are taken into account.

First, observe that it only makes sense to materialize certain types of nodes in a given Buffered Plan-DAG G. Materializing a node that either has all empty input buffers or no full output buffers will not break any existing deadlock: in the first case the node has literally nothing to materialize, and in the second case the node has nothing to wait on to produce its output. Let us say the nodes v of G with some full output buffer ($(v, w) \in E$ with $s(v, w) = \mathbf{F}$) and some non-empty input buffers ($(u, v) \in E$ with $s(u, v) \neq \mathbf{E}$) are *materialization candidates*. This set obviously changes over time, but can be easily be maintained by occasional buffer monitoring.

We assume for each node v that there is a materialization write cost $w(v)$ and a materialization read cost $r(v)$. As in previous work, the $r(v)$ is assumed to be linear in the outdegree of v in the Plan-DAG, *i.e.* $r(v) = a \cdot outdeg(v)$ for some constant $a > 0$. Below, we just think of these costs as arbitrary weights on the nodes.

When a node v is inserted in a (previously acyclic) waits-for graph that induces cycles in the graph, one naïve solution is to simply materialize v. However, the read and write costs of v may be enormous, so this is not necessarily the best plan at all. In general, we want to *find a minimum (materialization) cost subset of the materialization candidates, such that all induced deadlock cycles are broken in the waits-for graph*. Formally, we have (a) an arbitrary weighted directed graph G, (b) a distinguished subset S of vertices, and (c) a distinguished vertex v for which we know that, if we remove v, G will become acyclic. Our task is to find $S' \subseteq S$ such that S' is of minimum cost (under the appropriate weight function) and G without S' is acyclic.

We will give a polynomial time (quadratic) algorithm for this task, by showing it is actually tantamount to finding a minimum vertex cut in an undirected graph G', built from the waits-for graph.

4.3 Minimum Materialization in Quadratic or Linear Time

We reduce the above problem on a directed waits-for graph $G = (V, E)$ to one on an undirected graph $G' = (V', E')$. Let v be the distinguished vertex of G.

Consider the undirected connectivity graph of v (from Section 4.1) at this point: it has two occurrences of v, one on each "end" of it. To avoid confusion, let us distinguish these endpoints and call them s and t. We "bypass" the nodes that are not materialization candidates (say one of them is w) by placing edges from the nodes with edges to w, to the nodes in the connectivity graph to which w points in $\mathsf{Waits}(G)$. We have now reduced the optimal materialization problem to finding a minimum vertex cut between s and t in a weighted, undirected graph. This cut problem is known to take quadratic time to solve optimally, in the worst case.

Theorem 4.3 *(Henzinger* et al. *[9]) A minimum vertex cut in an weighted undirected n-node graph can be found in $O(n^2 + m)$ time.*

(Note there are no known lower bounds on the problem, this is just the current best algorithm known.) However, if the minimum vertex cut is at most three nodes, the problem can be solved very quickly. This is significant to our purpose, since in general we expect the total degrees of nodes to be small.

Theorem 4.4 *(Hopcroft and Tarjan [10]) If the minimum vertex cut in a weighted undirected graph is at most three nodes, then it can be found in $O(m + n)$ time.*

Therefore, we expect that specialized algorithms for the pipelined application at hand could possibly be very efficient at deadlock resolution.

4.4 Incorporating the Deadlock Resolver in a Query Engine

An important consideration when evaluating the deadlock detection and resolution scheme is the complexity of modifications that need to be made to a query execution engine, in order to support the scheme. We also want to ensure that the collection of runtime statistics required by the deadlock resolver to find an optimal vertex cut in the *waits-for graph* does not require a significant amount of memory or affect query execution times. Here we sketch a small set of modifications that need to be made to an execution engine to implement our detection and resolving scheme.

As described in Section 4.1, the execution engine needs to maintain a *waits-for graph* for a set of currently executing queries, and dynamically update the graph as tuple buffers became full, empty, or non-empty. This can easily be accomplished by changing the buffer's Insert() and Remove() routines to incorporate maintenance code for connectivity trees. Detecting full buffers is a potentially challenging task in execution engines that rely on a blocking Insert() implementation. In that case, we would need to change operators to use non-blocking interfaces for tuple operations.

In addition to connectivity trees for deadlock detection, we also have to maintain some runtime statistics for computing the materialization costs of query nodes. This can be done by monitoring the selectivities of every node in the query graph. Selectivity monitoring does not incur any additional runtime overhead, and results in a deadlock resolver with accurate materialization costs.

5 Experimental Evaluation

We evaluated the potential benefits of our dynamic materialization strategy on two workloads that can take advantage of query subplan sharing. The first workload uses a synthetic database based on the Wisconsin benchmark specification [4]. This benchmark suite specifies a simple schema with two large tables and a smaller one, and relatively simple select, project, join, and aggregate queries. We used 15,000,000 200-byte tuple tables for the big tables (*big1* and *big2* in the experiments) and 1,500,000 200-byte tuples for the small table. We only used queries 15 and 17, which have a form of 2 and 3-way joins with shared access to two large tables. The second experiment was conducted using the industry standard decision support benchmark TPC-H [13] with scale factor 10 (database size - 10Gb). We chose two queries (Q4 and Q10) with shared scans on a large LINEITEM table and a moderate ORDERS table.

