Join Selectivity Re-estimation for Repetitive Queries in Databases

Feng Yu[1], Wen-Chi Hou[1], Cheng Luo[2], Qiang Zhu[3], and Dunren Che[1]

[1] Southern Illinois University, Carbondale, IL 62901, USA
{fyu,hou,dche}@cs.siu.edu
[2] Coppin State University, Baltimore, MD 21216, USA
cluo@coppin.edu
[3] University of Michigan, Dearborn, MI 48128, USA
qzhu@umich.edu

Abstract. Repetitive queries refer to those queries that are likely to be executed repeatedly in the future. Examples of repetitive queries include those that are used to generate periodic reports, perform routine maintenance, summarize data for analysis, etc. They can constitute a large part of daily activities of the database system and deserve more optimization efforts. In this paper, we propose to collect information about joins of a repetitive query, called the trace, during execution. We intend to use this information to re-estimate selectivities of joins in all possible execution orders. We discuss the information needed to be kept for the joins and design an operator, called the extended full outer join, to gather such information. We show the sufficiency of the traces in computing the exact selectivities of joins in all plans of the query. With the exact selectivities of joins available, the query optimizer can utilize them to find truly the best join order for the query in its search space, guaranteeing "optimal" execution of the query in the future.

Keywords: Join Selectivity Estimation, Query Re-optimization.

1 Introduction

A primary problem in query optimization is to find the most efficient execution plan for a query, which is mainly determined by the join orders. In order to find the best join order, accurate cost estimations of alternative join orders must be known. Query optimizers generally use statistics stored in the database catalogs, such as histograms [3–5], etc., and assumptions about attribute values [1, 7] to estimate the cost. Unfortunately, due to the complexity of queries, sufficiency of statistics, and validity of assumptions, query optimizers often cannot find the most efficient join orders for the queries in their search spaces. Studies [1, 2] have shown that it could be orders of magnitude slower in speed when executing queries with sub-optimal plans. Thus, some database systems, like Sybase and Oracle, allows users to force the join orders; some, e.g., Sybase, even allows users

A. Hameurlain et al. (Eds.): DEXA 2011, Part II, LNCS 6861, pp. 420–427, 2011.

to explicitly edit the plans [6]. Unfortunately, such measures cannot guarantee success and can also be quite cumbersome and slow for complex queries. Clearly, there is a tremendous need for a mechanism that can automatically refine the execution plans of queries.

Repetitive queries refer to those that are likely to be posted repeatedly in the future. Many useful queries, such as those used for generating periodical reports, performing routine maintenances, summarizing and grouping data for analysis, etc., are repetitive queries. They are often stored in databases for convenient reuses for the long term. Any sub-optimality in the execution plans of such queries could mean repetitive and continued waste of system resources and time in the future. The efficiency of the executions of repetitive queries has a paramount effect on the performance of the system and thus deserves more optimization efforts. In this research, we attempt to gather information about a repetitive query while it is being executed. We show that the information gathered is sufficient for optimizers to compute the selectivities of joins accurately in all execution (or join) orders. It is worth mentioning that we do not intend to modify the search strategy of the optimizer, but just to provide it with sufficient and accurate information so that it can find truly the best join order for the query in its solution space, conveniently called the optimal plan here.

In this paper, we introduce the notion of the trace of a query, which contains information about how tuples from input relations are joined in the query. We propose to collect the trace of a query during execution and use it to re-estimate selectivities of joins in all alternative plans of the query derived by exchanging the input relations. We have designed operators to collect sufficient information in the traces so that the exact join selectivities in all execution orders can be computed. With the exact join selectivities known, the query optimizer can find the best execution plan for a repetitive query for future executions. Substantial saving can be obtained from running such queries with optimal plans, not to mention running them repeatedly. This work makes a major stride in the research of query re-optimization and can make significant improvement on the performance of the system.

The rest of the paper is organized as follows. Terminologies and definitions used in the paper are introduced in Section 2.Section 3 discusses selectivity estimation using traces for queries with acyclic join graphs. Due to space limitation, interested readers are referred to [8] for queries with cyclic join graphs. The analysis of overhead is included in section 4. Section 5 presents the conclusions and future work.

2 Framework and Terminology

In this section, we describe the selectivity re-estimation framework and introduce terminology used in the paper.

2.1 Join Selectivity Re-estimation Framework

We attempt to gather information about how tuples are joined in a query while the query is being processed. The information to gather here is called the query (or join) trace.

