Dependency-Aware Reordering for Parallelizing Query Optimization in Multi-Core CPUs

Wook-Shin Han
Department of Computer Engineering
Kyungpook National University
wshan@knu.ac.kr

Jinsoo Lee
Department of Computer Engineering
Kyungpook National University
jslee@www-db.knu.ac.kr

ABSTRACT

The state of the art commercial query optimizers employ cost-based optimization and exploit dynamic programming (DP) to find the optimal query execution plan (QEP) without evaluating redundant sub-plans. The number of alternative QEPs enumerated by the DP query optimizer can increase exponentially, as the number of joins in the query increases. Recently, by exploiting the coming wave of multi-core processor architectures, a state of the art parallel optimization algorithm [14], referred to as $\mathsf{PDP}_{\mathsf{sva}}$, has been proposed to parallelize the "time-consuming" DP query optimization process itself. While $\mathsf{PDP}_{\mathsf{sva}}$ significantly extends the practical use of DP to queries having up to 20-25 tables, it has several limitations: 1) supporting only the size-driven DP enumerator, 2) statically allocating search space, and 3) not fully exploiting parallelism. In this paper, we propose the first *generic* solution for parallelizing any type of bottom-up optimizer, including the graph-traversal driven type, and for supporting *dynamic* search allocation and *full* parallelism. This is a challenging problem, since recently developed, state of art DP optimizers such as $\mathsf{DP}_{\mathsf{cpp}}$ [21] and $\mathsf{DP}_{\mathsf{hyp}}$ [22] are very difficult to parallelize due to tangled dependencies in the join pairs they generate. Unless the solution is very carefully devised, a lot of synchronization conflicts are bound to occur. By viewing a serial bottom-up optimizer as one which generates a totally ordered sequence of join pairs in a streaming fashion, we propose a novel concept of *dependency-aware reordering*, which minimizes waiting time caused by dependencies of join pairs. To maximize parallelism, we also introduce a series of novel performance optimization techniques: 1) pipelining of join pair generation and plan generation; 2) the synchronization-free global MEMO; and 3) threading across dependencies. Through extensive experiments with various query topologies, we show that our solution supports any type of bottom up optimization, achieving linear speedup for each type. Despite the fact that our solution is generic, due to sophisticated optimization techniques, our generic parallel optimizer outperforms $\mathsf{PDP}_{\mathsf{sva}}$ tailored to size-driven enumeration. Experimental results also show that our solution is much more robust than $\mathsf{PDP}_{\mathsf{sva}}$ with respect to search space allocation.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems

General Terms

Algorithms

Keywords

Multi-cores, Parallel databases, Query optimization

1. INTRODUCTION

For the last few decades, the CPU performance has been significantly improved by increasing the clock rate according to Moore's law. However, fundamental physical limitations such as power consumption and heat generation clearly prevent us from relying on this trend any more [11, 12, 29, 30]. Instead, the industry has been improving the CPU performance by integrating more execution cores into each processor. The number of cores is expected to grow significantly over time [11, 35].

Recently, by exploiting this new wave of multi-core processor architectures, Han et al. [14] have proposed a novel framework referred to here as $\mathsf{PDP}_{\mathsf{sva}}$, to parallelize the "time-consuming" dynamic programming (DP) query optimization process itself. The DP query optimizer enumerates many alternative query execution plans (QEPs) for evaluating a declarative SQL query, while estimating the cost of each QEP, and then chooses the one with lowest estimated cost. The number of alternative QEPs enumerated by the DP query optimizer can increase exponentially, as the number of joins in the query increases. In fact, $\mathsf{PDP}_{\mathsf{sva}}$ significantly extends the practical use of DP to queries having up to 20-25 tables. We otherwise would have to depend on sub-optimal (randomized or greedy) heuristics [4, 19, 23, 31, 32] to complete query optimization in a reasonable time.

However, $\mathsf{PDP}_{\mathsf{sva}}$ has three limitations. First, it supports only one specific bottom-up optimizer, the size-driven DP optimizer. That is, it does not support recently developed, state of the art DP optimizers such as $\mathsf{DP}_{\mathsf{cpp}}$ [21] and $\mathsf{DP}_{\mathsf{hyp}}$ [22], which directly traverse a query graph to generate join pairs. Such optimizers have advantages over the size-driven enumeration. They can support early termination, since they can generate QEPs for all tables (more precisely, all quantifiers[1]) without generating QEPs for all smaller quantifier sets (i.e., not size-driven). Thus, as soon as we obtain a sufficiently good QEP or the estimated execution time of the obtained QEP is less than the expected remaining enumeration time, we can terminate the enumeration process early. $\mathsf{DP}_{\mathsf{hyp}}$ can handle

[1]Quantifiers correspond to the tuple variables seen in the FROM clause of the SQL query [24].

the widest class of non-inner joins very efficiently [22]. Therefore, there is a need for a generic framework that can parallelize any type of bottom-up optimizer so that it can support both existing and future bottom-up optimizers.

Secondly, assuming all cores are evenly loaded, PDP$_{\text{sva}}$ employs *static* search space allocation. Although the best allocation strategy of PDP$_{\text{sva}}$ can allocate search space to threads evenly [14], the slowest thread holds up all the other, faster threads, resulting in seriously unbalanced workloads. Therefore, the search allocation strategy must be *dynamic* to resolve this situation.

Lastly, PDP$_{\text{sva}}$ does not fully exploit parallelism since it merges per-thread MEMOs to the global MEMO in serial execution for each size of resulting quantifier sets. Here, each MEMO entry stores QEPs for a given quantifier sets. Thus, the best version of PDP$_{\text{sva}}$ achieves only up to 6.1 speedup for star queries[2] using 8 threads [14]. Therefore, in order to achieve linear speedup, all such serial steps must be executed by exploiting full parallelism.

In this paper, we propose the first *generic* solution for parallelizing any type of bottom-up optimizer, including the graph-traversal driven type, and for supporting *dynamic* search allocation and *full* parallelism. This is a challenging problem, since DP$_{\text{cpp}}$ and DP$_{\text{hyp}}$ are very difficult to parallelize [14]. Unless the solution is very carefully devised, a lot of synchronization conflicts are bound to occur. Figure 1 shows a motivating example using a sequence of join pairs (more precisely, a sequence of pairs of quantifier sets) generated by DP$_{\text{cpp}}$ or DP$_{\text{hyp}}$. Note that DP$_{\text{cpp}}$ and DP$_{\text{hyp}}$ generate the same sequence for equi-join. An arrow from one pair to another pair represents a dependency. As opposed to size-driven enumeration, the sizes of resulting quantifier sets do not monotonically increase. This leads to tradeoff between early termination and tangled dependencies. That is, since we obtain some QEPs for all quantifiers at the 17th pair of quantifier sets (q_1, $q_2q_3q_4$), we might be able to terminate the optimization process early, if the best QEP obtained thus far is good enough. On the other hand, the resulting tangled dependencies in the pairs of quantifier sets hinder parallelizing DP$_{\text{cpp}}$. For example, if a thread T_a processes the eighth pair (q_2, q_3q_4), and a thread T_b processes the fifth pair (q_3, q_4), T_a must wait until T_b finishes the processing of (q_3, q_4) first, since the quantifier set q_3q_4 has a dependency on the pair (q_3, q_4). If we change the order of the eighth and the eleventh pairs, T_a can process (q_1, q_4) without waiting.

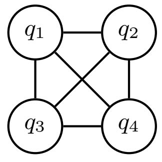
The overview of our solution is as follows. To parallelize any type of bottom-up enumeration, we view a serial bottom-up optimizer as one which generates a totally ordered sequence of pairs of quantifier sets in a streaming fashion. We buffer a fixed number of pairs and delay plan generation for the pairs buffered. Then on the fly, we convert the total order over these buffered pairs into a partial order over unordered groups of pairs, where threads can generate QEPs independently for all pairs within a group without waiting. These steps correspond to reordering of the original sequence so that the tangled dependencies in the original sequence are unraveled. We repeat these steps until we consume all pairs of quantifier sets.

Our contributions are as follows: 1) We propose the first generic framework for parallelizing any type of bottom-up optimization. 2) We propose a novel concept of dependency-aware reordering, which minimizes waiting time caused by dependencies of pairs of quantifier sets and propose a generic algorithm DPEGeneric for parallelizing query optimization. 3) To maximize parallelism, we propose a series of optimization techniques for DPEGeneric: pipelining of join pair generation and plan generation; the synchronization-free global MEMO; and threading across dependencies. 4) Through extensive experiments, we show that DPEGeneric supports any type of bottom up optimizer, achieving linear speedup for each type. Our algorithm is even better than the state of the art parallel optimizer tailored to size-based enumeration, PDP$_{\text{sva}}$. Our algorithm is also much more robust than PDP$_{\text{sva}}$ with respect to search space allocation.

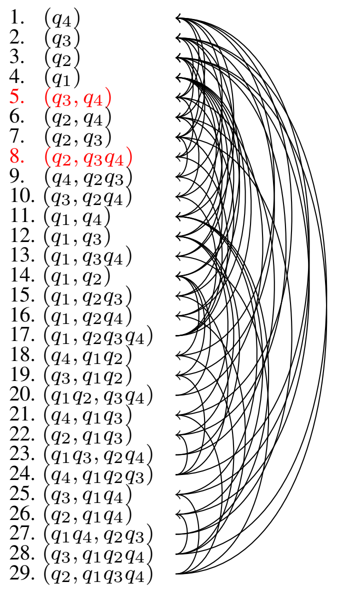
[2]A star query containing N quantifiers consists of a hub quantifier, N-1 neighboring quantifiers, and N-1 edges, where each neighboring quantifier is connected only to the hub quantifier.

SELECT $*$
FROM R_1 q_1, R_2 q_2,
R_3 q_3, R_4 q_4
WHERE $q_1.a_2 = q_2.a_1$ and
$q_1.a_3 = q_3.a_1$ and
$q_1.a_4 = q_4.a_1$ and
$q_2.a_3 = q_3.a_2$ and
$q_2.a_4 = q_4.a_2$ and
$q_3.a_4 = q_4.a_3$

SQL query statement



Query graph G



1. (q_4)
2. (q_3)
3. (q_2)
4. (q_1)
5. (q_3, q_4)
6. (q_2, q_4)
7. (q_2, q_3)
8. (q_2, q_3q_4)
9. (q_4, q_2q_3)
10. (q_3, q_2q_4)
11. (q_1, q_4)
12. (q_1, q_3)
13. (q_1, q_3q_4)
14. (q_1, q_2)
15. (q_1, q_2q_3)
16. (q_1, q_2q_4)
17. $(q_1, q_2q_3q_4)$
18. (q_4, q_1q_2)
19. (q_3, q_1q_2)
20. (q_1q_2, q_3q_4)
21. (q_4, q_1q_3)
22. (q_2, q_1q_3)
23. (q_1q_3, q_2q_4)
24. $(q_4, q_1q_2q_3)$
25. (q_3, q_1q_4)
26. (q_2, q_1q_4)
27. (q_1q_4, q_2q_3)
28. $(q_3, q_1q_2q_4)$
29. $(q_2, q_1q_3q_4)$

Figure 1: Pairs of quantifier sets generated by DP$_{\text{cpp}}$.

