Optimization Strategy of Top-Down Join Enumeration on Modern Multi-Core CPUs

Wanli Zuo
Key Laboratory of Symbol Computation and Knowledge Engineering of the Ministry of Education
College of Computer Science and Technology, Jilin University, Changchun, China
Email:wanli@jlu.edu.cn

Yongheng Chen
College of Computer Science and Technology, Jilin University, Changchun, China
Email:cyh771@163.com

Fenglin He and Kerui Chen
College of Computer Science and Technology, Jilin University, Changchun, China

Abstract—Most contemporary database systems query optimizers exploit System-R's bottom-up dynamic programming method (DP) to find the optimal query execution plan (QEP) without evaluating redundant sub-plans. The distinguished exceptions are Volcano/Cascades using transforms to generate new plans according to a top-down approach. As recent research has revealed, bottom-up dynamic programming can improve performance with respect to the shape of the join graph and parallelism. However top-down join enumeration dynamic programming method can derive upper bounds for the costs of the plans it generates which is not available to typical bottom-up DP method. In this paper, we propose a comprehensive and practical framework for parallelizing top-down dynamic programming query optimization with complex non-inner join in the multi-core processor architecture, referred as PTDhyp. We have implemented such a search strategy and experimental results show that can improve optimization time effective compared to known existing algorithms.

Index Terms—Multi-core, Query optimization, Join-Order, Dynamic Programming

I. INTRODUCTION

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 [3, 5, 6]. Instead, the industry has been improving the CPU performance by integrating more execution cores into each processor [2, 7].

Based on this trend, it has become tempting to revisit the concepts of database parallelism in the light of those emerging hardware architectures. Recently, by exploiting the new wave of multi-core processor architecture, Han et al. first propose a novel algorithm PDPsva[4] to parallelize query optimization process to exploit multi-core processor architectures whose main memory is shared among all cores. However, PDPsva generated optimal query plan for all smaller quantifiers sets. On contrary, DP optimizers such as DPcpp[8], which directly traverse a query graph to generate join pairs. Thus, plan generation mainly use of join pair without cross products, reduce execution time. While classical (inner) joins are by far the most important type of joins, some queries require non-inner joins like outer joins or antijoins. But DPcpp handle only simple (binary) join predicates and innerjoins. For this reason, taking the most efficient known join-ordering algorithm, DPcpp, as a starting point, DPhyp[9] models the query graph as hypergraph to exploit this capability to efficiently handle the widest class of non-inner joins. Furthermore [13] bring forward a solution for parallelizing query optimization based bottom-up and graph-traversal driven enumeration, referred as DPEGeneric, which sustain dynamic search allocation.

The algorithms discussed above which all constructed based on bottom-up join enumeration method. By contrast with the research about bottom-up method, the research about top-down join enumeration is relative less recently. In particular, to the best of our knowledge, the mind about the parallelizing query optimization of queries using dynamic programming method of top-down join enumeration that reference joins other than the standard “innerjoins”, specifically “outerjoins” and “antijoins” has not been proposed.

In this paper, we propose a novel framework to parallelize query optimization process which realizes top-down join enumeration methods with non-inner join predicates. Section 2 reviews the general bottom-up and top-down join enumeration algorithms. Section 3 gives a semantics expression and application of non-join predicates. Section 4 gives the method of constructing the connected join pairs. Section 5 gives the parallelizing query optimization algorithm of top-down join enumeration. Section 6 presents the results of performance evaluation. Section 7 concludes the paper.

© 2011 ACADEMY PUBLISHER
II. PREVIOUS WORD

This section gives an overview of existing join enumeration algorithms, divided by the search space.

A. Bottom-up Join Enumeration

An enumeration algorithm is named bottom-up if subproblem (e.g., deal with all smaller quantifier sets of both qs1 and qs2 before processing a pair of quantifier sets(qs1, qs2).)

1) Compositional Dynamic Programming

Compositional Dynamic Programming is a bottom-up technique that represents logical expressions as sets of relations, and enumerates over choices for already optimized sets V1 and V2 to obtain a new plan for V=V1 ∪ V2 [15]. There are two types Compositional Dynamic Programming which are frequently used by commercial optimizers based on the style of how the pairs of quantifier sets are generated. The first is size-driven; the second is graph-traversal driven.

