Multi-Weighted Tree Based Query Optimization Method for Parallel Relational Database Systems[1]

Jianzhong Li, Zhipeng Cai and Shuoying Chen
Department of Computer Science and Engineering
Harbin Institute of Technology, P.R.China
lijz@banner.hl.cninfo.net

Abstract

A multi-weighted tree based query optimization method for parallel relational databases is proposed in this paper. The method consists of a multi-weighted tree based parallel query plan model, a cost model for parallel qury plans and a query optimizer. The parallel query plan model models three types of parallelism of query execution, processor and memory allocation to operations, memory allocation to buffers in pipelines and data redistribution among processors. The cost model takes the waiting time of operations in pipelining execution into consideration and is computable in a bottom-up fashion. The query optimizer addresses the query optimization problem in the context of Select-Project-Join queries. Heuristics for determining the processor allocation to operations and the memory allocation to operations and buffers in pipelines are derived and used in the query optimizer. In addition, the query optimizer considers multiple join algorithms, and can make an optimal choice of join algorithm for each join operation in a query.

1. Introduction

In recent years, multiprocessor–based parallel database systems have drawn considerable attention due to high performance, high availability and scalability for data intensive applications. A key to the success of parallel database systems is parallel query optimization. During the last years much attention has been paid to the development of parallel query optimization techniques. Various methods have been proposed[1-26]. While considerable progress has been made in parallel query optimization, many problems still remain open. Eech of the existing paralllel query optimization methods only investigated one or a few aspects of the parallel query optimization. All of them have the following restrictions:

A. The models of parallel plans, such as right-deep, left-deep, bushy, segmented right-deep and zizag tree, are not enough to model the processor and memory allocation to operations, memory allocation to buffers between operations in pipelines and data redistribution among processors.

B. The cost models of query plans did not take the waiting time of the operations in pipelining execution into consideration so that the quality of query plans chosen as the best plans by optimizers can not be guaranteed.

C. Almost all the methods addresse optimization problems only in the context of multi-way join queries. Few methods considered the optimization problems of complex queries, such as selection-projection-join (SPJ for short) queries.

D. Almost all the methods are based one parallel join algorithm only and most of the methods dealed with only paralle hash join algorithm so that no choice of the optimal algorithm can be made for operations in a query. The choice of the optimal algorithms for operations in a query has significant effect on the performance of the query processing.

E. Most methods are not aware of memory resources in order to generate good-quality plans, especially the memory allocation to the buffers between operations in pipelining execution.

To counter the five restrictions above, a multi-weighted tree based query optimization method is proposed in this

[1] This work is supported by the Natinal Natural Science Foundation of China and 973 Plan of China.

paper. The method consists of a multi-weighted tree based parallel query plan model, a cost model for parallel qury plans and a query optimizer. The parallel query plan model is the first one to model all basic relational operations, all three types of parallelism of query execution, processor and memory allocation to operations, memory allocation to the buffers between operations in pipelining execution and data redistribution among processors. The query plan cost model takes the waiting time of the operations in pipelining execution into consideration and is computable in bottom-up fashion. A mathematics model of waiting time of the operations in pipelining execution is proposed. The query optimizer addresses the query optimization problem in the context of SPJ queries that are widely used in commercial DBMSs. Several heuristics for determining the processor allocation to operations are derived and used in the query optimizer. The query optimizer is aware of memory resources in generating good-quality plans. It includes the heuristics for determining the memory allocation to operations and buffers between operations in pipelining execution so that the memory resourse is fully exploit. Multiple join algorithms are consided in the query optimizer also. The query optimizer can make an optimal choice of join algorithm for each join operation in a query.

The rest of the paper is organized as follows. Section 2 describes the parallel query plan representation model. Section 3 presents the cost model for parallel query plans. Section 4 illustrates the parallel query optimizer. Section 5 concludes the paper and discusses the future research.

2. Parallel query plan model

Let R be a relation and ψ be a proposition expression with the attributes of R as variables. For any $t\in R$, $\psi(t)$ is a proposition produced by substituting the variables in ψ with the attribute values of t. Let π be an subset of attributes of R, $\pi(t)$ be a tuple with attributes set π whose values on π are the same with t, and $\theta\in\{=, \neq, <, >, \geq, \leq\}$. Follows are the definition of the basic operators in the query plan model proposed in this paper.

Scan(R, ψ, π) : compute $\{ t \mid \exists r\ (r\in R \text{ and } \psi(r)=true \text{ and } \pi(r)=t) \}$.

Sort(R, π): sort R according to the order of the values on the attributes specified by π.

Merge(R,S,π,θ): compute $\{r \mid r\in R, s\in S, \text{and } \pi(r)\theta\ \pi(s)\}$.

Nested_Loop(R, S, π, θ): use parallel nested-loops-join algorithm to compute the join of R and S[27].

Build-Hash-Table(R, π): build the hash table of R by applying a hash function to the values on the attributes specified by π[27].

Probe-Hash-Table(T, S, π): probe the hash table T using the tuples of S by the same hash function used in building T to perform a join of T and S[27].

CMD-Join(R, S): join two relations R and S that have been declustered by the CMD method[35].

B$^+$-Tree-Join(R, S): join two relations R and S one of which has a parallel B^+-tree index[34].

Let A be the set of join attributes of relations R and S. The algorithms for implemting the relational operations can be easily expreesed by the basic operation above. The selection, projection operations or their combination can be expressed by the Scan operator. The parallel sort-merge join algorithm, parallel nested-loops join algorithm, parallel hash join algorithm, parallel CMD-join algorithm and parallel B$^+$-tree-join algorithm can be expressed by Merge(Sort(R, π), Sort(S, π), π, θ), Nested-Loop(R, S, π, θ), Probe-Hash-Table(Build-Hash-Table(R, π), S, π), CMD-Join(R, S) and B$^+$-Tree-Join(R, S) respectively.

