Parallelizing Query Optimization *

Wook-Shin Han[1] Wooseong Kwak[1] Jinsoo Lee[1] Guy M. Lohman[2] Volker Markl[2]
[1] *Department of Computer Engineering, Kyungpook National University, Republic of Korea*
[2] *IBM Almaden Research Center, San Jose, California*

ABSTRACT

Many commercial RDBMSs employ cost-based query optimization exploiting dynamic programming (DP) to efficiently generate the optimal query execution plan. However, optimization time increases rapidly for queries joining more than 10 tables. Randomized or heuristic search algorithms reduce query optimization time for large join queries by considering fewer plans, sacrificing plan optimality. Though commercial systems executing query plans in parallel have existed for over a decade, the optimization of such plans still occurs serially. While modern microprocessors employ multiple cores to accelerate computations, parallelizing query optimization to exploit multi-core parallelism is not as straightforward as it may seem. The DP used in join enumeration belongs to the challenging *non-serial polyadic* DP class because of its non-uniform data dependencies. In this paper, we propose a comprehensive and practical solution for parallelizing query optimization in the multi-core processor architecture, including a parallel join enumeration algorithm and several alternative ways to allocate work to threads to balance their load. We also introduce a novel data structure called *skip vector array* to significantly reduce the generation of join partitions that are infeasible. This solution has been prototyped in PostgreSQL. Extensive experiments using various query graph topologies confirm that our algorithms allocate the work evenly, thereby achieving almost linear speed-up. Furthermore, for a star query joining 22 tables, our best algorithm using only 8 threads outperforms the conventional serial DP algorithm by two orders of magnitude, reducing optimization time from hours to a couple of minutes.

1. INTRODUCTION

The success of relational database management systems (RDBMSs) can largely be attributed to the standardization of the SQL query language and the development of sophisticated query optimizers that automatically determine the optimal way to execute a declarative SQL query by enumerating many alternative query execution plans (QEPs), estimating the cost of each, and choosing the least expensive plan to execute [18]. Many commercial RDBMSs such as DB2 employ dynamic programming (DP), pioneered by Selinger et al. [29]. Dynamic programming builds QEPs "bottom up" and exploits the *principle of optimality* to prune sub-optimal plans at each iteration (thereby saving space) and to guarantee that the optimal QEP is found without evaluating redundant sub-plans [12].

As the number of tables referenced in a query increases, however, the number of alternative QEPs considered by a DP-based optimizer can, in the worst case, grow exponentially [13]. This means that many real-world workloads that reference more than 20 tables (e.g., many Siebel queries) would have prohibitive optimization times using current DP optimization. In extreme cases (queries referencing a large number of relatively small tables), the time to optimize a query with DP may even exceed the time to execute it! Although randomized or heuristic (e.g., greedy) search algorithms [3, 13, 31, 33] reduce the join enumeration time by not fully exploring the entire search space, this can result in sub-optimal plans that execute orders of magnitude slower than the best plan, more than negating any savings in optimization time by such heuristics. And while the plan picked by the optimizer can sometimes be stored and reused, thereby amortizing the optimization cost over multiple executions, changes to the parameters in the query or the underlying database's characteristics may make this approach sub-optimal, as well.

In an era in which new chips are achieving speed-up not by increasing the clock rate but by multi-core designs [7, 32], it seems obvious to speed up CPU-bound query optimization by parallelizing it. Yet even though QEPs to execute a query in parallel have been common in commercial products for over a decade [6, 15], remarkably there have been no efforts, to the best of our knowledge, to parallelize the query optimization process itself! In the typical shared-nothing or shared-memory multi-node system, a single coordinator node optimizes the query, but many nodes execute it [10].

In this paper, we propose a novel framework to parallelize query optimization to exploit multi-core processor architectures whose main memory is shared among all cores. Our goal is to allocate parts of the optimizer's search space evenly among threads, so that speed-up linear in the number of cores can be achieved. Specifically, we develop a parallel

*This work was supported by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD) (KRF-2007-521-D00399).

VLDB '08, August 24-30, 2008, Auckland, New Zealand

DP-based join enumerator that generates provably optimal QEPs for much larger queries than can practically be optimized by today's query optimizers (> 12 tables). While parallelism doesn't negate the inherent exponential nature of DP, it can significantly increase the practical use of DP from queries having less than 12 tables to those having more than 20 or 25 tables, depending upon how strongly connected the query graph is, as we will see. Assigning the extra cores to other, concurrent queries might increase throughput, but would do nothing to improve the response time for individual queries, as our approach does.

Parallelizing query optimization that uses DP is not as simple as it might first appear. As we will see in Section 2, the DP algorithm used in join enumeration belongs to the *non-serial polyadic* DP class [8], which is known to be very difficult for parallelization due to its non-uniform data dependence [35]. Sub-problems in other applications of DP depend on only a fixed number of preceding levels (mostly, two), whereas sub-problems in join enumeration depend on all preceding levels. Thus, existing parallel DP algorithms [2, 5, 11, 34, 35] cannot be directly applied to our framework. Therefore, we develop a totally new method for parallelizing DP query optimization, which views join enumeration as a series of self-joins on the MEMO table containing plans for subsets of the tables (or *quantifiers*).

Parallel query optimization can speed up many other applications that exploit the query optimizer. It can help feedback-based query optimization such as progressive optimization (POP) [10, 20], especially for queries that have longer compilation time than execution time. Since POP repeatedly invokes an optimizer until it finds an optimal plan, parallel optimization can speed up such queries. Automatic physical database tools that exploit the query optimizer as a "What if?" tool, such as index advisors, are dominated by the time to re-optimize queries under different "What if?" scenarios, and so will also enjoy significantly improved execution times from parallelized query optimization.

Our contributions are as follows: 1) We propose the first framework for parallel DP optimization that generates optimal plans. 2) We propose a parallel join enumeration algorithm, along with various strategies for allocating portions to different threads to even the load. 3) We propose a novel index structure called a *skip vector array* and algorithms that exploit it to speed up our parallel join enumeration algorithm, especially for star queries. 4) We formally analyze why the various allocation schemes generate different sizes of search spaces among threads; 5) We perform extensive experiments on various query topologies to show that: (a) our parallel join enumeration algorithms allocate the work to threads evenly, thereby achieving almost linear speed-up; and b) our parallel join enumeration algorithm enhanced with our skip vector array outperforms the conventional generate-and-filter DP algorithm used by industrial-strength optimizers such as DB2 and PostgreSQL by up to two orders of magnitude for star queries.

The rest of this paper is organized as follows. Section 2 reviews the current serial, DP-based join enumeration. Section 3 gives an overview of our framework and our algorithm for parallelizing DP-based join enumeration. The next two sections give the details of an important subroutine to this algorithm – Section 4 details the basic algorithm, and Section 5 enhances the basic algorithm with the skip vector array to avoid generating many joins that will be infeasible because their quantifier sets aren't disjoint. In Section 6, we give a formal analysis of different strategies for allocating work to threads. Section 7 summarizes our experimental results. We compare our contributions with related work in Section 8, and conclude in Section 9.

2. DP BASED JOIN ENUMERATION

To understand how we parallelize query optimization, we must first review how DP is used today in serial optimization to enumerate join orders, as outlined in Algorithm 1, which we call *SerialDPEnum*. *SerialDPEnum* generates query execution plans (QEPs) in a "bottom up" fashion [12]. It first generates different QEPs for accessing a single table. Types of table access QEPs include a simple sequential scan, index scan, list prefetch, index ORing, and index ANDing [17]. *SerialDPEnum* 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., tables accessed, predicates applied, ordering of rows, partitioning, etc.) subsume those of QEP_1 (Line 3). *SerialDPEnum* then joins these best QEPs to form larger QEPs, and iteratively joins those QEPs together to form successively larger QEPs. Each QEP can be characterized by the set of tables, (or *quantifiers*), that have been accessed and joined by that QEP. QEPs for a given quantifier set are maintained in an in-memory quantifier set table (often called MEMO). Each entry in MEMO contains a list of QEPs for a quantifier set, and the entry typically is located by hashing the quantifier set.

