Optimizing Complex Queries with Multiple Relation Instances

Yu Cao   Gopal C. Das   Chee-Yong Chan   Kian-Lee Tan
School of Computing
National University of Singapore
{caoyu, gopal, chancy, tankl}@comp.nus.edu.sg

ABSTRACT

Today's query processing engines do not take advantage of the multiple occurrences of a relation in a query to improve performance. Instead, each instance is treated as a distinct relation and has its own independent table access method. In this paper, we present MAPLE, a *M*ulti-instance-*A*ware *PL*an *E*valuation engine that enables multiple instances of a relation to share one physical scan (called *SharedScan*) with limited buffer space. During execution, as *SharedScan* pulls a tuple for *any* instance, that tuple is also pushed to the buffers of other instances with matching predicates. To avoid buffer overflow, a novel *interleaved* execution strategy is proposed: whenever an instance's buffer becomes full, the execution is temporarily switched to a *drainer* (an ancestor blocking operator of the instance) to consume all the tuples in the buffer. Thus, the execution is interleaved between normal processing and drainers. We also propose a cost-based approach to generate a plan to maximize the shared scan benefit as well as to avoid interleaved execution deadlocks. MAPLE is light-weight and can be easily integrated into existing RDBMS executors. We have implemented MAPLE in PostgreSQL, and our experimental study on the TPC-DS benchmark shows significant reduction in execution time.

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems - Query processing

General Terms
Algorithms, Design, Performance

Keywords
shared scan, interleaved execution, query optimization, query processing

1. INTRODUCTION

Many applications use relational DBMSs (RDBMSs) as their data solutions to manage massive amount of data. In these applications, it is not uncommon for a single query to contain relations with multiple instances. For example, in traditional business oriented applications, a query over multiple views typically leads to multiple relational instances of the base tables as a result of view unfolding. In decision-support systems, such multi-instance queries are frequently posed: among the 99 queries in the TPC-DS benchmark [3], more than 60% of them contain at least one relation with multiple instances; the maximum number of instances for a relation is 8 (e.g., Q11 and Q88) and the maximum number of relations with multiple instances is 15 (e.g., Q78). Even in non-traditional applications such as web data management that stores XML (RDF) data in relational DBMSs, XPath/XQuery (SPARQL/RDQL) queries, when converted to SQL, comprise many self-joins [9, 5, 16].

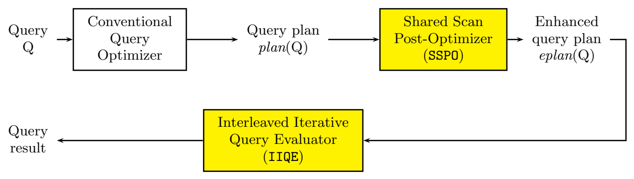
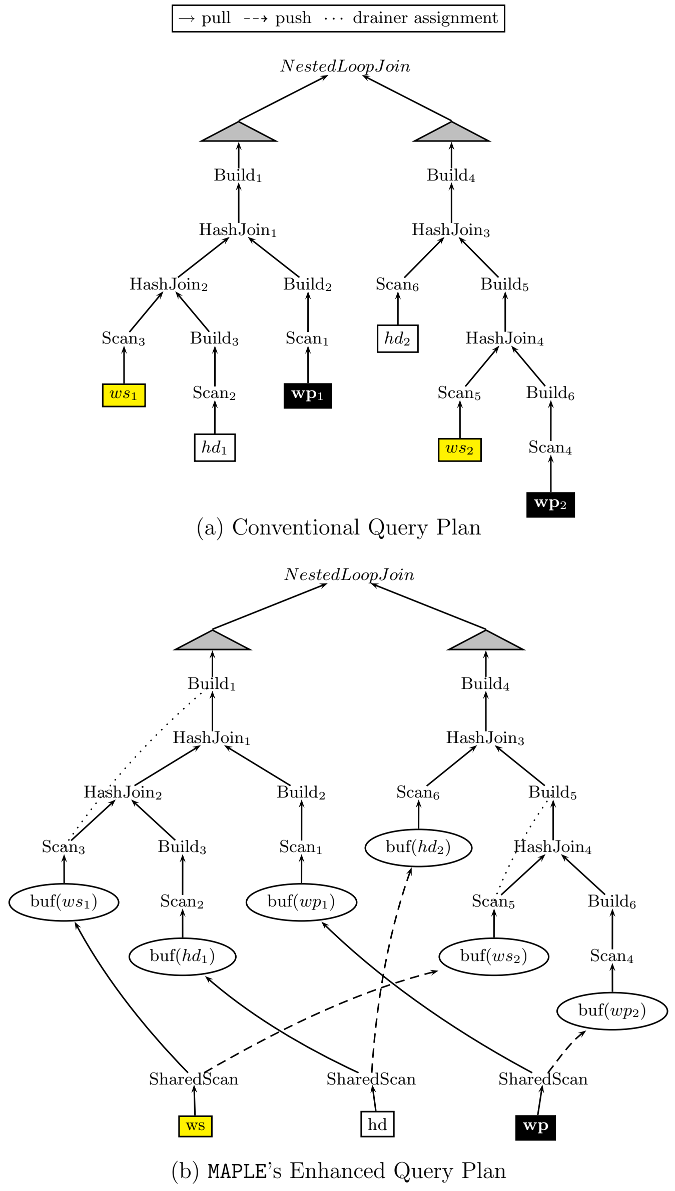

Query Q
Conventional Query Optimizer
Query plan plan(Q)
Shared Scan Post-Optimizer (SSPO)
Enhanced query plan eplan(Q)
Interleaved Iterative Query Evaluator (IIQE)
Query result

Figure 1: Architecture of MAPLE

Surprisingly, most of today's relational query engines do not explicitly recognize instances within queries during query optimization and/or evaluation. Instead, each instance is treated as a distinct relation and has its own independent access method (table or index scan). As such, the performance can be very bad even for an optimal plan especially when the relation with multiple occurrences is a large table.

While there have been some efforts to optimize multiple scans on the same table to minimize disk I/O cost, these works are limited in scope. In [1, 6, 10, 11, 12, 19], scans are coordinated for better buffer reuse (increasing buffer locality). In particular, the data-sharing opportunity arises mainly among scans from different queries running at the same time. The performance improvement is achieved by exhaustively exploiting the knowledge of query access patterns and carefully scheduling query executions. However, for a single query with multiple relational instances, it is not possible to synchronize the disk access patterns under the pull iterative execution model. As such, single multi-instance queries do not benefit much from these buffer reuse methods. Works in [7, 17] look at facilitating sharing of a single scan on the base relations at the operator level. However, these works are targeted at pipelining table tuples to consumers in different SQL [7] (OLAP [17]) queries handled by independent threads. Instances within a single query have, as we shall see, certain characteristics that these methods fail to accommodate. Yet another approach is to employ multi-query optimization (MQO) schemes (e.g., [14, 18]) to exploit common subexpressions in queries. However, MQO does not further optimize multiple scans on the materialized views of common subexpressions, which can be considered as base relations with multiple instances. Moreover, these techniques do not handle instances that are not part of the common subexpressions.



(a) Conventional Query Plan

(b) MAPLE's Enhanced Query Plan

Figure 2: Partial Query Evaluation Plans for Query Q90 in TPC-DS Benchmark

In this paper, we present MAPLE, a *M*ulti-instance-*A*ware *PL*an *E*valuation engine that takes advantage of multiple instances in single queries to reduce disk I/O cost. MAPLE comprises two key components (SSPO and IIQE) as shown in Fig. 1. First, a *shared scan post-optimizer (*SSPO*)* builds on a query evaluation plan (generated by any existing query optimizer) to produce an enhanced plan as follows. The SSPO opportunistically adds new materialize operators when required and bundles multiple instances of a relation into *share groups* such that instances within a group share one physical table scan (called SharedScan). For each instance of a relation that employs a SharedScan operator, it is allocated a *small* buffer. Moreover, for each instance with buffer overflow risk, an ancestor (blocking) operator in the query plan will be designated as its *drainer*. Second, an *interleaved iterative query evaluator* (IIQE) is used to execute the enhanced query plan produced by SSPO. IIQE adopts an *interleaved* pull iterative execution strategy to ensure that each SharedScan operator scans the table *only once* (for all instances within the same share group). Essentially, within a share group, as SharedScan pulls a tuple for *any* instance, that tuple is also *pushed* to other instances with matching predicates and placed in their buffers for later use. Whenever a buffer becomes full, the corresponding drainer becomes *active*. At this moment, query processing is temporarily switched to this drainer until it consumes all tuples in the buffer. Thus, query processing is interleaved between normal processing and active drainers.

Example 1.1 Fig. 2(a) shows the partial evaluation plan of Q90 in TPC-DS benchmark, generated by PostgreSQL [2]. Q90 contains two instances ws_1 and ws_2 for relation web_sales (denoted by ws), two instances wp_1 and wp_2 for relation web_page (denoted by wp), and two instances hd_1 and hd_2 for relation household_demographics (denoted by hd). Here the hash operator Build is used to build hash table in hash join. The plan tree contains one hash subtree in each side of the top nested-loop join and all instances are accessed by table scans.

MAPLE generates an enhanced plan, shown in Fig. 2(b), with three share groups: $\{ws_1, ws_2\}$, $\{wp_1, wp_2\}$ and $\{hd_1, hd_2\}$. No additional materialize operators are introduced. Each relation instance r_i is now associated with a buffer $buf(r_i)$ for storing the tuples pushed by the SharedScan operator. Under the iterative model, the execution starts from $\texttt{Build}_1$. Since both wp and hd are small tables, the shared scans on them did not incur buffer overflows in wp_2 and hd_2. However, when ws_1 calls its SharedScan, matching tuples pushed to ws_2 will fill up its buffer since ws is a very large table. Now, whenever $buf(ws_2)$ becomes full, the execution temporarily switches to $\texttt{Build}_5$, ws_2's drainer, which consumes all tuples in the buffer to partially construct the hash table, and then switches back to ws_1. The switched execution for ws_2 will complete the normal execution of $\texttt{Build}_6$ using cached tuples in $buf(wp_2)$. Finally, as all three shared scans finish, the remaining execution continues as in the traditional iterative model from $\texttt{Build}_4$ (which completes the execution of $\texttt{Build}_5$ and then conducts the hash join by probing the hash table with the cached tuples in $buf(hd_2)$).