Definition 2.1. Let $G = (V, E)$ be a tree, $E'\subseteq E$, $V'\subseteq V$, and S_1, ..., S_m, T_1, ..., and T_k be sets. A *multi-weighted tree* is a system $ET = (G, NF, EF)$, where NF is a function $V'\rightarrow(S_1\times...\times S_m)$, and EF is a function $E'\rightarrow(T_1\times...\times T_k)$.

Definition 2.2. Let Q be a query, N be the number of processing nodes, M be the memory size in bytes, OP be the set of basic operators in Q, and φ be the set of operand relations in Q. A *parallel query plan* of Q is a multi-weighted tree, $((V, E), F_1, F_2)$, where $V=OP\cup\varphi$, $E=\{(X, Y) \mid (X, Y\in OP) \text{ or } (Y\in OP \text{ and } X\in\varphi)\}$, F_1 is the function $OP\rightarrow\{0, 1, ..., N\}\times\{0, 1, ..., M\}$, F_2 is the function $E-\{(X, Y) \mid (Y\in OP \text{ and } X\in\varphi)\}\rightarrow\{0, 1, ..., M\}\times\{P, S\}$.

It is obvious that the allocation of processing nodes and memory to operators is modeled by function F_1. F_1 also models the intra-operation parallelism by assigning multi-processing nodes to an operator. The pipelining parallelism is modeled by function F_2. If F_2 puts weight (m, P) on edge (op_1, op_2) then the operators op_1 and op_2 are pipelining-parallel executable and the buffer size assigned to the pipeline is m. Otherwise, the weight (0, S) on edge (op_1, op_2) forces op_1 and op_2 to be executed sequentially. From the definition of the edge set E, all independent operators must be on different paths of the parallel query plan tree, which models the inter-operator parallelism.

3. Cost model of parallel query plans

The cost model of a multi-weighted tree based parallel query plans consists of two parts. One is the response time of individual operations in the plan. The other is to combine the response time of individual operations to obtain the response time of the entire plan. The response time of individual operations is a function $rtime(op, A, n, m, \rho, \theta)$, where op is operation, A is the algorithm to implement op, n is number of processing nodes assgned to op, m is the memory size assigned to op, ρ is the set of sizes of the operand relations, and θ is the size of the result. A lot of research work has been done to model the response time of individual operations. We concentrate on the cost model of entire parallel query plans in this paper.

3.1. Waiting time of operators

Let op_y and op_x be two operations in pipelining execution, op_y be the producer, and op_x be the consumer. By the defination of the query plan, op_x must be the parent of op_y and there is a pipelining buffer which can hold m tuples. op_y sends tuples to the buffer and op_x retrieves tuples from the buffer. The buffer may overflow or underflow. In case of underflow, op_x has to wait. In case of overflow, op_y has to wait. Let λ be the average rate at which op_y produces a tuple, μ be the average rate at which op_x consumes a tuple, and *output*(op_y) be the set of tuples produced by op_y and consumed by op_x. Since the function of each relational operation is to scan its input relations and generate new relation, it is reasonable to assume that

$$\mu=|\text{output}(op_y)|/\text{rtime}(op_x, A_x, n_x, m_x, \rho_x, \theta_x),$$
$$\lambda=|\text{output}(op_y)|/\text{rtime}(op_y, A_y, n_y, m_y, \rho_y, \theta_y).$$

Theorem 3.1. The average waiting time of op_X and op_y, WT^c_{xy} and WT^p_{xy}, are

$$WT^c_{xy}=\begin{cases}1/\lambda & \lambda \ge \mu\\ 1/\lambda+(|\text{output}(op_y)|-1)(1/\lambda-1/\mu) & \lambda<\mu\end{cases}$$

$$WT^p_{xy}=\begin{cases}0 & \lambda \le \mu\\ (|\text{output}(op_y)|-\lambda m/(\lambda-\mu))(1/\mu-1/\lambda) & \lambda>\mu\end{cases}.$$

Due to the length limitation of the paper, the proof of theorem 3.1 is omited. Let WT_X be the waiting time of an operator, op_x, in a parallel query plan tree. We define the waiting time of WT_X in five cases using theorem 3.1.

Case 1. op_x is the root and has one son op_y and w is the weight on edge (op_y, op_x). In the case, WT_X is

$$\begin{cases}WT^c_{xy} & \text{if } w=(m, P)\\ 0 & \text{if } w=(0, S)\end{cases}.$$

Case 2. op_x is the root and has two sons op_y and op_z, and w_1 and w_2 are the weights on edges (op_y, op_x) and (op_z, op_x), m_1 and m_2 are the sizes of the buffers on edges (op_y, op_x) and (op_z, op_x). In the case, WT_X is

$$\begin{cases}0 & \text{if } w_1=w_2=(0,S)\\ WT^c_{xy} & \text{if } w_1=(m_1,P),\ w_2=(0,S)\\ WT^c_{xz} & \text{if } w_1=(0,S),\ w_2=(m_2,P)\\ WT & \text{if } w_1=(m_1,P),\ w_2=(m_2,P)\end{cases}.$$

Since $\max\{WT^c_{xy}, WT^c_{xz}\} \le WT_X \le WT^c_{xy}+WT^c_{xz}$, we use $WT=(\max\{WT^c_{xy}, WT^c_{xz}\}+(WT^c_{xy}+WT^c_{xz}))/2$ to estimate WT_X.