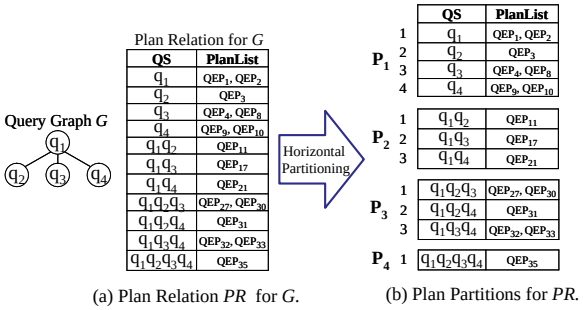
To produce a QEP representing quantifier sets of size S, *SerialDPEnum* successively generates and then joins quantifier sets $smallQS$ and $largeQS$ of size $smallSZ$ and $largeSZ = S - smallSZ$, respectively, where $smallSZ$ can vary from 1 up to half the size of S ($\lfloor S/2 \rfloor$). At each iteration, subroutine *CreateJoinPlans* does the bulk of the work, generating and estimating the cost of all join QEPs between the two given sets of quantifiers, $smallQS$ and $largeQS$, including QEPs in which either quantifier set is the outer-most (left input to the join) and alternative join methods (Line 13). *SerialDPEnum* iteratively increases the size S of the resulting quantifier set until it obtains the optimal QEP for all N quantifiers in the query.

Algorithm 1 SerialDPEnum

```
Input: a connected query graph with quantifiers q_1, ..., q_N
Output: an optimal bushy join tree
 1: for i ← 1 to N
 2:   Memo[{q_i}] ← CreateTableAccessPlans(q_i);
 3:   PrunePlans(Memo[{q_i}]);
 4: for S ← 2 to N
 5:   for smallSZ ← 1 to ⌊S/2⌋
 6:     largeSZ ← S − smallSZ;
 7:     for each smallQS of size smallSZ
 8:       for each largeQS of size largeSZ
 9:         if smallQS ∩ largeQS ≠ ∅ then
10:           continue; /*discarded by the disjoint filter*/
11:         if not(smallQS connected to largeQS) then
12:           continue; /*discarded by the connectivity filter*/
13:         ResultingPlans ← CreateJoinPlans(
                Memo[smallQS], Memo[largeQS]);
14:         PrunePlans(Memo[smallQS ∪ largeQS], ResultingPlans);
15: return Memo[{q_1, ..., q_N}];
```

Before calling *CreateJoinPlans*, *SerialDPEnum* first checks whether the two quantifier sets $smallQS$ and $largeQS$ can form a feasible join. To do so, a series of filters must be executed. For a more detailed description of the fil-

ters, refer to reference [24]. The two most important filters are a disjoint filter (in Line 9) and a connectivity filter (in Line 11). The disjoint filter ensures that the two quantifier sets $smallQS$ and $largeQS$ are disjoint. The connectivity filter verifies that there is at least one join predicate that references quantifiers in $smallQS$ and $largeQS$. Disabling the connectivity filter permits Cartesian products in the resulting QEPs. Note that the DP formulation in $SerialDPEnum$ is a *non-serial polyadic* formulation, since $SerialDPEnum$ has two recursive sub-problems (polyadic) (in Line 13), and sub-problems depend on all preceding levels (non-serial) (loop beginning on Line 5).



(a) Plan Relation *PR* for *G*. (b) Plan Partitions for *PR*.

Figure 1: Plan relation and its four plan partitions.

3. OVERVIEW OF OUR SOLUTION

In order to achieve linear speed-up in parallel DP join enumeration, we need to: (1) partition the search space evenly among threads, and (2) process each partition independently without any dependencies among threads. Our key insight is the following. In DP-based join enumeration, each sub-problem depends only on the results of all preceding levels. By partitioning sub-problems by their sizes – or, more precisely, the sizes of the resulting quantifier sets – sub-problems of the same resulting size are mutually independent. Furthermore, as the number of quantifiers increases, the number of sub-problems of the same size grows exponentially. This is especially true for star and clique queries, which will benefit most from parallel execution. In addition, each sub-problem of size S is constructed using any combination of one smaller sub-problem of size $smallSZ$ and another sub-problem of size $largeSZ$, such that $S = smallSZ + largeSZ$. Thus, we can further group the partitioned sub-problems of the same resulting size by the sizes of their two smaller sub-problems. In this way, we can solve the sub-problems of the same size by executing joins between their smaller sub-problems. With this approach, we can transform the join enumeration problem into multiple theta joins, which we call *multiple plan join*s (MPJs), in which the disjoint and connectivity filters constitute the join conditions. Each MPJ is then parallelized using multiple threads *without any dependencies* between the threads. Thus, by judiciously allocating to threads portions of the search space for MPJ, we can achieve linear speed-up.

To illustrate this more concretely, regard the MEMO table as a *plan relation* with two attributes, *QS* and *PlanList*. We horizontally partition this plan relation (by construction) into several partitions according to the size of the quantifier set *QS*. Thus, each partition of the plan relation, called a *plan partition*, has only tuples whose *QS* attributes are of same size. Let P_S denote the plan partition containing all quantifier sets of size S. As before, we maintain a hash index on the *QS* column to efficiently find the tuple in the plan relation having a given quantifier set. The plan partition P_S is generated by performing $\lfloor S/2 \rfloor$ joins from the start join between P_1 and P_{S-1} to the end join between $P_{\lfloor S/2 \rfloor}$ and $P_{S-\lfloor S/2 \rfloor}$. Figure 1 shows the plan relation for a query graph G. Since G has four quantifiers, we have four plan partitions, $P_1 \sim P_4$, as shown in Figure 1(b).

To parallelize the MPJ for P_S, we need to assign parts of the search space for the MPJ to threads. This step is called *search space allocation*. There are many possible ways to perform this allocation, some better than others. For example, the MPJ for P_4 in Figure 1 (b) must execute two plan joins, one between P_1 and P_3, and the other join between P_2 and P_2. If we try to evenly allocate all possible pairs of quantifier sets to two threads as shown in Figure 2, the workload looks balanced (thread 1 has 10 pairs, and thread 2 has 11 pairs). But in reality thread 2 will never invoke $CreateJoinPlans$, because all of its pairs will be discarded by the disjoint filter as infeasible! Thus, this seemingly even allocation unfortunately would result in seriously unbalanced workloads. This motivates us to more carefully allocate search spaces to threads, as we investigate further in Section 4.1. Note also that, as the number of quantifiers increases, the number of times the disjoint filter is invoked increases exponentially, dominating the join enumerator's performance. This motivates us to propose a novel index called a *skip vector array* (*SVA*) that minimizes the number of unnecessary invocations of the disjoint filter, as well as a new flavor of MPJ that exploits the SVA, in Section 5.

Figure 2: Allocating search spaces for building P_4 to two threads.

Algorithm 2 outlines our parallelized join enumeration algorithm, called $ParallelDPEnum$. We first allocate parts of the MPJ search space to m threads (Line 5), each of which then executes its allocated MPJs in parallel (Line 7). Here, we can use one of two different flavors of MPJ, depending on whether we exploit a skip vector array (SVA) or not. Both types of MPJ are useful, depending on the sizes of the plan partitions. If we choose not to exploit the SVA, at Line 7 we'll invoke the "basic" flavor of MPJ without SVAs, which will be explained in Section 4. Otherwise, we instead invoke at Line 7 the "enhanced" flavor of MPJ that exploits SVAs, which will be explained in Section 5. Once we've completed this parallel execution of MPJs for each size of quantifier sets, we need to merge results and prune expensive QEPs in the plan partition (Line 9). Then, if we are performing the SVA-enhanced MPJs, we must build an SVA for the plan partition we just constructed (Line 11), as will be explained in Section 5.1, to be exploited in subsequent MPJs. Note that the unit of allocation to threads in the SVA-enhanced MPJ is a pair of partitioned SVAs, whereas the unit of allocation to threads in the basic MPJ (without SVAs) is a pair of tuples.