The rest of this paper is organized as follows. Section 2 reviews the current bottom-up join enumeration algorithms and the state of the art parallel algorithm for the size-based enumeration. The next two sections give the details of two generic parallel enumeration algorithms. Section 3 gives a basic parallel enumeration algorithm that can support any type of bottom-up enumeration algorithms, and Section 4 gives a theoretical framework for the dependency-aware reordering and an enhanced parallel enumeration algorithm exploiting the dependency-aware reordering. Section 5 presents a series of performance optimization techniques to maximize parallelism. Section 6 presents the results of performance evaluation. We compare our contributions with related work in Section 7, and conclude in Section 8.

2. BOTTOM-UP ENUMERATION

An enumeration algorithm is called *bottom-up* if it processes all smaller quantifier sets of both qs_1 and qs_2 before processing a pair of quantifier sets (qs_1, qs_2). To avoid evaluating redundant subplans, the bottom-up enumerator exploits the principle of optimality and stores the optimal QEPs in an in-memory quantifier set table (a.k.a. MEMO) [18]. Each entry in MEMO contains a list of QEPs for a quantifier set, and MEMO is typically implemented by using a hash table with the quantifier set as the key.

Existing bottom-up enumerators can be classified into the following three categories based on how they generate pairs of quantifier sets: 1) size-driven enumeration; 2) subset-driven enumeration; and 3) graph-traversal driven enumeration. We omit explanation of subset-driven enumeration since it is far slower than the graph-traversal driven enumeration [21] and is not used by commercial optimizers. For a more detailed description on the subset-driven enumeration, refer to reference [21].

2.1 Size-Driven Enumeration

The size-driven optimizer iteratively increases the size of the resulting quantifier set until it obtains the optimal QEP for all quantifiers in the query, starting from sets containing only a single quantifier. The join enumerators of conventional optimizers, such as those of DB2 and PostgreSQL [26], belong to this category.

At each iteration, to produce all QEPs representing quantifier sets of size SZ, the optimizer uses "nested loops" between quantifier sets of $smallSZ$ and quantifier sets of $largeSZ$ such that $largeSZ = SZ - smallSZ$. Here, for each pair of quantifier sets, the optimizer must check whether the two quantifier sets can form a feasible join; the two quantifier sets are disjoint and connected using at least one join predicate between them. If the connectivity check is disabled, Cartesian products in the resulting QEPs are permitted. Note that, to avoid unnecessary generations of infeasible pairs (i.e., overlapped pairs) of quantifier sets, a special index called the skip vector array (SVA) can be used [14]. We omit explanation of how the SVA can be exploited during enumeration since this is not our focus.

Algorithm 1 outlines the state of the art parallel optimization algorithm for size-driven enumeration, $\mathsf{PDP_{sva}}$. For each size of the resulting quantifier sets, we first allocate parts of the search space to m threads (Line 3), each of which then executes its allocated nested loops in parallel (Line 4). In order to merge per-thread MEMOs and prune expensive QEPs in the global MEMO, we need to wait until all threads finish their processing (Line 5). After completing the parallel QEP generation for each size of resulting quantifier sets, $\mathsf{PDP_{sva}}$ merges per-thread MEMOs to the global MEMO in serial execution (Line 6). To speed up the process of finding feasible join pairs, the SVA must be built over MEMO entries we just constructed (Line 7).

Algorithm 1 $\mathsf{PDP_{sva}}$

```
Input: a connected query graph with quantifiers $q_1, \cdots, q_N$
Output: an optimal bushy join tree
1: create table access plans and prune expensive QEPs for each quantifier.
2: for $SZ \leftarrow 2$ to $N$
3:   allocate to $m$ threads portions of nested loops for QEPs representing
     quantifier sets of size $SZ$;
4:   each thread generates QEPs in parallel by using its allocated nested
     loops. ;
5:   wait until all threads finish generating QEPs representing quantifier
     sets of size $SZ$ ;
6:   merge per-thread MEMOs into a global MEMO;
7:   build the skip vector array for MEMO entries corresponding to quan-
     tifier sets of size $SZ$.
8: return $MEMO[q_1 \cdots q_N]$;
```

2.2 Graph-Traversal Driven Enumeration

By directly traversing the query graph, the graph-traversal based optimizer generates a pair of quantifier sets that are disjoint and connected. Two state of the art algorithms—$\mathsf{DP_{cpp}}$ [21] and $\mathsf{DP_{hyp}}$ [22]—belong to this category. They have been developed very recently.

Both behave similarly except that $\mathsf{DP_{hyp}}$ can handle non-inner and anti-join predicates as well. Both generate pairs of quantifier sets (qs_1, qs_2) such that qs_1 is generated by enumerating all connected subgraphs of the query graph, and qs_2 is generated by enumerating all other connected subgraphs that are disjoint and connected to qs_1.

As shown in Figure 1, the resulting dependencies in the pairs of quantifier sets generated by $\mathsf{DP_{cpp}}$ or $\mathsf{DP_{hyp}}$ would prevent cleanly parallelizing $\mathsf{DP_{cpp}}$ or $\mathsf{DP_{hyp}}$. This motivates us to dynamically reorder pairs of quantifier sets on the fly to exploit parallelism.

3. BASIC PARALLEL ENUMERATION

In this section, we propose a basic parallel enumeration algorithm, BPEGeneric, for parallelizing *any* type of bottom-up optimizer. Algorithm 2 outlines the algorithm of BPEGeneric. Each thread invokes BPEGeneric concurrently. At each iteration in BPEGeneric, the algorithm obtains a pair of quantifier sets by calling the subroutine GetNextQSPair (Line 2). When GetNextQSPair generates a pair of quantifier sets (qs_1, qs_2), qs_2 is set to empty if GetNextQSPair generates a singleton set. If qs_2 is empty, BPEGeneric invokes the subroutine CreateTableAccessPlans to generate QEPs for accessing a single table (Line 9). Otherwise, it invokes the subroutine CreateJoinPlans (Line 11) to generate various join QEPs by trying out different access paths, join methods, and join orders. BPEGeneric then calls PrunePlans to prune any plan QEP_1 if there is another plan QEP_2 such that $cost(QEP_1) > cost(QEP_2)$, and whose properties (e.g., ordering of rows and partitioning, etc.) subsume those of QEP_1 (Line 12).

Algorithm 2 BPEGeneric (Basic Parallel Enumeration)

```
Input:
• $G$: a query graph with quantifiers $q_1, \cdots, q_N$
• MEMO: a concurrent global MEMO
Output: an optimal bushy join tree
1: loop
2:   atomic {$(qs_1, qs_2, e) \leftarrow$ GetNextQSPair ($G$);}
3:   if $e$ = NO_MORE_PAIR then
4:     return;
5:   atomic {
6:     if CheckDependency($qs_1, qs_2$) = true then retry;
7:   }
8:   if $qs_2 = \emptyset$ then /*$qs_1$ must be a singleton.*/
9:     $newPlans \leftarrow$ CreateTableAccessPlans($qs_1$);
10:  else
11:    $newPlans \leftarrow$ CreateJoinPlans ($MEMO[qs_1]$, $MEMO[qs_2]$);
12:  PrunePlans($MEMO[qs_1 \cup qs_2]$, $newPlans$); )
```

Here, we assume that GetNextQSPair always returns a feasible join pair of quantifier sets. With the state of the art graph-traversal driven enumerators such as $\mathsf{DP_{cpp}}$ and $\mathsf{DP_{hyp}}$, we can directly generate feasible pairs of quantifier sets only. On the other hand, in other join enumerators, we must execute a series of filters to find a feasible join pair. BPEGeneric is generic in that any type of join enumerator can be employed by calling the overloaded subroutine GetNextQSPair of a specific join enumerator. We note that BPEGeneric must use a *concurrent* global MEMO table, since each thread can 1) concurrently access the MEMO table and 2) concurrently operate on MEMO entries to add and remove QEPs. We use the concurrent hash map in Intel Threading Building Block [28], which is known for its good scalability.

Before calling CreateJoinPlans, BPEGeneric invokes the subroutine CheckDependency to ensure that neither qs_1 nor qs_2 is dependent on any pair of quantifier sets (qs'_1, qs'_2) being processed by all the other threads. To do so, it checks whether either qs_1 or qs_2 is a superset of $qs'_1 \cup qs'_2$. For example, suppose that ($q_1q_2q_3$, q_4) is about to be processed in thread T_a, and (q_1q_2, q_3) is being processed in thread T_b. In this case, thread T_a must wait until thread T_b finishes the processing of (q_1q_2, q_3) since the quantifier set $q_1q_2q_3$ is dependent on (q_1q_2, q_3).

To minimize memory allocation/deallocation synchronization conflicts, each thread uses a per-thread memory manager. Thus, when a thread allocates a memory region or deallocates its own previously allocated memory region, it does not need any synchronization efforts. However, when a thread T_a tries to deallocate memory regions (e.g., QEPs) allocated by another thread T_b as in $PrunePlans$, T_a adds a free request to the queue maintained by T_b, and T_b periodically deallocates its free requests.

BPEGeneric employs "dynamic" search space allocation as opposed to static search space allocation employed in [14]. That is, each thread consumes only one join pair at a time by calling GetNextQSPair. However, in [14], for each size of the resulting quantifier set, the search space (i.e., nested loops) for that size is first evenly divided. Then, each divided search space is allocated to a thread. Thus, although perfectly evenly divided search spaces are allocated to threads, the slowest thread can hold up all the other, faster threads, leading to severely biased workloads. Since the unit of search space allocation in BPEGeneric is a pair of quantifier sets, the maximum delay incurred by BPEGeneric is the processing time of one pair of quantifier sets, which is negligible.

However, BPEGeneric pays the price for dynamic search space allocation, incurring a lot of synchronization overhead by using GetNextQSPair, CheckDependency, and the concurrent MEMO. Only one thread executes GetNextQSPair at a time, so the other threads must wait if they try to invoke it concurrently. Due to tangled dependencies, when a pair of quantifier sets qs is being processed by a thread T_a, T_a must execute CheckDependency to check whether there is another thread T_b currently processing any other pair that qs depends on.

This presents the question: "Can we avoid such synchronization overhead while using dynamic search space allocation?" This motivates us to develop a completely new approach that exploits 1) separation of join pair generation and plan generation, and 2) dependency-aware reordering. That is, to avoid the synchronization overhead incurred by GetNextQSPair, we separate join pair generation from plan generation. That is, we first generate pairs of quantifier sets by using one thread, and then, perform plan generation using multiple threads. We explain dependency-aware reordering in the next section.

4. SCALABLE PARALLEL ENUMERATION

The formal foundation of our dependency-aware reordering technique is presented in Section 4.1. More specifically, we propose the novel concepts of valid reordering, partial orders over search spaces, and dependency-aware reordering based on group topological sort. In Section 4.2, we propose an enhanced generic algorithm called DPEGeneric that exploits dependency-aware reordering as well as separation of join pair generation from plan generation, and we show the validity of the join pair sequence reordered by DPEGeneric.