Size-driven enumeration generates solutions for a larger problem in bottom-up fashion by combining solutions for smaller problems [1]. Reference to this description literally, we can start from a single quantifier and then, construct all query plans containing quantifier sets of size S2 by “nested loops” between quantifier sets of smallsz and quantifier sets of largesz such that largesz=sz smallsz. Although widely used, sized-driven enumeration is far from optimal because of attempted compositions of overlapping sets. However graph-traversal driven enumeration generates a pair of quantifier sets by directly traversing the query graph. The pair of quantifier sets (qs1,qs2) satisfy this conditions that qs1 is generated by enumerating all connected subgraphs of the query graph, and qs2 is generated by enumerating all other connected subgraphs that are disjoint and connected to qs1 [13].

2) Partitioning Dynamic Programming

Partitioning dynamic programming drives the enumeration by the choice of V, which is then partitioned into all choices for V1 and V2 such that V=V1 ∪ V2. The enumeration order of V must be carefully designed so that for any valid partition of V the sub-expressions have already been optimized [15].

Vance and Majer proposed [16] this method and used an efficient algorithm to search the space of bushy plans with Cartesian products for generating subsets to drive the ordering and partitioning of V.

B. Top-Down Join Enumeration

An algorithm is called top-down if it considers large logical expressions before small large ones. The top-down algorithm begins with a group consisting entirely of node, then considers generate all candidate logically equivalent multi-expression.

Transformational search is a top-down algorithm and amenable to top-down enhancements such as demand-driven interesting orders and branch-and-bound, which represents logical multi-expression as trees of ordered binary joins. After converting a query to an arbitrary logical join tree, the space of possible join plans is searched by top-down application of logical-to-logical and logical-to-physical transformations [15].

III. OUTJOIN AND ANTIJOIN REORDERING

“inner-join” is the most common type of join operation, which can be freely reordered. The freely reorder property endows the optimizer a great convenience and lead to a great deal of possible join orderings. Then the optimizer chooses the cheapest execution query.

Liking inner-join, improve the performance of query execution can be completed by reorder the outjoins and antijoins. Because the associative with each other of these types of joins are not always valid, reorder involving outerjoins and/or antijoins are complicated.

A. Existing Approaches

As with joins, the optimization performance of query execution can be completed by changing the order of evaluation of outerjoins and antijoins. However associability properties of outerjoin and antijion are limited, which cause the generation of equivalent plan a much more difficult task. Several proposals about the problem of outerjoin and antijion has been existing in many literature.

The difficulty when a query involving outerjoins and/or antijoins is that, unlike joins, a query graph without informa- tion on evaluation order is ambiguous. In [17], Galindo-Legaria and Rosenthal first overcome the ambiguity problem by simplifying the initial operator tree. Then Galindo-Legaria and Rosenthal capture the semantics of a query through query graph analysis to detect conflicting reordering [17]. This conflicting set for each join predicate is stored and used to form proper join orders in bottom-up join optimizer. This approach was improved to conflict analysis with paths in hypergraphs [18]. As pointed out by Rao et al. there is a much simpler ordering test using extended eligibility list or EEL to resolve this problem [18]. An EEL includes all additional tables needed by a predicate to preserve the semantics of the original query. Then two subplans can be joined only if all the tables in the EEL exist in the two subplans.

B. Reordering Algorithm Based On Top-Down

The reordering approaches discussed above are performed in a conventional bottom-up optimizer using dynamic programming. That is the two subplans quantifier sets must be dealt before processing larger quantifier sets. This is different from top-down optimizer. So according to the property of top-down optimizer an improved algorithm satisfying the top-down optimizer is proposed in the following.

1) The Semantics Expression

This subsection shows how to set the semantics of the original query when a query involving outerjoins and/or antijoins in the operator tree.

At the beginning of explanation of the algorithm, we briefly introduce the rules of conflict. All of the conflicts are summarized in Figure 1.
Figure 1. The conflicts of join predicate

The conflicting graph of Figure 1 provides the following information. An Inner join has no conflicts with outerjoin and antijoin. An antijoin has conflicts with outerjoin "pointing" inward in the preserving side and all kinds of joins in the null-producing side. An outerjoin conflicts with antijoin and inner join in the null-producing side.