Algorithm 2 *ParallelDPEnum*

Input: a connected query graph with quantifiers $q_1, \cdots, q_N$
Output: an optimal bushy join tree

```
1: for i ← 1 to N
2:   Memo[{q_i}] ← CreateTableAccessPlans(q_i);
3:   PrunePlans(Memo[{q_i}]);
4: for S ← 2 to N
5:   SSDVs ← AllocateSearchSpace(S, m);/*SSDVs: search space
     description vectors allocated for m threads */
6:   for i ← 1 to m  /*Execute m threads in parallel*/
7:     threadPool.SubmitJob(MutiplePlanJoin(SSDVs[i], S));
8:   threadPool.sync();
9:   MergeAndPrunePlanPartitions(S);
10:  for i ← 1 to m
11:    threadPool.SubmitJob( BuildSkipVectorArray(i) );
12:  threadPool.sync();
13: return Memo[{q_1, ··· , q_N}];
```

4. MULTIPLE PLAN JOIN

In our parallel DP optimizer, two important functions – *AllocateSearchSpace* and *MultiplePlanJoin* – still need to be defined. We first propose in Section 4.1 judicious ways to allocate plan joins to threads (*AllocateSearchSpace* in *ParallelDPEnum*). Then in Section 4.2, we detail the basic algorithm (without the skip vector array enhancement) for *MultiplePlanJoin* in *ParallelDPEnum*.

We assume that elements in a quantifier set are sorted in increasing order of their quantifier numbers, and thus sets can be regarded as strings. We also assume that each plan partition is sorted in lexicographical order of the quantifier sets.

4.1 Search Space Allocation Schemes

We compare four different schemes for allocating portions of the join enumeration space to threads: total sum, equi-depth, round-robin outer, and round-robin inner. Unlike a shared-nothing environment, in which tuples must be physically allocated to a thread on a specific node, we need only logically allocate partitions of the search space to each thread, because those partitions are stored in memory that is shared among cores.

4.1.1 Total Sum Allocation

When building the plan partition P_S in MPJ, there are $\lfloor S/2 \rfloor$ plan joins. Thus, the size of the search space for building P_S is $\sum_{smallSZ=1}^{\lfloor S/2 \rfloor} (|P_{smallSZ}| \times |P_{S-smallSZ}|)$.

Given m threads, with *total sum allocation*, we equally divide the search space into m smaller search spaces, and allocate them to the m threads. Each thread T_i executes MPJ for the i-th search space allocated. Figure 2 in Section 3 shows two allocated search spaces for building P_4 using total sum allocation.

This allocation method is useful when the number of *CreateJoinPlans* is evenly distributed among threads. However, depending on the topologies of the query graph, each plan join in the MPJ may invoke a considerably different number of *CreateJoinPlans*. To remedy this, we propose a concept of *stratified allocation*. Formal analysis about different allocation schemes for different query topologies will be given in Section 6.

4.1.2 Stratified Allocation

Stratified allocation divides the search space of MPJ for P_S into smaller strata, and then applies an allocation scheme to each stratum. Each stratum corresponds to the search space of one plan join in MPJ, and thus the number of strata is $\lfloor S/2 \rfloor$. Stratified allocation more evenly spreads the number of actual *CreateJoinPlans* invocations among threads than does total sum allocation. We propose the following three different stratified allocation schemes.

Equi-Depth Allocation

Given m threads, equi-depth allocation divides the whole range of the outer loop in each plan join between $P_{smallSZ}$ and $P_{largeSZ}$ into smaller contiguous ranges of equal size. That is, with equi-depth allocation, each thread loops through a range of size $\frac{|P_{smallSZ}|}{m}$ in the outer loop.

This allocation scheme is useful when the size of the outer is divisible by the number of threads, and the number of invocations of *CreateJoinPlans* are similar for contiguous and equally-partitioned ranges.

Round-Robin Outer Allocation

Given m threads, round-robin outer allocation logically assigns the k-th tuple in the outer partition to thread k **mod** m. As with equi-depth allocation, each thread loops through a range of size $\frac{|P_{smallSZ}|}{m}$ in the outer loop.

With round-robin outer allocation, outer tuples are distributed randomly across threads. Thus, this allocation scheme works well even when there is skew in the number of *CreateJoinPlans* invocations for different outer rows in the plan join. However, as in star queries, if the number of outer tuples is small and is not divisible by m, then some threads will have an extra outer tuple, and hence would invoke a considerably larger percentage of *CreateJoinPlans* than those without that extra row. This phenomenon will be explained further in Section 6.

Round-Robin Inner Allocation

Given m threads, round-robin inner allocation logically assigns a join pair (t_i, t'_j) to thread (j **mod** m), where t'_j is the j-th tuple in the inner plan partition. Unlike all other allocation methods, each thread using this allocation scheme loops through the entire range of the outer loop of MPJ, but inner tuples are distributed randomly across threads. This has an effect similar to randomly distributing all join pairs in a plan join across threads. Therefore, this scheme provides the most uniform distribution of *CreateJoinPlans* invocations among threads, regardless of query topologies.

4.2 Basic MPJ

Since the MPJ algorithm is executed in memory, it must be very cache conscious to make the best use of the CPU's cache. We therefore base MPJ upon the block nested-loop join [14], which is considered to be one of the fastest cache-conscious, in-memory joins [30], and we physically cluster tuples in plan partitions using arrays. The join enumerators of conventional optimizers, such as those of DB2 and PostgreSQL [25], effectively use a tuple-based nested-loop method and are less cache conscious, so suffer more cache misses than our approach, especially for large plan partitions. Note that those join enumerators were developed before cache-conscious techniques emerged. In a block-nested loop join of relations R_1 and R_2, the inner relation R_2 is logically divided into blocks, and then, for each block B in the relation R_2, it performs the tuple-based nested-loop join over B and the outer relation R_1.

To represent an allocated search space for each thread, we introduce a data structure called the *search space description vector* (SSDV). This vector is computed according to

the chosen search space allocation scheme described in Section 4.1. Each entry in SSDV gives the parameters for one problem to be allocated to a thread, in the form of a quintuple: $\langle smallSZ$, [$stOutIdx$,$stBlkIdx$,$stBlkOff$], [$endOutIdx$,$endBlkIdx$,$endBlkOff$], $outInc$, $inInc\rangle$. Here, $smallSZ$ corresponds to a plan join between $P_{smallSZ}$ and $P_{S-smallSZ}$; [$stOutIdx$,$stBlkIdx$,$stBlkOff$] specifies the start outer tuple index, the start block index, and the offset of the start inner tuple in the block; [$endOutIdx$,$endBlkIdx$,$endBlkOff$] gives the end outer tuple index, the end block index, and the offset of the end inner tuple in the block; and $outInc$ and $inInc$ specify increasing step sizes for the loops over the outer and inner plan partitions, respectively. Due to space limitations, we omit detailed explanations of how to compute the SSDV for each of the search space allocation methods discussed above.