4.1 Partial Orders over Search Spaces

To avoid synchronization conflicts, we convert the total order over join pairs into a partial order over unordered groups of pairs. The rational for grouping is that threads can generate QEPs independently for all pairs within a group without synchronization conflicts. However, grouping may reduce the opportunity for early termination. Thus, a grouping method supporting early termination must be devised. Grouping also may increase waiting time. For example, although a group G_1 is dependent on a group G_2, we may process some entries in G_1 before completely processing all entries of G_2. This phenomenon is explained in detail in Section 5.3.

Before explaining detailed grouping methods, we formally define several important concepts. When we reorder an incoming sequence of pairs of quantifier sets (called *join pair sequence*), we make sure that the reordered sequence is valid as well. Otherwise, we can not guarantee that the reordered sequence can generate the same final QEP as the original sequence. The following definition formally defines validity of a join pair sequence.

Definition 1. *A join pair sequence S is valid if any pair in S depends solely on its preceding pairs. Otherwise, the join pair sequence is invalid.*

We now define the important property of valid reordering in Definition 2.

Definition 2. *A reordering is valid if it transforms one valid join sequence into another valid join sequence.*

In order to find the valid reorderings for a streaming join pair sequence, we construct inherent partial orders over a set of unordered groups from the streaming join pair sequence, where no dependencies exist among entries in such a group. Then, we generate a totally ordered sequence by using group topological sort over the partial order. The formal definition of group topological sort is in Definition 3. Theorem 1 states the validity of a reordered sequence generated by group topological sort.

Definition 3. *For a given partial order P over a set of unordered groups, group topological sort performs topological sort [8] over P and obtains a totally ordered group sequence. Then, for each group G obtained, group topological sort generates a permutation for G.*

Theorem 1. *For a given partial order P over a set of unordered groups, any totally ordered sequence generated by group topological sort from P is valid.*

PROOF: We prove by contradiction. Assume that group topological sort generates an invalid sequence S'. Let S' be $s_1 s_2 ... s_m$. Then, by Definition 1, there exist two entries s_i and s_j in the sequence such that $i < j$, and s_i depends on s_j. Let unordered group G_1 and G_2 be the unordered groups that contain s_i and s_j, respectively. Note that there is no order in entries in a group. Thus, due to the property of the partial order, G_1 must depend on G_2. Since group topological sort performs topological sort over the set of unordered groups, all entries in G_2 must precede all entries in G_1. Thus, it contradicts the assumption above that s_i precedes s_j. □

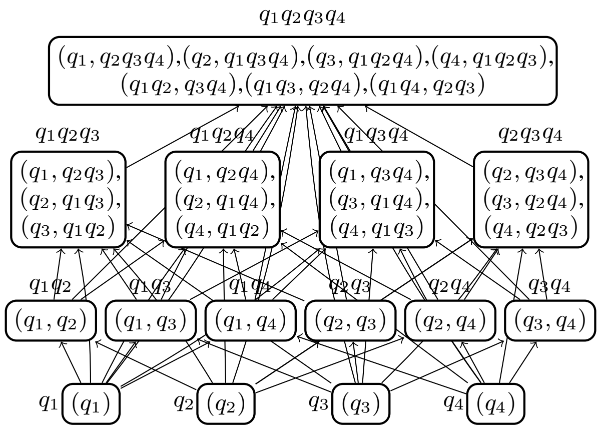
Since there can exist several partial orders over a set of unordered groups from the original join pair sequence, the one which best maximizes parallelism should be chosen. To measure goodness of a partial order, we use the following three criteria: 1) early termination is supported; 2) the cost of maintaining the partial order over streaming join pairs is minimized in the multi-threaded environment; and 3) waiting time due to dependencies is minimized. Therefore, if a partial order satisfies all three criteria, then parallelism is maximized while early termination is supported.

Different partial orders can be formed depending on how we group join entries as follows. We note that, in order to use Theorem 1, each group must have no dependencies among entries within the group.

1. Group by the resulting quantifier set. Each group is called an `RQS` group.
2. Group by the size of the resulting quantifier set. Each group is called an `SRQS` group.
3. Group by the size of the larger quantifier set in the join pair. Each group is called an `SLQS`.

4.1.1 *Group by the resulting quantifier set*

Let $U_{\texttt{RQS}}$ be a set of RQSs. We can define a binary relation $\preceq_{\texttt{RQS}}$ on $U_{\texttt{RQS}}$, where for any (rqs, rqs') in $\preceq_{\texttt{RQS}}$, rqs is a subset of rqs'. The binary relation $\preceq_{\texttt{RQS}}$ is a partial order since it is 1) reflexive ($rqs \preceq_{\texttt{RQS}} rqs$), 2) anti-symmetric (if $rqs \preceq_{\texttt{RQS}} rqs'$ and $rqs' \preceq_{\texttt{RQS}} rqs$, then rqs = rqs'), and 3) transitive (if $rqs \preceq_{\texttt{RQS}} rqs'$ and $rqs' \preceq_{\texttt{RQS}} rqs''$, then $rqs \preceq_{\texttt{RQS}} rqs''$), for all rqs, rqs', rqs'' in $U_{\texttt{RQS}}$. Figure 2 shows the partial order $\preceq_{\texttt{RQS}}$ for the sequence of pairs of quantifier sets in Figure 1. Each rounded box represents an RQS.



$q_1q_2q_3q_4$
$(q_1, q_2q_3q_4)$,$(q_2, q_1q_3q_4)$,$(q_3, q_1q_2q_4)$,$(q_4, q_1q_2q_3)$, (q_1q_2, q_3q_4),(q_1q_3, q_2q_4),(q_1q_4, q_2q_3)
$q_1q_2q_3$
(q_1, q_2q_3), (q_2, q_1q_3), (q_3, q_1q_2)
$q_1q_2q_4$
(q_1, q_2q_4), (q_2, q_1q_4), (q_4, q_1q_2)
$q_1q_3q_4$
(q_1, q_3q_4), (q_3, q_1q_4), (q_4, q_1q_3)
$q_2q_3q_4$
(q_2, q_3q_4), (q_3, q_2q_4), (q_4, q_2q_3)
q_1q_2
q_1q_3
q_1q_4
q_2q_3
q_2q_4
q_3q_4
(q_1, q_2)
(q_1, q_3)
(q_1, q_4)
(q_2, q_3)
(q_2, q_4)
(q_3, q_4)
q_1 (q_1)
q_2 (q_2)
q_3 (q_3)
q_4 (q_4)

Figure 2: An example of partial order $\preceq_{\texttt{RQS}}$.

We explain how to construct $\preceq_{\texttt{RQS}}$ from the incoming join pair sequence. For each entry (qs_1, qs_2), we create an RQS corresponding to $qs_1 \cup qs_2$ if the RQS has not been created, and add two incoming edges to that RQS, one from the RQS corresponding to qs_1 and the other from the RQS corresponding to qs_2. Due to the transitivity property of $\preceq_{\texttt{RQS}}$, one might think that we don't need to add any edge between an $\texttt{RQS}_i$ and an $\texttt{RQS}_j$ when $\texttt{RQS}_i$ is a subset of $\texttt{RQS}_j$ and $|\texttt{RQS}_i| < |\texttt{RQS}_j| + 1$. For example, the addition of the direct edge between RQS $q_1q_2q_3$ and RQS q_3 doesn't seem to be necessary. However, in general, its omission is not allowed. Consider a query where there is only one join predicate $q_1.a + q_2.b = q_3.c$ between q_1q_2 and q_3. This leads to a hyperedge [22] in the query graph. Thus, the direct edge between RQS $q_1q_2q_3$ and RQS q_3 must not be removed.

To use Theorem 1 for the partial order $\preceq_{\texttt{RQS}}$, all entries of each group in $\preceq_{\texttt{RQS}}$ have no dependencies among them. Lemma 1 guarantees this property.

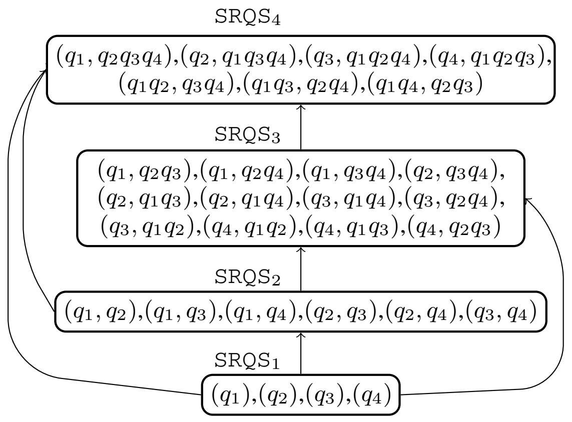
Lemma 1. *No dependencies exist among entries in an RQS.*

PROOF: Refer to [15]. □

We now discuss the pros and cons of the partial order $\preceq_{\texttt{RQS}}$. 1) In terms of early termination, $\preceq_{\texttt{RQS}}$ does not support early termination, since the topmost RQS $q_1q_2q_3q_4$ depends on all RQSs underneath. 2) In terms of maintenance cost, maintaining $\preceq_{\texttt{RQS}}$ would be expensive due to concurrent edge removals in multi-thread environments. That is, when thread T_a finishes processing an RQS, T_a must remove all of its outgoing edges. This can incur significant synchronization cost. 3) In terms of waiting time, as soon as the processing of an RQS is completed, we can find a set of RQSs that have no incoming edges, and thus, can be processed concurrently. The entries in an RQS can also be processed concurrently according to Lemma 1, minimizing waiting time.

4.1.2 *Group by the size of the resulting quantifier set*

Let $U_{\texttt{SRQS}}$ be a set of SRQSs. We denote $\texttt{SRQS}_i$ as an SRQS that contains every join pair whose resulting quantifier set size is i. We can define a binary relation $\preceq_{\texttt{SRQS}}$ on $U_{\texttt{SRQS}}$, where for any ($\texttt{SRQS}_i$, $\texttt{SRQS}_j$) in $\preceq_{\texttt{SRQS}}$, $i \leq j$. The binary relation $\preceq_{\texttt{SRQS}}$ is a partial order since it is 1) reflexive ($\texttt{SRQS}_i \preceq_{\texttt{SRQS}} \texttt{SRQS}_i$), 2) anti-symmetric (if $\texttt{SRQS}_i \preceq_{\texttt{SRQS}} \texttt{SRQS}_j$ and $\texttt{SRQS}_j \preceq_{\texttt{SRQS}} \texttt{SRQS}_i$, then $\texttt{SRQS}_i$ = $\texttt{SRQS}_j$), and 3) transitive (if $\texttt{SRQS}_i \preceq_{\texttt{SRQS}} \texttt{SRQS}_j$ and $\texttt{SRQS}_j \preceq_{\texttt{SRQS}} \texttt{SRQS}_k$, then $\texttt{SRQS}_i \preceq_{\texttt{SRQS}} \texttt{SRQS}_k$), for all $\texttt{SRQS}_i$, $\texttt{SRQS}_j$, $\texttt{SRQS}_k$ in $U_{\texttt{SRQS}}$. Figure 3 shows the partial order $\preceq_{\texttt{SRQS}}$ for the sequence of pairs of quantifier sets in Figure 1. Each rounded box represents an SRQS.