As illustrated, by using MAPLE, one share group reads the relation only once from the disk. In this example, we save one full scan on each ws, wp and hd. Our experimental results show significant benefit from the saving of one scan of ws since it is huge (1.5GB in 10GB TPC-DS dataset). On the contrary, the CPU overhead of execution switches is negligible. Intermediate results of execution switches are naturally consumed by the Build drainers without incurring additional I/O overhead. □

The key task of SSPO is to generate an enhanced plan that maximizes the benefits of SharedScan. Ideally, all instances of a relation should be grouped within a single share group without introducing any additional blocking operators. However, it turns out that this is not always possible due to several reasons (e.g., interleaved execution deadlocks). In this case, SSPO aims at finding a feasible shareable scan plan with maximum performance benefit.

MAPLE is light-weight and can be easily integrated into existing RDBMSs. We have prototyped our ideas in PostgreSQL. Our extensive performance study on the TPC-DS benchmark shows very significant reduction in execution time of up to 70% for some queries.

The rest of this paper is organized as follows. In Section 2, we present an overview of our MAPLE approach. Section 3 describes the shared scan post-optimizer. In Section 4, we present how to integrate IIQE into existing query executors. Section 5 presents results of an extensive performance study. Section 6 reviews related work, and finally, Section 7 concludes the paper and discusses directions for future work.

2. OVERVIEW OF MAPLE

In this section, we present an overview of our light-weight optimization approach named MAPLE.

We use *plan*(Q) to denote a query evaluation plan for Q generated by a conventional query optimizer, and use *eplan*(Q) to denote an enhanced query evaluation plan for Q produced by MAPLE based on *plan*(Q).

A query plan operator is classified as a *blocking operator* if it needs to completely consume its operand(s) before producing any output (e.g., sorting, building hash table, aggregation); otherwise, it is a *non-blocking operator* (e.g., scan, merge-join).

Given a multi-instance relation R in Q with n instances, $n > 1$, we use $G = \{r_1, r_2, \cdots, r_n\}$ to denote the instances of R.

2.1 Share Groups & Shared Scans

In contrast to the conventional pull-iterative execution engine [8], where the scans of instances of the same relation are performed independently, MAPLE tries to maximize the sharing of relation scans by partitioning the set of instances of a relation into a small number of subsets called *share groups*. Each relation instance r_i in a share group is allocated some small memory space, denoted by $buf(r_i)$, to hold the qualified tuples that satisfied the selection predicates for the scan of r_i. Each share group is associated with a new scan operator called the SharedScan operator[1] that can be invoked by any instance in that group. When a scan of an instance r_i is invoked, MAPLE will first check whether $buf(r_i)$ is empty. If a tuple is available in $buf(r_i)$, the scan of r_i will simply remove this tuple from $buf(r_i)$ and pass it to the scan's parent operator. However, if $buf(r_i)$ is empty, the scan of r_i will invoke the SharedScan operator for its share group. Besides pulling the qualified tuples for r_i into $buf(r_i)$, the SharedScan operator will also push qualified tuples for other instances r_j within the share group into their buffers $buf(r_j)$ as well. For space efficiency, the tuples stored in each $buf(r_i)$ only keep the relevant attributes of R for the scan of r_i[2].

In the ideal scenario, the tuples in each $buf(r_i)$ are consumed in a timely manner without causing any buffer overflows. However, in general, a shared scan can become *blocked* when the SharedScan operator (invoked by some other instance r_j in the same share group as r_i) tries to push qualified tuples into a full buffer $buf(r_i)$. In this case, we say that r_i is an *overflow instance* and $buf(r_i)$ *overflows*.

A naive approach to fix a blocked shared scan (under the iterative execution model) is to adopt a *drop-out scheme*, where the overflow instance r_i is dropped out of the shared scan of R, and the shared scan of R is allowed to continue among the remaining non-overflow instances of R within the share group. However, this scheme requires a separate partial scan of R to be initiated later to retrieve the remaining non-buffered qualified tuples for the overflow instance r_i, thereby limiting its effectiveness.

Note that if there is only one instance r_i in a group, the scan for r_i is not shared with any other instances of R; therefore, $buf(r_i)$ is not allocated and SharedScan is not used for this group.

2.2 Interleaved Executions with Drainers

MAPLE adopts a more aggressive approach to resolve blocked shared scans. Consider a shared scan invoked by r_i that becomes blocked due to the overflow of $buf(r_j)$. Instead of dropping r_j out of the shared scan of R, MAPLE tries to "unblock" the shared scan by suspending the execution of the scan and switching the execution control to another operator, called the *drainer* of r_j, denoted by $drainer(r_j)$. $drainer(r_j)$ is an ancestor of r_j, whose execution will result in "draining" the tuples from the full buffer $buf(r_j)$. Once all the tuples in $buf(r_j)$ have been consumed (i.e., $buf(r_j)$ becomes empty), the suspended shared scan of R becomes unblocked and can be resumed by r_i. It is possible for nested execution control switches to occur, where the execution of the query subplan under a drainer operator causes another execution control switch to another drainer, and so on. We refer to the enhanced iterative execution model used by MAPLE as *interleaved iterative execution*.

2.2.1 Drainer Operators

When $buf(r_j)$ overflows during a shared scan that is invoked by another instance r_i, MAPLE will try to switch execution to a drainer operator, $drainer(r_j)$, to clear the buffer $buf(r_j)$. Thus, $drainer(r_j)$ must necessarily be an ancestor operator of r_j in the query plan so that the scan of r_j will get evaluated as part of the evaluation of the subquery plan rooted at $drainer(r_j)$.

Consider the scenario where all the ancestor operators of r_j up to and including $drainer(r_j)$ are non-blocking operators. In this case, any tuple produced by the evaluation of $drainer(r_j)$ has to be either cached (possibly incurring disk I/O) or returned to the parent operator of $drainer(r_j)$. The latter option is not possible (under the iterative execution model) since the execution control is passed to $drainer(r_j)$ and not to its parent operator. To avoid incurring unnecessary disk I/O for caching output tuples from $drainer(r_j)$, it makes sense to assign a blocking operator as a drainer. In this way, the evaluation of the blocking drainer will not generate any output tuple until its entire query subplan has been completely evaluated. To minimize the number of op-

[1]Currently, MAPLE considers shared scans only for table scans.

[2]An alternative buffering scheme is to have a single buffer shared among all instances within the share group. But this not only requires storing the entire tuple (in general), but also involves a more elaborate tracking of the tuples that are qualified for each instance scan.

erator evaluations for draining $buf(r_j)$, `MAPLE` chooses the *closest* ancestor blocking operator of r_j as its drainer.

Clearly, a drainer operator does not always exist for an overflow instance. We can classify an overflow instance as a *drainable instance* if it has an ancestor blocking operator in the query plan; otherwise, the overflow instance is considered to be *non-drainable*.

Since a drainer operator cannot be assigned for a non-drainable instance r_j, it is not possible to drain $buf(r_j)$ (if it becomes full) via an interleaved execution. Thus, non-drainer instances cannot participate in shared scans (i.e, a separate physical scan is necessary for each non-drainable instance). However, a non-drainable instance r_j can be made drainable by inserting an explicit materialize operator *op* in the query plan such that *op* becomes an ancestor operator of r_j (i.e., $drainer(r_j) = op$).

Consider the example in Fig. 2(b), where ws_1 and ws_2 are assumed to be overflow instances, the drainer assignment for each overflow instance r_j is indicated by a dotted line between $scan(r_j)$ and $drainer(r_j)$.

2.2.2 Deadlock-free Interleaved Execution

To maximize shared scans, an ideal query plan is to have a single share group for each distinct multi-instance relation R that contains all its instances. In this way, only a single physical scan of R is required to scan all its instances. However, this is not always feasible due to two reasons: (1) the existence of non-drainable instances; and (2) the existence of *interleaved execution deadlocks*.

Basically, an interleaved execution deadlock arises whenever an interleaved execution that is triggered to drain a full buffer $buf(r_j)$ eventually leads to more tuples being pushed into $buf(r_j)$. The following example illustrates a simple example of an execution deadlock.

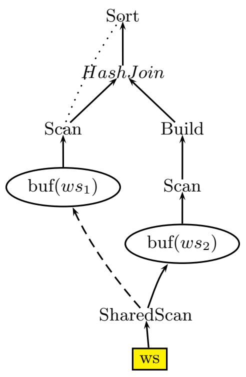


Figure 3: Simple Execution Deadlock

Example 2.1 Fig. 3 shows a self-join between two instances ws_1 and ws_2 of the relation *web_sales* in TPC-DS, where ws_1 is an overflow instance sharing a scan with ws_2. The execution starts with the scan of ws_2. During the scan of ws_2, $buf(ws_1)$ will become full and the execution will be switched to $drainer(ws_1)$, which is the *Sort* operator. However, since the hash table has not been completely constructed yet, before the tuples from ws_1 can be processed, it is necessary to complete the scan of ws_2. But since $buf(ws_1)$ is already full, the execution is deadlocked. □

The following example illustrates a more complex deadlock scenario.

Example 2.2 Consider again the Q90 query plan in Fig. 2(b). Suppose that hd_2 is now an overflow instance. The execution will start with $Build_1$. During the shared scan of hd_1 and hd_2, $buf(hd_2)$ becomes full and the execution switches to $Build_4$, which is the drainer for hd_2. This eventually triggers the execution of the scan of ws_2 and hence a shared scan of ws_1 and ws_2 which results in $buf(ws_1)$ becoming full. Consequently, the execution now switches over to $Build_1$, which is the drainer for ws_1. Here, a deadlock occurs since both $buf(ws_1)$ and $buf(hd_2)$ are full but there are more tuples to be pushed into them. □

To generate a deadlock-free query plan that maximizes shared scans, `MAPLE` uses a cost-based approach to optimize both the usage of explicit materialize operators as well as the partitioning of share groups. Explicit materialize operators can be used not only to enable non-drainable instances to become drainable (and therefore allowing them to participate in shared scans) but also to avoid deadlock situations.

2.3 Architecture of MAPLE

Fig. 1 shows the architecture of `MAPLE` which consists of two components: the shared scan post-optimizer (`SSPO`) and the interleaved iterative query evaluator (`IIQE`).

An input query Q is optimized by `MAPLE` in two steps. First, a conventional query optimizer is used to generate a query evaluation plan (*plan*(Q)). Next, *plan*(Q) is used as input for `SSPO` to produce an enhanced query plan (*eplan*(Q)). An *eplan*(Q) enhances *plan*(Q) by using share groups, `SharedScan` operators, and possibly explicit materialize operators.