We assume that there are mechanisms in the database that can differentiate a repetitive query from an ordinary query. We also assume an optimizer knows how to compute the selectivities of joins from the trace, which will be discussed in the next two sections.

Figure 1(a) depicts the framework. A query is first optimized by the query optimizer as usual. If it is a repetitive query, its trace will be gathered when the query is executed. Once the execution completes, the trace collected will be provided to the optimizer to compute selectivities of joins of alternative plans and select the best plan. Physical execution plans are generated based on the best logical plans by the optimizer and stored in the database for future invocations of the query. Certainly, after the database has gone through substantial changes, the process of trace gathering and re-optimization can be re-invoked.

In this paper, we discuss the information to be gathered, and how to gather them. In addition, we show the sufficiency of the gathered information for computing the exact selectivities of joins of alternative plans.

2.2 Join Graph

A query can be modeled by a graph, called the join graph, that describes the join relationships among participating relations.

Definition 1 (Join Graph). *The join graph $G(V, E)$ of a query consists of a set of vertices $V = \{R_1, R_2, ..., R_n\}$ and a set of edges E. Each vertex denotes an input relation of the query and each edge $(R_i, R_j) \in E, R_i, R_j \in V$, denotes the existence of join conditions between R_i and R_j in the query.*

Example 1 (Join Graph). Fig. 1(b) shows the join graph of a query where there are join conditions placed between R_1 and R_2, R_2 and R_3, and R_3 and R_4.

If the join graph of a query is disconnected, we can consider each connected component separately (and then merge them by Cartesian products). Therefore, we shall assume hereafter all queries have connected join graphs or all join graphs are connected. For simplicity, we further assume all joins are equi-join, though our approach can be applied to other joins, such as non-equi joins, and Cartesian products.

In this research, we assume every (execution) plan P is in the form of a left-deep tree $P = (...((R_1 \bowtie R_2) \bowtie R_3)...) \bowtie R_n$ [6] because most, if not all, commercial database systems generate such plans for executions. We assume all $\bowtie$'s are equi-joins and no Cartesian product appears in P. It is worth mentioning that the method proposed is applicable to bushy trees and right-deep trees as well.

Let $G(V, E)$ be the join graph of a query Q. In this research, we are interested in estimating the selectivities of all possible subqueries that are joins of some or all of the relations in V. Note that we assume all subgraphs $G'(V', E')$ are connected, and no Cartesian product exists in any of the subqueries or plans.

Definition 2 (Joinable Relations). *A pair of relations R_i and R_j are said to be joinable in a query if there is an edge (R_i, R_j) in the join graph of the query.*

Definition 3 (Joinable Tuples). *A pair of tuples t_i and t_j, $t_i \in R_i, t_j \in R_j$, are said to be joinable if t_i and t_j have the same value for the join attributes of R_i and R_j. Joinable tuples are also referred to as match tuples.*

Theorem 1. *Given a connected join graph for a query and an execution plan $P = (...((R_1 \bowtie R_2) \bowtie R_3)...) \bowtie R_n$, each relation $R_k, 2 \leq k \leq n$, has join attributes with exactly one prior relation $R_i, i < k$, in P if and only if the join graph of the query has no cycle.*

Proof. See [8].

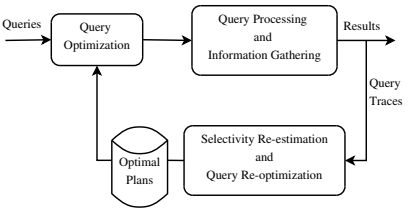
2.3 Query Trace

When a query is being processed, information about how tuples are joined is gathered. We intend to use this information, called query (or join) trace, to estimate selectivities of joins in all execution orders.

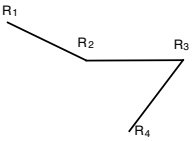
Example 2. Fig. 1(b) is the join graph of a query, and Fig. 1(c) shows the matching of join attribute values between tuples. For simplicity, we have represented a tuple only by its added IDs without reference to other attribute values, that is, $R_1 = \{1, 2, 3\}, R_2 = \{a, b\}, R_3 = \{A, B\}, R_4 = \{\text{I, II}\}$. For example, tuple 1 of R_1 matches tuple a of R_2, and tuples 2 and 3 match tuple b of R_2. Tuples a and b of R_2 match tuples A and B of R_3, respectively. Finally, tuples A and B of R_3 match tuples I and II of R_4, respectively.