$\texttt{SRQS}_4$
$(q_1, q_2q_3q_4)$,$(q_2, q_1q_3q_4)$,$(q_3, q_1q_2q_4)$,$(q_4, q_1q_2q_3)$, (q_1q_2, q_3q_4),(q_1q_3, q_2q_4),(q_1q_4, q_2q_3)
$\texttt{SRQS}_3$
(q_1, q_2q_3),(q_1, q_2q_4),(q_1, q_3q_4),(q_2, q_3q_4), (q_2, q_1q_3),(q_2, q_1q_4),(q_3, q_1q_4),(q_3, q_2q_4), (q_3, q_1q_2),(q_4, q_1q_2),(q_4, q_1q_3),(q_4, q_2q_3)
$\texttt{SRQS}_2$
(q_1, q_2),(q_1, q_3),(q_1, q_4),(q_2, q_3),(q_2, q_4),(q_3, q_4)
$\texttt{SRQS}_1$
(q_1),(q_2),(q_3),(q_4)

Figure 3: An example of partial order $\preceq_{\texttt{SRQS}}$.

Now, we explain how to construct $\preceq_{\texttt{SRQS}}$ from the incoming join pair sequence. Given a query having n quantifiers, we create in advance n SRQSs and add outgoing edges from $\texttt{SRQS}_i$ to $\texttt{SRQS}_j$ where $i + 1 \leq j \leq n$. For each entry (qs_1, qs_2) in the join pair sequence, we add that entry to $\texttt{SRQS}_{|qs_1 \cup qs_2|}$.

Lemma 2. *No dependencies exist among entries in an SRQS.*

PROOF: Refer to [15]. □

Next, we discuss the pros and cons of the partial order $\preceq_{\texttt{SRQS}}$. Like $\preceq_{\texttt{RQS}}$, $\preceq_{\texttt{SRQS}}$ does not support early termination either, since the topmost SRQS depends on all SRQSs underneath. However, the cost of maintaining $\preceq_{\texttt{SRQS}}$ is negligible, since we only need to access SRQSs in the increasing order of the size of the resulting quantifier sets. No synchronization is needed to process entries within the same SRQS, thus minimizing waiting time.

4.1.3 *Group by the size of the larger quantifier set in the join pair*

Let $U_{\texttt{SLQS}}$ be a set of SLQSs. We denote $\texttt{SLQS}_i$ as an SLQS that contains every join pair whose larger quantifier set size is i. To represent a set of single quantifiers (not join quantifiers), $\texttt{SLQS}_0$ is used. We can define a binary relation $\preceq_{\texttt{SLQS}}$ on $U_{\texttt{SLQS}}$, where for any ($\texttt{SLQS}_i$, $\texttt{SLQS}_j$) in $\preceq_{\texttt{SLQS}}$, $i \leq j$. We can easily verify that the binary relation $\preceq_{\texttt{SLQS}}$ over $U_{\texttt{SLQS}}$ is a partial order. Figure 4 shows the partial order $\preceq_{\texttt{SLQS}}$ for the sequence of pairs of quantifier sets in Figure 1. Each rounded box represents an SLQS.

The construction of $\preceq_{\texttt{SLQS}}$ from the incoming join pair sequence is similar to that of $\preceq_{\texttt{SRQS}}$. That is, we create the partial order for a given query in advance. For each pair (qs_1, qs_2) in the incoming join sequence, we add this pair to $\texttt{SLQS}_{max(|qs_1|,|qs_2|)}$ if qs_2 is not empty. If qs_2 is empty, we add the pair to $\texttt{SLQS}_0$.

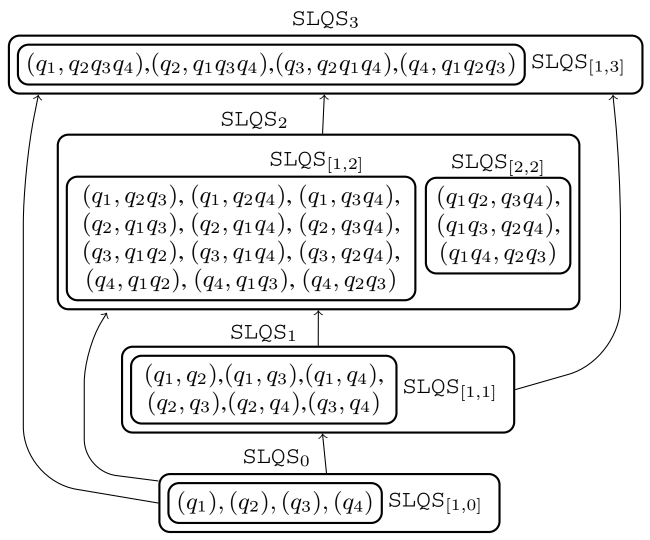

$\mathtt{SLQS}_3$
$(q_1, q_2q_3q_4)$,$(q_2, q_1q_3q_4)$,$(q_3, q_2q_1q_4)$,$(q_4, q_1q_2q_3)$ $\mathtt{SLQS}_{[1,3]}$
$\mathtt{SLQS}_2$
$\mathtt{SLQS}_{[1,2]}$
(q_1, q_2q_3), (q_1, q_2q_4), (q_1, q_3q_4), (q_2, q_1q_3), (q_2, q_1q_4), (q_2, q_3q_4), (q_3, q_1q_2), (q_3, q_1q_4), (q_3, q_2q_4), (q_4, q_1q_2), (q_4, q_1q_3), (q_4, q_2q_3)
$\mathtt{SLQS}_{[2,2]}$
(q_1q_2, q_3q_4), (q_1q_3, q_2q_4), (q_1q_4, q_2q_3)
$\mathtt{SLQS}_1$
(q_1, q_2),(q_1, q_3),(q_1, q_4), (q_2, q_3),(q_2, q_4),(q_3, q_4) $\mathtt{SLQS}_{[1,1]}$
$\mathtt{SLQS}_0$
(q_1), (q_2), (q_3), (q_4) $\mathtt{SLQS}_{[1,0]}$

Figure 4: An example of partial order $\preceq_{\mathtt{SLQS}}$.

Lemma 3. *No dependencies exist among entries in an* SLQS*.*

PROOF: Refer to [15]. □

We can further divide $\mathtt{SLQS}_i$ into a set of subgroups by the size of the smaller quantifier set of the join pair. $\mathtt{SLQS}_{[j,i]}$ ($j \leq i$) denotes a subgroup of $\mathtt{SLQS}_i$ that contains every join pair whose smaller quantifier set size is j. As a special case, $\mathtt{SLQS}_{[1,0]}$ represents $\mathtt{SLQS}_0$.

Let us discuss pros and cons of the partial order $\preceq_{\mathtt{SLQS}}$. Unlike $\preceq_{\mathtt{RQS}}$ and $\preceq_{\mathtt{SRQS}}$, $\preceq_{\mathtt{SLQS}}$ supports early termination. Suppose that a given query has N quantifiers. Then, we have N levels in $\preceq_{\mathtt{SLQS}}$. When we reach level $\lceil \frac{N}{2} \rceil$, we can obtain QEPs containing all quantifiers by processing join entries in $\mathtt{SLQS}_{[N-\lceil \frac{N}{2} \rceil, \lceil \frac{N}{2} \rceil]}$. For example, when we process entries in $\mathtt{SLQS}_{[2,2]}$ in Figure 4, we can obtain QEPs containing all four quantifiers in the query. Thus, we may terminate the optimization process if the obtained QEPs are good enough. Like $\preceq_{\mathtt{SRQS}}$, the cost of maintaining $\preceq_{\mathtt{SLQS}}$ is negligible, and waiting time is minimized since it has the same number of edges in the partial order as $\preceq_{\mathtt{SRQS}}$.

It is clear that the partial order $\preceq_{\mathtt{SLQS}}$ is the best one among the three we have considered. Using the same experimental setup in Section 6, we empirically verified that, with $\preceq_{\mathtt{SLQS}}$, the first join entry that contains all quantifiers is processed just after processing 33.4% $\sim$ 62.5% of the total join entries, depending on query topologies. Thus, hereafter we use $\preceq_{\mathtt{SLQS}}$ as a default partial order. We will empirically show that we can achieve linear speedup using $\preceq_{\mathtt{SLQS}}$ in Section 6.

4.1.4 Discussion on grouping

We proposed three grouping methods and concluded that grouping by SLQS was the best grouping method of the three. However, there might exist better approaches than SLQS. In fact, an SLQS may be partitioned into smaller groups. In this case, parallelism and the partial order management overhead could grow as we allow more groups. Thus, an interesting future topic would be to find the optimal grouping method.

4.2 Dependency-Aware Parallel Enumeration

To convert a totally ordered join pair sequence to a partial order $\preceq_{\mathtt{SLQS}}$[3], we allocate a concurrent dependency buffer $\mathcal{B}_{[j,i]}$ for each $\mathtt{SLQS}_{[j,i]}$. We denote $\mathcal{B}_{[*,i]}$ as a set of dependency buffers corresponding to $\mathtt{SLQS}_i$. The dependency buffer $\mathcal{B}_{[j,i]}$ is implemented as a concurrent queue that supports two core operations: Push and Pop. Here, we do not need any synchronization for Push, since we separate join pair generation from plan generation. We need synchronization for Pop since multiple threads can consume entries from a dependency buffer concurrently. However, the overhead for such synchronization is negligible as we will see in our extensive experiments in Section 6. Note that we do not use monitoring-based load balancing techniques as in [25], since such monitoring can incur non-negligible overhead in CPU-bound jobs. We also note that we have to buffer only a fixed size of join pairs to avoid a huge memory footprint size.

Algorithm 3 shows a dependency-aware parallel enumeration algorithm called DPEGeneric. The main thread invokes DPEGeneric. It then invokes the subroutine EnumAndBuildPartialOrder to convert a fixed number of join pairs into a partial order over SLQSs. To do so, EnumAndBuildPartialOrder repeatedly invokes GetNextQSPair to generate a join pair and pushes the pair to the corresponding dependency buffer, delaying plan generation for the pairs generated (Line 2). Here, to control the maximum number of join pairs to generate, MAXENUMCNT is used. More specifically, in EnumAndBuildPartialOrder, if a join pair generated is (qs_1, qs_2), this pair is pushed to $\mathcal{B}_{[|qs_1|,|qs_2|]}$. After that, for each $\mathtt{SLQS}_i$ (Line 4), all threads concurrently consume join pairs in $\mathcal{B}_{[j,i]}$ for all j by executing GenerateQEPs (Lines 5 and 6). This step corresponds to group topological sort in parallel. After the processing of $\mathtt{SLQS}_i$ is completed (Line 7), the main thread merges per-thread MEMOs into the global MEMO (Line 8). We repeat these steps until all join pairs are consumed.