The generated *eplan*(Q) is then evaluated by the `IIQE` component which is a variant of the conventional iterative query execution engine enhanced to support shared scans as well as interleaved operator executions.

3. SHARED SCAN POST-OPTIMIZER

In this section, we describe how the *shared scan post-optimizer* (`SSPO`) component of `MAPLE` generates an enhanced query plan that supports shared scans and interleaved operator executions.

3.1 Overflow Instances

Since `SSPO` optimizes a query plan statically, it needs to estimate the potential for an instance r_i to overflow and assign a drainer to r_i if necessary. Specifically, for each instance r_i within a share group in the query plan, `SSPO` uses statistical information on R (to estimate the number of qualified tuples for the scan of r_i) as well as information about the allocated memory space for $buf(r_i)$ to decide whether r_i has the potential to overflow. If the total estimated qualified tuples for r_i cannot fit in $buf(r_i)$, r_i is considered to be an *overflow instance*, and `SSPO` then assigns $drainer(r_i)$ to be the closest ancestor blocking operator of r_i if r_i is drainable.

Consider an instance r_i that is determined by `SSPO` to be a non-overflow instance (i.e., no drainer has not assigned to r_i). If r_i actually overflows at runtime, then `MAPLE` has no choice but to dynamically materialize the contents of $buf(r_i)$.

3.2 Interleaved Execution Deadlocks

In this section, we provide a characterization of interleaved execution deadlocks in terms of *execution dependencies* and *overflow dependencies*.

3.2.1 Execution & Overflow Dependencies

Execution Dependencies. Whenever $buf(r_i)$ overflows during a shared scan and execution control switches to $drainer(r_i)$ which in turn causes the scan of some other relation instance s_j (where s_j is a descendant of $drainer(r_i)$) to be evaluated, we say that there is an *execution dependency* from r_i to s_j (denoted by $r_i \rightarrow s_j$). Here, r_i and s_j can be instances of the same relation or different relations. Note that execution dependencies are transitive: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. Moreover, if $a \rightarrow b$ and $b \rightarrow a$, then both $drainer(a)$ and $drainer(b)$ must be the same.

Overflow Dependencies. Consider two instances r_i and r_j within a share group. If $buf(r_j)$ becomes full during a shared scan invoked by r_i, we say that there is an *overflow dependency* from r_i to r_j (denoted by $r_i \dashrightarrow r_j$).

Instance Dependency Cycles. We can now characterize interleaved execution deadlocks in terms of execution and overflow dependencies. An interleaved execution deadlock occurs when there is an *instance dependency cycle* among a set of relation instances $\{r_1, s_2, t_3, \cdots, z_n\}$, $n > 1$, that consists of an alternating sequence of $\dashrightarrow$ and $\rightarrow$ dependencies of the form $r_1 \dashrightarrow s_2 \rightarrow t_3 \dashrightarrow \cdots \dashrightarrow z_n \rightarrow r_1$.

Observe that in Example 2.1, there is an instance dependency cycle $ws_2 \dashrightarrow ws_1 \rightarrow ws_2$; and in Example 2.2, there is an instance dependency cycle $hd_1 \dashrightarrow hd_2 \rightarrow ws_2 \dashrightarrow ws_1 \rightarrow hd_1$.

3.2.2 Eliminating Dependencies

The above characterization of interleaved execution deadlocks provides two ways to break deadlocks by eliminating overflow or execution dependencies. For an overflow dependency $r_i \dashrightarrow r_j$, which arises when a shared scan for a group containing r_i and r_j causes $buf(r_j)$ to overflow, the overflow dependency can be eliminated by separating r_i and r_j into two different share groups.

For an execution dependency $r_i \rightarrow s_j$, the dependency can be eliminated by introducing a materialize operator *op* into the query plan such that *op* becomes the closest ancestor blocking operator for r_i (i.e., *op* is a descendant of $drainer(r_i)$) and s_j is outside of the query subtree rooted *op*. In this way, $drainer(r_i)$ becomes *op* and the evaluation of this new drainer for r_i will not cause the scan of s_j to be evaluated.

Example 3.1 Consider once more Example 2.2 in Fig. 2(b), where each distinct relation (i.e., *hd*, *wp*, and *ws*) has a single share group for all its instances, and hd_2 is an overflow instance. There is an execution deadlock in this plan due to the instance dependency cycle $hd_1 \dashrightarrow hd_2 \rightarrow ws_2 \dashrightarrow ws_1 \rightarrow hd_1$. The execution dependency $hd_2 \rightarrow ws_2$ can be eliminated by introducing a materialize operator above $Scan_6$ which will then become the new drainer for hd_2. The overflow dependency $hd_1 \dashrightarrow hd_2$ can be eliminated by separating hd_1 and hd_2 into two separate share groups.

3.2.3 Deadlock Avoidance

There are two approaches to handle interleaved execution deadlocks. The first is a dynamic approach that detects and breaks instance dependency cycles at run-time to resolve deadlocks. The second is a static approach that avoids deadlocks altogether by generating and processing only deadlock-free query plans. `MAPLE` adopts the simpler static approach as it provides a light-weight solution that can be easily integrated into existing query engines. We plan to explore the dynamic approach as part of our future work.

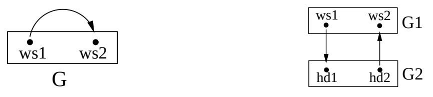
Due to the absence of run-time information on execution and overflow dependencies, the deadlock-free plans generated by a static approach are necessarily more conservative. Specifically, in `MAPLE`, if a relation instance r_i in a share group G is considered to be an overflow instance, then `MAPLE` will conservatively assume the following:

- for every other instance r_j in G, there is an overflow dependency $r_j \dashrightarrow r_i$; and
- if r_i is a drainable instance, then for every other instance s_j within the query subtree rooted at $drainer(r_i)$, there is an execution dependency $r_i \rightarrow s_j$.

Given the above conservative assumptions regarding execution and overflow dependencies, we can now generalize the notion of instance execution dependencies to derive a simpler and "higher level" characterization of interleaved execution deadlocks in terms of *group execution dependencies*.

Group Execution Dependencies. Consider two share groups G_1 and G_2. We say that there is a *group execution dependency* from G_1 to G_2, denoted by $G_1 \rightarrow G_2$, if there is an instance x in G_1 and an instance y in G_2 such that $x \rightarrow y$. We refer to x and y as *participants* of the group execution dependency $G_1 \rightarrow G_2$. Note that G_1 and G_2 are not necessarily distinct.

Group Dependency Cycles. We say that there is a *group dependency cycle* among a set of share groups $\{G_1, \cdots, G_n\}$, $n \geq 1$, if there is a cycle of group dependencies $G_1 \rightarrow G_2 \rightarrow \cdots \rightarrow G_n \rightarrow G_1$ such that for each G_i, $i \in [1, n]$, the two participants of the two group execution dependencies involving G_i are distinct.



(a) Example 2.1 (b) Example 2.2

Figure 4: Examples of Group Dependency Cycles

Example 3.2 Consider the examples in Fig. 4, where instances within the same share group are boxed and the directed edges between instances represent instance execution dependencies. Fig. 4(a) represents the group dependency cycle in Example 2.1 formed within a single share group G (i.e., $G \rightarrow G$). Fig. 4(b) represents the group dependency cycle in Example 2.2 formed between share groups $G1$ and $G2$. □

Note that each group in a group dependency cycle must be involved in two group execution dependencies. For example, in Fig. 4(b), we have $G_1 \rightarrow G_2$ and $G_2 \rightarrow G_1$. Moreover, the two participants in each group must necessarily be distinct; otherwise, it would imply that a shared scan that is invoked by the scan of an instance r_i causes its own buffer $buf(r_i)$ to overflow, which is impossible.

The following results state a useful sufficient condition on deadlock-free interleaved executions based on the absence of group dependency cycles.

THEOREM 3.1. *If there are no group dependency cycles in a query plan P, then there are also no instance dependency cycles in P.*

COROLLARY 3.2. *If there are no group dependency cycles in a query plan P, then P is free of interleaved execution deadlocks.*

3.3 Enhanced Query Plan Optimization

In this section, we describe how SSPO generates an enhanced query plan *eplan*(Q) from the optimal query plan *plan*(Q) produced by a conventional optimizer such that *eplan*(Q) maximizes shared scans without any interleaved execution deadlocks. Specifically, an enhanced plan for *plan*(Q), denoted by *eplan*(Q) = (*plan*(Q), $\mathcal{G}$, $\mathcal{M}$), specifies two additional components:

1. a list of share groups $\mathcal{G} = \{G_1, \cdots, G_k\}$, where each G_i contains a subset of instances from the same relation, $\bigcup_{i=1}^{k} G_i$ is the set of all relation instances in Q, the G_i's in $\mathcal{G}$ are pairwise disjoint. Clearly, $\mathcal{G}$ must contain at least one group for each distinct multi-instance relation in Q, and the maximum number of share groups occurs when each group is a singleton (i.e., without any shared scans).

2. a set (possibly empty) of materialize operators $\mathcal{M} = \{M_1, \cdots, M_n\}$ to be added to *plan*(Q).

Following the discussion in Section 3.2.2, both $\mathcal{G}$ and $\mathcal{M}$ help to eliminate some dependencies, while $\mathcal{M}$ also serves to enable some non-drainable instances to become drainable.

For notational convenience, given an enhanced query plan P, we use $G(P)$ to refer to the share group list component of P, and use $M(P)$ to refer to the materialize operator set component of P.

Cost Model. We now explain the cost model used by SSPO to select an optimal enhanced plan. Let $\mathcal{R} = \{R_1, \cdots R_d\}$ denote the set of distinct multi-instance relations in query Q, and n_i denote the number of instances of R_i. Given the share group list $\mathcal{G}$, let g_i denote the number of groups in $\mathcal{G}$ that have instances of $R_i \in \mathcal{R}$. Thus, each $n_i > 1$ and each $g_i \geq 1$. In Example 1.1, we have $d = 3$, and $n_i = 2$, $g_i = 1$, $i \in [1, 3]$.

For each $R_i \in \mathcal{R}$, let $scanCost(R_i)$ denote the cost of a single complete scan of R_i. For each $M_i \in \mathcal{M}$, let $matCost(M_i)$ denote the materialization cost of M_i, which includes the cost of writing the intermediate results to disk and the cost of reading them back later. Given $M \subseteq \mathcal{M}$, we define $matCost(M) = \sum_{M_i \in M} matCost(M_i)$.

