SQL Query Optimization Methods of Relational Database System

Dandan Li
Department of Computer Software
Beijing City University
Beijing, China
LiDandan1976@163.com

Lu Han
Library of Beijing Institute Of Technology
Beijing Institute of Technology
Beijing, China

Yi Ding
Institute of Physical Geography and Information Technology
China University of Geosciences
Beijing, China

***Abstract*—Oracle provides several options to aid performance, such as partitioning large tables, using materialized views, storing plan outlines, and many others. This chapter examines how DBAs can use these techniques to aid developers' efforts to increase the efficiency of their application code. This chapter introduces the SQL Tuning Advisor to help you tune SQL statements. You can then use the recommendations of this advisor to rewrite poorly performing SQL code. I begin the chapter with a discussion of how to approach performance tuning. More than the specific performance improvement techniques you use, your approach to performance tuning determines your success in tuning a recalcitrant application system.**

Keywords: Query optimization, SQL Server, Relational database

I. INTRODUCTION

Optimization has always been, and still is, a central topic in database research. The last 30 years have seen a tremendous amount of work in the topic, and that work still continues strong today. This is especially true in the context of Decision Support, where the amount of data keeps growing, queries keep on getting more and more complex, and answers are still expected quickly. SQL, the de facto standard for relational query languages, supports more operations now than they did just a few years ago (cubes, windows, etc.), and this also adds to the complexity. Hence, query optimization continues to be a relevant research topic Being a declarative language, SQL relies on optimization techniques to provide responses in a timely manner. However, it is known that complex queries (with subqueries, grouping, and large number of joins) generate so many choices for execution (i.e., possible query plans),
that most optimization algorithms cannot guarantee that an *optimal* plan is chosen. It has been noted that the structure of SQL is itself to blame for the complexity of some queries [16].

In particular, queries that involve *aggregate* functions, very common in Decision Support environments, must first compute the aggregate in a subquery before they can use its value in a comparison. For instance, in table ITEM (itemid, supid, price), where item itemid is supplied by supplier supid at price price, selecting the cheapest item calls for a query with an embedded subquery: select itemid from item where price = (select min(price)from item)The subquery will calculate the price of the cheapest item; the main query will use this information to select the items with such price. Note that both the main query and the subquery will run over the same table, ITEM. This is quite a common situation, since the aggregate is usually computed over data closely related to the data which will use the value of the aggregate for selection. We say that such queries contain *intra-query redundancy*, that is, redundancy between a main query block and a subquery block, where redundancy means overlap in the FROM clause and possibly in the WHERE clause. Obviously, queries containing intra-query redundancy are not infrequent. A similar argument is made by Zhu et al. [16], where the overlapping parts are called "common subparts". There are several workarounds for this problem: the simplest one is to write the query as a sequence of SQL queries, each one creating a partial solution in the form of a table or view which is later used to compute the final result. However, it is difficult to optimize sequences of SQL queries [10,20]. Extending the SQL language to deal with some of the problems involving aggregation has been proposed [2,3]. The SQL standard group, mindful of the problem, has added the ability to define queries in a WITH clause or inside the FROM subclause, as well as the ability to use a CASE statement in the SELECT clause to define results conditionally. This alleviates, but does not solve, the problem.

II. RELATED WORK AND MOTIVATION

One of the most powerful features of SQL is nested queries. A *nested query* is a query that has another query embedded within it. An embedded query may appear in the FROM clause (called a *derived table*), or in the WHERE or HAVING clause (called a *subquery*). The query that contains a subquery is called a *main* query or an *outer* query (we use the term "the main query" and "the outer query" interchangeably in this paper). The subquery can be either aggregate or non-aggregate. If the subquery has an aggregate function (MIN, MAX, SUM, COUNT, AVG) in its SELECT clause, we say it is an *aggregate subquery*; otherwise, we say it is a *non-aggregate subquery*. The subquery can be either independent of the outer query or correlated to the outer query. A *correlated subquery* contains a predicate (called a *correlated predicate*) that references a table in the outer query; thus, the subquery result depends on each tuple in the outer query. A query is a *multi-level* query if it has multiple subqueries. A query is a *one-level* query if all of its subqueries are flat queries which have no subqueries; a query is a *two-level* queries if at least one of its subqueries is a one-level query; and so on. Optimization of nested queries has received significant attention since the 1980s. It was originated from the observation that executing nested queries with correlated subqueries using the traditional *nested iteration* method can be very inefficient [13]. To improve performance of such queries, *query unnesting*, i.e., rewriting nested queries into flat forms, has been studied and proposed Nested queries with aggregate subqueries are widely used in SQL. In some

978-0-7695-3982-9/10 $26.00 © 2010 IEEE
DOI 10.1109/ICCEA.2010.113

cases, the aggregation in a subquery is computed over the same data (or closely related data) as the data in the outer query. As a result, some SQL queries present a large degree of redundancy in FROM and WHERE clauses between the outer query and the subquery. It is important to point out that redundancy is present because of the structure of SQL, which necessitates a subquery in order to declaratively state the aggregation to be computed. With the addition of user-defined methods to SQL, detecting and dealing with redundancy is even more important, as many time such methods are expensive to compute and it is hard for the optimizer to decide whether to push them down or not [12]. The following query from the TPC-H benchmark [4] gives some intuition about this problem.