Example 1. *Recall Figure 2 where total sum allocation is used. For ease of explanation, let the block size be the size of the inner plan partition (= tuple-based nested loop). The SSDV for thread 1 is* $\{\langle 1, [1,1,1], [4,1,1], 1, 1\rangle, \langle 2, [-1,-1,-1], [-1,-1,-1], 1, 1\rangle\}$. *The first entry in the SSDV represent the first 10 pairs as shown in Figure 2. Since thread 1 does not execute a plan join between P_2 and P_2, ranges in the second entry are set to [-1,-1,-1]. The SSDV for thread 2 is* $\{\langle 1, [4,1,2], [4,1,3], 1, 1\rangle, \langle 2, [1,1,1], [3,1,3], 1, 1\rangle\}$. *The first entry represents the 11th and 12th pairs, and the second represents all 9 pairs for a plan join between P_2 and P_2.* □

Algorithm 3 represents a basic *MultiplePlanJoin* (MPJ) that can be used with the various allocation schemes discussed in Section 4.1. The inputs of the algorithm are an SSDV and the size S of quantifier sets for the plan partition to build. The loop iterates over the $SSDV$, calling *PlanJoin*. In *PlanJoin*, the first loop iterates over blocks in the inner plan partition $P_{S-smallSZ}$. The second loop iterates over tuples in the outer plan partition $P_{smallSZ}$. The last loop iterates over tuples in the current block of the outer relation. According to the current block number and the current offset, we compute ranges for outer tuples (Line 5) and the offsets for inner tuples in the block (Line 7). We omit detailed explanations of how to compute these values, which is not the focus of our paper. When $smallSZ = largeSZ$, we can use a simple optimization called *NoInnerPreceding*, since the plan join becomes a self-join [25]. That is, we skip any cases where the index of the inner tuple $t_i \leq$ that of the outer tuple t_o. We used *NoInnerPreceding* in all experiments in Section 7; however, for ease of explanation, we do not show this distinction in the algorithm.

Algorithm 3 MultiplePlanJoin

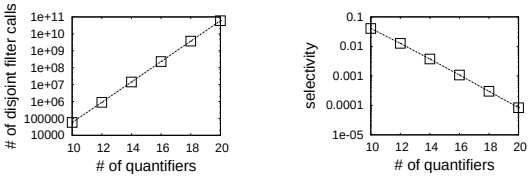
Input: $SSDV$, S
1: **for** $i \leftarrow 1$ **to** $\lfloor S/2 \rfloor$
2: **PlanJoin**($SSDV[i]$, S);

Function PlanJoin
Input: $ssdvElmt$, S
1: $smallSZ \leftarrow ssdvElmt.smallSZ$;
2: $largeSZ \leftarrow S - smallSZ$;
3: **for** $blkIdx \leftarrow ssdvElmt.stBlkIdx$ **to** $ssdvElmt.endBlkIdx$
4: $blk \leftarrow blkIdx$-th block in $P_{largeSZ}$;
5: $\langle stOutIdx, endOutIdx\rangle \leftarrow$ GetOuterRange($ssdvElmt, blkIdx$);
6: **for** $t_o \leftarrow P_{smallSZ}[stOutIdx]$ **to** $P_{smallSZ}[endOutIdx]$ **step by** $ssdvElmt.outInc$
7: $\langle stBlkOff, endBlkOff\rangle \leftarrow GetOffsetRangeInBlk(ssdvElmt, blkIdx,$ offset of $t_o)$;
8: **for** $t_i \leftarrow blk[stBlkOff]$ **to** $blk[endBlkOff]$ **step by** $ssdvElmt.inInc$
9: **if** $t_o.QS \cap t_i.QS \neq \varnothing$ **then continue**;
10: **if not**($t_o.QS$ connected to $t_i.QS$) **then continue**;
11: $ResultingPlans \leftarrow$ CreateJoinPlans(t_o, t_i);
12: PrunePlans($P_S, ResultingPlans$);

5. ENHANCED MULTIPLE PLAN JOIN

The basic MPJ in Section 4 requires invoking the disjoint filter for all possible pairs of tuples in the inner and outer plan partitions. Furthermore, as the number of quantifiers increases, the number of these disjoint filter invocations increases exponentially, especially in star queries, dominating the overall performance.

To measure the impact of both the number and the selectivity of these filter invocations, we performed some experiments for star queries as the number of quantifiers increased. Figure 3(a) confirms that the number of invocations of the disjoint filter increases exponentially with the number of quantifiers. Figure 3(b) shows that the selectivity of the disjoint filter decreases exponentially as the number of quantifiers increases. We observed a similar trend for variants of star queries, such as snowflake queries, which occur frequently in enterprise data warehouses. This escalating cost for invoking the disjoint filter motivated us to develop Skip Vectors, and the enhanced MPJ algorithm that uses them, to minimize unnecessary invocations of the disjoint filter.



(a) # of disjoint filter calls. (b) Selectivities.

Figure 3: Disjoint filter selectivity tests for star queries by varying the number of quantifiers.

5.1 Skip Vector Array

To avoid unnecessary invocations of the disjoint filter, we introduce a new index structure, called a Skip Vector, for speeding retrieval of disjoint quantifier sets. We augment each row in the plan partition with a Skip Vector for the quantifier set in that row. Collectively, the Skip Vectors for all rows in a plan partition are called the Skip Vector Array (SVA). The ith element of the Skip Vector for any given quantifier set qs gives the row k of the first quantifier set that does not contain the ith element of qs. Since we number quantifiers in a specific order and maintain quantifier sets for a plan partition in lexicographical order, the Skip Vector thus enables us to skip large groups of quantifier sets that are known to contain any element of a given quantifier set qs. In order to cluster together quantifiers likely to be joined together, we initially number quantifiers in the query graph in depth-first order, starting from the node having the maximum number of outgoing edges. For example, the hub node in a star query would typically be numbered one, since it usually has the maximum number of outgoing edges.

Let us define the Skip Vector more precisely. In the following, we will represent quantifier sets as text strings. For example, a quantifier set $\{q_1, q_3, q_5\}$ is represented as a string $q_1q_3q_5$.

The i-th row of plan partition P_S thus contains its quantifier set $P_S[i].QS$ and its corresponding Skip Vector $P_S[i].SV$,

as well as the plan list for $P_S[i].QS$. Then element j of $P_S[i].SV$, $P_S[i].SV[j]$, is defined as

$$\min\{k | P_S[i].QS[j] \text{ does not overlap } P_S[k].QS, k > i\}.$$

Example 2. *Consider plan partition P_3 in Figure 4. The Skip Vector is shown as the third column of the plan partition. Consider the first entry of the first skip vector $P_3[1].SV[1]$ (for quantifier set $q_1q_2q_3$), which is 8. This indicates that if any quantifier set qs matches $P_3[1].QS$ on its first element (q_1), then qs can skip to row 8 ($P_3[8]$), thereby bypassing rows 2 through 7 in P_3, because their quantifer sets all start with the same quantifier (q_1). Similarly, if qs matches on the first two elements (q_1q_2), then the second element, $P_3[1].SV[2]$, which contains 5, points to the first row (5) in P_3 at which the prefix q_1q_2 changes to another value (q_1q_3).* □

P_1

	QS	PlanList	SV
1	q_1	...	2
2	q_2	...	3
3	q_3	...	4
4	q_4	...	5
5	q_5	...	6
6	q_6	...	7
7	q_7		8
8	q_8	...	9

P_3

	QS	PlanList	SV		
1	$q_1q_2q_3$	...	8	5	2
2	$q_1q_2q_4$	...	8	5	3
3	$q_1q_2q_5$	...	8	5	4
4	$q_1q_2q_6$	...	8	5	5
5	$q_1q_3q_4$	...	8	6	8
6	$q_1q_4q_7$	...	8	8	7
7	$q_1q_4q_8$	...	8	8	8
8	$q_2q_5q_6$	...	9	9	9
9	$q_4q_7q_8$	...	10	10	10

Figure 4: Example SVAs.