Let $cost(plan(Q))$ refer to the total cost of scanning each relation instance in *plan*(Q) independently; i.e.,

$$cost(plan(Q)) = \sum_{R_i \in \mathcal{R}} (scanCost(R_i) \times n_i) \qquad (1)$$

Let $cost(eplan(Q))$ refer to the sum of the total relation scan cost of $\mathcal{G}$ and the total materialization cost of $\mathcal{M}$ incurred by $eplan(Q)$; i.e.,

$$cost(eplan(Q)) = \sum_{R_i \in \mathcal{R}} (scanCost(R_i) \times g_i) + matCost(\mathcal{M}) \qquad (2)$$

The *benefit* of *eplan*(Q) over *plan*(Q), which measures the savings in the evaluation cost of using *eplan*(Q) instead of *plan*(Q), is given by

$$benefit(eplan(Q)) = cost(plan(Q)) - cost(eplan(Q)) \quad (3)$$

Ideal Enhanced Plan. Based on Equations (1) to (3), the upper bound for $benefit$ is given by $\sum_{R_i \in \mathcal{R}} (scanCost(R_i) \times (n_i - 1))$ which happens when *eplan*(Q) scans each distinct relation exactly once (i.e., there is exactly one share group for each distinct relation), and *eplan*(Q) does not incur any materialization cost (i.e., $\mathcal{M}$ is empty). We refer to such a *eplan*(Q) as an *ideal enhanced query plan*.

We can now state the query optimization problem for SSPO more formally as follows.

Enhanced Plan Optimization Problem. Given an optimal query plan *plan*(Q) produced by a conventional optimizer for a query Q, find an enhanced query plan *eplan*(Q) = (*plan*(Q), $\mathcal{G}$, $\mathcal{M}$) such that *eplan*(Q) is free of interleaved execution deadlocks and benefit(*eplan*(Q)) is maximized.

The above optimization problem is (not surprisingly) a difficult problem as indicated by the following result for a simplified version of the problem.

THEOREM 3.3. *Given* plan*(Q) and a set of materialize operators $\mathcal{M}$, the problem of finding a share group list $\mathcal{G}$ such that* eplan*(Q) = (*plan*(Q), $\mathcal{G}$, $\mathcal{M}$) is free of interleaved execution deadlocks and benefit(*eplan*(Q)) is maximized is NP-hard.*

3.4 Optimization Algorithm

Given the hardness of the enhanced plan optimization problem, SSPO uses a heuristic approach that is shown in Algorithm 1.

Consider a query Q consisting of d distinct multi-instance relations $R_1, \cdots, R_d$ with a query plan *plan*(Q). For each instance r_j of each R_i, SSPO first estimates whether r_j is an overflow instance and initializes the drainer for each drainable relation instance, $drainer(r_j)$, to be the closest ancestor blocking operator of r_j (steps 1 to 4).

Next, SSPO checks whether a deadlock-free ideal enhanced query plan exists for *plan*(Q) (steps 5 to 9). Recall that an ideal enhanced query plan has an "ideal" enhancement with an empty set of materialize operators and a share group list given by $\mathcal{G} = \{G_1, \cdots, G_d\}$, where each share group G_i contains all the instances of R_i except for non-drainable instances. If the set of group dependency cycles in $\mathcal{G}$, specified by C, is empty and all the overflow instances in *plan*(Q) are drainable, then the constructed plan P_{opt} is indeed a deadlock-free ideal enhanced plan, in which case SSPO returns P_{opt} and terminates.

If the constructed enhanced plan P_{opt} is not a deadlock-free ideal enhanced plan, SSPO then optimizes P_{opt} by refining its share group list $\mathcal{G}$ and/or adding materialize operators using a two-phases approach. In the first phase (steps 10 to 17), SSPO generates a collection of candidate materialize operator sets. In the second phase (steps 18 to 30), SSPO takes each candidate materialize operator set $\mathcal{M}$ to create a deadlock-free candidate enhanced plan P with $M(P) = \mathcal{M}$ and $G(P) = \mathcal{G}_{opt}$, where $\mathcal{G}_{opt}$ is an optimized refinement of $\mathcal{G}$ (w.r.t. $\mathcal{M}$). Among all the candidate enhanced plans generated, SSPO returns the plan with the maximum benefit as the optimized enhanced query plan.

The details of the two phases are presented in the rest of this section.

3.4.1 *Generating Materialize Operator Sets*

Useful Materialized Operator Sets. Let M_{all} denote the set of all possible materialize operators that can be inserted into *plan*(Q). Instead of generating all possible subsets of M_{all}, SSPO considers only candidate materialize operator

Algorithm 1 `Post-Optimizer`

```
Input: optimal plan $plan$(Q) for query $Q$
Output: enhanced query plan $eplan(Q)$
1: let $\mathcal{R}_{multi} = \{R_1, \cdots, R_d\}$ be the set of distinct multi-instance relations in $Q$
2: for each $R_i \in \mathcal{R}_{multi}$ do
3:     for each overflow instance $r_j$ of $R_i$ do
4:         initialize $drainer(r_j)$ if $r_j$ is drainable
5: let $\mathcal{G} = \{G_1, \cdots, G_d\}$, where each share group $G_i$ contains all instances of $R_i$ except for non-drainable instances
6: let $P_{opt} = (plan(\text{Q}),\ \mathcal{G},\ \emptyset)$
7: let $C$ be the set of group dependency cycles in $\mathcal{G}$
8: if $(C = \emptyset)$ and (every overflow instance is drainable) then
9:     return $P_{opt}$
10: let $M_{all}$ be the set of all possible materialize operators that can be inserted into $plan$(Q)
11: let $M_{drain} = \{M_i \in M_{all} \mid \ drainSet(M_i) \neq \emptyset\}$
12: let $\mathcal{S}_{drain}$ be the collection of all useful subsets of $M_{drain}$
13: let $M_{cycle} = \{M_i \in M_{all} \mid cycleSet^-(M_i, C) \neq \emptyset\}$
14: for each $\mathcal{M}_{drain} \in \mathcal{S}_{drain}$ do
15:     let $C' = C\ \cup\ cycleSet^+(\mathcal{M}_{drain}, C)$
16:     let $\mathcal{S}_{cycle}(\mathcal{M}_{drain})$ be the collection of all useful subsets of $M_{cycle}$ w.r.t $C'$
17: let $S = \{(\mathcal{M}_{drain}, \mathcal{M}_{cycle}) \mid \mathcal{M}_{drain} \in \mathcal{S}_{drain},\ \mathcal{M}_{cycle} \in \mathcal{S}_{cycle}(\mathcal{M}_{drain})\}$
18: initialize $P_{best} = (plan(\text{Q}), \emptyset, \emptyset)$
19: for each $(\mathcal{M}_{drain}, \mathcal{M}_{cycle}) \in S$ do
20:     for each instance $r_j \in drainSet(\mathcal{M}_{drain})$ do
21:         $drainer(r_j)$ = the closest ancestor operator of $r_j$ from $\mathcal{M}_{drain}\ \cup\ \mathcal{M}_{cycle}$
22:     let $\mathcal{G}' = \{\{r_i\} \mid r_i$ is a non-drainable instance $\}$
23:     let $\mathcal{G}_{new} = \{G_1, \cdots, G_d\}$, where each share group $G_i$ contains all instances of $R_i$ except for non-drainable instances
24:     if $(\mathcal{R}_{multi} = \{R_1\})$ and (no two drainable instances in $R_1$ have the same drainer) then
25:         $\mathcal{G}_{new}$ = OptimalGrouping $(G_1)$
26:     else
27:         $\mathcal{G}_{new}$ = HeuristicGrouping $(\mathcal{G}_{new})$
28:     $P = (plan(\text{Q}),\ \mathcal{G}_{opt},\ \mathcal{M}_{drain} \cup \mathcal{M}_{cycle})$, where $\mathcal{G}_{opt} = \mathcal{G}_{new}\ \cup\ \mathcal{G}'$
29:     if $(cost(P) < cost(P_{best}))$ then
30:         $P_{best} = P$
31: return $P_{best}$
```

sets that are *useful*. Intuitively, a set of materialize operators $\mathcal{M} \subseteq M_{all}$ is considered to be *useless* (or not useful) if there exists a deadlock-free enhanced query plan P' with $M(P') \neq \mathcal{M}$ such that for every deadlock-free enhanced query plan P'' with $M(P'') = \mathcal{M}$, $cost(P') < cost(P'')$. Thus, a useless set of materialize operators can be safely ignored without affecting the optimality of the enhanced query plan.

We now provide a more concrete characterization of the notion of a useful set of materialize operators. Recall that adding a materialize operator M to *plan*(Q) can help enhance its performance in two ways. First, M can enable a non-drainable instance r_i to become drainable thereby allowing r_i to participate in a shared scan. Second, M can eliminate some execution dependencies thereby enabling a plan to become deadlock-free (i.e., $C = \emptyset$). These two benefits of M can be formalized in terms of its *drain set* and *remove-cycle set* defined as follows.

The *drain set* of M, denoted by $drainSet(M)$, is defined to be the set of non-drainable instances in *plan*(Q) that become drainable if M is added to *plan*(Q). Thus, M becomes the drainer operator for each of the instances in $drainSet(M)$.

The *remove-cycle set* of M (w.r.t. C), denoted by $cycleSet^-(M, C)$, is defined to be the subset of group dependency cycles in C that are eliminated by the addition of M to *plan*(Q).

The following result states a useful relationship between $drainSet(M)$ and $cycleSet^-(M, C)$.

LEMMA 3.4. *At most one of $drainSet(M)$ and $cycleSet^-(M, C)$ can be non-empty.*

Lemma 3.4 follows from the observation that if $drainSet(M) \neq \emptyset$ (i.e., M becomes a drainer for some non-drainable instance r_i), then r_i cannot have any ancestor drainer operator prior to the addition of M, which implies that there are no instance execution dependencies (and hence group execution dependencies) that M can eliminate. Hence $cycleSet^-(M, C) = \emptyset$. Conversely, if $cycleSet^-(M, C) \neq \emptyset$, then M is able to eliminate some group dependency cycle (via the elimination of some instance execution dependency) which implies that there must exist some drainer operator that is an ancestor of M. Hence, there cannot be any non-drainable instances within the query subtree rooted at M (i.e., $drainSet(M) = \emptyset$).

Based on Lemma 3.4, the useful materialize operators (w.r.t. C) can be partitioned into two disjoint sets M_{drain} and M_{cycle} defined as follows:

$$M_{drain} = \{M \in M_{all} \mid drainSet(M) \neq \emptyset\}$$
$$M_{cycle} = \{M \in M_{all} \mid cycleSet^-(M, C) \neq \emptyset\}$$

A materialize operator that is not contained in $M_{drain} \cup M_{cycle}$ is useless.