Let us now formalize this approach. As usual, $\circ$ denotes the operator. A NS includes relations referenced by the join predicate, which is used to associate each join predicate. An ES includes additional relations needed by a predicate to preserve the semantics of the original query. We denote by $\text{ref}(p)$ the set of tables referenced by the join predicate $p$. The $\text{ref}(\text{preserving}(p))$ and $\text{ref}(\text{nullproducting}(p))$ denote the set of table reference by the join predicate $p$ in the preserving side and null-producing side. An set_outerjoin contains all the relations associated by inner join or antijoin predicates. An set_antijoin includes the relations that are linked to each relation through outerjoins pointing to each relation.

Algorithm 1 the construction of ES

GetES

Input: original query tree $G = (V,E)$ with quantifiers $q_i = \{ q_i, \ldots, \}$

Output: a set contains the NS and ES of each join predicate

1. Loop //for each join predicate $p$
2. if $\circ p$ is inner join or antijoin
3. for each relation $r \in \text{ref}(p)$
4. $s = s + \text{set\_outerjoin}(r)$;
5. for each member of $s$
6. set the set\_outerjoin of the member to be $s$
7. if $\circ p$ is outerjoin
8. for each relation $r \in \text{ref}(\text{nullproducting}(p))$
9. $v = v + \text{set\_outerjoin}(r)$;
10. set the ES of $p$ to be $v$
11. for each relation $r \in \text{ref}(\text{preserving}(p))$
12. $u = u + \text{set\_antijoin}(r)$;
13. for each relation $r \in \text{ref}(\text{nullproducting}(p))$
14. add all the tables in $u$ to the set\_antijoin of $r$;
15. if $\circ p$ is antijoin
16. for each $r \in \text{ref}(\text{preserving}(p))$
17. $w = w + \text{set\_antijoin}(r)$;
18. set the ES of $p$ to be $w$

After initializing ES with NS for every join predicate $p$ and an set\_outerjoin and an set\_antijoin for each table in the join to include only the table itself. Algorithm 1 sets the ES of outerjoin, antijoin, inner join predicates.

Example 1

Let us now formalize this approach. As usual, $\circ$ denotes the operator. A NS includes relations referenced by the join predicate, which is used to associate each join predicate. An ES includes additional relations needed by a predicate to preserve the semantics of the original query. We denote by $\text{ref}(p)$ the set of tables referenced by the join predicate $p$. The $\text{ref}(\text{preserving}(p))$ and $\text{ref}(\text{nullproducting}(p))$ denote the set of table reference by the join predicate $p$ in the preserving side and null-producing side. An set_outerjoin contains all the relations associated by inner join or antijoin predicates. An set_antijoin includes the relations that are linked to each relation through outerjoins pointing to each relation.

Algorithm 1

GetES

Input: original query tree $G = (V,E)$ with quantifiers $q_i = \{ q_i, \ldots, \}$

Output: a set contains the NS and ES of each join predicate

1. Loop //for each join predicate $p$
2. if $\circ p$ is inner join or antijoin
3. for each relation $r \in \text{ref}(p)$
4. $s = s + \text{set\_outerjoin}(r)$;
5. for each member of $s$
6. set the set\_outerjoin of the member to be $s$
7. if $\circ p$ is outerjoin
8. for each relation $r \in \text{ref}(\text{nullproducting}(p))$
9. $v = v + \text{set\_outerjoin}(r)$;
10. set the ES of $p$ to be $v$
11. for each relation $r \in \text{ref}(\text{preserving}(p))$
12. $u = u + \text{set\_antijoin}(r)$;
13. for each relation $r \in \text{ref}(\text{nullproducting}(p))$
14. add all the tables in $u$ to the set\_antijoin of $r$;
15. if $\circ p$ is antijoin
16. for each $r \in \text{ref}(\text{preserving}(p))$
17. $w = w + \text{set\_antijoin}(r)$;
18. set the ES of $p$ to be $w$

Example 1. The algorithm 1 is illustrated using Example 1. In Example 1, the ES for predicate Prs is $\{ R, S, T \}$, which requires table $S$ be joined with table $T$ before $R$. The ES for predicate Puv includes $\{ R, T, U, V \}$ and thus Puv can only be applied at the end. The order of the two outer joins can be switched. So $((R,((S,T),U)),V)$ is valid order equivalent with original query.