Case 3. op_x is an inner node and has a parent op_y and a son op_z, and w_1 and w_2 are the weights on edges (op_x, op_y) and (op_z, op_x). In this case, WT_X is

$$\begin{cases}0 & \text{if } w_1=w_2=(0,S)\\ WT^p_{yx}, & \text{if } w_1=(m_1,p), w_2=(0,S)\\ WT^c_{xz}, & \text{if } w_1=(0,s), w_2=(m_2,P)\\ (\max\{WT^p_{yx}, WT^c_{xz}\}+(WT^p_{yx}+WT^c_{xz}))/2, & \text{if } w_1=(m_1,p), w_2=(m_2,P)\end{cases}$$

Case 4. op_x is inner node and has parent op_y and sons op_z and op_w, and w_1, w_2 and w_3 are the weights on edges (op_x, op_y), (op_z, op_x) and (op_w, op_x). In this case, WT_X is

$$\begin{cases}0, & \text{if } w_1=w_2=w_3=(0,S)\\ WT^c_{xz}, & \text{if } w_1=(0,s), w_2=(m_2,p), w_3=(0,S)\\ WT^c_{xw}, & \text{if } w_1=(0,s), w_2=(0,s), w_3=(m_3,P)\\ WT_1, & \text{if } w_1=(0,s), w_2=(m_2,p), w_3=(m_3,P)\\ WT^p_{yx}, & \text{if } w_1=(m_1,p), w_2=(0,s), w_3=(0,S)\\ WT_2, & \text{if } w_1=(m_1,p), w_2=(m_2,p), w_3=(0,S)\\ WT_3, & \text{if } w_1=(m_1,p), w_2=(0,s), w_3=(m_3,P)\\ WT_4, & \text{if } w_1=(m_1,p), w_2=(m_2,p), w_3=(m_3,P)\end{cases}$$

where, $WT_1=(\max\{WT^c_{xz}, WT^c_{xw}\}+(WT^c_{xz}+WT^c_{xw}))/2$, $WT_2=(\max\{WT^p_{yx}, WT^c_{xz}\}+(WT^p_{yx}+WT^c_{xz}))/2$, $WT_3=(\max\{WT^p_{yx}, WT^c_{xw}\}+(WT^p_{yx}+WT^c_{xw}))/2$, and $WT_4=(\max\{WT^p_{yx}, WT^c_{xz}, WT^c_{xw}\}+(WT^p_{yx}+WT^c_{xz}+WT^c_{xw}))/2$.

Case 5. op_X is a leaf node and has a parent op_y, and w is the weight on edge (op_X, op_y). In this case, WT_X is

$$\begin{cases}WT^p_{yx} & \text{if } w=(m, P)\\ 0 & \text{if } w=(0, S)\end{cases}.$$

It should be noted that WT_X=0 if op_X is not executed in pipelining manner with any operation.

3.2. Cost model for entire query plans

In the following discussion, let P be a parallel query plan, $\alpha=\{a_1, ..., a_k\}$, $\eta=\{n_1, ..., n_k\}$, $\sigma=\{m_1, ..., m_k\}$, $\rho=\{s_1, ..., s_k\}$, and $\theta=\{o_1, ..., o_k\}$, where, a_i, n_i and m_i are the algorithm, number of processing nodes and memory size assigned to the i^{th} operator in P, s_i and o_i are the set of sizes of the operand relations and the size of the result of the i^{th} operator in P.

Definition 3.1. Let op be a operator in P. The elapse time of op is the elapse time from the beginning of the execution of P to the termination of the execution of op, denoted by *etime*(op).

Definition 3.2. The respose time of a parallel query plan P is the elapse time from the time of P being started to the termination time of the execution of the root operator in P.

Now, we give a recursive algorithm to compute the response time of P. For each node directly connected to relation nodes (leaf nodes) op_x in P, *etime*(op_x)=*rtime*(op_x, a_x, n_x, m_x, s_x, o_x)+WT_x. For each other operator op_x in P, six cases need to be considered.

Case 1. op_x has a son op_y, and the weight on edge (op_y, op_x) is (0, S), i.e., op_x and op_y are sequentially executed. In this case, *etime*(op_x)=*rtime*(op_x, a_x, n_x, m_x, s_x, o_x) + *etime*(op_y) + WT_X.

Case 2. op_x has a son op_y, and the weight on edge (op_y, op_x) is (m, P), i.e., op_x and op_y are executed in pipelining manner. In this case, $etime(op_x)$ = ($etime(op_y)$-$rtime(op_y)$-WT_y) + WT_x + $rtime(op_x, a_x, n_x, m_x, s_x, o_x)$, where ($etime(op_y)$ - $rtime(op_y)$ - WT_y) is the start time of op_x.

Case 3. op_x has two sons op_y and op_z, and the weights on edge (op_y, op_x) and (op_z, op_x) are (0, S) and (0, S), i.e., the execution of op_x and op_y is sequential, the execution of op_x and op_z is sequential, and op_y and op_z are executed in parallel. In this case, $etime(op_x)$=$rtime(op_x, a_x, n_x, m_x, s_x, o_x)$ + WT_x + $\max\{etime(op_y), etime(op_z)\}$.

Case 4. op_x has two sons op_y and op_z, and the weights on edge (op_y, op_x) and (op_z, op_x) are (m_1, P) and (m_2, P), i.e., op_x and op_y are executed in pipelining manner, op_x and op_z are executed in pipelining manner, and op_y [illegible] op_z are executed in parallel. In this case, $etime(op_x)$=$rtime(op_x, a_x, n_x, m_x, s_x, o_x)$ + WT_x + $\min\{etime(op_y) - rtime(op_y) - WT_y, etime(op_z) - rtime(op_z) - WT_z\}$.