However, adding a materialize operator M to *plan*(Q) not only incurs a processing cost (i.e., $matCost(M)$) but could also introduce additional group dependency cycles. We characterize the latter cost for M as follows. The *add-cycle set* of M (w.r.t. C), denoted by $cycleSet^+(M, C)$, is defined to be the set of new group dependency cycles (i.e., not contained in C) that are introduced by the addition of M to *plan*(Q).

The following result states that adding a materialize operator from M_{cycle} to *plan*(Q) does not create any new group dependency cycles.

LEMMA 3.5. *$cycleSet^+(M, C) = \emptyset$ for each $M \in M_{cycle}$.*

Lemma 3.5 can be established by contradiction. Suppose $cycleSet^+(M, C) \neq \emptyset$. Then the addition of M must have introduced a new instance execution dependency $r_i \to s_j$ (that contributed to a new group dependency cycle), where both r_i and s_j are within the query subtree rooted at M. However, $M \in M_{cycle}$ implies that there must be a drainer operator that is an ancestor of M in the query plan which contradicts the fact that $r_i \to s_j$ is a new dependency.

The definitions of $drainSet(M)$, $cycleSet^+(M, C)$, and $cycleSet^-(M, C)$ can be generalized naturally for a set of materialize operators $\mathcal{M} \subseteq M_{all}$ (e.g., $drainSet(\mathcal{M}) = \bigcup_{M \in \mathcal{M}} drainSet(M)$).

By Lemmas 3.4 and 3.5, we can define a useful materialize operator set in terms of its two disjoint subsets: a useful subset of M_{drain} and a useful subset of M_{cycle} as follows.

We say that $\mathcal{M} \subseteq M_{drain}$ is *useful* (w.r.t. C) if there does not exist another $\mathcal{M}' \subseteq M_{drain}$ such that all the following four conditions hold: (1) $drainSet(\mathcal{M}) \subseteq drainSet(\mathcal{M}')$, (2) $cycleSet^+(\mathcal{M}, C) \supseteq cycleSet^+(\mathcal{M}', C)$, (3) $matCost(\mathcal{M}) \geq matCost(\mathcal{M}')$, and (4) at least one of the three previous conditions is strict.

Similarly, we say that $\mathcal{M} \subseteq M_{cycle}$ is *useful* (w.r.t. C) if there does not exist another $\mathcal{M}' \subseteq M_{cycle}$ such that all the following three conditions hold: (1) $cycleSet^{-}(\mathcal{M}, C) \subseteq cycleSet^{-}(\mathcal{M}', C)$, (2) $matCost(\mathcal{M}) \geq matCost(\mathcal{M}')$, and (3) at least one of the two previous conditions is strict.

Finally, consider a set $\mathcal{M} \subseteq M_{all}$, where $\mathcal{M} = \mathcal{M}_{drain} \cup \mathcal{M}_{cycle}$, $\mathcal{M}_{drain} \subseteq M_{drain}$, and $\mathcal{M}_{cycle} \subseteq M_{cycle}$. We say that $\mathcal{M}$ is *useful* (w.r.t. C) if $\mathcal{M}_{drain}$ is useful (w.r.t. C) and $\mathcal{M}_{cycle}$ is useful (w.r.t. $C \cup cycleSet^{+}(\mathcal{M}_{drain}, C)$).

A materialize operator set that is not useful cannot form an optimal enhanced query plan.

Algorithm. `SSPO` (steps 10 to 17 in Algorithm 1) generates a collection of useful candidate materialize operator sets (denoted by S) as follows. First, `SSPO` generates $\mathcal{S}_{drain}$, the collection of all useful subsets of M_{drain}. Next, `SSPO` takes each $\mathcal{M}_{drain} \in \mathcal{S}_{drain}$ to generate $\mathcal{S}_{cycle}(\mathcal{M}_{drain})$, the collection of all useful subsets of M_{cycle} w.r.t. C', where $C' = C \cup cycleSet^{+}(\mathcal{M}_{drain}, C)$. The final collection S is given by $\{\mathcal{M}_{drain} \cup \mathcal{M}_{cycle} \mid \mathcal{M}_{drain} \in \mathcal{S}_{drain},\ \mathcal{M}_{cycle} \in \mathcal{S}_{cycle}(\mathcal{M}_{drain})\}$. Note that as the empty set is contained in both $\mathcal{S}_{drain}$ and $\mathcal{S}_{cycle}(\mathcal{M}_{drain})$, an empty set of materialize operators is also generated by `SSPO`.

Although the time complexity of the procedure above is exponential in the number of materialize operators in M_{drain} and M_{cycle}, this number is reasonably small in practice. Alternatively, some heuristic can be applied to generate smaller M_{drain} and M_{cycle} so as to reduce the running time, in the cost of missing some useful candidate materialize operator sets in S.

Example 3.3 Fig. 5(a) shows two useful materialize operators, M_1 and M_2, that can be used to break the execution dependency cycle $hd_1 \dashrightarrow hd_2 \rightarrow ws_2 \dashrightarrow ws_1 \rightarrow hd_1$ for the query plan of Q90 in Example 2.2. M_1 breaks the cycle by eliminating $ws_1 \rightarrow hd_1$ while M_2 breaks the cycle by eliminating $hd_2 \rightarrow ws_2$. Ignoring the $matCost(.)$ component, there are three useful sets of materialize operators: $\emptyset$, $\{M_1\}$ and $\{M_2\}$. □

3.4.2 Optimizing Share Group List

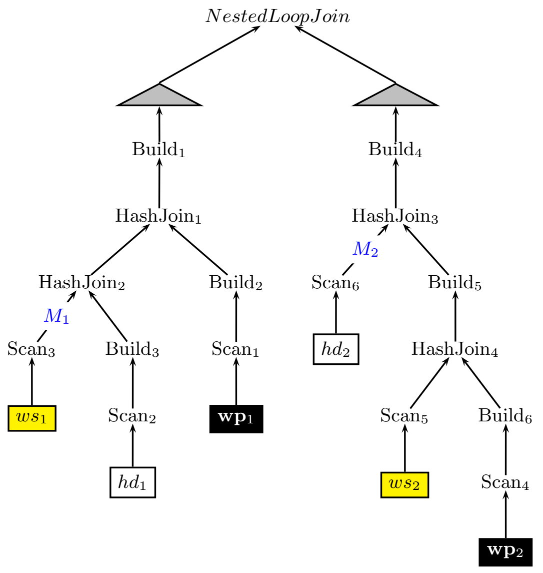
Given a candidate set of materialize operators $\mathcal{M}$, the second phase of `SSPO` (steps 18 to 30 in Algorithm 1) computes an optimized share group list $\mathcal{G}$ to produce a deadlock-free enhanced plan P with $M(P) = \mathcal{M}$ and $G(P) = \mathcal{G}$. `SSPO` has two algorithms for this computation: an optimal algorithm (that can compute an optimal share group list) is used if *plan*(Q) meets certain conditions; otherwise, a greedy heuristic algorithm is used.

Optimal Grouping. An optimal share group list can be computed using Algorithm 2 when *plan*(Q) satisfies two conditions:

(C1) there is exactly one multi-instance relation R_1 in *plan*(Q); and

(C2) the drainers for all the drainable-instances of R_1 are all distinct.

Algorithm 2 takes a single share group G_1 as input, where G_1 contains all the instances of R_1 except for non-drainable instances. The algorithm first constructs a directed graph G, where the nodes in G are instances in G_1, and the edges represent execution dependencies among the instances in G_1. By condition (C2), G must be a directed acyclic graph. The algorithm then iteratively refines G_1 into a collection



(a) Candidate Materialize Operators

Plan	M	Share Groups
P_1	$\emptyset$	$\{ws_1, ws_2\}$, $\{hd_1\}$, $\{hd_2\}$, $\{wp_1, wp_2\}$
P_2	$\{M_1\}$	$\{ws_1, ws_2\}$, $\{hd_1, hd_2\}$, $\{wp_1, wp_2\}$
P_3	$\{M_2\}$	$\{ws_1, ws_2\}$, $\{hd_1, hd_2\}$, $\{wp_1, wp_2\}$

(b) Candidate Enhanced Query Plans

Figure 5: Enhanced Query Plans for Example 2.2

of share groups $G'_1, \cdots, G'_n$ such that $G'_1 \rightarrow G'_2 \cdots \rightarrow G'_n$. The time complexity of Algorithm 2 is $O(m^2)$, where m is the number of instances in G_1.

Heuristic Grouping. Algorithm 3 is a greedy heuristic approach to optimize an input share group list $\{G_1, \cdots, G_d\}$, where each G_i contains all the instances of relation R_i excluding the non-drainable instances. The share groups are ordered such that $scanCost(R_1) \leq \cdots \leq scanCost(R_d)$. The heuristic refines each share group G_i by splitting it into a collection of smaller groups S_i in the order $G_1, \cdots, G_d$. The intuition behind processing the share groups in non-descending order of the scan cost of the associated relations is to minimize the total scan cost of the refined share groups. For each group G_i, the heuristic tries to split G_i into the smallest number of groups by iteratively removing from G_i, the instance that is involved in the largest number of cycles. The removed instance is inserted into an existing split group of G_i whenever possible; otherwise, it is inserted into a new split group. The insertions into split groups are performed such that no new group dependency cycles are formed. The time complexity of of Algorithm 3 is $O(n^2)$, where $n = \sum_{i=1}^{d} |G_i|$.