2) Implementation of ES in Top-Down Algorithm

In this subsection a new algorithm will be introduced to detect whether the join reorder is rational through applying the ES to top-down join enumeration optimizer using dynamic programming. At first, we briefly introduce the process of detecting the join reorder through using the query of Example 1, as is showed in Figure 2.

Outer curve of Figure 2 denotes the process of traversal of top-down optimizer. The process applying ES is along the curve and detects the generated and adjacent two join predicates. There are three rectangular in Figure 2, namely A, B and C, and every rectangular includes two join predicates need be detected. The overall order of detecting is $A, B, C$.
Algorithm 2 the application of ES in the top-down optimizer

Call ES
Input: two adjacent join predicates $\circ p_1$ and $\circ p_2$. $\circ p_2$ is the just preceded and $\circ p_1$ is the current join predicate
Output: whether the reorder is rational
1. if the union $ES(\circ p_2)$ and $NS(\circ p_1)$ consists the set of $ES(\circ p_1)$
2. return true; //the reorder is rational
3. else return false; //the reorder is not rational

Algorithm 3 Optimistic Partitioning

Partition
Input: a connected query trivial join graph $G = (V,E)$ with quantifiers $q=q_1, q_2$...
Output: all MultiExp of $G$
1. for all $v \in V$
   2. PairQueue $+=\{v\}$
   3. for each $s_i \in$ PairQueue
      4. $S_2+2\in CmpSub(s_1)$
      5. for each $s_2 \in S_2$
         6. return $(s_1,s_2)$ and $(s_2, s_1)$

MinOptimistic
Input: disjoint sets $S, T \subseteq V$
Output: minimum cuts extended from $S$
1. for $v \in (N(S) \setminus T), v \neq \emptyset$ // $N(v) = V \setminus \{t\}$
2. do $S' + S \cup \{v\}$
3. return $(S')$
4. $\text{MinCutOptimistic}(S', T')$

CmpSub
Input: a connected query trivial join graph $G = (V,E)$ with quantifiers $q=q_1, q_2$... and a connected subset $S_1$
Output: all connected subset $S_2$ supplementing $S_1$
1. $X = \text{CmpSub}(s_1)$ union $S_1$
2. $N(\{s_1\}) \times X$
3. for all $vi \in N$ by descending $i$
4. return $(vi)$
5. $\text{MinOptimistic}(\{vi\}, X \cup N)$

This section shows how to produce the non-empty connected compare pairs based on graph-traversal driven. We will use the non-empty connected join pairs to replace the logical-logical transformation through traditional transformational rule.

Let us start the exposition by fixing some notations. For a node $v \in V$ define the neighborhood $N(v)$ of $v$ as $N(v):=\{v' \mid (v, v') \in E\}$. For a connected subset $S \subseteq V$ of $V$ we define the neighborhood of $S$ as $N(S):=\cup v \in S N(v) \subseteq S$. The neighborhood of a set of nodes thus consists of all nodes reachable by a single edge. The complement of $S$ is a connected subgraph comprised by $\overline{V \setminus S}$.

The following statement gives a hint on how to construct the connected subsets. Let $S$ be a connected subset of an undirected graph $G$ and $S'$ be any subset of $N(S)$. Then $S \cup S'$ is connected. As a consequence, a connected subset can be enlarged by adding any subset of its neighborhood using a breadth-first. Algorithm 3 provides a skeleton framework generating all connected subsets and the complement subsets.