Case 5. op_x has two sons op_y and op_z, and the weights on edge (op_y, op_x) and (op_z, op_x) are (0, S) and (m, P), i.e., op_x and op_y are executed sequentially, op_x and op_z are executed in pipelining manner, and op_y and op_z are executed in parallel. In this case, $etime(op_x)$=$rtime(op_x, a_x, n_x, m_x, s_x, o_x)$ + WT_x + $\max\{etime(op_y), etime(op_z)$-$rtime(op_z)$-$WT_z\}$.

Case 6. op_x has two sons op_y and op_z, and the weights on edge (op_y, op_x) and (op_z, op_x) are (m, P) and (0, S). This case is similar to case 5.

Finaly, the responde time of P is $etime(op_{root})$, where op_{root} is at the root of P.

4. Parallel query optimizer

The parallel query optimizer (PQO) consists of six phases. Given a query Q, phase 1 is to generates join trees of Q. Phase 2 chooses appreciate parallel join algorithms for join operations in each join tree. Phase 3 transformed the join trees with join algorithms assigned to all join operations into operator trees. Phase 4 determines the pipelines in each operator tree and the execution strategy of the pipelines. Phase 5 assigns memory and processing nodes to the operators and assigns memory to the buffers in each operator tree. Finally, phase 6 chooses optimal query plan from the set of operator trees resulted in phase 5. We present the techniques in the 6 phases in the following subsections.

4.1. Generating join trees

Join tree is a model of multi-join query. The internal nodes express join operations. The leaf nodes are relations. Given a query Q with n join operations, we first generate the join trees of Q ignoring the selection and projection operations. The selection and projection operations will be considered in the generation of operator trees. The algorithm for generating join trees uses the dynamic programming technique. The set of the join trees with one join operation, Join_tree[1], are gennerated first. Generally, after the set of join trees with k join operations, Join_tree[k], being generated from Join_tree[k-1], the set of join trees with k+1 join operations, Join_tree[k+1], are genenrated from Join_tree[k]. Finally, the set of join trees with n join operations, Join_tree[n], are generated from Join_tree[n-1]. The size of the join results are estimated while the join trees are constructed. Given the number of join operations, n, and the set of join relations, *R_SET*, the followoing **G-Join-Tree** returns all the sets, *Join_tree*[k] with k join operations for $0 \le k \le n$, and the size, *size*[t], of each join tree t.

ALGORITHM G-Join-Tree(*n*, *R_SET*)

```
(1)  Join_tree[0]:={Join_tree(0, R) | R∈R_SET };
     /* Join_tree(0, R) generatetes special join trees with
       only one join relation R */
(2) FOR  i =1  TO  n+1  DO  size[Join-tree(0, R_i)] := |R_i|;
(3) FOR  i =1  TO  n  DO  Join_tree[k] := empty set;
(4) FOR  k =1  TO  n,  i=0  TO  k-1   DO
(5)    FOR ∀(Ltree, Rtree) ∈Join_tree[i]×Join_tree[k-1-i];
(6)        IF  leaves(Ltree)∩leaves(Rtree)= empty set
(7)        THEN Join_tree[k+1]:=Join_tree[k]∪{Join-tree(Ltree, Rtree)};
(8)                 size[Join-tree(Ltree, Rtree)]:=Join-size(size[Ltree],
                                                    size[Rtree]).
```

Function *leaves*(T) gives the set of leaf nodes in tree T. Function *Join-tree*(L, R) combines join trees L and R to form a new tree of which L and R are the left and right subtrees. Function *Join-size*(x, y) computes the size of join result of relations with sizes x and y. There are seveval methods of estimating the sizes of join results[28, 29]. We do not discuss them again in the paper.

4.2. Choosing parallel join algorithms

PQO adopts a heuristic method to choose appreciate parallel join algorithm for each join operation in the join tree generated by algorithm **G-Join-Tree**. Let J be a join operation, R and S be the join relations of J, A be the join attribute set, the size of R be smaller than that of S, C_0=(one of R or S can be fit into memory), C_1=(R or S has ordered index on A), C_2=(R or S is sorted in the order of the values of A), C_3=(the operation after J needs sorted input relation), C_4=(Number of processing nodes<δ), C_5=(the *skew* of R or S on A is great than ϑ), C_6=(J is qual join), C_7= (The difference of the sizes of R and S is greater than Δ), where ϑ, δ and Δ are variable with different parallel computing systems. They need to be decided by experements. C_4 expresses the number of processing nodes is very small. C_5 expresses the data of R or S is very skew on A. Let PR be the rerult of projection of R on A, n is the size of PR, K_i is the number of tuples whose value on A is the i^{th} value in PR, $|R|$ is the size of R. The *skew* of R on A is defined as $\sum_{i=1}^{n}\left|k_i - \frac{|R|}{n}\right|$. C_7 expresses the difference of the sizes of R and S is very big. From the experimental result and theoretic analysis[30-35], we derived the following heuristic rules.

Rule 1 $C_0 \rightarrow$ (choose parallel nested_loops join algorithm)

Rule 2 (R or S has parallel B^+_tree index on A) $\rightarrow$ (choose parallel B^+_tree based join algorithm)

Rule 3 (R or S is stored in parallel CMD file) $\rightarrow$ (choose parallel join algorithmbased on CMD files).

Rule 4 $C_6 \wedge (C_1 \vee C_2 \vee C_3) \rightarrow$ (choose parallel sort_merge join algorithm).