Example 3.4 Continuing with Example 3.3, Fig. 5(b) shows the optimized share group lists computed for each candidate materialize operator set using the heuristic algorithm in Algorithm 3. □

Algorithm 2 `OptimalGrouping`

```
Input: a single share group $G_1$ containing the instances of $R_1$ excluding non-drainable instances
Output: an optimal list of share groups
 1: let $G = (V, E)$, where
    $V = G_1$ and $E = \{(a, b) \mid a, b \in V,\ a \rightarrow b\}$
 2: initialize $n = 0$
 3: repeat
 4:     $n = n + 1$; $G'_n = \{v \in V \mid$ v has in-degree of 0 in $G\}$
 5:     remove each $v \in G'_n$ from $G$ and its incident edges
 6: until $V = \emptyset$
 7: return $\{G'_1, \cdots, G'_n\}$
```

Algorithm 3 `HeuristicGrouping`

```
Input: $\{G_1, \cdots, G_d\}$, where each $G_i$ is a share group containing all instances of relation $R_i$ excluding non-drainable instances such that $scanCost(R_1) \leq \cdots \leq scanCost(R_d)$
Output: an optimized list of share groups
 1: let $C$ = set of group dependency cycles among $G_1, \cdots, G_d$
 2: for $i = 1$ to $d$ do
 3:     initialize $S_i = \{G_i\}$
 4:     while ($C$ contains a cycle involving $G_i$) do
 5:         let $r_j$ be the instance in $G_i$ that participates in the largest number of cycles in $C$
 6:         if $r_j$ can be added into some $G_k \in S_i$, $k \neq i$, without introducing any new group dependency cycles then
 7:             add $r_j$ into $G_k$
 8:         else
 9:             create a new group $G' = \{r_j\}$
10:             add $G'$ into $S_i$
11:         remove $r_j$ from $G_i$
12:         remove cycles in $C$ that involved $r_j$
13: return $S_1 \cup \cdots \cup S_d$
```

4. INTERLEAVED ITERATIVE EXECUTION

In this section, we explain how the `IIQE` component of `MAPLE` can be implemented by making only moderate modifications to the conventional iterative query execution engine; thus, demonstrating that `MAPLE` is indeed a light-weight approach to optimize complex queries with multiple relation instances.

For each relation instance r_i in *eplan*(Q), `IIQE` maintains the following static information: (1) a boolean flag, denoted by *switchEnabled*(r_i), which has a `true` value if and only if r_i is estimated by `SSPO` to be an overflow instance; and (2) *drainer*(r_i) if if r_i is a drainable instance. In addition to the above information, which remains unchanged during the execution of the query, `IIQE` also maintains some global runtime information that is updated dynamically as the query execution progresses. Specifically, each relation instance r_i is associated with a status variable for its drainer, denoted by *drainerStatus*(r_i), which has three possible values: *inactive*, *active*, and *successful*, indicating, respectively, that the drainer is not active, the drainer is active and the draining is in progress, and the drainer is active and the draining has completed. The value of *drainerStatus*(r_i) is initialized to *inactive* for each relation instance r_i before the execution of *eplan*(Q). Whenever $buf(r_j)$ becomes full during the shared scan of some other instance of r (say r_i) and `IIQE` decides to switch execution to *drainer*(r_j), the value of *drainerStatus*(r_j) is updated to *active*. Subsequently, when all the tuples in the full buffer $buf(r_j)$ have been consumed, the scan operator for r_j will update the value of *drainerStatus*(r_j) from *active* to *successful*. When the execution control is returned from *drainer*(r_j), the value of *drainerStatus*(r_j) is reset to *inactive*.

Algorithm 4 `Scan`

```
Input: $r_i$, the instance being scanned
 1: let $Op$ be the parent operator of $Scan(r_i)$ in query plan
 2: if ($buf(r_i)$ is not empty) then
 3:     let $t$ be the first tuple in $buf(r_i)$
 4:     deliver $t$ to $Op$
 5:     remove $t$ from $buf(r_i)$
 6: else
 7:     if ($drainerStatus(r_i) = active$) then
 8:         $drainerStatus(r_i) = successful$
 9:         deliver a dummy-null tuple to $Op$
10:     else
11:         SharedScan ($r_i$)
```

Algorithm 5 `SharedScan`

```
Input: $r_i$, the instance being scanned
 1: let $G_x$ be the share group that $r_i$ belongs to
 2: initialize $continueScan$ = true
 3: while ($continueScan$) do
 4:     let $t$ be the next tuple from relation $r$
 5:     if ($t$ is null) or ($t$ qualifies for $Scan(r_i)$) then
 6:         deliver $t$ to the parent operator of $Scan(r_i)$
 7:         $continueScan$ = false
 8:     for each ($r_j \in G_x$, $r_j \neq r_i$) do
 9:         if ($t$ is null) or ($t$ qualifies for $Scan(r_j)$) then
10:             if ($buf(r_j)$ is full) and switchEnabled($r_j$) then
11:                 SwitchExecution ($r_j$)
12:             append $t$ into $buf(r_j)$
```

Algorithm 6 `SwitchExecution`

```
Input: $r_i$, an overflow instance
 1: $drainerStatus(r_i) = active$
 2: transfer execution control to $drainer(r_i)$ operator
 3: $drainerStatus(r_i) = inactive$
```

Recall that in the iterative execution model, each operator is specified in terms of three functions: *open*, *getNext*, and *close*. Algorithm 4 highlights the modifications required for the *getNext* procedure of the table scan operator (referred to as `Scan`). Given a relation instance r_i, `Scan` first checks whether its associated buffer $buf(r_i)$ is empty: if it is not empty, the first tuple in $buf(r_i)$ will be returned to the parent operator of *Scan*(r_i) and then removed from $buf(r_i)$. The key modification for the `Scan` algorithm occurs when the buffer is empty, where there are two cases to consider. In the first case (steps 8-9), if the scan of r_i has been initiated to drain its buffer (i.e., *drainerStatus*(r_i) = *active*), then it means that the draining process has completed successfully. The value of *drainerStatus*(r_i) is then updated to *successful*, and a `dummy-null` tuple is returned to the parent operator of *Scan*(r_i). The use of `dummy-null` tuples is important to distinguish a successful draining process (i.e., all the tuples in a full buffer have been consumed) from a completed relation scan event (i.e., there are no more tuples to be placed in the buffer). In the latter case, an actual `null` tuple is returned. In this way, whenever an operator *op* receives a `dummy-null` tuple from its child operator, *op* will know that the tuple is due to a completed draining process and will therefore pass the `dummy-null` tuple up to its parent operator, and so on. The upward propagation of the `dummy-null` tuple in the query plan tree continues until the tuple is received by a successful drainer operator *op* (i.e., $op = drainer(r_k)$ and $drainerStatus(r_k) = successful$). Thus, the drainer operator *op* then returns execution control to the interrupted relation scan that initiated *op*. The value of $drainerStatus(r_k)$ is also reset to *inactive*. In the second case (step 11), where the scan of r_i is a "normal" scan (i.e., not initiated for buffer draining), the `SharedScan` of r_i will be invoked.

The details of `SharedScan` are shown in Algorithm 5. Essentially, `SharedScan` continues scanning r for the next tuple that qualifies for r_i; i.e., satisfies the selection predicate conditions associated with the scan of r_i. For each scanned tuple t, `SharedScan` also checks if t qualifies for other instances r_j that are in the same share group as r_i; the qualified tuples are pushed into the appropriate buffers. If some buffer $buf(r_j)$ becomes full, then there are two cases to consider. If `SSPO` has correctly estimated that r_j is an overflow instance (i.e., $switchEnabled(r_j)$ has a `true` value), a drainer operator $drainer(r_j)$ would have been assigned by `SSPO` and the execution control then switches to this drainer (step 11) by invoking the `SwitchExecution` function. Otherwise, if the overflow of $buf(r_j)$ has not been anticipated by `SSPO`, the full buffer $buf(r_j)$ will not be drained and it will instead be implicitly materialized; i.e., in `IIQE`, whenever a tuple is added to a full buffer $buf(r_j)$, the buffer contents will be materialized.

In general, the `SharedScan` of r_i could lead to full buffers for multiple instances in the same share group as r_i. When this happens, there is the issue of the execution order of the multiple drainers. The current implementation of `MAPLE` simply picks an arbitrary sequence; possible optimization of this ordering is part of our future work.

The `SwitchExecution` function (shown in Algorithm 6) is invoked to switch execution to $drainer(r_i)$ for an overflow instance r_i. The function needs to update the *activeDrainer* status of r_i to *active* before transferring control to the drainer and reset the *activeDrainer* status to *inactive* upon its return.

In summary, implementing the `IIQE` component of `MAPLE` requires only moderate modifications to the traditional iterative execution evaluation engine used by most RDBMSs. Specifically, the main changes include: two new functions `SharedScan` and `SwitchExecution`; and minor modifications to operator code to distinguish between `null` and `dummy-null` tuples.

5. PERFORMANCE STUDY

We validated our techniques using an experimental prototype built on PostgreSQL 8.1.3. All experiments were performed on a Dell workstation with a Quad-Core Intel Xeon 2.33GHz processor, 3GB of memory, one 160GB SATA disk and another 750GB SATA disk, running Linux 2.6.20. Both the operating system and PostgreSQL system are built on the 160GB disk, while the databases of PostgreSQL are stored on the 750GB disk.

Since PostgreSQL 8.1.3 does not support the WITH clause, we replaced those WITH procedures in queries with VIEW definitions. In this way, PostgreSQL applies view unfolding to replace the views by their definitions during optimization.

As default, the initial system buffer pool in PostgreSQL is set to 1,000 8K-pages. We also tested with larger buffer pool sizes. The results were similar and thus omitted.

5.1 Test Queries

As mentioned, more than 60% (61 out of total 99) of the TPC-DS queries contain multiple instances. We have conducted experiments on many of these queries. We present here a representative set that offers some interesting insights. A query is chosen if it satisfies *all* the following criteria: (a) It contains multiple instances that are eligible for scan sharing, i.e., apply sequential scan on the same table. (b) It contains multiple instances of at least one of the three big relations: store_sales (ss), catalog_sales (cs) and web_sales (ws). (c) It is executable by PostgreSQL. Some operators in the queries are not recognized/supported by PostgreSQL. (d) It can be optimized by PostgreSQL's *dynamic programming* (DP) optimizer. For queries that are too complex to optimize using the DP method, PostgreSQL provides another *genetic optimizer(geqo)*. However, since geqo does not guarantee to generate consistent plans for the same query, we cannot use it. (e) It is not a *batch* query which contains a batch of separate queries that run in parallel. We have to exclude batch queries because our current implementation only supports single queries, although our techniques can be easily extended to support batch queries. (f) Its execution time is affordable for us. Some queries, like Q74 and Q95, require super long-time executions. Table 1 presents a summary of the 49 queries excluded according to the criteria above.

criterion	a	b	c	d	e	f
# of queries	5	28	8	2	4	2

Table 1: Queries Filtered by Each Criterion

Finally we are left with 12 queries listed in Table 2, along with the instance number of *ss*, *cs* and *ws* inside. However, all other instances within a chosen query, irrespective of their sizes, were also considered by `MAPLE`.

	rel ∗ inst#		**rel ∗ inst#**
Q2	cs ∗ 2, ws ∗ 2	**Q61**	ss ∗ 2
Q4	ss ∗ 4, cs ∗ 4	**Q65**	ss ∗ 2
Q11	ss ∗ 4, ws ∗ 4	**Q72**	ss ∗ 2, cs ∗ 2
Q31	ss ∗ 3, ws ∗ 3	**Q88**	ss ∗ 8
Q51	ss ∗ 2	**Q90**	ws ∗ 2
Q59	ss ∗ 2	**Q97**	ws ∗ 2

Table 2: Test Queries in Experiments

Since TPC-DS queries are all very complex, we cannot afford to draw full queries/plans in the paper due to space limitation.

