Adaptive Multi-Join Query Processing in PDBMS

Sai Wu *1, Quang Hieu Vu *2, Jianzhong Li †3, Kian-Lee Tan *4

1School of Computing, National University of Singapore, Singapore
2{wusai, tankl}@comp.nus.edu.sg, 3hieuv@nus.edu.sg
3Harbin Institute of Technology, Harbin, China
4lijzh@hit.edu.cn

Abstract—Traditionally, distributed databases assume that the (small) set of nodes participating in a query is known apriori, the data is well placed, and the statistics are readily available. However, these assumptions are no longer valid in a Peer-based DataBase Management System (PDBMS). As such, it is a challenge to process and optimize queries in a PDBMS. In this paper, we present our distributed solution to this problem for multi-way join queries. Our approach first processes a multi-way join query based on an initial query evaluation plan (generated using statistical data that may be obsolete or inaccurate); as the query is being processed, statistics obtained on-the-fly are used to (continuously) refine the current plan dynamically into a more effective one. We have conducted an extensive performance study which shows that our adaptive query processing strategy can reduce the network traffic significantly.

I. INTRODUCTION

In a DataBase Management System (DBMS), multi-join queries are frequently posed to retrieve information from multiple tables. While multi-join queries have been well studied in the traditional DBMS, it is still unknown who to process and optimize such queries in a Peer-based DataBase Management System (PDBMS). Existing works on PDBMS largely focused on building a unify framework [8], indexing data to process queries [9] or mapping heterogeneous schemas [15]. To our knowledge, there is no reported work that addresses the multi-join optimization problem in PDBMS.

In a DBMS, query optimization is the most challenging task [13]. Conventional centralized DBMSs solve this task by first building a query execution plan for the query from data statistics and then continuously refining the plan based on actual results while the query is being processed [7]. This solution is also employed in distributed DBMSs where centralized servers are dedicated to maintain data statistics and to coordinate distributed nodes in query execution and optimization. However, to apply this solution in PDBMSs, where centralized servers do not exist, we need to solve two main challenges:(1) how to build and optimize an initial query plan in a distributed manner; and (2) how to adapt and refine an existing query plan in the absence of centralized coordinators. In this paper, we address these challenges, and propose a distributed and adaptive scheme to process multi-join queries in PDBMS.

The rest of the paper is organized as follows. In Section II, we review related work. In Section III, we give the problem definition. We introduce our query processing solution in Section IV. We present results of our experimental study in Section V. Finally, we conclude in Section VI.

II. RELATED WORK

A. Peer-based DataBase Management System

PIAZZA [15], Hyperion [14] and PeerDB [12] can be considered as the first generation Peer-based Database Management Systems (PDBMS). These systems share the same property that they are built on unstructured P2P networks, and hence cannot support query processing efficiently. PIER [8], [9] focused on building a unified framework for PDBMS. This framework is designed on structured P2P networks where the basic ideas of data indexing and query processing were proposed. However, building and maintaining the routing index incurs high overheads. To address this problem, PISCES [17] proposed a partial indexing scheme.

B. Query Processing Strategies

Traditional DBMS such as system R [5] adopts an “optimize-then-execute” strategy. In this strategy, the system generates an optimal query plan and then processes the query based on the plan. In real applications, data distribution is often skewed and attributes in a table are usually correlated. As such, a plan that is supposedly optimal at compilation time may actually be inferior at runtime. To overcome the above problem, adaptive query processing approaches [2], [10], [11] have been proposed to refine query plans that are suboptimal at runtime. In this strategy, the system continues to optimize the initial query plan during query processing based on accurate statistics obtained from actual results at runtime.

III. PROBLEM DEFINITION

In our system, peer nodes manage their database independently. However, all databases in the system must follow the same schema or they can be mapped to a mediated schema (note that, we do not consider the schema mapping problem since it is beyond the scope of this paper). Each peer node maintains a database containing one or more relations. Different peer nodes can maintain the same relation e.g. different peer nodes owned by different companies can publish the same “product” relation. This partition is similar to the hierarchical partition in a distributed database where data in a relation can be disseminated to different nodes in the system. The only difference, however, is that in a P2P system there is no control over the data distribution and each node produces and manages its own data. This autonomy at peer nodes makes
query processing in P2P systems more challenging. Now, let $n$ denote a peer node in the system and $n.R$ denote the existence of a partition of a relation $R$ at $n$. We define a group of nodes having the same relation in the following definition.