Rule 5 $C_6 \wedge \neg(C_1 \vee C_2 \vee C_3) \wedge C_5 \wedge C_4 \rightarrow$ (choose parallel sort_merge Join algorithm).

Rule 6 $C_6 \wedge \neg(C_1 \vee C_2 \vee C_3) \wedge C_5 \wedge \neg C_4 \rightarrow$ (choose parallel nested_loops join algprithm).

Rule 7 $C_6 \wedge \neg(C_1 \vee C_2 \vee C_3) \wedge C_7 \rightarrow$ (choose parallel nested_loops join algorithm).

Rule 8 $\neg C_6 \wedge C_4 \rightarrow$ (choose parallel sort_merge join algorithm).

Rule 9 $\neg C_6 \wedge \neg C_4 \rightarrow$ (choose parallel nested_loops join algorithm).

Whenever PQO chooses an appreciate algorithm for a join operation, it first try to choose one of the algorithms of parallel sort_meger join, parallel nested_loops join, parallel join based on B^+_tree index and parallel join based on CMD files using the nine rules. If all the conditions of the rules can not be satisfied, the parallel hash join algorithm is choosen.

4.3. Transform join tree into operator tree

Let T be a join tree of a query Q, and all the join operations in T have been assigned parallel algorithms. T is transformed to an operator tree in two steps.

The first step is to insert the selection and projection operations of Q into appropriate positions of T to form a SPJ tree. Since selection and projection are expressed by scan operators, actually the first step is to insert scan operators into T. Let v and w be a pair of parent-son nodes of T, v be the parent and w be the son. The heuristic rules of inserting scan operators is as follows.

Rule 1. If w is a leaf node (relation), then insert a scan operator between v and w.

Rule 2. If a selection or projection operation need to be executed on the output of w, then insert a scan operator between v and w.

Rule 3. If v is the root of T and a selection or projection operation need to be executed on the output of v, then insert a scan operator as the new root of T and the parent of v.

Rule 4. Move the Scan operator as close as possible to the leaf nodes accoding to the algebra rule of the relational operations.

When a Scan operator is inserted into the join tree, the output size of the Scan node, *size*[Scan], can be estimated. During the execution of a query, some intermediate rerults may need to be distributed over multiple processing nodes. A new operator, Dis(R, P), which distributes relation R among P processing nodes is needed. The following rule is for inserting the Dis operators.

Rule 5. If w is an internal join operation node and the output of w need to be distributed over multiple processing nodes, then insert a Dis operation between v and w.

The second step replaces each join node in SPJ tree with a subtree of basic operators according to the algorithm assigned to the node to form a operator tree. Let join operation node J have been assigned algorithm Alg. The input, R and S, of J may be initial relations or intermediate result relations, and R is smaller than S. The rules for constructing an operator tree from a SPJ tree are as follows.

Rule 1. If Alg is paralle hash join, then replace J according to Fig.4.1 (A).

Rule 2. If Alg is sort-merge join, then replace J according to Fig. 4.1 (B).

Rule 3. If Alg is nested-loops join, then replace J according to Fig. 4.1 (C).

Rule 4. If Alg is parallel B^+-tree based join (B-Tree-Join), then replace J according to Fig. 4.1 (D).

Rule 5. If Alg is parallel CMD join (CMD-Join), then replace J according to Fig. 4.1 (E).

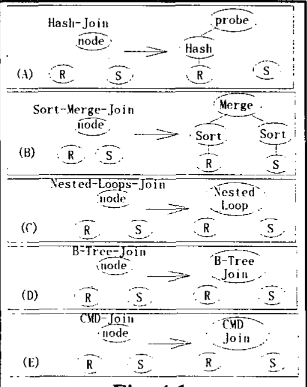

Fig. 4.1.

4.4. Determine pipelines in an operator tree

Given a operator tree T, the first task of determining pipelines in T is to compute the weight, (m, x), on each edge (w, v) of T, where v is the parent of w. We first discuss how to compute m. If v and w are executed sequentially then m=0, otherwise m is computed by the following theorem 4.1 that is derived from theorem 3.1.

Theorem 4.1. Let λ be the average rate at which w produces tuples, and μ be the average rate at which v consumes the tuples produced by w. Assume that w and v are executed in pipelining manner. If m is computed as

$$m = \begin{cases} \text{Size of one tuple} & \text{if } \lambda \leq \mu \\ (\lambda - \mu)\,|\,\text{output}(w)\,|\,/\,\lambda & \text{if } \lambda > \mu \end{cases}$$

then the average waiting times of w and v are minimized.

The second component, x, of the weight (m, x) on the edge (w, v) can be computed as follows.

If v = Scan, x=P.
If v = Sort, x=S.
If v = Hash, x=P.
If v = Probe and w = Hash, x=S.
If v = Probe and $w \neq$Hash, x=P.
If v = Merge, x=P.
If v = Nested-loop, x=P.
If w=Dis, x=S.
If v = CMD-Join, x=S.
If v = B-Tree-Join, x=S.

Definition 4.1. Let T be a operator tree with weight on each edge. A *natural pipeline* of T is a subtree, ST, of T satisfying (1) all edges have weigh of type (m, P), and (2) there is no subtree ST' in T such that ST' satisfies (1) and ST is a subtree of ST'.

It is obvious that each subtree in the forest obtained by removing the edges with weight (0, S) from T is a natural pipeline. Thus, the set of natural pipelines of T can be obtained by removing the edges with weight (0, S) from T. If the natural pipelines are regarded as nodes and the removed edges are regarded as edges, we obtain a special tree. This tree is called *natural pipeline tree*.