For every element $\{v\}$ of $V$ (line 1), $\text{Partition}$ expands $\{v\}$ by calling a routine $\text{MinOptimistic}$ that extends a given connected set to bigger connected set (line 2). Then it iteratively gets subgraph $S_1$ produced by the $\text{MinOptimistic}$, and generates the complement subgraph of $S_2$ by calling $\text{CmpSub}$(line 4). $\text{CmpSub}$ considers all neighbors of $S_1$ generated by $\text{MinOptimistic}$. First, they are used to determine those $S_2$ that contain only a single node. Then, for each neighborhood of $S_1$(line 3), it recursively calls $\text{MinOptimistic}$ to create those $S_2$ that more than a single node (line 5). $\text{MinOptimistic}$ is an iteration function and mainly expands the node $S$ by calculating the neighborhood $N(S)$. It returns the connected subset $S$ and $S \cup \{v\}$ (line 2) and recursively calls $\text{MinCutOptimistic}$.

Example 2. The Algorithm 3 is illustrated using Figure 4. This table of Figure 4, $S$ and $T$ are the arguments of $\text{MinOptimistic}$. The column Out($S_1$) contains the connected subset. Return$S_1$ is the Output of $\text{MinOptimistic}$ which then becomes the argument of the recursive call to $\text{MinOptimistic}$ (labeled by $\rightarrow$) and at the same time the input of $\text{CmpSub}$. Return (S2) contains the connected subset complementing the return (S1) and is the argument of the recursive call to $\text{MinOptimistic}$. Thick lines mark the recursive call to $\text{MinOptimistic}$ from $\text{CmpSub}$(labeled by $\rightarrow$).
In order to support non-join, we need to partition independently without any dependencies among threads, and then process each sub-problem depend only on space evenly among threads, and then process each down join enumeration, we need to partition the search result of all preceding levels. By partitioning sub-

The next subsection 5.1 discusses the allocating of the search space among multi-threads. Subsection 5.2 shows the processing of each threads. Subsection 5.3 discusses the concrete realization algorithm of join predicate and relation of single thread. 

A. Multiple Plan Join

In the join enumeration based on dynamic programming, each, each sub-problem depends only on the results of all preceding levels. By partitioning sub-problems by their sizes, more precisely, or the sizes of the resulting quantifier sets, sub-problems of the same resulting size are mutually independent. Taking advantage of this property, we reorder the join pairs by the size of the quantifier set. This is illustrated using the following example.

Example 3. Figure 5 contains all reordered join pairs and codes of them generated by Figure 4. The join pairs are grouped by the size of the resulting quantifier set.

To parallelize the top-down enumeration, we need to assign the Group[n] (n is the total number of relations) for the threads. 

Algorithm 4 allocating the search space among multithread

ParallelTD

Input: a connected query trivial join graph G = (V,E) with quantifiers q={q0,…,qn} and Lower Bound B

Output: optimal plan with cost not exceeding B, or null if no such plan exists

1 GetES();
2 Group←Reorder(Partition(G));
3 SSM ←AllocateSearch([n],m); /*SSM: search space allocated to m threads*/
4 for i=1 to m  // m thread parallel implement Group[n]
5 pool.SubmitJob (TD_Thread(SSM[i], B, ThreadMemoi));
6 pool.Sync();
7 MergeAndPrunePlans(MEMO, {ThreadMemo1,…,ThreadMemo_m});
8 Return MEMO[q1,…,qn];

Algorithm 4 outlines our parallelized top-down join enumeration. The ParallelTD first construct the ES by calling GetES at line 1. Then the function of reorder groups the join pairs generated by traveling the join graph at line 2. At the same time, the join pairs belong to the Group[n] are allocated among the multi-threads at line 3. Then the m threads parallel execute the join pairs of Group[n] (line 5). MergeAndPrunePlans function (line 7) selects the Optimal plan with cost not exceeding B or null if no such plan existing.

B. Single Thread Implemental Algorithm

Each thread performs the join pairs by the function of TD_Thread of Algorithm 5.
### Algorithm 5: single thread implemental algorithm

**TD_Thread**

**Input:** logical join pair expression MultiExp, Lower Bound B and local thread ThreadMemo

**Output:** optimal plan with cost not exceeding B, or null.

1. ThreadMemo ← null;
2. for each expression of MultiExp
3.  $\lor$ p = the join predicate of GL and GR
4. for each MultiExp in (GL, GR)
5.  Memo_operatorL ← BestPlan(GL, B)
6.  Memo_operatorR ← BestPlan(GR, B)
7.  if Memo_operatorL.Memo <> null and Memo_operatorR.Memo <> null and CallIES(p, pl=relation and pr=relation) and CallIES(Memo_operatorR.operator, $\lor$ p)
     then BestPlan_Join ← {Memo[V], operator }
     and CallIES(Memo_operatorR.operator, $\lor$ p)
8.  then BestPlan ← Memo_operatorL.Memo $\lor$ p Memo_operatorR.Memo
9.  ThreadMemo[ ] ← BestPlan
10. Return ThreadMemo;