Since the plan partition is sorted in lexicographical order, its SVA can be constructed in linear time, whenever the number of quantifiers in a query graph is constant. To compute the indexes for skip vectors efficiently, the algorithm *BuildSkipVectorArray* constructs skip vectors backwards, that is, from the last skip vector to the first one. Suppose that we are constructing the i-th skip vector $P_S[i].SV$ of P_S. We will have already constructed from the $(i+1)$-th skip vector of P_S to its end. If $P_S[i].QS[j]$ does not overlap $P_S[i+1].QS$, then $i+1$ is assigned to $P_S[i].SV[j]$. Otherwise, – i.e., if $P_S[i].QS[j]$ is equal to $P_S[i+1].QS[l]$ for some l – $P_S[i+1].SV[l]$ is assigned to $P_S[i].SV[j]$. For example, consider $P_3[4].SV$. $P_3[5].SV[1](=8)$ is assigned to $P_3[4].SV[1]$, since $P_3[4].QS[1]$ $(=q_1)$ is equal to $P_3[5].QS[1]$. $P_3[4].SV[2]$ is assigned to 5, since $P_3[4].QS[2](=q_2)$ does not overlap $P_3[5].QS(=q_1q_3q_4)$. Similarly, $P_3[4].SV[3]$ is assigned to 5. Since quantifier sets are lexicographically ordered, the time complexity of constructing a skip vector is $O(S)$.

Theorem 1. *Given a plan partition P_S, the time complexity of BuildSkipVectorArray is $O(|P_S| \times S)$.*

Now, we describe how to use the SVA in our parallel DP join enumerator. To use a pair of partitioned SVAs as the unit of allocation to threads, we first partition each plan partition into sub-partitions. To support MPJ with SVA using total sum allocation or equi-depth allocation, the plan partition needs to be partitioned using equi-depth partitioning. To support MPJ with SVA using round-robin inner or outer allocation, the plan partition needs to be partitioned using round-robin partitioning. Ideally, the total number of sub-partitions for a plan partition should be a multiple of the number of threads, in order to assign an equal number of sub-partitions pairs to threads when we use *NoInnerPreceding* optimization. We denote the j-th sub-partition of P_S as $P_{\{S,j\}}$. Next, we build the SVAs for all the sub-partitions. Here, for fast clustered access, we can embed skip vectors within sub-partitions. Figure 5 shows an example of partitioned SVAs using the equi-depth partitioning. The plan partition P_3 is first partitioned into four sub-partitions, $P_{\{3,1\}}$, $P_{\{3,2\}}$, $P_{\{3,3\}}$, and $P_{\{3,4\}}$. We next build embedded SVAs for the four sub-partitions.

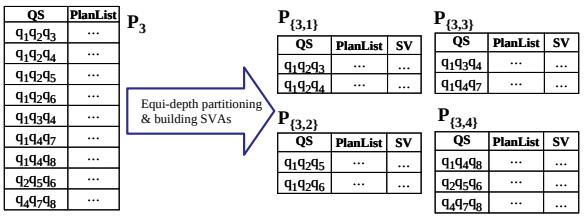


Figure 5: An example of a plan partition divided into four sub-partitions.

5.2 MPJ With Skip Vector Array

We first explain how to allocate search spaces when MPJ with SVA is executed. As explained in the previous section, the unit of allocation to threads is a pair of partitioned SVAs. Except for using a different allocation unit, we can reuse the same allocation schemes developed in Section 4.1.

Algorithm 4 represents the enhanced MPJ algorithm, *MultiplePlanJoinWithSVA*, that exploits SVAs. The inputs of the algorithm are an $SSDV$ and the size S of quantifier sets for the plan partition to build. The loop iterates over the $SSDV$, calling *PlanJoinWithSVA*. In *PlanJoinWithSVA*, the first loop iterates over sub-partitions in the outer plan partition $P_{smallSZ}$. The second loop iterates over sub-partitions in the inner plan partition $P_{largeSZ}$ and invokes Skip Vector Join SVJ subroutine, described next, for $P_{\{smallSZ,outerPartIdx\}}$ and $P_{\{largeSZ,innerPartIdx\}}$.

Algorithm 4 MultiplePlanJoinWithSVA

Input: $SSDV$, S
1: **for** $i \leftarrow 1$ **to** $\lfloor S/2 \rfloor$
2: **PlanJoinWithSVA**$(SSDV[i], S)$;

Function PlanJoinWithSVA

Input: $ssdvElmt$, S
1: $smallSZ \leftarrow ssdvElmt.smallSZ$;
2: $largeSZ \leftarrow S - smallSZ$;
3: **for** $outerPartIdx \leftarrow ssdvElmt.stOuterPartIdx$ **to** $ssdvElmt.endOuterPartIdx$ **step by** $ssdvElmt.outInc$
4: $\langle stInnerPartIdx, endInnerPartIdx\rangle \leftarrow$ GetInnerRange$(ssdvElmt, outerPartIdx)$;
5: **for** $innerPartIdx \leftarrow stInnerPartIdx$ **to** $endInnerPartIdx$ **step by** $ssdvElmt.inInc$
6: $outerPartSize \leftarrow |P_{\{smallSZ,outerPartIdx\}}|$;
7: $innerPartSize \leftarrow |P_{\{largeSZ,innerPartIdx\}}|$;
8: **SVJ**$(\langle P_{\{smallSZ,outerPartIdx\}}, 1, outerPartSize\rangle, \langle P_{\{largeSZ,innerPartIdx\}}, 1, innerPartSize\rangle)$;

Note that there are two differences between *MultiplePlanJoin* (Section 4.2) and *MultiplePlanJoinWithSVA* algorithms. First, *MultiplePlanJoinWithSVA* uses loops over sub-partitions, whereas *MultiplePlanJoin* uses loops over tuples. Secondly, *MultiplePlanJoinWithSVA* invokes the Skip Vector Join subroutine for each inner and outer sub-partition to skip over partitions that won't satisfy the disjoint filter, whereas *MultiplePlanJoin* performs a block nested-loop join on all pairs, resulting in many unnecessary invocations of the disjoint filter. Apart from these differences, the two algorithms are the same.

Algorithm 5 defines the Skip Vector Join (*SVJ*) subroutine, which is an indexed join for two sub-partitions exploiting their embedded SVAs. The inputs of the algorithm are

(a) the inner/outer sub-partitions $P_{\{smallSZ,outerPartIdx\}}$ (= R_1) and $P_{\{largeSZ,innerPartIdx\}}$ (=R_2), (b) the start indexes idx_{R_1} and idx_{R_2} of tuples in R_1 and R_2, respectively, and (c) the end indexes $endIdx_{R_1}$ and $endIdx_{R_2}$ of R_1 and R_2, respectively. *SVJ* checks whether two tuples are disjoint (Lines 3-4). If so, *SVJ* invokes the connectivity filter and generates join results (Lines 5-7). After that, *SVJ*s are recursively called to join all remaining join pairs of the two sub-partitions (Lines 8-9). If the two tuples are not disjoint, we first obtain skip indexes for the first overlapping element (Lines 11-15). Then, we skip all overlapping pairs using the skip indexes obtained, and recursively call *SVJ*s. (Lines 16-17). Note that, for fast performance, the iterative version of *SVJ* is used in our experiments.