Due to the limitation of the memory size, all the operation on a pipeline may not be executed in parallel. If the total size of all the pipeline buffers on a pipeline is greater than the size of availble memory, the pipeline should be partitioned into multiple subpipelines such that the total size of all the buffers on each subpipeline is smaller than the size of availble memory. The resulting pipelines are called *executable pipelines*. All the operations on a executable pipeline can be executed in parallel. The algorithm to generate the *executable pipeline tree* is as follows.

Algorithm Partition
Input: natural pipeline tree T, availble memory size M.
Output: executable pipelines tree T'.

```
(1) FOR each natural pipeline $F$ in $T$ DO
(2)     IF total size of all buffers on $F$ is greater than $M$
(3)     THEN Partition $F$ into multiple subpipelines in top down manner
               so that the total size of all the buffers on each subpipeline
               is not greater than $M$;
(4)          Change partition edges into dashed lines with weight (0,S)
```

4.5. Determine strategies of executing Pipelines

Some executable pipelines in an executable pipeline tree can be executed in parallel. Give an executable pipeline tree EPT, a strategy of executing pipelines in EPT can be expressed by a gragh, called *execution strategy gragh*. A node of the gragh is a set of pipelines in EPT that can be executed in parallel, where "can be executed in parallel" means that all the pipelines contained in the node are in different pathes of EPT and the total memory required by them is smaller than the available memory. An edge (x, y) of the execution strategy gragh expresses that the pipelines in x must be executed before the pipelines in y. The algorithm to determine the execution strategy and to generate execution strategy gragh is as follows.

Input: Executable pipeline tree $EPT=(V, E)$, and size M of available memory in bytes.
Output: Execution strategy gragh $G=(V', E')$.

```
(1) FOR each $v \in V$ DO $W(v) := 0$;
(2) FOR each $v \in V$ DO
(3)     $W(v)$ := time of executing pipeline of $v$ by one processing node;
           /* $W(v)$ is called the work of $v$ */
(4) Construct gragh $G=(V', E')$ that is isomorphic with the
    longest path $P=(V'', E'')$ of $EPT$:
      A. For $\forall v \in V''$, add one and only one node $N_v$ to $V'$, $N_v$
         initially contains the pipelines in $v$, $W(N_v):=W(v)$;
      B. For $\forall(v, w) \in E''$, add one and only one edge $(N_v, N_w)$ to $E'$;
(5) \* Insert pipelines in nodes of $(V-V')$ into $V'$ of $G$ *\
    \* Assume that $EPT$ has $L$ levels and the level numbers
     from the root of $EPT$ are $L, L-1, ..., 2$ and $1$ *\
    FOR $i = 1$ TO $L$ DO
      FOR $\forall v \in (V-V')$ that has the largest work $W(v)$ and
          at level $i$ of $EPT$ DO
        A. Find a node $N_w$ in $G$ that satisfies the
           following two conditions:
             $(a)$. (the level of $N_w$ in $EPT$) $\le$ (the level of $v$ in $EPT$);
             $(b)$. $N_w$ is the node that satisfies condition $(a)$
                 and has the smallest work $W(N_w)$;
        B. $N_w := N_w \cup \{\text{pipelines in } v\}$;
           $W(N_w) := W(N_w) + W(v)$;
           remove $v$ and related edges from $EPT$;
(6) FOR each $N_v \in G$ DO
      A. If the required memory size by all pipelines in $N_v$ is greater
         than $M$, then $N_v$ is partitioned into $n$ nodes, $N_{v1}$, $N_{v2}$, ... and
         $N_{vn}$, which satisfy the following conditions:
         $(a)$. For $1 \le i \le n$, the memory size required by all the pipelines
             in $N_{vi}$ is not greater than $M$,
         $(b)$. $n$ is the minimal integer that satisfies condition $(a)$;
      B. substitute path $(N_{vp}, N_v, N_{vs})$ in $G$ by the path $(N_{vp},$
         $N_{v1}, ..., N_{vn}, N_{vs})$.
         /* $N_{vp}$ and $N_{vs}$ are the parent and son of $N_v$ */
```

4.6. Memory assignment

The method of assigning memory to pipeline buffers has been discussed in section 4.4. This section describes the algorithm, *M_Assign*, of assigning memory to pipelines and operators. Given a executable pipeline tree EPT, algorithm *M_Assign* assigns memory to the pipelines and operators in EPT based on the execution strategy gragh of EPT, the ratio of the data volumes processed by the pipelines, and the ratio of the data volumes processed by the operators. The *data volume processed by a pipeline* is the total size of all the relations and intermediate results on the pipeline. The *data volume processed by a operator* is the total size of all the input relations of the operator. Since the nodes in an execution strategy gragh are executed sequentially, we assume that each node of the execution strategy gragh has been assigned whole available memory. Algorithm *M_Assign* is as follows.