**BestPlan_G(B)**

**Input:** join graph G = (V, E), cost budget B

**Output:** optimal plan with cost not exceeding B, or null if no such plan exists

1. BestPlan ← null
2. if Memo[ ] contains plan of G with cost $\leq$ B
3. then BestPlan ← Memo[ ]
4. else if Memo[ ] is empty or contains lower bound < B
5. then if |V| = 1
6. then BestScan_operator ← BestPlan_SCAN (G, B)
7. operator ← BestScan_operator.operator;
8. BestPlan ← BestScan_operator.BestPlan;
9. else BestJoin_operator ← BestPlan_Join(G, B)
10. operator ← BestJoin_operator.operator;
11. BestPlan ← BestJoin_operator.BestPlan;
12. if BestJoin_operator.BestPlan = null
13. then Memo[ ] ← lower bound B
14. else Memo[ ] ← BestPlan
15. operator ← BestJoin_operator.operator;
16. Memo_operator ← {Memo[ ], operator }
17. return Memo_operator

**TD_Thread** iteratively invokes the logical join pair expression allocated the single thread at line 2. Then for every part of the logical expression, the BestPlan is called at line 5 and line 6 in order to get the optimal plan of left and right subtree of logical join pair. If every part of the logical join pair exist the optimal query plan, the reorder rationality’s detection of current join predicate with the root operator of left partition and right partition is invoked at line 7.

**BestPlan** function firstly checks the content of Memo. If the cost of G is less then upper bound and there is a optimal plan of G, the algorithm return the BestPlan (line 3). Otherwise, if the Memo is empty or contains lower bound is less then upper bound (line 4), BestPlan_SCAN(line 6) and BestPlan_Join(line 9) will be called in accordance to the number of nodes in join graph. BestPlan_SCAN and BestPlan_Join have two return values, respectively the join predicate and the query optimal plan. The join predicate of BestPlan_SCAN is a fixed identity of “relation”. However the join predicate of BestPlan_Join is the root join operator of the current query graph. The concrete realization of BestPlan_SCAN and BestPlan_Join will be introduced in the next subsection.

### Algorithm 6: join predicate and single relation algorithm

**BestPlan_SCAN(G, B)**

**Input:** trivial join graph G = (|R|), cost budget B

**Output:** optimal plan satisfying B, or null and the operator identity “relation”

1. BestPlan ← null Let Cost(null) = $\infty$
2. for each operator Scan(R)
3. do CurrPlan ← Scan(R)
4. if Cost(CurrPlan) $\leq$ Cost(BestPlan) and Cost(CurrPlan) $\leq$ B
5. then BestPlan ← CurrPlan
6. operator ← relation; BestScan_operator ← { BestPlan , operator }
7. return BestScan_operator

**BestPlan_Join(G, B)**

**Input:** non-trivial join graph G = (V, E), cost budget B

**Output:** optimal join plan satisfying B and B, or null

1. BestPlan ← null Let Cost(null) = $\infty$
2. for each partition (GL, GR) in MatchSearch(G)
3. do for each operator GL $\lor$ p GR
4. do C $\lor$ p ← cost of operator $\lor$ p
5. B $\lor$ p ← Min(B, Cost(BestPlan) $\lor$ $\lor$ p)
6. Memo_operator ← BestPlan(GL, B’)
7. P_L ← Memo_operator. Memo[ ];
8. $\lor$ p = Memo_operator/operator;
9. if P_L $\lor$ p = null
10. then B’ ← B’ $\lor$ Cost(P_L)
11. Memo_operator ← BestPlan(GR, B’)
12. P_R ← Memo_operator. Memo[ ];
13. $\lor$ p = Memo_operator/operator;
14. if P_R $\lor$ p = null
15. then B’ ← B’ $\lor$ Cost(P_R)
16. if $\lor$ p = relation and $\lor$ p = relation
17. and CallIES( $\lor$ p, $\lor$ p) is true
18. or $\lor$ p = relation and $\lor$ p = relation
19. and CallIES( $\lor$ p, $\lor$ p) is true
20. or $\lor$ p = relation and $\lor$ p = relation
21. and CallIES( $\lor$ p, $\lor$ p) is true
22. then BestPlan ← P_L $\lor$ p P_R
23. operator = $\lor$ p; BestJoin_operator = {BestPlan, operator}
24. return BestJoin_operator