5.2 Experiment Design

In our implementation, `MAPLE` is integrated into the original system. By setting a flag, we can switch between the *original mode* and `MAPLE` *mode*. In the original mode, the original execution engine will be used; in the `MAPLE` mode, the `MAPLE` engine will be used. Both engines share the same query optimizer.

In each experiment below, we ran the same test query in both the original mode and the `MAPLE` mode to compare the execution time difference. When a test query was running, no other queries was running in parallel. Between queries we restarted the operating system to clear caches.

In PostgreSQL, *each* sorting and hashing operation has a dedicated *operator memory*. In MAPLE, besides the operator memories, each overflow instance uses additional buffer memory, which we shall refer to as *instance-buffer*. For a fair comparison, in each experiment we distributed the total amount of instance-buffer used in MAPLE mode evenly to each operator memory in original mode.

We studied the effect of three experiment parameters: operator memory (operator_mem), instance-buffer size (buffer) and the *dataset* size. For the latter, we used both 10 GB and 100 GB TPC-DS datasets.

The TPC-DS datasets are imported into PostgreSQL's databases, which are stored on the 750GB disk. In the experiments, the same disk was used to store the temporary files generated during query execution.

The default settings that we used for our experiments are 1 MB (instance) buffer, 10 MB operator memory and a 10 GB dataset.

In following subsections, we shall refer to the system under original mode as PostgreSQL.

5.3 Optimization Overhead

	psql	MAPLE		psql	MAPLE
Q2	90	125	**Q61**	366	434
Q4	113	126	**Q65**	311	351
Q11	117	133	**Q72**	137570	137789
Q31	104	115	**Q88**	397	413
Q51	427	502	**Q90**	346	354
Q59	88	119	**Q97**	420	473

Table 3: Optimization times (in *microsecond*) with Default Settings

It is desirable to measure the optimization overhead of MAPLE, which is incurred mainly by SSPO. Therefore, we compared the actual optimization times of PostgreSQL and MAPLE with default parameter settings. In order to eliminate any first-level instruction cache effect in query optimization, we restarted the operating system between optimizations. The optimization times of PostgreSQL and MAPLE can be found in Table 3. It is very clear that the optimization overhead of MAPLE is low, and as we shall see shortly, it is also negligible compared to the query execution time.

5.4 Operator Memory

In this experiment, we study the effect of operator_mem. We use three different sizes: 5 MB, 10 MB and 20 MB.

Fig. 6 shows the performance improvements (in %) of MAPLE over the PostgreSQL; and Fig. 7 shows the corresponding query execution times in MAPLE and PostgreSQL. In Fig. 7, the execution times of PostgreSQL can be computed by adding the execution time of MAPLE with the time MAPLE saved. Fig. 8 depicts the *expected saving* and the *actual saving* for all queries with 5 MB operator_mem. The expected saving refers to the time MAPLE is expected to save over PostgreSQL. The actual saving is the saving of MAPLE over PostgreSQL for the actual total query execution time. Due to space limitations, we shall not present detail query-by-query analysis. Instead, we will summarize the more interesting findings here.

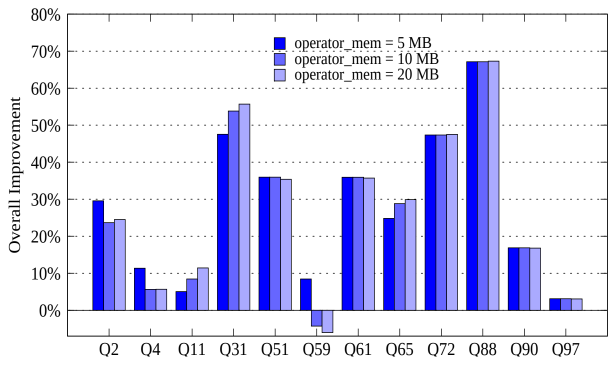


Figure 6: Performance Improvements By MAPLE

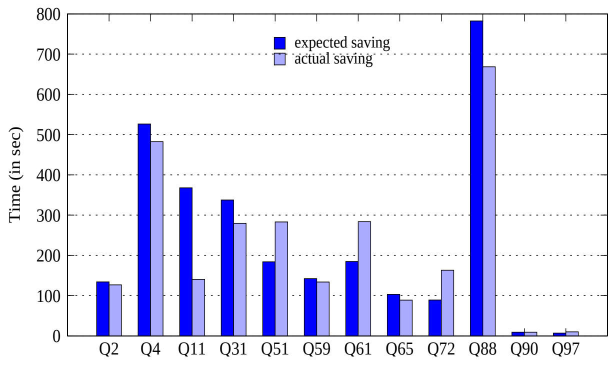


Figure 8: Expected Saving and Actual Saving With 5MB operator_mem

First, as shown, MAPLE offers significant performance improvement in almost all queries (except Q59, for which we will explain shortly). The average improvement is around 30% and the highest improvement is 67% achieved by Q88. In terms of absolute time, the savings range from a few seconds to 700 seconds. These results are expected as MAPLE requires only one scan of multiple instances of a relation. Second, we also observe that MAPLE remains superior as we vary operator_mem.

Second, we note that, for some queries (Q2, Q4, Q11, Q31, Q59 and Q65), the execution times of both MAPLE and PostgreSQL vary with different operator_mem. There are two main reasons for this:(a) The query plan generated by PostgreSQL (and hence MAPLE) may be different under different operator_mem size. In the experiment, the plans for Q4 and Q65 are different when we change operator_mem from 5 MB to 10/20 MB; for Q11, there are three different plans for the three operator_mem sizes; for the other queries, their plans remain the same for the three operator_mem sizes. (b) A larger operator_mem may reduce the I/O cost, e.g., for sorting, the number of runs may be reduced, and for hybrid hash join, the amount of data in the partitions to be written out and re-read will also be smaller. This reduces the execution times. With a reduced execution time, the savings for MAPLE over PostgreSQL may correspondingly reduce.

Third, from Fig. 8, we find that the actual savings in MAPLE are close to the expected savings for most queries. The difference is mainly due to the additional overhead (like the

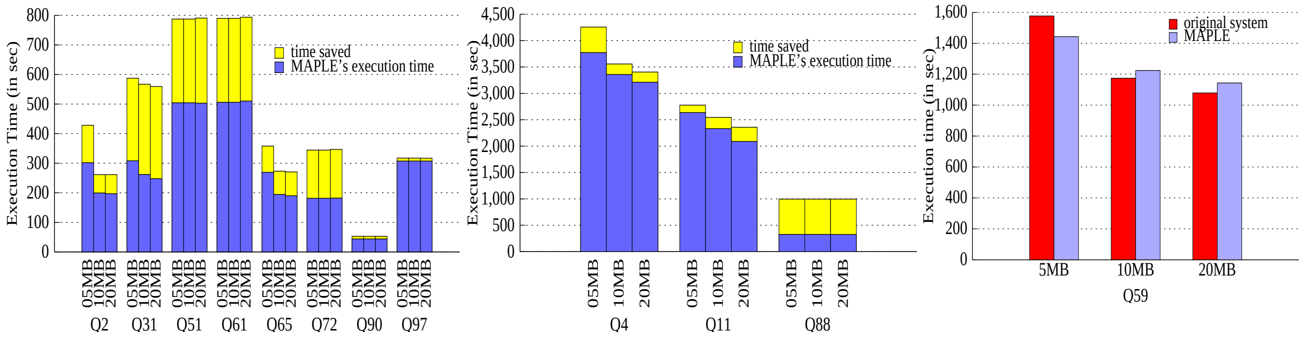


Figure 7: Query Execution Times

cost of copying tuples to buffers) incurred by interleaved execution. However, for Q11, the actual saving is significantly lower than expected, whereas for Q51, Q61 and Q72, the actual savings are much higher than expected! Our investigation shows these are contributed by several effects of the interleaved execution: (a) *FragmentedReadWrite effect.* Under the interleaved execution model, the processing of one drainer may trigger other drainers to become active. As a result, when the processing of these drainers involve disk accesses (e.g., sorting), the intermediate results written (and subsequently read) are more fragmented across the disk (than it would be had there been only one single drainer running, as in PostgreSQL). (b) *BufferHit effect.* This effect arises when both an active drainer and an interrupted drainer share some cache content. As a result, when the active drainer requires some data, it finds it in the buffer, and when the suspended drainer resumes processing, it also finds its required data in the buffer. Clearly, the FragmentedReadWrite is a negative effect while the BufferHit is a positive effect.

For Q51, Q61 and Q72, we observe the BufferHit effect For example, in Q51, there are some common index scans in subtrees of two drainers: while execution switches between these two drainers, the index pages fetched from disk can be shared via the system buffer. As such, besides the expected savings from using SharedScan, the sharing of these indexed pages also contributes to the actual savings.

On the other hand, for Q11, it turns out that the drainers that are processed in an interleaved fashion need to write out large amount of intermediate results, resulting in the FragmentReadWrite effect that reduces the savings.

For Q90 and Q97, little improvement opportunity was left to `MAPLE` due to the *OS CacheHit* effect. This is because in these two queries ws is the only large table for which a large part is cached by the OS in the 3GB RAM.

Finally, for Q59, the plan involves a sort operator on a large intermediate result produced by a hash join operator. When the operator_mem is small (5 MB), the buffer is not sufficient to hold the entire hash table, and the sort operation incurs more disk I/O cost. As such, although there is a FragmentReadWrite effect in `MAPLE`, this is relatively small and hence `MAPLE` outperforms PostgreSQL. However, when the operator_mem increases to 10/20 MB, the hash join can be processed in memory, and the sort operator incurs lesser I/O cost. As a result, PostgreSQL's execution time reduces significantly. On the contrary, the FragmenReadWrite effect remains in `MAPLE`. It turns out that this effect far outweighs the benefits of SharedScan, resulting in its poorer performance than PostgreSQL. We note that we can statically determine the number of switches (which gives a hint on how fragmented the drainer's output will be). If the value is above a certain threshold, we will not post-optimize the PostgreSQL plan. We plan to explore this further.