Algorithm 5 SVJ (Skip Vector Join)

Input: $\langle P_{\{smallSZ,outerPartIdx\}}(=R_1), idx_{R_1}, endIdx_{R_1}\rangle$, $\langle P_{\{largeSZ,innerPartIdx\}}(=R_2), idx_{R_2}, endIdx_{R_2}\rangle$

1: $S \leftarrow smallSZ + largeSZ$;
2: **if** $idx_{R_1} \le endIdx_{R_1}$ **and** $idx_{R_2} \le endIdx_{R_2}$ **then**
3: $overlapQS \leftarrow R_1[idx_{R_1}].QS \cap R_2[idx_{R_2}].QS$;
4: **if** $overlapQS = \varnothing$ **then** /*the case for join*/
5: **if** ($R_1[idx_{R_1}].QS$ connected to $R_2[idx_{R_2}].QS$) **then**
6: $ResultingPlans \leftarrow$ CreateJoinPlans($R_1[idx_{R_1}], R_2[idx_{R_2}]$);
7: PrunePlans($P_S, ResultingPlans$);
8: SVJ($\langle R_1, idx_{R_1}+1, endIdx_{R_1}\rangle$,$\langle R_2, idx_{R_2}, endIdx_{R_2}\rangle$);
9: SVJ($\langle R_1, idx_{R_1}, idx_{R_1}\rangle$,$\langle R_2, idx_{R_2}+1, endIdx_{R_2}\rangle$);
10: **else** /*the case for skip*/
11: $elmt \leftarrow$ FirstElmt($overlapQS$);
12: $lvl_{R_1} \leftarrow$ GetLevel($R_1[idx_{R_1}].QS, elmt$);
13: $lvl_{R_2} \leftarrow$ GetLevel($R_2[idx_{R_2}].QS, elmt$);
14: $jpIdx_{R_1} \leftarrow R_1[idx_{R_1}].SV[lvl_{R_1}]$;
15: $jpIdx_{R_2} \leftarrow R_2[idx_{R_2}].SV[lvl_{R_2}]$;
16: SVJ($\langle R_1, jpIdx_{R_1}, endIdx_{R_1}\rangle$,$\langle R_2, idx_{R_2}, endIdx_{R_2}\rangle$);
17: SVJ($\langle R_1, idx_{R_1}, \min(jpIdx_{R_1}-1, endIdx_{R_1})\rangle$, $\langle R_2, jpIdx_{R_2}, endIdx_{R_2}\rangle$);

Example 3. *Consider the SVJ for plan partitions P_1 and P_3 exploiting their SVAs in Figure 4. Suppose that $SVJ(\langle P_1, 1, 8\rangle, \langle P_3, 1, 9\rangle)$ is invoked. Since the first entries of the partitions overlap (q_1 and $q_1q_2q_3$), we skip to the second entry of the first partition using $P_1[1].SV[1](=2)$ and skip to the eighth entry of the second partition using $P_3[1].SV[1](=8)$. We then recursive call $SVJ(\langle P_1, 2, 8\rangle, \langle P_3, 1, 9\rangle)$ and $SVJ(\langle P_1, 1, 1\rangle, \langle P_3, 8, 9\rangle)$. For $SVJ(\langle P_1, 1, 1\rangle, \langle P_3, 8, 9\rangle)$, since the first entry in P_1 and the eighth entry in P_3 are disjoint, we join the two quantifiers, and then, recursively call $SVJ(\langle P_1, 2, 1\rangle, \langle P_3, 8, 9\rangle)$ and $SVJ(\langle P_1, 1, 1\rangle, \langle P_3, 9, 9\rangle)$.* □

Theorem 2. *Given two plan partitions P_M and P_N, Algorithm SVJ correctly generates all feasible QEPs using P_M and P_N for the plan partition P_{M+N}.*

As an interesting alterative method for *SVJ*, we could exploit inverted indexing techniques used for documents to efficiently determine overlapping quantifier sets for a given quantifier set qs [9, 19]. In this approach, sets are treated as documents, and elements as keywords. We first compute the corresponding inverted list for each quantifier in qs. Next, we UNION all of these inverted lists, that is, all overlapping sets. By then accessing the complement of the UNIONed set, we will find all disjoint sets for qs. By storing inverted lists as bitmaps, we can obtain the complement of the UNIONed set very easily. Here, we need to execute bit operations to find bits having 0 from the UNIONed set. Given two partitions $P_{smallSZ}$ and $P_{largeSZ}$, the time complexity of this inverted-index scheme is $O(|P_{smallSZ}| \times smallSZ \times I_{largeSZ})$, where $smallSZ$ is the size of the quantifier set in $P_{smallSZ}$ and $I_{largeSZ}$ is the size of the inverted list for $P_{largeSZ}$. Observe that $I_{largeSZ}$ is in proportion to $|P_{largeSZ}|$. The time complexity of the basic MPJ is $O(|P_{smallSZ}| \times |P_{largeSZ}|)$. Thus, the inverted-index variant of MPJ outperforms the basic MPJ when $|P_{largeSZ}| > smallSZ \times I_{largeSZ}$. The time complexity of *SVJ* is $O(\#\ of\ disjoint\ pairs)$. So *SVJ* is much faster than the other two join methods for joins over large plan partitions. Note also that the SVA can be used for both one-index and two-index joins. However, we do not use the algorithm in [19] to compute the complement of a set that requires two inverted indexes, because of the expense of building a two-dimensional bitmap for the set that can be constructed only after a join.

6. FORMAL ANALYSIS OF DIFFERENT ALLOCATION SCHEMES

In this section, we formally analyze why the different search allocation schemes generate different numbers of *CreateJoinPlans* among threads. For a given allocation scheme, the number of *CreateJoinPlans* invoked per thread is determined by the topology of the query graph. Figure 6 shows four different representative query topologies: linear, cycle, star, and clique.

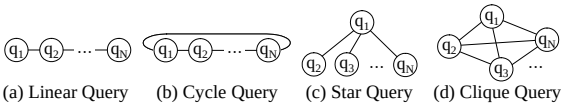


Figure 6: Different query graph topologies.

For each quantifier set qs in the outer plan partition $P_{smallSZ}$, we calculate $NumCJP(qs)$, which is the number of *CreateJoinPlans* invoked for the quantifier set qs. We note that *CreateJoinPlans* is called only when two quantifiers to join are disjoint and connected. We assume that both plan partitions are sorted in lexicographical order. We also assume that the *NoInnerPreceding* optimization is not used. In Section 7, we analyze the effect of the *NoInnerPreceding* optimization.

Linear Query

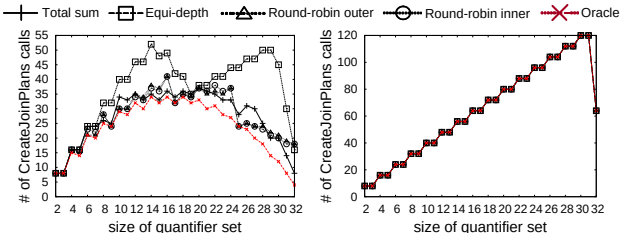
Theorem 3. *Consider a linear query with N quantifiers where nodes in the query graph are numbered from one to N in depth first order, starting with the node having only one connected node as shown in Figure 6 (a). Given any plan join between $P_{smallSZ}$ and $P_{largeSZ}$ for P_S such that $S = smallSZ + largeSZ$, consider a quantifier set qs in the outer plan partition, where $qs = \{q_{a_i}, ..., q_{a_{i+smallSZ-1}}\}$. Case* 1) *If $(a_i < largeSZ+1) \wedge (a_i > N-S+1)$, $NumCJP(qs) = 0$; Case* 2) *if $largeSZ+1 \le a_i \le N-S+1$, $NumCJP(qs) = 2$; Case* 3) *otherwise, $NumCJP(qs) = 1$.*

Proof: See Appendix B. □

With Theorem 3, we see that quantifier sets in different contiguous ranges in Cases $1 \sim 3$ invoke three different numbers of *CreateJoinPlans*. With equi-depth allocation, contiguous ranges are allocated to threads, and thus it would invoke skewed numbers of *CreateJoinPlans* among the threads. With total sum allocation, all outer tuples in $\lfloor S/2 \rfloor$ plan joins are allocated to threads in equi-depth fashion across joins. Thus, it would invoke very similar numbers

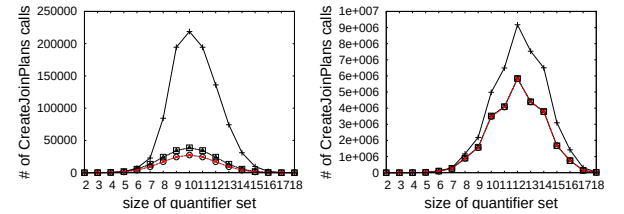
of *CreateJoinPlans* among threads. The round-robin inner (or outer) schemes also invoke almost similar numbers of *CreateJoinPlans* among threads, since inner (or outer) tuples in any contiguous range are randomly distributed.