Since we currently do not have a database engine that implements both a conservative static and an optimistic dynamic scheme, we relied on Microsoft SQL server's EXPLAIN facility for runtime estimates of the submitted queries. The SQL Server optimizer uses sophisticated cost models which are more accurate

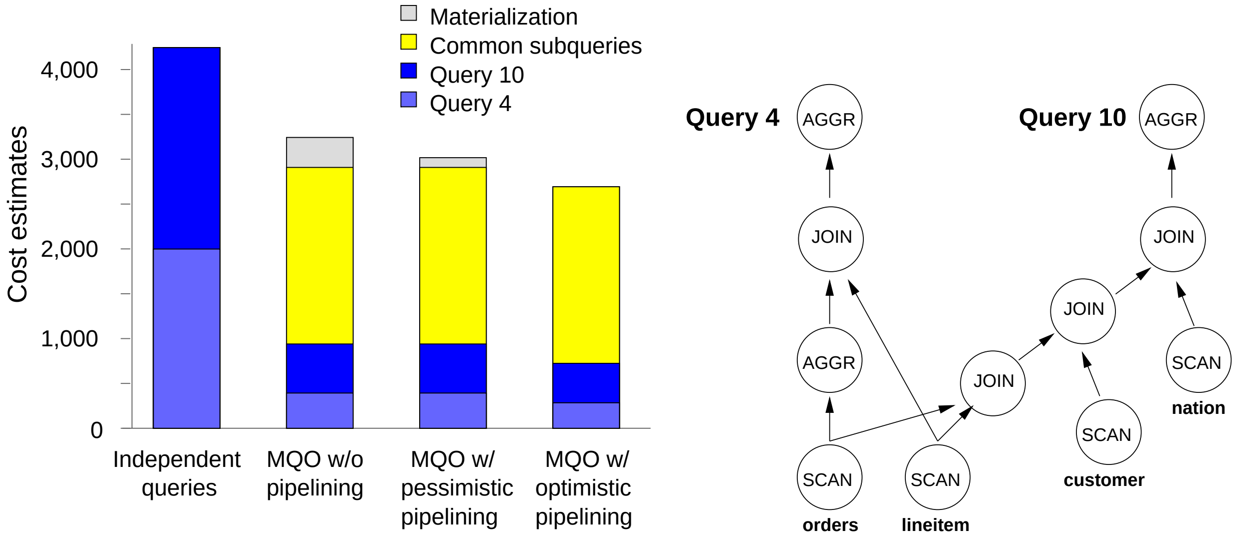

Figure 3: **Experiment 1: TPC-H.**

for estimating relative query costs than the simplistic cost models largely used in MQO literature. Since SQL Server does not support multi-query optimization, we mimic it by manually materializing common subexpressions and appropriately adjusting the query costs to reflect materialization or multiple reads of shared results. Instead of physically materializing their shared subplans, we create corresponding entries in the system catalogs, to trick the optimizer into thinking that materialized plans actually exist.

All experiments were conducted on a Dual Pentium 3 900Mhz SMP machine with 1GB of RAM running SQL Server 2000 on Windows Server 2003.

5.1 Experiment 1 - TPC-H

The first experiment compares the potential savings obtained by employing multi-query optimization, the relative costs of common result computation, and materialization. We used TPC-H queries Q4 and Q10, which share scans on the LINEITEM and ORDERS relations. The merged query plan graph is shown in Figure 3. It's easy to see that the Plan-DAG contains a C-cycle, and that static deadlock analysis will need to materialize all edges coming out of one of the relations. Since the scan on LINEITEM is significantly more expensive than the scan on ORDERS, the greedy static algorithm will choose to pipeline the results of the scan on LINEITEM and will be forced to materialize the scan on ORDERS. However, it is easy to establish by visual query inspection that the consumption rates for the ORDERS and LINEITEM relations are not radically different for different consumers, so an optimistic scheme that pipelines everything will never reach deadlock.

The cost estimates for running the batch of 2 queries are shown in the bar-graph of Figure 3. The Y-axis indicates units of time used by SQL Server. This graph demonstrates that even the full-materialization scheme is fairly efficient: it allows us to reduce the overall time for batch execution to 76% of the scenario when no MQO is employed. Applying the conservative static pipelining algorithm reduces the execution time to 60% by avoiding materialization and incurring read costs on the ORDERS relation. Our aggressive approach further removes any materialization costs and shared read costs by using a fully pipelined Plan-DAG, and further reduces the total running time to 52.8%.