C. Concrete Realization of Join Predicate and Relation

In this subsection we will focus on introducing the realization of BestPlan_Join and BestPlan_Join.

**BestPlan_Join** of Algorithm 6 mainly calculates the root cost (line 4) and achieves graph’s partition from top to down (line 2). It is nontrivial to define the cost of every logic join pair MultiExp. A MultiExp’s root operator has a cost, but its inputs are groups, not expressions, and it is not clear how to calculate the cost of a group. By recursively searching input groups (line 6, 10), the cost of a MultiExp is thus calculated.

**MatchSearch** acquire the logical join pairs of the Group of figure 10 according by the number of the
quantifier set. This is illustrated using the following example.

Example 4. The R0R1R2 in Group [4] can be logical expressed via traveling every row in Group [3] and estimates whether the LVA ∪ RVA equals the {0, 1, 2}. The collection satisfying the condition is considered as the logical expressing of R0R1R2.

This method makes the Top-Down dynamic programming not relying on transformation rule of traditional. It is optimal with respect to the join graph and avoids the Cartesian products which can extremely decreasing the search space.

The line from 14 to 16 of Algorithm 6 uses the CallES to detect the rationality of join predicate reorder. If each partition of logic join pair is single relation, the BestPlan can be acquired directly by the return BestPlan of each partition of logic join pair at line 13. If left partition of logic join pair is single relation and right is not, the join predicate reorder rationality of current operator and the root operator of right partition will firstly be detected at line 14. Line 15 is a similar although inverse manner with line 14. When each partition of logic join pair is not single relation, the reorder rationality’s detection of current join predicate with the root operator of left partition and right partition is invoked at line 16.

BestPlan_Scan calculates the cost of single relation by recursively accessing the operator of relation. It first generates different query optimal plans for accessing a single table. Types of table access query optimal plans include a simple sequential scan, index scan, list prefetch, index ORing, and index ANDing. The function of BestPlan_Scan will prune any plan QEP1 if there is another plan QEP2 such that cost (QEP1)>cost (QEP2), the value of cost (QEP2) and whose properties (e.g. tables accessed, predicates applied, ordering of rows, partitioning, etc.) subsume those of QEP1. By this way the bestplan of the single table will be selected.

D. Correctness Proof of Algorithm

Lemma 5.1 Algorithm MincutOptimistic terminates if G is a finite graph.

Proof MincutOptimistic is called with two arguments, S and T. As G=(V, E) is a finite graph and S ⊆ V ∩ T ⊆ V, S and T are also finite. In each recursion, MincutOptimistic considers the neighbors v ⊆ V of S, ignoring vertices in T. It then evaluates each non-empty subset of v, calling MincutOptimistic recursion enlarging T by N. As T ⊆ V and T is enlarged by each call, the recursion depth of MincutOptimistic is limited by |V|.

Lemma 5.2 Algorithm 3 terminates if G is a finite graph.

Proof Algorithm 3 performs a finite number of loop iterations (N(s) < |V|). After each iteration, it constructs a finite set and passes it as argument to MincutOptimistic. Thus CmpSub terminates if MincutOptimistic terminates for all inputs. MincutOptimistic terminates as shown in Lemma 5.1.

Lemma 5.3 The optimal query plan resulted from Algorithm 4 is (a) the plan of valid join orders (b) with no Cartesian.

Proof (a) There are two type of operator in the CallES, current and the just preceded operator. The valid join order’s test of query tree generated by top-down join enumeration is composed by the order test of current and the just preceded operator. The query tree is composed by three type subtrees divided according to the number of operators.

So the proof of whether the optimal query plan is valid join orders can be divided three cases.