Algorithm 3 DPEGeneric (Dependance-Aware Parallel Enumeration)

```
Input:
• G: a connected query graph with quantifiers $q_1, \cdots, q_N$
• MEMO: a non-concurrent global MEMO
• $memo_t$: a local memo for thread $t$
Output: an optimal bushy join tree
    Variable enumeration buffer $\mathcal{B}$
 1: loop
 2:   $e \leftarrow$ EnumAndBuildPartialOrder($G$, $\mathcal{B}$, MAXENUMCNT);
 3:   if $e$ = NO_MORE_PAIR then break;
 4:   for $i \leftarrow 0$ to $N-1$ /*increase the larger quantifier set size*/
 5:     for $t \leftarrow 1$ to $m$ /*Execute $m$ threads in parallel*/
 6:       $pool$.SubmitJob(GenerateQEPs($\mathcal{B}_{[*,i]}$,$memo_t$));
 7:     $pool$.Sync();
 8:     MergeAndPrunePlans(MEMO, $\{memo_1, \cdots, memo_m\}$);
 9: return MEMO$[q_1 \cdots q_N]$;
Function GenerateQEPs
Input:
• $\mathcal{B}_{[*,i]}$: a set of enumeration buffers large size is $i$
• $memo_t$: a local memo for thread $t$
 1: $j \leftarrow 1$;
 2: repeat
 3:   loop
 4:     atomic $\{(qs_1, qs_2, e) \leftarrow$ Pop ($\mathcal{B}_{[j,i]}$);$\}$
 5:     if $e$ = NO_MORE_PAIR then break;
 6:     if $qs_2 = \emptyset$ then /*$qs_1$ must be a singleton.*/
 7:       $newPlans \leftarrow$ CreateTableAccessPlans($qs_1$);
 8:     else
 9:       $newPlans \leftarrow$ CreateJoinPlans(MEMO$[qs_1]$, MEMO$[qs_2]$);
10:     PrunePlans($memo_t[qs_1 \cup qs_2], newPlans$);
11:   $j \leftarrow j + 1$;
12: until $j \leq i$ and $j + i \leq N$
```

[3]For ease of exposition, we use SLQS as a default partial order. However, other partial orders can easily applied to Algorithm 3.

Like BPEGeneric, DPEGeneric employs dynamic search space allocation. That is, in GenerateQEPs, each tread only consumes one join pair at a time by calling Pop (Line 4). Since the unit of search space allocation in DPEGeneric is a pair of quantifier sets, the maximum delay among threads incurred by GenerateQEPs is the processing time of one pair of quantifier sets, which is negligible. As opposed to BPEGeneric, DPEGeneric has no GetNextQSPair and CheckDependency conflicts.

Theorem 2. *For a given valid sequence of join pairs, DPEGeneric always generates a valid reordered sequence of the join pairs.*

PROOF: Let the original join sequence generated by the serial join pair enumerator be $S = s_1 \cdots s_n$. S must be valid. The main thread divides S into a series of subsequences by buffering each subsequence in EnumAndBuildPartialOrder at a time. Here, the size of each subsequence except the last subsequence is MAXENUMCNT. Thus, it is sufficient to show that DPEGeneric generates the valid reordered sequence for each subsequence buffered using Lines 4 - 6 in DPEGeneric. The for loop in Line 4 (corresponding to topological sort over SLQSs) and the repeat-until loop in GenerateQEPs (corresponding to a permutation using multiple threads) correspond to group topological sort. Due to Theorem 1, DPEGeneric generates a valid reordered sequence for each buffered subsequence. This completes the proof. □

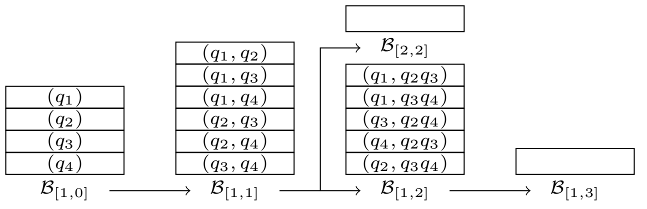
Example 1. *Figure 5 depicts how DPEGeneric operates using an example. Suppose that MAXENUMCNT is set to 15. The main thread first invokes EnumAndBuildPartialOrder(G, $\mathcal{B}$, 15) in serial (Line 2). Figure 5(a) shows a snapshot of a set of dependency buffers corresponding to a partial order over SLQSs after buffering the first 15 entries from the incoming sequence of join pairs of Figure 1. For each dependency buffer in increasing order of the larger quantifier set size, each thread concurrently executes GenerateQEPs using its per-thread MEMO (Lines 4 $\sim$ 6). That is, each thread T_a concurrently consumes entries in $\mathcal{B}_{[*,i]}$ using $memo_a$. After that, the main thread merges all per-thread MEMOs into the global MEMO. We repeat these above steps until we consume all join entries. That is, the main thread invokes EnumAndBuildPartialOrder(G, $\mathcal{B}$, 15) again (Line 2). Figure 5(b) shows a snapshot of a set of dependency buffers after the remaining 14 entries are buffered (Line 2).*

5. MAXIMIZING PARALLELISM, IN DEPTH

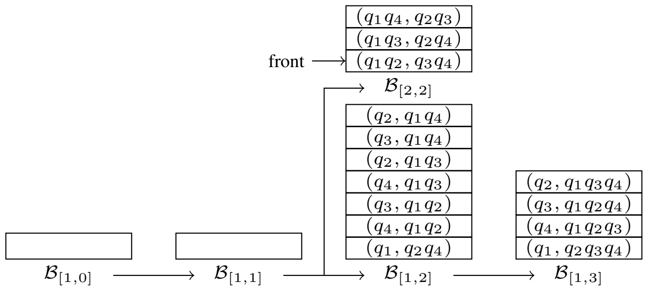
Although DPEGeneric solves fundamental problems of BPEGeneric (GetNextQSPair and CheckDependency conflicts), two steps in DPEGeneric still run in serial: 1) join pair generation and 2) the merging of per-thread MEMOs into the global MEMO. The third problem with DPEGeneric is that the dependency buffers must be processed in sequence, so buffered join pairs in other dependency buffers may be ready for processing, but the threads can't process such pairs in advance of the current dependency buffer. This is especially problematic when the current dependency buffer does not have a sufficient number of join pairs for all threads, since some threads are not fully utilized. In this section, we propose three optimization techniques resolving all three problems.

5.1 Pipelining of Join Pair Generation and Plan Generation

Suppose that there are m threads available. In order to utilize the other m - 1 threads while the main thread generates join pairs, we leverage *pipeline parallelism* [13]. To do so, we regard the main thread generating join pairs as a producer and the other threads generating QEPs for the join pairs generated as consumers.



(a) After the first 15 entries are buffered.



(b) After the remaining 14 entries are buffered.

Figure 5: An example of DPEGeneric.

More specifically, the main thread first generates and buffers a fixed number of join pairs. After that, all threads except the main thread consume the buffered join pairs. At the same time, the main thread generates a fixed number of join pairs for the next iteration. Note that the time needed to generate a fixed number of join pairs is much smaller than the time needed to generate QEPs for those join pairs generated. After the main thread finishes join pair generation, together with the other threads, it participates in consuming the remaining buffered sequence. This way, all m threads are fully utilized. To avoid synchronization conflicts on dependency buffers, we use dual buffers for each dependency buffer $\mathcal{B}_{[j,i]}$.

5.2 Synchronization-Free Global MEMO

Unlike BPEGeneric, DPEGeneric uses per-thread MEMOs to avoid synchronization conflicts during plan generation. However, it needs the additional step of merging per-thread MEMOs to the global MEMO. Another problem with per-thread MEMOs is that the total memory footprint size, in the worst case, can grow in proportion to the number of threads. By carefully analyzing operational semantics for the global MEMO, we develop a synchronization-free global MEMO resolving these problems. To this end, we propose two novel concepts: 1) equivalence class grouping and 2) earmark-and-pinning.

Before explaining these concepts in detail, we briefly explain the structure of the MEMO table. The MEMO table of conventional optimizers, such as those of DB2 and PostgreSQL [26], is typically implemented as a chained hash table with the quantifier set as the key. The MEMO table can be dynamically reconstructed using a different hash function and a different hash table size, e.g., when the length of a hash chain is larger than a predefined threshold. Each hash bucket corresponds to a MEMO entry that contains a resulting quantifier set and a pointer to the plan chain for the quantifier set. Figure 6 depicts a MEMO table.

As shown in Figure 6, we note that there are two different types of linked lists in the MEMO table: the hash chain and the plan chain. Thus, two different types of synchronization conflicts can

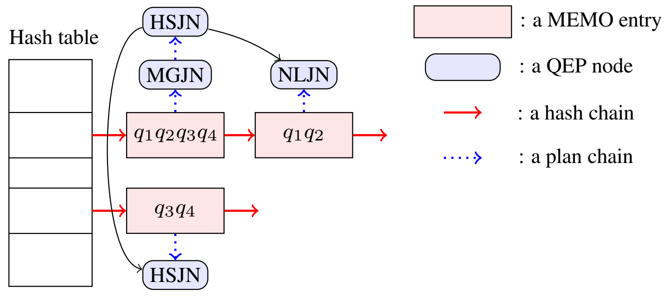


Figure 6: A MEMO table.

occur if threads manipulate these chains concurrently. To remove synchronization conflicts for the plan chain, we propose the concept of equivalence class grouping. To remove synchronization conflicts for the hash chain, we propose the concept of earmark-and-pinning.

Equivalence class grouping: We first define an equivalence class in Definition 4, and describe how equivalence classes are used to remove synchronization conflicts for the plan chain.

Definition 4. *An equivalence class in the dependency buffer $\mathcal{B}_{[j,i]}$ is a set of join pairs in $\mathcal{B}_{[j,i]}$ whose resulting quantifier sets are the same.*

Equivalence class grouping for $\mathcal{B}_{[j,i]}$ subgroups join pair entries in $\mathcal{B}_{[j,i]}$ by their equivalence classes. The set of equivalence classes in $\mathcal{B}_{[j,i]}$ defines a partition over the set of join pairs in $\mathcal{B}_{[j,i]}$. To support equivalence class grouping, each entry in $\mathcal{B}_{[j,i]}$ stores an equivalence class, rather than a single join pair. The following lemma states that, with equivalence class grouping, threads incur no synchronization conflicts for any plan chain in the global MEMO.

Lemma 4. *Given dependency buffers $\mathcal{B}_{[*,i]}$ currently being processed by threads, if all entries in an equivalence class are consumed by only one thread, there occurs no synchronization conflict for plan chains.*

PROOF: A plan chain is updated only by the subroutine PrunePlans, where newly created QEPs are added to the plan chain, and expensive QEPs are pruned from the plan chain. Thus, it is clear that, unless any two threads invoke PrunePlans for the same MEMO entry, no synchronization conflict for plan chains occurs. Suppose that two threads T_a and T_b process two different join pairs (qs_1, qs_2) in $\mathcal{B}_{[j,i]}$ and (qs'_1, qs'_2) in $\mathcal{B}_{[j',i]}$ concurrently. When $j \neq j'$, T_a and T_b invoke PrunePlans for different MEMO entries, since $|qs_1 \cup qs_2| \neq |qs'_1 \cup qs'_2|$. Even when $j = j'$, T_a and T_b invoke PrunePlans for different MEMO entries, since both join pairs are from different equivalence classes, i.e., $qs_1 \cup qs_2 \neq qs'_1 \cup qs'_2$. □

We now discuss the ratio of the maximum size of an equivalence class over the total number of join pairs[4] according to the topology of the query graph. The ratio must be small, since the unit of allocation to threads is an equivalence class. For example, if we have a clique query[5] having 18 quantifiers, the ratio is only 0.023%. Furthermore, since we buffer a fixed number of join pairs at a time, the size of a buffered equivalence class is much smaller than these theoretical values.