5.5 Instance-buffer Size

We next study the effect of instance-buffer size. In this experiment, we use two different instance-buffer sizes: 100 KB, 1 MB. Recall that for PostgreSQL, the total amount of instance-buffer sizes used for `MAPLE` goes to its operator memories. The results for this experiment are shown in Fig. 9.

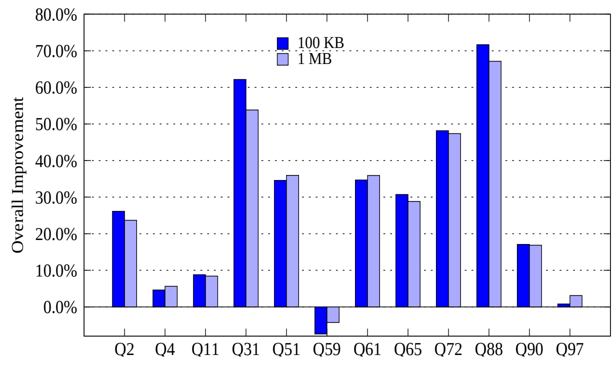


Figure 9: `MAPLE` Effect of Changing Instance-buffer Size

Generally, the performance of `MAPLE` under different buffer sizes are more or less the same. We see two different effects of the buffer size. For `MAPLE` a larger instance-buffer reduces the number of interleaved executions, and hence less FragmentReadWrite effect. On the other hand, for PostgresSQL, a larger operator memory (recall that the instance-buffer of `MAPLE` are distributed to the operator memory) reduces the I/O cost of sort and hash operators. For some queries (e.g., Q4, Q11, Q51, Q59, Q61, Q97), the improvement over PostgreSQL increases with larger instance-buffer. However, for some queries, like Q72 and Q88, the performance improvement over PostgreSQL degraded marginally with increased buffer sizes.

5.6 Dataset

We also conducted an experiment with a 100 GB dataset to study the scalability of MAPLE. Here, we use 10 MB operator_mem and 1 MB buffer. Fig. 10 shows both the results of 100GB dataset and the results of 10GB dataset with the same parameter settings.

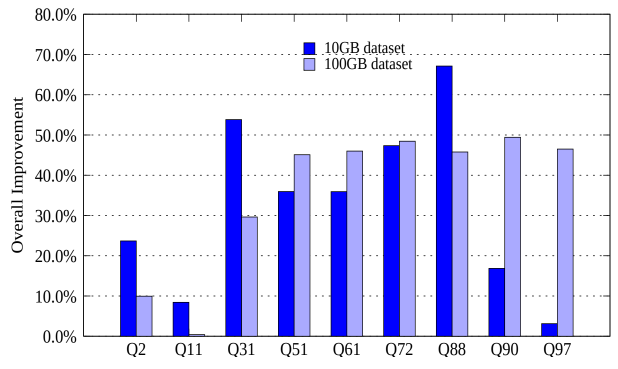


Figure 10: MAPLE Effect in 100GB Dataset

Since the execution times of queries on 100GB dataset were very long, we did not finish all 12 queries. From the figure, we see that MAPLE still performs well with a larger dataset.

For some queries like Q2, Q31 and Q88, the improvements over PostgreSQL (in %) with the 100 GB dataset is lower than that with the 10 GB dataset. There are two main reasons for this behavior: a) while (big) relation sizes have increased ten fold from 10 GB to 100 GB dataset, their scan times have not increased proportionally. For example, the scan time of web_sales in Q2 increased from 20 seconds to 90 seconds in 100 GB dataset. b) With the 100 GB dataset, in PostgreSQL the ratio of the total table scan time of instances to the total execution time is reduced compared to that of 10 GB dataset. For example, the total table scan time of instances of Q2 took around 49% and 39% of the total execution time in 10GB and 100GB dataset, respectively.

On the other hand, for Q90 and Q97, MAPLE performs much better in the 100 GB dataset. This is because the OS CacheHit effect present in the 10 GB dataset disappeared in 100 GB case, and the gain of shared scan becomes more significant.

6. RELATED WORK

The need to efficiently coordinate multiple disk scans on the same table to exploit data-sharing has long been recognized. Early work focused on designing buffer replacement algorithms (e.g., LRU-K [13]) to maximize buffer locality. However, these works do not explicitly optimize data sharing. Moreover, their effectiveness is limited especially for large tables that do not fit in the cache. Several commercial Database systems have implemented various forms of *circular scans* on database relations (Teradata [4], RedBrick [6] and Microsoft SQL Server [1]). The basic idea is to let a newly starting scan attach to an ongoing scan to reuse buffer pages brought by the ongoing scan. In QPipe [10], Harizopoulos et al. propose to maintain one scan thread that keeps scanning a table while table scan operators can attach to and detach from this thread in order to share the scanned buffer pages. However, the degree of sharing the buffer pool provided in these methods is extremely sensitive to the speed diversity of scans. Recently, a modified circular scan has been proposed in IBM DB2 system [11, 12] by adding explicit group control and allowing throttling of faster scans. Zukowski et al. [19] introduce an enhanced buffer manager that dynamically schedules disk reads of scan operators such that multiple concurrent scans reuse the same buffer pages.

Sharing scan of base relations and pipelining of common subexpression results reduce disk access costs on the level of query processing operators. Zhao et al. [17] consider sharing scans and pipelining subexpressions among OLAP queries (aggregation on a join of fact table with dimension tables). Nilesh et al. [7] discussed the feasibility of pipelining in multi-query optimization. They aim to pipeline results of a common subexpression or tuples of a base relation to consumers in different SQL queries. Our work can also be extended to handle common subexpressions and multiple queries.

In [7], the determination of a valid pipeline schedule has a similar motivation as our deadlock avoidance method in Section 3.2.3. However, the two are actually different. As an example, consider Q90 in Fig. 2. The schedule of sharing the scan of all three relations will be considered valid by [7] as each cycle has two opposite materialized edges (the build edges of hash join). However, as we discussed in the paper, whether it is a valid sharing scan schema depends on whether hd_2 is an overflow instance or not (see example 1.1 and example 2.2). In fact, the interleaved execution deadlock described in this paper is different from the deadlock situation in [7, 10].

In multi-query optimization(MQO) [14, 18], exploiting common subexpressions in (multi-instance) queries indirectly leads to avoiding multiple scans on the same relation table. However, the materialized results of a common subexpression need be separately read by different consumers, just like the independent scans of relation instances. Moreover, MQO is not able to optimize scans of instances that are outside the common subexpression. Therefore, for multi-instance queries, MAPLE can be either applied independently or ultilized as the next optimization step after MQO.

The philosophy under our interleaved execution strategy is that when event a is blocked, process event b to continue a. The *query scrambling* [15] technique follows another similar but different philosophy: when event a is blocked, process event b until a resume itself. Used in distributed query processing, query scrambling reacts to unexpected delays in obtaining initial requested tuples from remote sources by performing other useful work which would normally be scheduled for a later point in the execution.

We also note that Graefe has hinted on the idea of switched execution in [8]. However, there is no discussion on how to realize it. We are the first to investigate the interleaved execution model and demonstrate its practical effectiveness.

7. CONCLUSION

In this paper, we have presented MAPLE, a *M*ulti-instance-*A*ware *PL*an *E*valuation engine. MAPLE enables multiple instances of a relation in single queries to share one physical scan with limited buffer space. MAPLE is light-weight and can be easily integrated into existing RDBMS executors.

We have developed a prototype in PostgreSQL, and our experimental study using the TPC-DS benchmark showed that MAPLE can significantly reduce the execution time (compared to the original plans produced by PostgreSQL).

There are several directions for future work. First, MAPLE can be easily extended to support common subexpressions within a single query (instead of just table scans) as well as across multiple queries. The result of a common subexpression, either pipelined or materialized, can be treated as a virtual table and shared "scanned" by all instances. For multiple queries, common subexpressions or tables across multiple queries can be shared in a similar manner. In addition, we note that these queries can be processed simultaneously without any execution dependency between them. We plan to complete our implementation to support these.

Second, as shown in our experimental study, interleaved execution of operators may impact performance of a query in a negative way, i.e., the FragmentReadWrite effect. We plan to explore how we can extend MAPLE to consider these factors. In particular, we need to ensure that a MAPLE-enhanced plan must not be inferior to the corresponding PostgreSQL plan.

Finally, MAPLE is a post-optimization strategy. As such, it only enhances a single plan generated by the optimizer. We also plan to explore an integrated strategy, i.e., to extend the search space of a query optimizer to support instance-awareness as a plan is built. In this way, the generated plan is expected to be superior over that of MAPLE's plan.

8. REPEATABILITY ASSESSMENT RESULT

The repeatability committee has not been able to repeat the experiments of this paper due to the lack of appropriate hardware.

Acknowledgements This research is supported in part by NUS Grant R-252-000-271-112.

9. REFERENCES

[1] Microsoft SQL Server Library. `http://msdn2.microsoft.com/en-us/library/bb545450.aspx`.

[2] PostgreSQL. `http://www.postgresql.org/`.

[3] TPC BENCHMARK Decision Support. `http://www.tpc.org/tpcds/`.

[4] R. Bhashyam. TPC-D: the challenges, issues and results. *SIGMOD Rec.*, 25(4):89–93, 1996.

[5] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *VLDB*, pages 1216–1227, 2005.

[6] L. S. Colby, R. L. Cole, E. Haslam, N. Jazayeri, G. Johnson, W. J. McKenna, L. Schumacher, and D. Wilhite. Redbrick vista: Aggregate computation and management. In *ICDE*, pages 174–177, 1998.

[7] N. Dalvi, S. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. *Journal of Computer and System Sciences*, 66(4):728–762, 2003.

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

[9] T. Grust, M. V. Keulen, and J. Teubner. Accelerating xpath evaluation in any rdbms. *ACM Trans. Database Syst.*, 29(1):91–131, 2004.

[10] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[11] C. A. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *ICDE*, pages 1136–1145, 2007.

[12] C. A. Lang, B. Bhattacharjee, T. Malkemus, and K. Wong. Increasing buffer-locality for multiple index based scans through intelligent placement and index scan speed control. In *VLDB*, pages 1298–1309, 2007.

[13] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD*, pages 297–306, 1993.

[14] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[15] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. *SIGMOD Rec.*, 27(2):130–141, 1998.

[16] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *SWDB*, pages 131–150, 2003.

[17] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *SIGMOD*, pages 271–282, 1998.

[18] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, pages 533–544, 2007.

[19] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *VLDB*, pages 723–734, 2007.