**Definition 1: Relation Group, $G(R)$**
A relation group of a relation $R$, denoted as $G(R)$, is a set of nodes satisfying two conditions:

1. $\forall n_x \in G(R), \exists n_x.R$.
2. $\forall n_y \notin G(R), \nexists n_y.R$.

In this definition, we assume that there are no peer nodes joining or leaving the network. However, in a dynamic network, Relation Groups are changing over time and even the snapshot semantics cannot be guaranteed [3]. To cope with this challenge, we relax the requirement to adopt the single-site validity semantics defined in [3]. Based on this definition, the result of a query $q = \sigma_p(R_i \bowtie_{R_i.a=R_j,b} R_j)$ denoted as $RS(q)$ is defined as follows.

$$RS(q) = \bigcup_{n_x \in G(R_i), n_y \in G(R_j)} \sigma_p(n_x.R_i \bowtie_{R_i.a=R_j,b} n_y.R_j)$$

In this paper, we address the problem of finding a good query plan for a query defined above. For simplicity, we assume that no indexing schemes such as those in [9] exist in our system. We note that our proposed query processing method is actually orthogonal to the indexing schemes. The existence of indices only provides a potentially better query plan. Their existence does not affect the correctness of our query processing scheme.

**IV. ADAPTIVE MULTI-JOIN PROCESSING IN P2P NETWORK**

In this section, we examine multi-join optimization and processing. We focus only on equi-join throughout this paper.

**A. Generating an Initial Query Plan**

When a node issues a query $q = R_1 \bowtie R_2 \cdots \bowtie R_n$, the query processor first retrieves histograms of $R_1, R_2, \cdots, R_n$ from the nodes’ local DBMS. Recall that there may exist several partitions of a relation $R_i$ at different nodes and we denote the existence of a partition of $R_i$ at $n_x$ as $n_x.R_i$. Based on retrieved histograms of $R_1, R_2 \cdots R_n$, the query processor estimates (1) the size of each $n_x.R_i, R_i \in q$ as $ES(n_x.R_i)$ and (2) the result size of each pair of joining between $n_x.R_i$ and $n_y.R_j, R_i \bowtie R_j \in q$ as $ES(n_x.R_i \bowtie n_y.R_j)$. Furthermore, from this estimation, the result of joining $n_1.R_1 \bowtie n_2.R_2 \cdots \bowtie n_k.R_k, k \geq 3$, can be calculated as $ES(n_1.R_1 \bowtie n_2.R_2 \cdots \bowtie n_k.R_k) = \frac{F_1}{F_2}$, where $F_1$ and $F_2$ are computed as follows.

$$F_1 = \prod_{i=1, k-1} ES(n_i.R_i \bowtie n_{i+1}.R_{i+1})$$

$$F_2 = \prod_{i=2, k-1} ES(n_i.R_i)$$

To construct an initial query plan for $q$, let us define the plan search space as a directed joining search tree $JST(q)$ as follows.

![Joining Search Tree](image)

**Fig. 1. Joining Search Tree of $q = R_1 \bowtie R_2 \bowtie R_3$**

**Definition 2: Joining Search Tree, $JST(q)$**

Given $q$, $JST(q)$ is a tree satisfying the following properties:

1) A special node $V$ is created at level 0 of $JST(q)$. This node is the root of the tree.

2) A tree node $tn(n_x.R_i)$ is created at level 1 of $JST(q)$ and is set as a child of $V$ if $\exists n_x.R_i$. $R_i \in q$.

3) A tree node $tn(n_1.R_1 \cdots n_k.R_k)$ is created at level $k > 1$ of $JST(q)$ if either or both of the following conditions is satisfied:

   - $\exists tn(n_1.R_1 \cdots n_{k-1}.R_{k-1})$ in level $(k-1)$ of $JST(q)$ and $\exists n_k.R_k, R_k \bowtie R_k \in q$ and $n_k.R_k$ does not appear in $n_1.R_1 \cdots n_{k-1}.R_{k-1}$.
   - $\exists tn(n_1.R_1 \cdots n_{k-1}.R_{k-1})$ in level $(k-1)$ of $JST(q)$ and $\exists n_1.R_1, R_1 \bowtie R_2 \in q$ and $n_1.R_1$ does not appear in $n_2.R_2 \cdots n_k.R_k$.