Algorithm M_Assign(*EPT, G, M, m, size*)
Input: Executable pipeline tree EPT, execution strategy gragh G, available memory size M, and array *size*, where *size*[*op*] is the size of the output of *op* in EPT which has been evaluated by algorithms in sections 4.1 and 4.3.
Output: Array *PMS*, where *PMS*[*P*] is the size of memory assigned to pipeline *P*, array *PM*, where *PM* [*P*] is the memory assigned to pipeline *P*, array *OMS*, where *OMS*[*op*] is the size of memory assigned to operator node *op*. and array *OM*, where *OM*[*op*] is the memory assigned to operator node *op*.

```
(1) FOR $\forall v \in G$ DO /* Assign memory to each pipeline in each $v \in G$ */
(2)     $Data := 0$;
```

```
(3)  FOR each pipeline $P$ in $v$ DO
(4)    $PD[P]$ := sum of the sizes of all relations on $P$;
(5)    FOR each operation $op$ on $P$ DO $PD[P] := PD[P] + size[op]$;
(6)    $Data := Data + PD[P]$;
(7)  FOR each pipeline $P$ in $v$ DO /* Assign memory to $P$ */
(8)    $PR[P] := PD[P]/Data$; $PMS[P] := \lfloor M \times PR[P] \rfloor$;
(9)    IF $PMS[P] < 1$ THEN $PMS[P] := 1$;
(10)   $PM[P]$ := location of $PMS[P]$ bytes of memory assigned to $P$;
(11) IF there is residual memory
     THEN Assign the residual memory to some pipelines so that the
          ratios, ( $PMS[P]/PD[P]$ )'s, of all the pipelines are
          as close as possible;
(12) FOR each $P$ in $v$ DO /* Assign memory to operation nodes */
(13)   $Data := 0$;
(14)   FOR each $op$ on $P$ DO
(15)     IF $op$ has two input relations $R$ and $S$
(16)     THEN $OD[op]:=size(R)+ size(S)$; /*size($X$)=size of $X$ */
(17)     ELSE $OD[op] := size(R)$; /* $op$ has one input relation $R$ */
(18)     $Data := Data + OD[op]$;
(19)   FOR each $op$ on $P$ DO    /* Assign memoery to $op$ */
(20)     $OR[op] := OD[op]/Data$;
(21)     $OMS[op] := \lfloor PMS[P] \times OR[op] \rfloor$;
(22)     IF $OMS[op] < 1$ THEN $OM[op] := 1$;
(23)     $OM[op]$ := the location of the $OMS[op]$ bytes of memory
            in $PM[P]$ assigned to $P$
(24)   IF the memory assigned to $P$ has not used up
(25)   THEN Assign the residual memory to some operator nodes so
          that the ratios, ($OMS[op]/OD[op]$)'s, of all the perator
          nodes on $P$ are as close as possible.
```

4.7. Processing node assignment

Given a executable pipeline tree *EPT*, processing node assignment algorithm, *P_Assign,* assigns processing nodes to the pipelines and operators in *EPT* based on the execution strategy gragh of *EPT*, the ratio of the costs of the pipelines, and the ratio of the costs of the operators. The *cost of an operator* is the execution time of the operator using one processing node. The *cost of a pipeline* is the total cost of all the operators on the pipeline. Since the nodes in an execution strategy gragh are executed sequentially, each node of the execution strategy gragh has been assigned all the processing nodes initially. The algorithm *P_assign* is similar to **M_Assign**.

4.8. Selecting and executing optimal query plan

Given a query *Q*, PQO uses the algorithms in sections 4.1, 4.2, 4.3, 4.4 and 4.5 to generate a set of executable pipeline trees and execution strategy graghs. For each pair of executable pipeline tree and execution strategy gragh, PQO uses the algorithms in sections 4.6 and 4.7 to assign processing nodes and memory to the operators in the executable pipeline trees. The memory assignment for pipeline buffers and the determination of the weights on the edges of the executable pipeline trees have been performed in section 4.4. Let *OM*(v) and *OPROC*(v) be the memory and processing nodes assigned to the operator node v. For each v in the executable pipeline tree, let (*OM*(v), *OPROC*(v)) be the weight of v. Finaly, PQO changes each dashed line in every excutable pipeline tree into a normal edge with weight (0, S) to obtain the set, *QPS*, of multi-weighted tree based parallel query plans of *Q*. For each plan *P* in *QPS*, PQO computes the parallel execution time of *P* using the recursive algorithm in section 3.2, chooses the plan P_0 with the minimum parallel execution time, and executes P_0 according to it's execution strategy gragh to perform the processing of *Q*.

5. Summary and Future Research

A multi-weighted tree based query optimization method is proposed in this paper. The parallel query plan model of the method is the first one to model processor and memory allocation to operations, memory allocation to the buffers between operations in pipelines and data redistribution among processors. The cost model of the method takes the waiting time of the operations in pipelining execution into consideration. The query optimizer of the method addresses the query optimization problem in the context of SPJ queries. Several heuristics determining the processor and memory allocation to operators and buffes are derived and used in the query optimizer. Multiple algorithms for implementing join operations are also consided by the query optimizer. The query optimizer can make an optimal choice of join algorithm for each join operation in a query.

In the future research, we will address the issues of heterogeneous parallel computing architectures, dynamic/ pre-emptive optimization, space-time tradeoffs, evaluation of optimization algorithms, and accuracy of the cost model.

References

[1] W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS", in Proc. of the 1st Int'l Conf. on Parallel and Distributed Information Systems, IEEE CS Press, 1991, pp..

[2] M.S. Chen, P.S.Yu, and K.L. Wu, "Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries", in Proc. of IEEE Int'l Conf. on Data Engineering, 1992, pp. 58-67.

[3] H. Lu, M.C. Shan, and K.L. Tan, "Optimization of Multi-Way Join Queries for Parallel Execution", in Proc. of Int'l Conf. on Very Large Data Bases, 1991.

[4] M.C. Murphy and M.C. Shan, "Execution Plan Balancing: A Practical Technique for Multiprocessor Query Optimization", in Proc. of Int'l Conf. on Data Engineering, 1991.

[5] M.S. Chen, et al., "Using Segmented Right-Deep Tree for the Execution of Pipelined Hash Joins", in Proc. of Int'l Conf. on Very Large Data Bases, 1992.