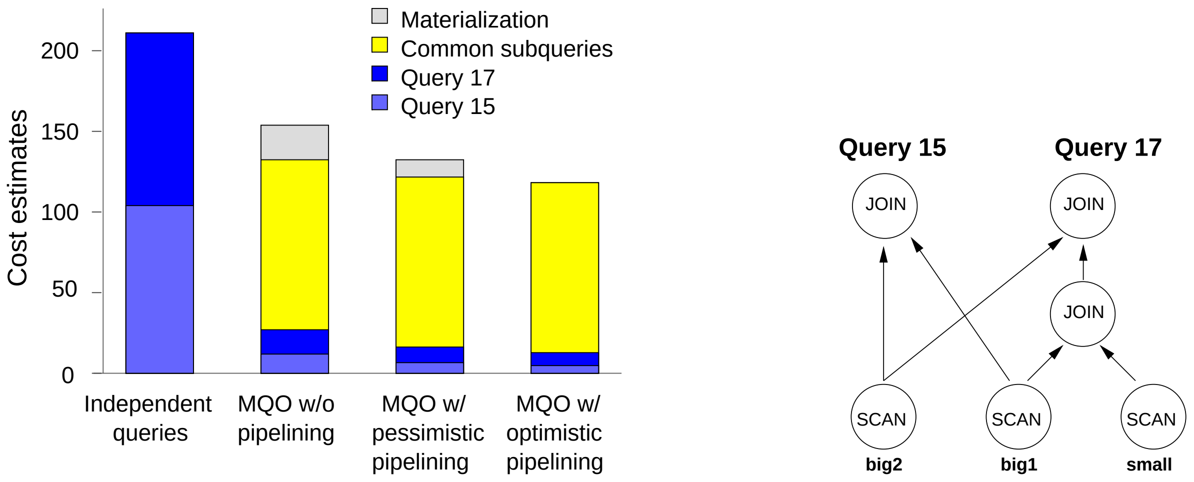

Figure 4: **Experiment 2: Wisconsin.**

5.2 Experiment 2 - Wisconsin Benchmark

For the second experiment we used two join queries (Q15 and Q17) from the Wisconsin benchmark suite. The merged query plan graph is shown in Figure 4. Q15 involves a two-way join between two large tables *big1* and *big2*; Q17 in addition joins the result with relation *small*. It is not possible to directly share the results of *big1* and *big2*, since the selection predicates for scans on *big1* and *big2* are different for the two queries.

Similarly to the queries used in Experiment 1, the Plan-DAG for these queries also contains a C-cycle. The two potential candidates that static deadlock analysis will consider for materialization have identical read costs, so either one of them will be materialized to break the potential deadlock. Again, it is easy to convince ourselves that all the consumers consume the results of the shared scans at about the same rate, so the optimistic dynamic approach will be beneficent for this query.

The cost estimates for running the batch of 2 queries are shown in the bar-graph of Figure 4. The Y-axis here indicates the time units used by SQL Server. In this graph, even the full-materialization scheme is fairly efficient, and allows us to reduce the overall time for batch execution to 73% of the scenario when no MQO is employed. Applying the static pipelining algorithm reduces the execution time to 63% by avoiding the materialization cost of the *big1* relation. Our aggressive approach results in a fully pipelined Plan-DAG, and further reduces the total running time to 56%.

6 Conclusion

We proposed a new model that characterizes deadlock in query plan DAGs in terms of the state of buffers used for inter-operator tuple transfer. Minimum cost deadlock prevention at optimization time is NP-hard, and even if one solves that problem, the materialization choices might still be suboptimal. We propose pipelining as much as possible in the query plan, avoiding unnecessary materialization costs but facing possible runtime deadlock. We proposed efficient methods for detecting and resolving deadlock at run-time, where the resolution algorithm minimizes the cost required to resolve a given deadlocked scenario. Experiments indicate that our strategy safely eliminates unnecessary materialization costs in queries with commonalities.

References

[1] C. Cook. Database Architecture: The Storage Engine. Miscrosoft SQL Server 2000 Technical Article, July 2001. Available at: http://msdn.microsoft.com/library/

[2] N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. J. Comput. Syst. Sci. 66(4):728–762 2003.

[3] C. Demetrescu and G. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In Proc. IEEE Foundations of Computer Science, 381-389, 2000.

[4] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future. The Benchmark Handbook, J. Gray, ed., Morgan Kaufmann Pub., San Mateo, CA, 1991.

[5] P. M. Fernandez. Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms. In Proc. ACM SIGMOD International Conference on Management of Data, 1994.

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

[7] P.V. Hall. Common subexpression identification in general algebraic systems. Technical Report UKSC 0060, IBM United Kingdom Scientific Centre, Nov. 1974.

[8] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. To appear in Proc. ACM SIGMOD International Conference on Management of Data, 2005.

[9] M. R. Henzinger, S. Rao and H. N. Gabow, Computing vertex connectivity: new bounds from old techniques, Proc. 37th IEEE Foundations of Computer Science, 462–471, 1996.

[10] J. Hopcroft, R.E. Tarjan, Dividing a graph into triconnected components. SIAM J. Comput. 2:135–158, 1973.

[11] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In Proc. of ACM Symposium on Theory of Computing, 184–191, 2004.

[12] T. K. Sellis. Multiple-Query Optimization. ACM Trans. Database Syst. 13(1):23–52, 1988.

[13] Transaction Processing Performance Council. TPC-H Benchmark Specification. Published at http://www.tcp.org/hspec.html.