$tn(n_1.R_1 \cdots n_k.R_k)$ is set as a child of either or both $tn(n_1.R_1 \cdots n_{k-1}.R_{k-1})$ and $tn(n_2.R_2 \cdots n_k.R_k)$.

4) Each link from a parent node $tn(n_1.R_1 \cdots n_k.R_k)$ to its children is assigned a cost $C = ES(n_1.R_1 \bowtie \cdots \bowtie n_k.R_k)$. The cost of links from $V$ is 0.

**Example 1:** Figure 1 shows the joining search tree $JST(q)$ of $q = R_1 \bowtie R_2 \bowtie R_3$, where $R_1$ has only one partition at $n_1$; $R_2$ has two partitions at $n_2$ and $n_3$; $R_3$ has three partitions at $n_4$, $n_5$ and $n_6$. There are 6 nodes in level 1 of $JST(q)$ corresponding to 6 partitions of queried relations. There are 8 nodes in level 2 of $JST(q)$ corresponding to 8 possible combinations of nodes in level 1 of $JST(q)$ with 6 basic partitions. Similarly, there are 6 nodes in level 3 of $JST(q)$ corresponding to 6 possible combination of nodes in level 2 of $JST(q)$ with 6 basic partitions.

**Definition 3: Sub-query plan, $subQP(q)$**

Given $JST(q)$, $subQP(q)$ is a plan of joining partitions of queried relations by following a path from the root to a leaf node of $JST(q)$. The cost of a $subQP(q)$ is the total cost of links along the traversal path.

Based on the definition of a sub-query plan, we give a definition of a query plan and the optimal query plan for a $JST(q)$ as follows.

**Definition 4: Query Plan, $QP(q)$**

Given $JST(q)$, a query plan of $q$ denoted as $QP(q)$ is a set of $subQP(q)$ such that each leaf node in $JST(q)$ is reached by exactly one $subQP(q)$. The cost of $QP(q)$ denoted as $C(QP(q))$ is the total cost of all sub-query plans inside it.
Algorithm 1 PDH\textsubscript{JST}(\textit{JST}(q))

1: SET init\textit{QP}(q) = \emptyset
2: for each node \textit{tn} in \textit{JST}(q) do
3:   SET Path(\textit{tn}) = \emptyset
4:   for \textit{i} = 0 to max\_level(\textit{JST}(q)) − 1 do
5:     for each node \textit{tn}_\textit{i} in level \textit{i} do
6:       for each link \textit{L} connecting \textit{tn}_\textit{x} and \textit{tn}_\textit{y} in level \textit{i} + 1 do
7:         Path(\textit{tn}_\textit{y}) = Path(\textit{tn}_\textit{x}) ∪ \textit{L}
8:         Path(\textit{tn}_\textit{y}).cost = Path(\textit{tn}_\textit{x}).cost + \textit{L}.cost
9:       for each leaf node \textit{tn} do
10:          sub\textit{QP}(q) = the path with minimal cost in Path(\textit{tn})
11:         init\textit{QP}(q) = init\textit{QP}(q) ∪ sub\textit{QP}(q)
12: return \textit{R}.

Definition 5: Optimal Query Plan, opt\textit{QP}(q).

The optimal query plan of \textit{q} denoted as opt\textit{QP}(q) is a \textit{QP}(q) such that \textit{QP}(q) \subseteq opt\textit{QP}(q) and \textit{cost}(opt\textit{QP}(q)) < \textit{cost}(\textit{QP}(q)).

To search a good query plan for a query \textit{q}, we only consider left-deep query plans in \textit{JST}(q). Even though we may miss some query plans, by using the same assumptions as system \textit{R} [5], the space of left-deep plans is sufficient to avoid bad query plans while allowing good enough plans to be generated at a lower cost.