To verify our analysis, we performed experiments using all four allocation schemes for a linear query with 32 quantifiers. In addition, we plotted the curve generated by an optimal "oracle" allocation scheme that always divides the total number of *CreateJoinPlans* evenly among threads. Here, we use 8 threads. Figure 7 plots the maximum number of *CreateJoinPlans* invocations made by all threads as a function of the size of the quantifier sets being built. With the exception of equi-depth allocation, all other allocation schemes generate very similar numbers of *CreateJoinPlans* invocations among threads as does the oracle allocation.



(a) Linear query with 32 quantifiers.

(b) Cycle query with 32 quantifiers.



(c) Star query with 18 quantifiers.

(d) Clique query with 18 quantifiers.

Figure 7: Distribution of # of *CreateJoinPlans* invocations by 8 threads for different allocation schemes.

Cycle Query

Theorem 4. *Consider a cycle query with N quantifiers, where nodes in the query graph are numbered from one to N in depth-first order, as shown in Figure 6 (b). Given any plan join between $P_{smallSZ}$ and $P_{largeSZ}$ for P_S such that $S = smallSZ + largeSZ$, consider a quantifier set qs in the outer plan partition, where $qs = \{q_{a_i}, ..., q_{a_{i+smallSZ-1}}\}$. If $S = N$, $NumCJP(qs) = 1$. Otherwise, $NumCJP(qs) = 2$.*

Proof: See Appendix C. □

In Theorem 4, it is clear that all allocation schemes generate the same *CreateJoinPlans* invocation numbers. We can verify our analysis with experiments for a cycle query with 32 quantifiers as in Figure 7(b).

Star query

Theorem 5. *Consider a star query with N quantifiers where nodes in the query graph are numbered from one to N in depth first order as shown in Figure 6 (c). Given any plan join between $P_{smallSZ}$ and $P_{largeSZ}$ for P_S such that $S = smallSZ + largeSZ$, consider a quantifier set qs in the outer plan partition. Case 1) If $(smallSZ > 1) \vee ((qs = \{q_1\}) \wedge (largeSZ > 1))$, $NumCJP(qs) = 0$; Case 2) if $qs = \{q_1\} \wedge largeSZ = 1$, $NumCJP(qs) = N - 1$; Case 3) otherwise, $NumCJP(qs) = \binom{N-2}{largeSZ-1}$.*

Proof: See Appendix D. □

With Theorem 5, we see that the number of *CreateJoinPlans* calls are extremely skewed with respect to $smallSZ$, depending upon the allocation method. For total sum allocation, the number of *CreateJoinPlans* invocations are very skewed. For equi-depth and round-robin outer, threads invoke different numbers of *CreateJoinPlans* depending on whether outer tuples contain the hub quantifier or not. Note that, with equi-depth and round-robin outer, the maximum difference of outer tuples to process per thread is 1. This difference is negligible in other topologies, since $\lfloor |P_{smallSZ}|/m \rfloor$ (# of outer tuples to process per thread) is much larger than m. However, in star queries, we can call *CreateJoinPlans* only if $|P_{smallSZ}|$ is the number of quantifiers, and thus, $\lfloor |P_{smallSZ}|/m \rfloor$ is also very small. Thus, this difference is no longer negligible. With round-robin inner allocation, we invoke nearly the same numbers of *CreateJoinPlans* among threads, since we evenly partition inner tuples for each outer tuple. Our analysis is verified by Figure 7(c).

Clique Query

Theorem 6. *Consider a clique query with N quantifiers where nodes in the query graph are numbered in depth first order as shown in Figure 6 (d). Given any plan join with $P_{smallSZ}$ and $P_{largeSZ}$ for P_S such that $S = smallSZ + largeSZ$, consider a quantifier set qs in the outer plan partition. $NumCJP(qs) = \binom{N - smallSZ}{largeSZ}$.*

Proof: See Appendix E. □

All allocation methods except for total sum generate the same invocation numbers of *CreateJoinPlans* among threads. $NumCJP(qs)$ for clique queries depends on the value of $smallSZ$. Thus, total sum allocation generates considerably different invocation numbers as shown in Figure 7(d).

7. EXPERIMENTS

The goals of our experiments are to show that: 1) our algorithms significantly outperform the conventional serial DP algorithm, in Section 7.1; and 2) both the basic and enhanced MPJ algorithms achieve almost linear speed-up, in Sections 7.2 and 7.3, repectively. We evaluated four different query topologies: linear, cycle, star, and clique. Since smaller plan partitions rarely benefit from parallelism, our parallel DP optimizer is invoked only when the sizes of plan partitions exceed a certain threshold, ensuring that our solution never slows down optimization.

All the experiments were performed on a Windows Vista PC with two Intel Xeon Quad Core E5310 1.6GHz CPUs (=8 cores) and 8 GB of physical memory. Each CPU has two 4Mbyte L2 caches, each of which is shared by two cores. We implemented all algorithms in PostgreSQL 8.3 [25] 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. We generate one merge key for each qualified pair of quantifier sets when there are no interesting orders in the higher levels, since there is little benefit to vary merge keys.

We used the $NoInnerPreceding$ optimization, explained in Section 4.2, for all experiments. That is, we skip any cases where the index of the inner tuple $t_i \leq$ that of the outer tuple t_o. To evenly distribute the number of disjoint filter calls among threads under this optimization, round-robin outer allocation executes in a *zig-zag* fashion. That is, suppose that the $i(\geq 0)$-th tuple in the outer partition is being assigned to thread j $(0 \leq j \leq m\text{-}1)$. If $\lfloor i/m \rfloor$ (=the number of tuples allocated so far for the thread) is even, the next tuple index to allocate for the thread is $i+2m-2j-1$; otherwise, the index is $i+2j+1$. We also applied this technique to the round-robin inner allocation that was used for all parallel algorithms.

Our performance metrics are the number of disjoint filter invocations, the number of $CreateJoinPlans$ invocations, and the speed-up, where speed-up is defined as the execution time of the serial algorithm divided by that of the parallel algorithm. Table 1 summarizes the experimental parameters and their values. We omit all experimental results for linear and cycle queries, because the sizes of their plan partitions are generally too small to benefit from parallelization. For clique queries, we vary the number of quantifiers only up to 18 because optimization would take too long with the serial optimizer, and the trends observed do not change when the number of quantifiers is larger than 16.

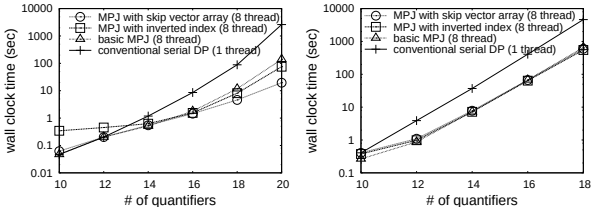
Table 1: Experimental parameters and their values.

Parameter	Default	Range
query topology	star, clique	star, clique
# of quantifiers	20, 18	10, 12, 14, 16, 18, 20
# of threads	8	$1 \sim 8$

7.1 Overall comparison of different algorithms

Experiment 1: Effect of # of quantifiers and query topologies. Figure 8 shows our experimental results for star and clique queries exploiting 8 threads, using our different allocation algorithms as the number of quantifiers increases.