Earmark-and-Pinning: In order to remove synchronization conflicts for the hash chain, the main thread generating join pairs preallocates a MEMO entry for each equivalence class buffered if such MEMO entry has not been previously created. This step is called *earmarking*. To pin the MEMO entry earmarked, a join pair entry in the dependency buffer stores a pointer to it. Thus, to support earmark-and-pinning, each entry in the dependency buffer is changed to the form of ($m[qs_1 \cup qs_2]$, $m[qs_1]$, $m[qs_2]$), where $m[qs]$ represents the memory address of the MEMO entry corresponding to qs. This way, during plan generation, threads directly access MEMO entries without accessing any hash chain or hash array in the MEMO table.

Example 2. *Figure 7 depicts a synchronization-free global MEMO that exploits equivalence grouping and earmark-and-pinning. Recall Example 1. Suppose that the main thread invokes EnumAndBuildPartialOrder(G, $\mathcal{B}$, 15) for the second time. As shown in this figure, each entry in a dependency buffer is an equivalence class. For example, in $\mathcal{B}_{[2,2]}$, the MEMO entry for the equivalence class $q_1q_2q_3q_4$ has not been created in the previous iteration. The main thread earmarks a MEMO entry for the equivalence class. During plan generation, when an equivalence class is popped by Thread T_a, T_a directly accesses the MEMO entry corresponding to the equivalence class using $m[q_1q_2q_3q_4]$ without accessing any hash chain or hash array. Note that all the other threads except T_a never access $m[q_1q_2q_3q_4]$ concurrently according to Lemma 4.*

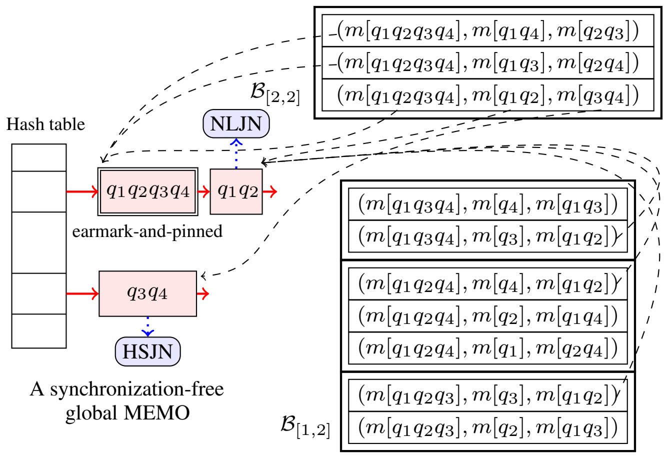

$(m[q_1q_2q_3q_4], m[q_1q_4], m[q_2q_3])$
$(m[q_1q_2q_3q_4], m[q_1q_3], m[q_2q_4])$
$(m[q_1q_2q_3q_4], m[q_1q_2], m[q_3q_4])$
$\mathcal{B}_{[2,2]}$
Hash table
NLJN
$q_1q_2q_3q_4$
q_1q_2
earmark-and-pinned
q_3q_4
HSJN
A synchronization-free global MEMO
$(m[q_1q_3q_4], m[q_4], m[q_1q_3])$
$(m[q_1q_3q_4], m[q_3], m[q_1q_2])$
$(m[q_1q_2q_4], m[q_4], m[q_1q_2])$
$(m[q_1q_2q_4], m[q_2], m[q_1q_4])$
$(m[q_1q_2q_4], m[q_1], m[q_2q_4])$
$(m[q_1q_2q_3], m[q_3], m[q_1q_2])$
$(m[q_1q_2q_3], m[q_2], m[q_1q_3])$
$\mathcal{B}_{[1,2]}$

Figure 7: A synchronization-free global MEMO exploiting equivalence grouping and earmark-and-pinning.

5.3 Threading Across Dependencies

When there is an insufficient number of join pairs at the current level to fully utilize all threads, the threading across dependencies technique enables threads to process dependency buffers in upper levels without waiting. More specifically, when a thread processes entries in $\mathcal{B}_{[*,i]}$, the other threads can process an entry e in $\mathcal{B}_{[*,k]}$ ($k > i$) without waiting whenever all entries upon which e depends are already consumed either in the current or previous iterations. Such entries are called *dependency-free entries*.

In order to determine which join entry is dependency-free, each MEMO entry e has an additional field named *numEntry* that stores the number of buffered join entries for e. Given a join entry ($m[qs_1 \cup qs_2]$, $m[qs_1]$, $m[qs_2]$) in a dependency buffer, if the values of the *numEntry* fields of both $m[qs_1]$ and $m[qs_2]$ are 0, this join entry is dependency-free.

To maintain the correct number of buffered join entries in each MEMO entry, we perform the following operations. Before the main thread pushes an entry ($m[qs_1 \cup qs_2]$, $m[qs_1]$, $m[qs_2]$) to a

[4]The join pairs here include all singleton quantifier sets.
[5]Each quantifier in a clique query with N quantifiers is connected to all the other quantifiers using N-1 edges.

dependency buffer, it increases the *numEntry* field of the MEMO entry $m[qs_1 \cup qs_2]$ by one, using a hardware atomic instruction. For example, in Figure 7, the MEMO entry for $q_1q_2q_3q_4$ has three buffered entries in $\mathcal{B}_{[2,2]}$. The MEMO entry for q_1q_2 has no buffered entries. During plan generation, after a thread consumes an entry ($m[qs'_1 \cup qs'_2]$, $m[qs'_1]$, $m[qs'_2]$) in a dependency buffer, the thread decreases the *numEntry* field of the MEMO entry $m[qs'_1 \cup qs'_2]$ by one, also using a hardware atomic instruction. In this way, each MEMO entry maintains the correct number of remaining buffered entries.

During plan generation, when a thread consumes a join entry ($m[qs_1 \cup qs_2]$, $m[qs_1]$, $m[qs_2]$) in the dependency buffer, it first checks, using hardware atomic instructions, whether both this entry is dependency-free. If not, the thread must wait until the entry has no dependency. To minimize such wait time due to threading across dependencies, we move dependency-free entries to the front of each dependency buffer during join pair generation. Thus, when some threads start to process dependency buffers in upper levels during plan generation, they access dependency-free entries first. With equivalence grouping, an entry in $\mathcal{B}_{[*,i]}$ is an equivalence class which has a set of join pairs whose resulting quantifier sets are the same. Thus, if all join pairs in the equivalence class are dependency-free, we move the equivalence class to the front of the dependency buffer.

Example 3. *Figure 8 depicts an example of how the threading across dependencies technique operates. Buffers on the left-hand side show a snapshot of dependency buffers before applying the technique. The right-hand side shows dependency buffers after dependency-free entries are moved to the front. With threading across dependencies, thread T_b can concurrently consume dependency-free entries from $\mathcal{B}_{[*,i+1]}$ without any synchronization conflict while T_a processes entries from $\mathcal{B}_{[*,i]}$.*

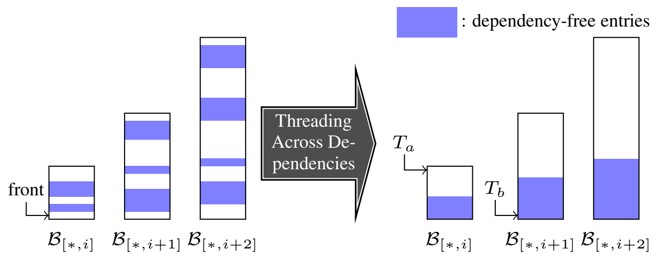


Figure 8: An example of threading across dependencies.

Theoretically, with threading across dependencies, threads can process entries of dependency buffers in many different levels, and thus, the threads may access the plan chain of the same MEMO entry concurrently. Thus, to guarantee the correctness of the threading across dependencies technique, we need synchronization when we access the plan chain. However, in reality, synchronization conflicts on the same plan chain due to threading across dependencies rarely occur. Through extensive experiments in Section 6, the ratio of the number of synchronization conflicts over the total number of join pairs is only $0\% \sim 0.00064\%$, depending on query topologies. We will empirically show in Section 6 that the wait time due to synchronization overhead of the threading across dependencies technique is negligible.

6. PERFORMANCE EVALUATION

The goals of our experiments are as follows:

- To determine how much the three optimization techniques proposed in Section 5 contribute to maximize the parallelism of DPEGeneric (in Section 6.1)
- To show that our parallel algorithm supports any type of bottom up enumeration, achieving linear speedup for each type (in Section 6.2).
- To show that our parallel algorithm is even better than the state of the art parallel optimizer tailored to size-driven enumeration, PDP$_{\text{sva}}$ (in Section 6.3).
- To show that our parallel algorithm is much more robust than PDP$_{\text{sva}}$ with respect to search space allocation, especially when a core is heavily loaded, (in Section 6.4)

We evaluated the same four representative query topologies as [14]: linear, cycle, star, and clique. In order to ensure that our solution never slows down optimization, our parallel optimizer is invoked only when the number of join pairs buffered exceeds a certain threshold, i.e., when there is a sufficient number of joins pairs to fully utilize multiple threads.

All the experiments were conducted on a PC with two Intel Xeon Quad Core E5310 1.6GHz CPUs (=8 cores) and 8 GB RAM, running Windows Vista. Each CPU has two 4Mbyte L2 caches, each of which is shared by two cores. We implemented all algorithms in PostgreSQL 8.3 [26] to see the performance trends in a full-fledged DBMS. Since the optimization component in PostgreSQL was not thread safe, we modified it significantly in order to be thread-safe. Furthermore, there were many places in the original code where memory was not released during query optimization. We also fixed all such problems by calling memory deallocation functions efficiently. Since fixed-sized structures (such as list cells) are extensively used, we used two types of memory managers to minimize the total memory allocation size, one for variable-sized structures and the other for fixed-sized structures. Unlike a commercial DBMS, during plan pruning, PostgreSQL uses a fuzzy costing comparison function that considers both the total cost and the startup cost of a plan, which tends to accumulate unnecessary plans in the plan chain. However, this costing mechanism is only useful for top-k plans having the LIMIT clause. Since we focus on non-top-k plans, we only exploited the total cost of a plan during plan pruning.

The performance metrics are the speed-up and the elapsed time, where the speedup is defined as the ratio of the elapsed time of the serial algorithm over that of its parallel counterpart. Table 1 summarizes the experimental parameters and their values. Note that we used the same parameter values as [14]. We omit all experimental results for linear and cycle queries, because the total number of join pairs are generally too small to benefit from parallelization. Our main focus is to reduce compilation times for large data warehouse OLAP queries which are typically star-shaped. OLTP queries are not our focus, since they can be optimized very fast. An experimental study on a real DB2 query workload [18] verified that compilation time is dominated by the number of join re-orderings. Thus, we believe that star queries (for varying the number of joins) indeed model representative real data warehouse queries and are sufficient to show our claim. For the same reason as [14, 21, 22], we used large clique queries to show the worst case scenario, which we believe is theoretically meaningful.