Case 1: the subtree consisting only one operator
In this case, there is no the just preceded operator. Therefore does not require the join order’s test. Algorithm 6 (line 13) considers this case.

Case 2: the subtree consisting two operators
In this case, the current node is the root operator of subtree and the just preceded operators is the root operator of left subtree and right subtree.

So the test of the join order between the root operator of subtree and the just preceded operator is needed. Algorithm 6 (line 14, 15) considers this case.

Case 3: the subtree consisting three operators
In this case, the current node is the root operator of subtree and the just preceded operator is the root operator of left subtree. The root operator of right subtree is the current node relatively to the root operator of left subtree, the the just preceded operator relatively to the root operator of subtree.

So the test of the join order between the root operator of subtree and the just preceded operator is needed. Algorithm 6 (line 16) considers this case.

Through the discussed above, we know every case is considered in algorithm 6. Moreover the query tree is composed by three type subtrees. So the optimal query plan resulted from Algorithm 4 is the plan of valid join orders.

Proof (b) in algorithm 4, every logical expression is consisted two parts, i.e. GL and GR, resulted from algorithm 3. And GR is produced by enlarge the neighbors v ⊆ V of GL (line 4 of Partition), so GL and GR is connected. So every logical expression is connected. We can concluded the optimal query plan resulted from Algorithm 4 is the plan of valid join orders without Cartesian.

VI. PERFORMANCE ANALYSIS

The goals of out experiments are to show that our algorithms significantly outperform the method of PDPSva, DPEGeneric.

All the experiments were performed on a Windows Vista PC with two Intel Xeon Quad Core E540 1.6GHz CPUs(=8 cores) and 8GB of physical memory. Each CPU has two 4Mbyte L2 caches, each of which is shared by two cores. The experimental parameters and their values are illustrated by Table I.
Table I. Experimental Parameters and Their Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Enumeration Style</th>
<th>Join pair enumerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-down</td>
<td>Parallel Connected-subgraph Pairs</td>
<td>PTDhyp</td>
</tr>
<tr>
<td>Bottom-up</td>
<td>Parallel Graph_Driven Parallel Size-Driven</td>
<td>DPEGeneric PDPsva</td>
</tr>
</tbody>
</table>

We ran several experiments to evaluate the different algorithms under different settings. Due to lack of space, we selected the two typical experiments.

Figure 6 shows experimental results for varying the number of threads for chain and cycle queries. *PTDhyp* achieves linear speedup for the chain queries as the number of threads increases, consistently outperforming *PDPsva* and *DPEGeneric*. For cycle queries, the overall trend is similar.

Figure 7 show the relative performance results for varying the number of quantifiers for star and clique queries. As the optimization time varies greatly with the query size, all performance numbers are given relative to *PTDhyp*, e.g. the optimization time of *PTDhyp* is always 3.

Again, *PTDhyp* is superior to *DPEGeneric* from chain queries. As the query size increases, *PDPsva* becomes suboptimal because for every enumerated join operator, they incur a great deal of cost testing connectivity on number of Cartesian products. Cycle queries show similar experiments.

This shows that our parallel algorithm is better than the state of the art parallel optimizer tailored to size-driven *PDPsva* and *DPEGeneric*.

**VII. SUMMARY**

By exploiting the public index structure of the reorder join pairs, we first proposed a novel framework for parallelizing join enumeration of top-down dynamic programming with respect to the query graph in the multi-core processor architecture. This improves extremely conventional top-down join enumeration performance. Especially, the integration of non-join method makes the top-down optimizer more practical. Experimental results also showed that our algorithm is much more robust than the *DPEGeneric* and *PDPsva* algorithm.

The future work includes exploiting the parallelism of branch-and-bound pruning, because the problem of a shared cost budget B among multi-thread is failed to be resolved. Overall, we believe the proposed solutions provide comprehensive insight and a substantial framework for future research.

**ACKNOWLEDGMENT**

The research is under the support of the Specialized Research Foundation for the Doctoral Program of Higher Education of China under Grant No. 200801830021; Nature Science Foundation of China under Grant No. 60973040, 60903098; The Science and Technology Development Program of Jilin Province of China under...
REFERENCES