For star queries having ≤ 14 quantifiers, the basic MPJ performs the best. However, as the number of quantifiers increases over 16, plan partitions become big enough to benefit from our SVA. Specifically, MPJ with SVA outperforms the basic MPJ by up to 7.2 times, inverted index-based MPJ by up to 3.8 times, and the conventional serial DP (PostgreSQL DP join enumerator whose algorithm is outlined in Algorithm 1) by up to 133.3 times. This is because, as the number of quantifiers in the query increases, the number of overlapping join pairs increases exponentially as well. In another experiment using a star query with 22 quantifiers, MPJ with SVA outperforms the basic MPJ by up to 18.1 times, inverted index-based MPJ by up to 10.2 times, and the conventional serial DP by up to 547.0 times, from 14 hours 50 minutes to 98 seconds! For clique queries, the basic MPJ slightly outperforms all other methods when the number of quantifiers ≤ 12. All parallel algorithms have almost the same performance for clique queries having more than 12 quantifiers. This is because invocations of $CreateJoinPlans$ dominate the execution time in clique queries, and we used the best allocation scheme, round-robin inner, for all parallel algorithms.

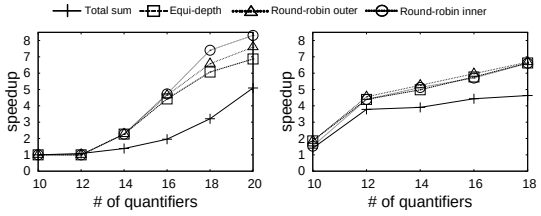


(a) Star queries. (b) Clique queries.

Figure 8: Experimental results using all different algorithms (8 threads).

7.2 Sensitivity analysis for basic MPJ

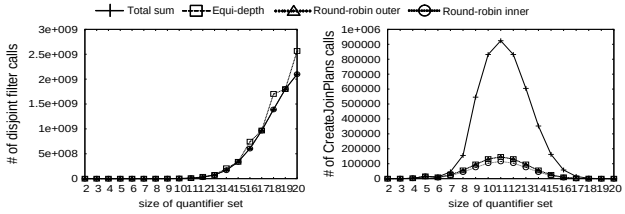
Experiment 2: Effect of # of quantifiers and query topologies. Figure 9 shows the experimental results for varying the number of quantifiers for star and clique queries using 8 threads. The speed-up of the parallel basic MPJ methods over the serial basic MPJ using various allocation schemes is shown for star queries in Figure 9(a) and for clique queries in Figure 9(b). With star queries, only round-robin inner achieves linear speed-up when the number of quantifiers ≥ 18. This is because plan partitions are large enough to exploit 8 parallel threads, and round-robin inner evenly allocates search spaces to threads. Since threads access the same inner/outer plan partitions, we achieve 8.3 times speed-up for quantifier sets of size 20 with round-robin inner, due to caching effects. Clique queries achieve higher overall speed-ups than comparable star queries with the same number of quantifiers because the numbers of *CreateJoinPlans* calls in clique queries are much larger than those in equally-sized star queries. Note that the maximum speedup for clique queries is about 7. This is because 1) in clique queries, the number of invocations for *CreateJoinPlans* dominates performance, and 2) each thread accesses the main memory using the per-thread memory manager, in order to generate sub-plans in *CreateJoinPlans*, which then results in some cache contention.



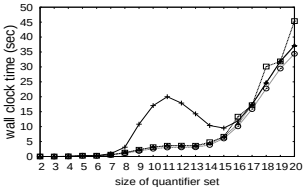
(a) Star queries. (b) Clique queries.

Figure 9: Experimental results for speed-up by varying the number of quantifiers (8 threads).

Figure 10 compares our three different performance metrics – the number of disjoint filter calls, the number of *CreateJoinPlans* calls, and wall clock time – for a star query with 20 quantifiers; and Figure 11 does the same for the clique query with 18 quantifiers. These figures plot the maximum performance metric among all threads as a function of the size of the quantifier sets being built.

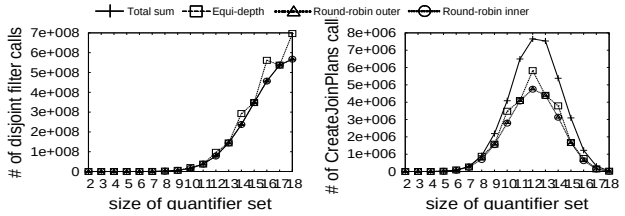


(a) # of disjoint filter calls. (b) # of *CreateJoinPlans* calls.

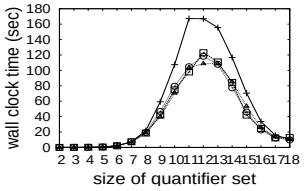


(c) Wall clock time.

Figure 10: Distributions of performance figures using basic MPJ for the star query (8 threads).



(a) # of disjoint filter calls. (b) # of *CreateJoinPlans* calls.



(c) Wall clock time.

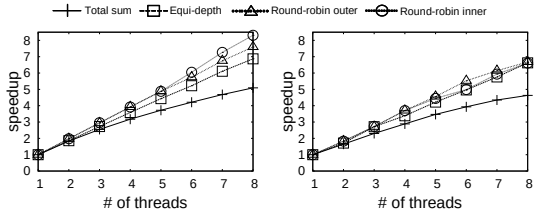
Figure 11: Distributions of performance figures using basic MPJ for the clique query (8 threads).

The general trend of all plots in Figure 10(c) is that the elapsed time first increases until the size of quantifier sets reaches 11, and then decreases until the size reaches 15, after which it sharply increases. This trend is explained as follows. The elapsed time mostly depends on the number of invocations of both *CreateJoinPlans* and the disjoint filter. However, the cost of *CreateJoinPlans* is much higher than that of the disjoint filter. As the size of the quantifier sets increases, the number of the disjoint filter calls increases exponentially for the star query, as shown in Figure 10(a). At the same time, the number of *CreateJoinPlans* calls first increases until the quantifier set size is 11, and then decreases, forming a bell shape, as in Figure 10(b). Combining these two costs, we obtain plots such as in Figure 10(c). Note also that equi-depth allocation does not evenly distribute the number of disjoint filter calls among threads, since we applied the *NoInnerPreceding* optimization. This optimization is used only when the plan is a self-join, and thus we see a skewed number of disjoint filter calls when the sizes of quantifier sets to build are even numbers.

For clique queries, the trend of plots in Figure 11(c) is the same as that in Figure 11(b). This is because the number of *CreateJoinPlans* calls is only 100 times smaller than the number of disjoint filter calls, and the cost of *CreateJoinPlans* is much higher than that of the disjoint filter.

Experiment 3: Effect of # of threads and query topologies. Figure 12(a) shows the speed-up of the parallel basic MPJ with various allocation schemes over the serial basic MPJ for star queries; Figure 12(b) shows the same for clique queries.

Regardless of query topologies, round-robin inner allocation achieves almost linear speed-up as the number of threads increases. For star queries, the second best allocation is round-robin outer, the third is equi-depth, and total sum allocation performs the worst, with a speed-up of 5.1 using 8 threads. For clique queries, all allocation methods except total sum allocation achieve nearly the same performance.



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

Figure 12: Experimental results for speed-up by varying the number of threads.

7.3 Sensitivity analysis for Enhanced MPJ

Experiment 4: Effect of # of quantifiers and query topologies. Figure 13 shows the performance of our Enhanced MPJ with SVA for star and clique queries, varying the number of quantifiers. Figure 13(a) shows the speed-up of the parallel MPJ with SVA using various allocation schemes among 8 threads, versus the serial MPJ with SVA for star queries; Figure 13(b) does the same for clique queries. The SVA reduces the cost of disjoint filter invocation to almost negligible. However, for star queries, merging results after executing MPJ constitutes about 5% of the overall execution time. Thus, we achieve 6.1 times speed-up with round-robin inner for star queries. Attempting to reduce the merging time would be interesting future work. Note that equi-depth and round-robin outer perform comparably to round-robin inner at 16 quantifiers, since 16 is divisible by the number of threads (=8), and thus all threads process an equal number of outer tuples.

Figure14(a) analyzes the performance for our three performance metrics. Again, the SVA reduces the number of disjoint filter calls to near the theoretical lower bound. Thus, the trend of plots in Figure 14(b) is the same as that in Figure 14(c). Clique queries have performance similar to that of Figure 11, so we omit the figures for them.