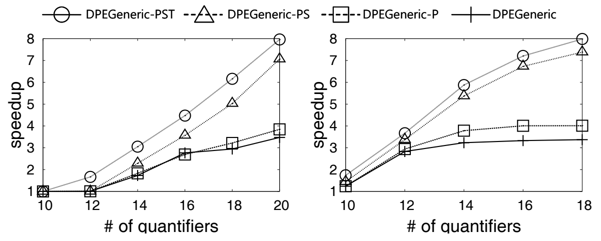
Table 1: Experimental parameters and their values.

Parameter	Default	Range
join pair enumerator	DP_{cpp}	DP_{cpp}, DP_{hyp}, DP_{sva}
query topology	star, clique	star, clique
# of quantifiers	20, 18	10, 12, 14, 16, 18, 20
# of threads	8	$1 \sim 8$

6.1 Impact of the Three Optimization Techniques

We are interested in learning the impact of the three optimization techniques (in Section 5) on DPEGeneric, with respect to query topologies and the number of threads. The three optimization techniques are called P, S, and T. P stands for the pipelining of join pair generation and plan generation; S for synchronization-free global MEMO; and T for threading across dependencies. For example, DPEGeneric-PST denotes the DPEGeneric algorithm with the three optimization techniques.

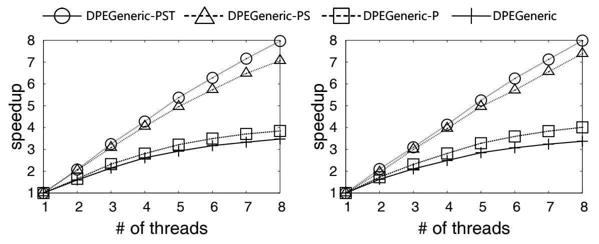
Figure 9 shows the experimental results for varying the number of quantifiers for star and clique queries using 8 threads. The results are presented using the speedup between the serial DP_{cpp} algorithm and the DPEGeneric algorithm with the specified optimization technique(s). For star queries, only DPEGeneric-PST, which exploits the three optimization techniques, achieves linear speedup when the number of quantifiers is 20. This is because the number of join pairs is large enough to exploit 8 parallel threads, and the three optimization techniques maximize parallelism. DPEGeneric-PS achieves 7.1 times speedup, and DPEGeneric-P achieves 3.9 times speedup, while DPEGeneric alone achieves 3.3 times speedup. Clique queries achieve higher overall speedups than comparable star queries having the same number of quantifiers, because the number of join pairs in clique queries are much larger than those in equally-sized star queries. For clique queries, DPEGeneric-PST also achieves linear speedup due to the sophisticated optimization techniques. Note that the threading across dependency technique is more effective in star queries than in clique queries, since join entries buffered in star queries are spread across many dependency buffers, and thus, there exist dependency buffers that have insufficient number of pairs to fully utilize all threads.



(a) Star queries. (b) Clique queries.

Figure 9: Effect of three optimization techniques by varying the number of quantifiers (8 threads).

Figure 10(a) shows the impact of the three optimization techniques on DPEGeneric for varying the number of threads over the serial DP_{cpp} for star queries; Figure 10(b) shows the same for clique queries. Regardless of query topologies, DPEGeneric-PST achieves linear speedup as the number of threads increases. The speedup rates for DPEGeneric and DPEGeneric-P diminish as the number of threads increases. This is because, as the per-thread MEMOs increase, the cost of merging them into the global MEMO also increases. In some cases, we notice that slightly superlinear speedups happen. Note that this can often happen in multi-core applications due to cache sharing. That is, threads in different cores can access shared data structures, such as the MEMO table, catalog structures, and enumeration codes, in 4 Mbytes L2 cache shared by two cores.



(a) Star query with 20 quantifiers. (b) Clique query with 18 quantifiers.

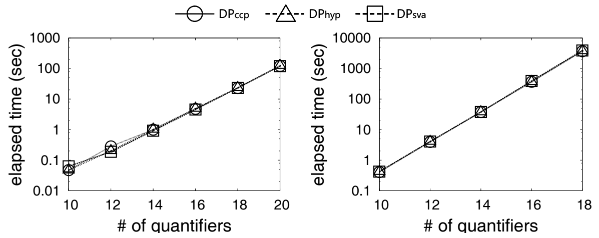
Figure 10: Effect of three optimization techniques by varying the number of threads.

In the following experiments, we apply all three optimization techniques to DPEGeneric.

6.2 Generality of Our Framework

The generality experiment is to show that our parallel algorithm supports any type of bottom up enumeration, achieving linear speedup for each type. To this end, we use the three state of the art serial bottom-up enumerators: DP_{cpp}; DP_{hyp}; and the size-driven enumerator using the skip vector array called DP_{sva}. Note that DP_{sva} is the fastest size-driven enumerator which avoids generating infeasible join pairs using the skip vector array [14].

We first perform experiments using the three serial enumerators before showing speedup using their parallel counterparts, since there have been no experimental comparisons reported of DP_{cpp}, DP_{hyp}, and DP_{sva}. Figure 11 shows experimental results for varying the number of quantifiers for star and clique queries. As shown in this figure, the elapsed time for all three enumerators is nearly the same, although DP_{cpp} and DP_{hyp} that generate join pairs by directly traversing a query graph are marginally faster than DP_{sva} for clique queries having 16 and 18 quantifiers (about 6%).

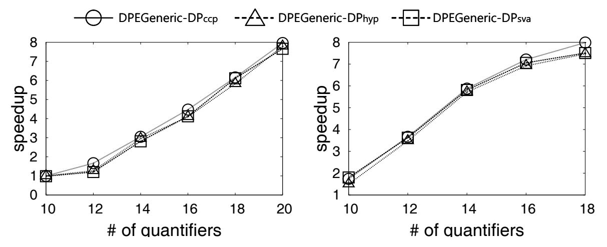


(a) Star queries. (b) Clique queries.

Figure 11: Experimental results of DP_{cpp}, DP_{hyp}, and DP_{sva} by varying the number of quantifiers.

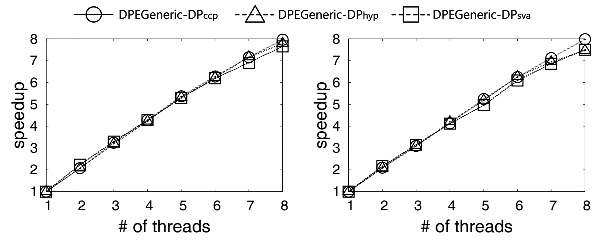
Figure 12 shows the generality of our framework. As we see here, DPEGeneric supports all three state of the art enumerators. We achieve perfect linear speedup for DP_{cpp}, and nearly linear speedups for DP_{hyp} and DP_{sva}.

Figure 13 shows the generality of our framework by varying the number of threads. Again, DPEGeneric achieves (nearly) linear speedup for all three enumerators.



(a) Star queries. (b) Clique queries.

Figure 12: Effect of various enumerator by varying the number of quantifiers (8 threads).



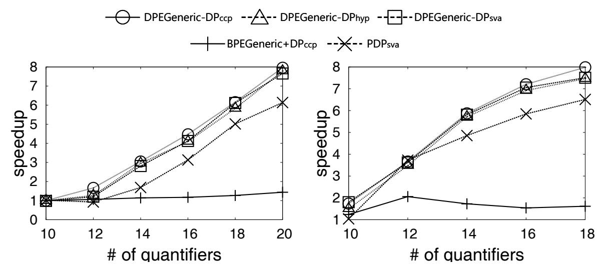
(a) Star query with 20 quantifiers. (b) Clique query with 18 quantifiers.

Figure 13: Effect of various enumerator by varying the number of threads.

6.3 Comparison with Direct Competitors

This experiment provides comparison of DPEGeneric with BPEGeneric (in Section 3) and PDP_{sva} (the state of the art parallel enumerator for DP_{sva}). The three state of the art serial enumerators are applied to DPEGeneric.

Figure 14 shows experimental results for varying the number of quantifiers for star and clique queries. BPEGeneric performs the worst due to serious synchronization overhead. DPEGeneric achieves linear speedup, while PDP_{sva} only achieves up to 6.1 speedup for star queries using 8 threads. That is, 23% of the total cores are not utilized! For clique queries, the overall trend is similar.

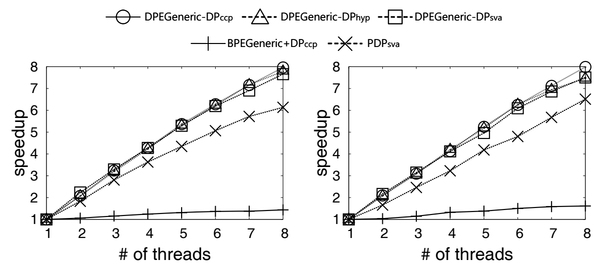


(a) Star queries. (b) Clique queries.

Figure 14: Experimental results for speed-up by varying the number of quantifiers (8 threads).

Note that DPEGeneric-DP_{sva} and PDP_{sva} use the same enumerator, but DPEGeneric-DP_{sva} consistently outperforms PDP_{sva} due to the three sophisticated optimization techniques. Thus, in spite of being a generic solution, DPEGeneric is highly effective.

Figure 15 shows experimental results for varying the number of threads. Again, DPEGeneric achieves linear speedup as the number of threads increases, consistently outperforming PDP_{sva} and BPEGeneric. The performance curve of BPEGeneric increases very slowly as the number of threads increases, due to substantial synchronization overhead.



(a) Star query with 20 quantifiers. (b) Clique query with 18 quantifiers.

Figure 15: Experimental results for speed-up by varying the number of threads.

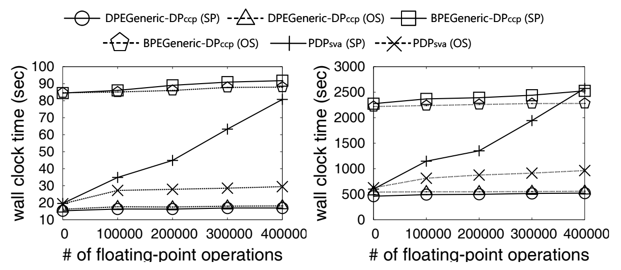
6.4 Robustness

This experiment demonstrates the robustness of our parallel algorithm in a controlled setting, with respect to search space allocation. For this, we place a CPU-intensive process on a specific core. This process loops infinitely, executing two steps in each loop: the first step executes 100,000 floating-point operations and the second sleeps for 1 millisecond. To increase loads on a specific core, we increase the number of floating-point operations accordingly. We use two thread scheduling policies. One policy, called static placement denoted as SP, places thread i to core i. The other policy relies on the underlying operating system thread scheduling policy denoted as OS.