Consider a left-deep tree execution plan $P = ((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$. To generate the trace, an ID attribute is added to every relation and the attribute is to be preserved in the outputs of all operators. Thus, the result of $R_1 \bowtie R_2$, as shown in Fig. 1(d), besides its normal set of attributes, denoted by Result-Attrs, has additional attributes R_1-ID and R_2-ID, called the trace of $R_1 \bowtie R_2$, denoted by $T(R_1 \bowtie R_2)$.

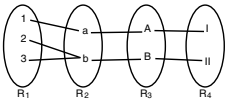
Fig. 1(f) shows the trace of $((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$, denoted by $T(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4)$. Once a query is completely processed, we can extract the final trace, e.g., $T(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4)$ in Example 2, from the "extended" query result by a simple projection on all the added ID attributes.



(a) Selectivity Re-estimation Framework



(b) A Join Graph



(c) Matching of Tuples

Result-Attrs	R_1-ID	R_2-ID
...	1	a
...	2	b
...	3	b

(d) Result and Trace of $R_1 \bowtie R_2$

R_1-ID	R_2-ID	R_3-ID
1	a	A
2	b	B
3	b	B

(e) Trace of $(R_1 \bowtie R_2) \bowtie R_3$

R_1-ID	R_2-ID	R_3-ID	R_4-ID
1	a	A	I
2	b	B	II
3	b	B	II

(f) Trace of $((R_1 \bowtie R_2) \bowtie R_3) \bowtie R4$

Fig. 1. Query Traces

3 Selectivity Estimation for Acyclic Join Graphs

In this and next sections, we discuss information that needs to be incorporated into the traces in order to estimate selectivities of joins accurately.

Let Q be a query with an acyclic join graph $G(V, E)$ and P an execution plan of the query. Let $T(P)$ be the final trace of P. Let $G'(V', E')$ be a vertex-induced connected subgraph of $G(V, E)$, in which $V' = \{R_{i_1}, ..., R_{i_m}\} \subseteq V$ and $E' \subseteq E$, representing a subquery Q' of Q. The selectivity of Q' can be estimated as

$$\widetilde{sel}(Q') = \frac{|\pi_{R_{i_1}\text{-ID},...,R_{i_m}\text{-ID}} T(P)|}{|R_{i_1}| \times ... \times |R_{i_m}|} \tag{1}$$

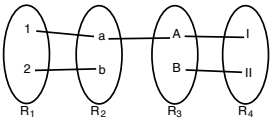
in which $\pi_{R_{i_1}\text{-ID},...,R_{i_m}\text{-ID}} T(P)$ is the projection of trace $T(P)$ on attributes R_{i_1}-ID, ..., R_{i_m}-ID, without duplicate.

3.1 No Dangling Tuples in the Joins

Here, we assume no dangling tuple exists in any of the joins in the query, i.e. every tuple in one relation finds at least one matching tuple in another relation with which there is a join edge in the join graph. The relations in Fig. 1(c) satisfy this condition.

Theorem 2. *Let P be an execution plan of a query Q with a connected acyclic join graph $G(V, E)$. Let Q' be a subquery of Q that has a vertex-induced connected join subgraph $G'(V', E'), V' = \{R_{i_1}, ..., R_{i_m}\} \subseteq V$. If there is no dangling tuple in any join of P, Eq. (1) derives the exact selectivity of Q' from $T(P)$.*

Proof. See [8] .



(a) Matching of Join Attribute Values

R_1-ID	R_2-ID	R_3-ID	R_4-ID
1	a	A	I

(b) No Information about Dangling Tuples

R_1-ID	R_2-ID
1	a
2	b

(c) $T(R_1 \bowtie R_2)$ Generated by Outer Joins

R_1-ID	R_2-ID	R_3-ID
1	a	A
2	b	
		B

(d) $T((R_1 \bowtie R_2) \bowtie R_3)$ Generated by Outer Joins

R_1-ID	R_2-ID	R_3-ID	R_4-ID
1	a	A	I
2	b		
		B	II

(e) $T(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4)$ Generated by Outer Joins

Fig. 2. Dangling Tuples in Relations

overlap	90%		80%		70%	
Rel. size	time	result size	time	result size	time	result size
10k	3.12%	21.03%	3.87%	47.60%	4.98%	81.51%
100k	2.46%	20.23%	5.46%	45.79%	6.30%	78.19%

Fig. 3. Overhead: outerjoin vs. join

3.2 Dangling Tuples in Joins

Dangling tuples are lost in the joins. To retain matching information about dangling tuples, we replace the joins in the query by the full outer joins ($\overset{\circ}{\bowtie}$). Fig. 2(c) to 2(d) show the traces generated at different stages of query execution, where the joins are replaced by the full outer joins. The trace in Fig. 2(c) is the same as it were generated by a join because there is no dangling tuple in the join. The trace in Fig. 2(d) retains information about dangling tuples b in R_2 and B in R_3 by the outer join. Fig. 2(d) is the final trace that will be retained and used in later selectivity estimation.