[6] K.A. Hua, Y.L. Lo, and H.C. Young, "Including the Load Balancing Issue in the Optimization of Multi-Way Join Queries

for Shared-Nothing Database Computers", in Proc. of the 2nd Int'l Conf. Parallel and Distributed Information Systems, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 74-83.

[7] R.S.G. Lanzelotte, P. Valduriez, and M. Zait, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces", in Proc. of Int'l Conf. on Very Large Data Bases, 1993.

[8] D. Schneider and D.J. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines", in Proc. of Int'l Conf. on Very Large Data Bases, 1990.

[9] E.J. Shekita, H.C. Young, and K.L.Tan, "Multi-Join Query Optimization for Symmetric Multi-Processors", in Proc. Of the 19th Int'l Conf. on Very Large Data Bases, Morgan Kaufman, San Mateo, Calif., 1993.

[10] M. Ziane, M. Zait and P. Borla-Salamet, "Parallel Query Processing in dbs3," in Proc. of the 2nd Int'l Conf. Parallel and Distributed Information Systems, IEEE CS Press, Los Alamitos, Calif., 1993.

[11] L. Haas, et al., "Extensible Query Processing in Starburst", in Proc. of ACM SIGMOD Int'l Conf. on Management of Data, 1989.

[12] J. Srivastava, and G. Elsesser, "Optimizing Multi-Join Queries in Parallel Relational databases", in Proc. of the 2nd Int'l Conf. Parallel and distributed Information Systems, IEEE CS Press, Los Alamitos, Calif., 1993, pp.84-92.

[13] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query Optimization for Parallel Execution", in Proc. of ACM SIGMOD Int'l Conf. on Management of Data, 1992.

[14] W. Hong, "Exploiting Inter-Operator Parallelism in XPRS", in Proc. of ACM SIGMOD Int'l Conf. on Management of Data, 1992.

[15] M.L. Lo, et al., "On Optimal Processor Allocation to Support Pipelined Hash Joins", in Proc. of ACM SIGMOD Int'l Conf. on Management of Data, 1993.

[16] N. Duppel, "Modelling and Optimization of Complex Database Queries in a Shared-Nothing System", in Proc. of 3rd IEEE Symp. Parallel and Distributed Processing, IEEE CS Press, Los Alamito, Calif., 1991.

[18] H.I. Hsiao, M.S. Chen and P.S. Yu, "Processor Allocation for Parallel Execution of Hash Joins," in Proc. of Int'l Conf. Parallel and Distributed Systems, 1993.

[19] M. Mehta, V. Soloview, and D.J. Dewitt, "Batch Scheduling in Parallel Database Systems," in Proc. of IEEE Int'l Conf. on Data Engineering, 1993.

[20] E. Rahm and R. Marek, "Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems", in Proc. of Int'l Conf. on Very Large Data Bases, 1993.

[21] K. L. Tan and H. Lu, "Pipeline Processing of Multi-Way Join Queries in Shared Memory Systems," in Proc. of the 22nd Int'l Conf. on Parallel Processing, Vol.1, 1993.

[22] K.L. Tan and H. Lu, "On Resource Scheduling of Multi-Join Queries in Parallel Database Systems", Information Processing Letters, Vol. 48, No. 4, Nov. 1993.

[23] K.A. Hua, W. Tavanapong, and H.C. Young, "A Performance Evaluation of Load Balancing for Join Operations on Multicomputer Database Systems", in Proc. of IEEE Int'l Conf. on Data Engineering, 1995.

[24] H-I. Hsiao, M-S. Chen, and P.S. Yu, "On Parallel Execution of Multiple Pipelined Hash Joins", in Proc. of ACM SIGMOD Int'l Conf. on Management of Data, 1994.

[25] A.N. Wilschut, J. Flokstra, and M.G. Apers, "Parallel Evaluation of Multi-Join Queries", in Proc. of ACM SIGMOD Int'l Conf. on Management of Data, 1995.

[26] C. Lee and Z-A. Chang, "Workload Balance and Page Access Scheduling for Parallel Joins in Shared-Nothing Systems", in Proc. of Int'l Conf. on Data Engineering, 1993.

[27] D.A. Schneider and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", in Proc. ACM SIGMOD, 1989.

[28] R.J. Lipton and J.F. Naughton, "Query Size Estimation by Adaptive Sampling", J. Computer and System Science, 51(1), 1995.

[29] J.D. Ullman, Principles of Database and Knowledge-Base Systems, Volume II, Computer Science Press, Inc., 1989.

[30] D.A. Schneider and D.J. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", in Proc. of ACM Int'l Conf. on Management Data, 1989, pp.110-121.

[31] P.V. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine" , ACM Transactions on Database Systems, Vol.9, No.1, 1984.

[32] D. DeWitt, J. Naughton, and J. Burger, "Nested Loops Revisited", in Proc. of the 2nd Parallel and Distributed Information Systems Conf., IEEE CS Press, Los Alamitos, Calif., 1993, pp.230-242.

[33] D. BittON, "Parallel Algorithms for the Execution of Relational Database Operations", ACM Trans. databae Systems, Vol.8, No.3, 1983, pp.324-353.

[34] Wunjun Sun, Jianzhong Li, and Hong Chang, "Design, Analysis and Implementation of B^+-Tree Based Parallel Join Algorithms", Computer Journal, Vol. 21, No. 1, 1998.

[35] Jianzhong Li and Wei Du, "Parallel CMD Join Algorithms on Parallel Databases", Chinese Journal of Software, Vol.9, No.4, 1998.