Theorem 1: The problem of finding the optimal query plan in a \textit{JST}(q) can be reducible to the Steiner tree search problem.

Proof: In a \textit{JST}(q), if we consider the root as the starting node, the leaves as termination nodes and internal nodes as Steiner nodes, by constructing the Steiner tree on \textit{JST}(q), we generate an optimal query plan for query \textit{q}.

The problem of finding Steiner tree is widely known to be NP-Complete [16]. In our system, we employ the Pruned Dijkstra Heuristic (PDH) algorithm [4] to find an initial query plan. PDH guarantees to find a plan whose cost is less than two times of the cost of the optimal plan. Algorithm 1 presents our adapted PDH algorithm to find an initial query plan for a \textit{JST}(q).

Theorem 2: The PDH\textsubscript{JST} algorithm has the complexity of \textit{O}(\textit{E}), where \textit{E} is the number of links in the \textit{JST}.

Proof: In Algorithm 1, since we iterate every link only once, the complexity is linear to \textit{E}.

In general, to construct an initial query plan, the query processor first collects the corresponding statistics of histograms to build the joining search tree. After that, it applies Algorithm 1 to generate an initial query plan. This plan is then sent to all related peer nodes to process the query.

B. Adaptive Query Processing

The initial query plan estimated from the local histograms may not be the optimal one because employing histograms to estimate the join result size may lead to high error rates. In this paper, we apply the adaptive query processing technique. After processing some tuples based on the old plan, the system can reevaluate the cost and generate a better query plan if necessary.

Our adaptive scheme contains two parts, re-estimating the cost of links in joining search tree and regenerating query plans based on the updated costs. In this section, we focus on how to re-estimate the costs of links. Once the costs are correctly updated, we can apply Algorithm 1 to compute the new query plans.

Definition 6: Join Selectivity \textit{JS}(\textit{T} \bowtie \textit{n}_\textit{x}.\textit{R}_\textit{y})

Given \textit{T} and \textit{n}_\textit{x}.\textit{R}_\textit{y}, \textit{T} can be either a single relation \textit{n}_\textit{x}.\textit{R}_\textit{y} or a joining result of a set of relations \{(\textit{n}_1,\textit{R}_1) \bowtie \cdots \bowtie (\textit{n}_k,\textit{R}_k)\}, the join selectivity of \textit{(T} \bowtie \textit{n}_\textit{x}.\textit{R}_\textit{y}) is

\[
\text{JS}(\textit{T} \bowtie \textit{n}_\textit{x}.\textit{R}_\textit{y}) = \frac{\textit{S}(\textit{T} \bowtie \textit{n}_\textit{x}.\textit{R}_\textit{y})}{\textit{S}(\textit{T}) \times \textit{S}(\textit{n}_\textit{x}.\textit{R}_\textit{y})}
\]

Where \textit{S}(\textit{T} \bowtie \textit{n}_\textit{x}.\textit{R}_\textit{y}) is the result size of the join operation between \textit{T} and \textit{n}_\textit{x}.\textit{R}_\textit{y} and \textit{S}(\textit{T}) and \textit{S}(\textit{n}_\textit{x}.\textit{R}_\textit{y}) are input sizes of \textit{T} and \textit{R}_\textit{y}.

Example 2: According to the initial query plan, which is built from histograms of queried relations presented in the previous section, the join selectivity of \{(\textit{n}_1,\textit{R}_1) \bowtie \textit{n}_2.\textit{R}_2\} can be computed as \textit{JS}(\textit{n}_1.\textit{R}_1 \bowtie \textit{n}_2.\textit{R}_2) = \frac{\textit{ES}(\textit{n}_1.\textit{R}_1,\textit{n}_2.\textit{R}_2)}{\textit{ES}(\textit{n}_1.\textit{R}_1,\textit{n}_2.\textit{R}_2)}.