Figure 16 shows experimental results for varying loads on a core for a star query and a clique query. As the number of floating-point operations increases, the elapsed time for PDP_{sva} using SP increases sharply, meaning that the real workload is highly unbalanced. PDP_{sva} using OS performs much better than PDP_{sva} using SP, indicating that the OS thread scheduling policy in Windows Vista is much better for PDP_{sva}. PDP_{sva} using OS achieves 4.1 $\sim$ 6.1 speedups, while DPEGeneric achieves 7.2 $\sim$ 8.0 speedups. Note that both level off after reaching 400,000 floating point operations (i.e., one core is heavily loaded). This phenomenon can be analyzed as follows. Suppose that the total elapsed time for a serial algorithm is t, and there are m cores available. Assume that there is no load on any core, and the parallel counterpart achieves linear speedup. Then the ideal elapsed time for each thread of the parallel algorithm is $\frac{t}{m}$. Since one core is heavily overloaded, the worst scenario is that, after $m-1$ threads finish running their jobs concurrently, (i.e., the elapsed time becomes $\frac{t}{m}$, the last thread then executes its job (i.e., the elapsed time for this thread is $\frac{t}{m}$.). Thus, the total elapsed time is $\frac{2t}{m}$ (= $\frac{t}{m}+\frac{t}{m}$), and the speedup is $\frac{m}{2}$. Therefore, the speedup of PDP_{sva} converges to 4. However, due to dynamic search space allocation, DPEGeneric achieves $m-1$ speedup when one core is heavily loaded. Therefore, its speedup converges to 7, which is ideal.

6.5 Summary and Discussion

In a series of tests, we have shown how DPEGeneric performs with different numbers of threads and quantifiers. By using dependency-aware reordering along with sophisticated optimization techniques, we have achieved linear speedup. That is, the tangled dependencies in the original sequence are unraveled by dependency-



(a) Star query with 20 quantifiers. (b) Clique query with 18 quantifiers.

Figure 16: Experimental results for varying loads on a core.

aware reordering. Moreover, by using sophisticated optimization techniques, discussed in Section 5, the wait time caused by the synchronization required in DPEGeneric proved to be immaterial for the different parameters we tested. Due to the synchronization-free global MEMO, other than the dependency buffers, DPEGeneric has no more memory overhead than the serial optimizer. Note that the maximum size of all dependency buffers we tested is only 34Mbytes.

We have also shown that DPEGeneric supports all three state of the art serial enumerators, and achieves linear speedup for each one. Due to the sophisticated optimization techniques, DPEGeneric consistently outperforms PDP_{sva} for all parameter values we tested. Experimental results also show that DPEGeneric is much more robust than PDP_{sva} with respect to search space allocation. Given m cores in a CPU, when one core is heavily loaded, PDP_{sva} only achieves $\frac{m}{2}$ speedup, while DPEGeneric achieves $m-1$ speedup.

One interesting question is "Given a fixed optimization budget, how much larger of a query can we handle using parallelism?" This question is related to how many cores are available in the machine. Suppose that the optimization budget is the elapsed time of the serial DP_{cpp} (the best serial enumerator) for a star query having 14 quantifiers. Then, according to our estimation, we need 122 cores to handle a star query having up to 20 quantifiers with the same budget. As additional cores are placed in a chip, it becomes more interesting for future work to perform experiments using even more cores.

7. RELATED WORK

There has been a lot of work on "parallel query" optimization, which generates QEPs for parallel execution [7, 10, 16, 29, 20]. Commercial shared-nothing and shared-everything DBMSs also support such parallel query optimization, so that QEPs optimized by a serial optimizer can be executed in many nodes in parallel. Most recently, Han et el. [14] proposed the first framework to parallelize the size-based serial enumeration in a multi-core architecture.

Extensive work has been done on heuristic or randomized query optimization to reduce query optimization time for large join queries [4, 19, 23, 31, 32]. PostgreSQL uses a thresholding logic—if the number of the quantifiers in an input query is less than or equal to twelve, it uses DP optimization; otherwise, it uses a genetic algorithm [26]. Although such heuristic or randomized query optimization reduces the query optimization time by not fully exploring the entire search space, this can result in slower sub-optimal plans by up to several orders of magnitude.

The dynamic programming used in bottom-up join enumeration belongs to the *non-serial polyadic class*, which is known to be very difficult to parallelize [14]. Furthermore, sub-problems in join enumeration depend on all preceding levels, whereas sub-problems in other applications of DP depend on only a fixed number of preceding levels (usually, two). Thus, existing parallel DP algorithms [3, 9, 17, 33, 34] cannot be readily applied to DP query optimization to achieve linear speedup. Loop partitioning techniques in compilers [1, 27], which partition iterations in nested loops of DP, can be applied to DP query optimization. However, this can result in seriously unbalanced workloads, since the cost of each iteration can significantly differ, depending on whether CreateJoinPlans is invoked. To overcome this problem, reference [14] proposed round-robin inner *static* allocation along with the skip vector array. However, the round-robin inner static allocation in turn works only when all cores are evenly loaded. If not, the slowest thread holds up all the other, faster threads, still resulting in seriously unbalanced workloads anyway. In contrast to static allocation, we employ dynamic allocation to cope with cases where some cores are heavily overloaded. With our sophisticated performance optimization techniques along with dynamic partitioning, we show that our generic solution is much more robust than that of [14] tailored to size-based enumeration.

Our approach is related to work on job and DAG scheduling [2, 5, 6]. Although the basic idea of dependency-aware execution and communication of results shares some similarity, our work is significantly different from this work in that we generate a DAG on-the-fly from the algebraic structure of the search space while the DAG of [2, 5, 6] is given as input to a DAG scheduler. We also provided the formal foundation and several sophisticated tuning techniques specific to parallelization of query optimization.

8. CONCLUSIONS

Recently, [14] proposed a novel framework for parallelizing the process of optimizing queries. The parallel framework outperforms the conventional serial generate-and-filter DP algorithm optimizer by up to two orders of magnitude using 8 threads due to linear speedup using parallelism, and an order of magnitude performance improvement using the skip vector array-based enumeration [14]. However, the framework has three limitations: 1) supporting only the size-driven DP enumerator, 2) statically allocating search space, and 3) not fully exploiting parallelism.

In this paper, we proposed the first *generic* solution for parallelizing any type of bottom-up optimizer, including the graph-traversal driven type, and for supporting *dynamic* search allocation and *full* parallelism. Specifically, by viewing a serial bottom-up optimizer as one which generates a totally ordered sequence of pairs of quantifier sets in a streaming fashion, we developed a novel concept of dependency-aware reordering, which minimizes waiting time caused by dependencies of pairs of quantifier sets. By exploiting dependency-aware reordering and dynamic search space allocation, we devised a scalable parallel enumeration algorithm called DPEGeneric. To maximize parallelism, we proposed a series of optimization techniques that can be applied to DPEGeneric: 1) pipelining of join pair generation and plan generation; 2) the synchronization-free global MEMO; and 3) threading across dependencies. Through extensive experiments with various query topologies, we have shown that our solution supports any type of bottom up optimization, achieving linear speedup for each type. Despite the fact that our solution is generic, due to sophisticated optimization techniques, our parallel optimizer using the size-driven DP enumerator outperforms the state of the art parallel optimizer tailored to size-driven enumeration. Experimental results also showed that our algorithm is much more robust than the state of the art algorithm with respect to search space allocation.

Overall, we believe the proposed concepts and solutions provide comprehensive insight and a substantial framework for future research.

Acknowledgement

We appreciate the SIGMOD reviewers' careful comments, which significantly enhanced the presentation. We thank Jaume Baixeries for discussions on the initial idea of Wook-Shin Han. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation (KOSEF), grant number R11-2008-007-03003-0.

9. REFERENCES

[1] A. Agarwal, D. A. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE TPDS*, 6(9), 1995.

[2] K. Agrawal, Y. He, and C. E. Leiserson. An empirical evaluation ofwork stealing with parallelism feedback. In *ICDCS*, 2006.

[3] C. E. R. Alves, E. Cáceres, and F. K. H. A. Dehne. Parallel dynamic programming for solving the string editing problem on a cgm/bsp. In *SPAA*, 2002.

[4] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis. A genetic algorithm for database query optimization. In *ICGA*, 1991.

[5] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[7] C. Chekuri, W. Hasan, and R. Motwani. Scheduling problems in parallel query optimization. In *PODS*, 1995.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[9] M. Elkihel and D. E. Baz. Load balancing in a parallel dynamic programming multi-method applied to the 0-1 knapsack problem. In *PDP*, pages 127–132, 2006.

[10] S. Englert, R. Glasstone, and W. Hasan. Parallelism and its price: A case study of nonstop sql/mp. *SIGMOD Record*, 24(4):61–71, 1995.

[11] J. Erickson. Multicore and gpus: One tool, two processors. *Dr. Dobb's Journal*, 2007, http://www.ddj.com/hpc-high-performance-computing/199501192.

[12] M. B. et al. Pam: a novel performance/power aware meta-scheduler for multi-core systems. In *SC*, 2008.

[13] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. McGraw-Hill, 3rd edition, 1994.

[14] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. In *VLDB*, 2008.

[15] W.-S. Han and J. Lee. Dependency-Aware Reordering for Parallelizing Query Optimization in Multi-Core CPUs. http://wshan.org/HanL09Reordering.pdf.

[16] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. *Distrib. Parallel Databases*, 1(1):9–32, 1993.

[17] S.-H. S. Huang, H. Liu, and V. Viswanathan. Parallel dynamic programming. *IEEE TPDS*, 5(3), 1994.

[18] I. F. Ilyas, J. Rao, G. M. Lohman, D. Gao, and E. T. Lin. Estimating compilation time of a query optimizer. In *SIGMOD*, 2003.

[19] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD*, 1990.

[20] R. S. G. Lanzelotte, P. Valduriez, M. Zaït, and M. Ziane. Industrial-strength parallel query optimization: issues and lessons. *Inf. Syst.*, 19(4), 1994.

[21] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, 2006.

[22] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, 2008.

[23] T. Morzy, M. Matysiak, and S. Salza. Tabu search optimization of large join queries. In M. Jarke, J. A. B. Jr., and K. G. Jeffery, editors, *EDBT*, volume 779, pages 309–322, 1994.

[24] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, 1990.

[25] N. W. Paton, V. Raman, G. Swart, and I. Narang. Autonomic query parallelization using non-dedicated computers: An evaluation of adaptivity options. In *ICAC*, 2006.

[26] Postgresql version 8.3. http://www.postgresql.org.

[27] F. Rastello and Y. Robert. Automatic partitioning of parallel loops with parallelepiped-shaped tiles. *IEEE TPDS*, 13(5):460–470, 2002.

[28] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Inc, Sebastopol, 2007.

[29] P. Stenstrom. Ipdps panel: Is the multi-core roadmap going to live up to its promises? *IPDPS*, 2007.

[30] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[31] A. N. Swami. Optimization of large join queries: Combining heuristic and combinatorial techniques. In *SIGMOD*, 1989.

[32] A. N. Swami and A. Gupta. Optimization of large join queries. In *SIGMOD*, 1988.

[33] G. Tan, S. Feng, and N. Sun. Biology - locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In *SC*, 2006.

[34] G. Tan, N. Sun, and G. R. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *SPAA*, 2007.

[35] D. Wentzlaff and A. Agarwal. The Case for a Factored Operating System (fos). http://hdl.handle.net/1721.1/42894.