The estimated selectivities for $R_1 \bowtie R_2$, $R_2 \bowtie R_3$, and $R_3 \bowtie R_4$ are now, by Eq. (1), $\frac{1}{2}(=\frac{2}{2\times 2})$, $\frac{1}{4}(=\frac{1}{2\times 2})$, and $\frac{1}{2}(=\frac{2}{2\times 2})$, respectively, which are exact. Note that, as mentioned earlier, a trace tuple having a null for any of the projected attributes is not accounted for in the respective $|\pi_{R_{i_1}\text{-ID},\ldots,R_{i_1}\text{-ID}}T(P)|$ because a null in a R_{i_j}-ID column of a trace tuple indicates that there is no match found in R_{i_j} for the respective combination of tuples to generate an output in the (sub)query. One can easily verify that the estimated selectivities for all other subqueries are all exact.

Theorem 3. *Let P be an execution plan of a query Q with a connected acyclic join graph $G(V, E)$. Let Q' be a subquery of Q that has a vertex-induced connected*

join subgraph $G'(V', E'), V' = \{R_{i_1}, \ldots, R_{i_m}\} \subseteq V$. *Eq. (1) derives the exact selectivity of Q' from the trace obtained by replacing the joins in the query with the full outer joins, denoted by* $T(P)$.

Proof. See [8]

4 Preliminary Experimental Results

4.1 Preliminary Experimental Results

We test two cases where the synthetic relations have 10K and 100K tuples. Each input relation and the result relation has 5 attributes. By overlapping parts of the domains of join attributes, we generate match tuples, partial match, and no-match tuples.

Relations are read into memory for processing. The CPU cost accounts for the cost of all processing and the writing of outputs to memory. We use result size as a measure for potential I/O cost if the result cannot fit in memory . Table 3 shows the CPU and result size (in terms of the number of tuples) overheads for the outer join operator. The overheads are computed as $(OJ - J)/J$, where OJ and J represent the CPU time and the result sizes of outer join and join, respectively.

The CPU overheads (i.e., 3.12%, 3.87%, 4.98%) are still quite small. This is because the same amounts of computations, for hashing and comparisons, need to be performed for both the join and outer join. Only copying dangling tuples to the output relation (in memory) is extra, which does not take much time. It is noted that the results could have been better or worse depending on the amounts of dangling tuples generated in the relations.

From the experiments, we observe that CPU overhead is much more acceptable than result size overhead. Therefore, if the memory is large enough to hold the trace at each stage, the trace gathering can be performed with query evaluation with too much of delay. On the other hand, if the memory is too small to hold the traces, the result size overhead could dramatically slow down the query processing. If that is the case, we may have to gather the trace off-line by running the query again in spare time.

5 Conclusions and Future Work

In this paper, we propose to collect information about joins, called traces, to re-estimate the selectivities of joins of repetitive queries. We have shown that the exact selectivities of joins in all execution orders of a query can be computed from its trace. In the future, we shall empirically study the overheads incurred in the the process of trace gathering more thoroughly.

References

1. Christodoulakis, S.: Implications of certain assumptions in database performance evauation. ACM Trans. Database Syst. 9, 163–186 (1984)
2. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., Cilimdzic, M.: Robust query processing through progressive optimization. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD 2004, pp. 659–670. ACM, New York (2004)
3. Muralikrishna, M., DeWitt, D.J.: Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In: SIGMOD Conference, pp. 28–36 (1988)
4. Piatetsky-Shapiro, G., Connell, C.: Accurate estimation of the number of tuples satisfying a condition. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD 1984, pp. 256–276. ACM, New York (1984)
5. Poosala, V., Ioannidis, Y.E.: Selectivity estimation without the attribute value independence assumption. In: Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB 1997, pp. 486–495. Morgan Kaufmann Publishers Inc., San Francisco (1997)
6. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd edn. McGraw-Hill, New York (2003)
7. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, SIGMOD 1979, pp. 23–34. ACM, New York (1979)
8. Yu, F., Hou, W.-C., Luo, C., Zhu, Q., Che, D.: Join selectivity re-estimation for repetitive queries in databases, `http://www2.cs.siu.edu/~fyu/main-trace.pdf`