Assume that we have two sub-query plans: one is \textit{n}_1.R_1 \bowtie \textit{n}_2.\textit{R}_2 \bowtie \textit{n}_3.\textit{R}_3 and the other is \textit{n}_2.\textit{R}_2 \bowtie \textit{n}_3.\textit{R}_3 \bowtie \textit{n}_1.R_1, and hence these two plans share the same tree node \textit{tn}(\textit{n}_1.\textit{R}_1\textit{n}_2.\textit{R}_2\textit{n}_3.\textit{R}_3). This node has two parent nodes \textit{tn}(\textit{n}_1.\textit{R}_1\textit{n}_2.\textit{R}_2) and \textit{tn}(\textit{n}_2.\textit{R}_2\textit{n}_3.\textit{R}_3), which are in the two plans. Now if the first sub-query plan is executed, \textit{JS}'(\textit{n}_1.\textit{R}_1 \bowtie \textit{n}_2.\textit{R}_2 \bowtie \textit{n}_3.\textit{R}_3) and \textit{ES}'(\textit{n}_1.\textit{R}_1 \bowtie \textit{n}_2.\textit{R}_2 \bowtie \textit{n}_3.\textit{R}_3) are updated. Based on the definition of join selectivity, these new values can be used to recalculate a new \textit{ES}'(\textit{n}_2.\textit{R}_2 \bowtie \textit{n}_3.\textit{R}_3) of the second query plan by the following formula:

\[
\text{ES}'(\textit{n}_2.\textit{R}_2 \bowtie \textit{n}_3.\textit{R}_3) = \frac{\textit{F}_1}{\textit{F}_2},
\]

where

\[
\text{F}_1 = \text{JS}(\textit{n}_1.\textit{R}_1 \bowtie \textit{n}_2.\textit{R}_2 \bowtie \textit{n}_3.\textit{R}_3) \cdot \text{ES}(\textit{n}_1.\textit{R}_1)
\]

\[
\text{F}_2 = \text{ES}(\textit{n}_1.\textit{R}_1 \bowtie \textit{n}_2.\textit{R}_2 \bowtie \textit{n}_3.\textit{R}_3)
\]

Applying the above formulas, we can adjust the edge weights of the joining search tree. Then the Query Executor and Optimizer may generate a new sub-query plan with a lower cost. If necessary, it needs to terminate the current plan and replaces it with the newly generated plan.

C. Distributed Mechanism

In our system, we propose a distributed mechanism to maintain the joining search tree. The query initiator only needs to construct the \textit{JST}(\textit{q}) for a query \textit{q}. After that, it disseminates the \textit{JST}(\textit{q}) to all related peer nodes. These peer nodes then cooperate to create the initial query plan. Query plan optimization is also done in a distributed manner. In this way, a single node failure does not affect the functionality of the optimizer. In particular, our distributed mechanism works as follows. For each tree node in \textit{JST}(\textit{q}), its information, including its estimation size, parent information, children information and link costs, is sent to a specific peer. The estimation size and link costs are used for generating query plan, while the parent and children information are used to communicate among peers involving the \textit{JST}(\textit{q}) later. In our system, a tree node \textit{tn}_\textit{x}(\textit{n}_1.\textit{R}_1 \cdots \textit{n}_\textit{R}_\textit{n}) is assigned an identifier \textit{ID}(\textit{tn}_\textit{x}) = \textit{n}_1.\textit{R}_1 \cdots \textit{n}_\textit{R}_\textit{n} and the information of \textit{tn}_\textit{x} is indexed in the P2P network at a node holding key \textit{k}(\textit{tn}_\textit{x}) = \textit{H}(\textit{ID}(\textit{tn}_\textit{x})), where \textit{H} is the hash function defined for the overlay.
Involving a join operation from 10 to 50, and hence the total number of peers involving a query having 3 join operations may be up to 500 peers. The results in Figure 3 show that even though the average network cost increases significantly due to an increase in the number of peers/tables involving the query, the average execution time does not increase much. This is simply because our query processing scheme allows a join operation to be processed in parallel at different pairs of tables.

VI. CONCLUSION

In this paper, we have proposed an adaptive multi-join solution for reducing the network cost of processing distributed joins. We model distributed query processing in P2P networks as a tree model and apply an approximate algorithm to find a good enough initial query plan. We further proposed a new adaptive scheme, which can adjust the query plan if necessary based on the real-time statistics obtained during runtime. Experiments show that our adaptive query processing mechanism can greatly reduce both the network cost and time for query processing.

REFERENCES