Query 1 *A query with complete redundancy*. This query lists the suppliers who can supply the parts with the desired size and type in a given region at the minimum cost.

select s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment from part, supplier, partsupp, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 15 and p_type like '%BRASS' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name='EUROPE' and ps_supplycost = (select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'EUROPE')

The most noticeable feature of Query 1 is intra-query redundancy because the tables and conditions of the subquery are totally included in those of the outer query. We say it has *complete* redundancy (fig.1). As a nested query with a correlated, aggregate subquery, Query 1 can be evaluated using most unnesting techniques proposed in the literature (e.g., [5,13,14]).However, without redundancy recognized, the common parts (tables and conditions) between the outer query and the subquery have to be computed repeatedly.

III. Heuristic Strategies for Query Processing

The use of the cost-based optimization technique isn't the only way to perform query optimization. A database can also use less systematic techniques, known as heuristic strategies, for query processing. A join operation is called a binary operation, and an operation such as selection is called a unary operation. A successful strategy in general is to perform the unary operation early on, so the more complex and time-consuming binary operations use smaller operands. Performing as many of the unary operations as possible first reduces the row sources of the join operations. Fig.1 is complete redundancy. Here are some of the common heuristic query-processing strategies:

- Perform selection operations early so you can eliminate a majority of the candidate rows early in the operation. If you leave most rows in until the end, you're going to do needless comparisons with the rows that you're going to get rid of later anyway.
- Perform projection operations early so you limit the number of columns you have to deal with.
- If you need to perform consecutive join operations, perform the operation that produces the smaller join first.
- Compute common expressions once and save the results.

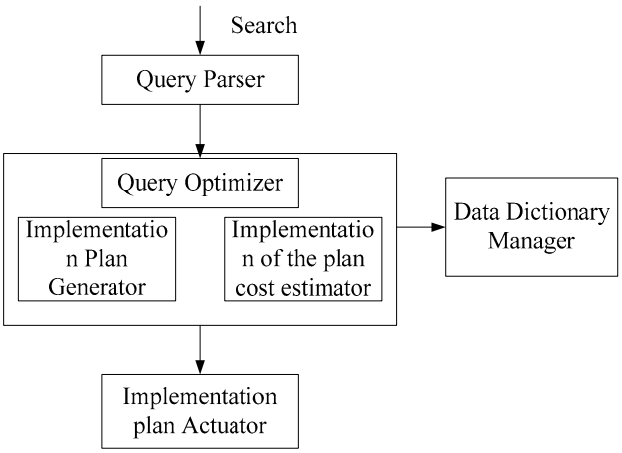

Fig.1 complete redundancy

IV. Performance Experiments

Contrast to [1], we focus on the query performance. We all know that the original intention of data cube is to speed up query performance. Since Dwarf already has high compression ratio, we consider that it is worthy to get better query performance at the cost of construction time and storage of Dwarf. The data set has 10 dimensions and 4×105 tuples. The cardinalities are 10, 100, 100, 100, 1000, 1000, 2000, 5000, 5000, and 10000, respectively. The aggregation operation is SUM. All the experiments are run on a Pentium IV with 2.6GHZ CPU and 512MB RAM, running Windows XP. We use ODBC connection to SQL Server 2000.In the following, non-clustering represents non-clustering Dwarf, Dwarf represents computational dependencies between the group-by relationships, R-Dwarf represents recursion clustering, and H-Dwarf represents hierarchical clustering.

Test 1: We test 7 to 10 dimensions with 4 × 105 tuples data set, respectively.

The construction time costs are shown in Fig.5 and Fig.6. The construction time of our algorithm is about one time more than the original one. The storage space of our algorithm is about 0.3 times more than the original one. The main reason is that the idle space manager, the node-indexing, and the logical clustering mechanism need additional time and space.

Test 2: The data cubes are the four Dwarfs used in Test 1 with 4×105 tuples. We create 1000 point queries randomly and run them continuously. Fig.7 shows the time cost. The recursion clustering outperforms the computational dependencies method by 10%. Hierarchical clustering behaves too badly, and it drops behind

V. Related work

This is an extension of the earlier work by including (a) the notion of extended xpath (Sect. 2) and its application in query translation and query answering (Sect. 3), (b) revised translation algorithms (Sects. 4, 5), in particular a new

algorithm. CycleEX for handling the descendant axis of xpath; and (c) an extensive experimental study. As remarked earlier, extended xpath is useful not only in query translation from xpath to sql, but also in developing native xml query engines [1,19] and answering xml queries over xml views. In particular, the use of variables in extended xpath allows us to represent each sub-query q in an extended xpath expression only once, no matter where and how often q appears in the query. This yields the lowpolynomial bound of CycleEX; in contrast, simply adopted Tarjan's algorithm for finding a regular-expression representation of all matching paths, which, in the worst case, may be of an exponential size. There has been a host of work on querying xml using an rdbms, over xml data stored in an rdbms or xml views published from relations for a comprehensive survey). At least two approaches have been proposed to querying xml data stored in relations. One approach is based on middleware and xml views, and the other is by translating xpath queries to sql.

The middleware-based approach, e.g., XPERANTO and SilkRoute [16], provides clients with an xml view of the relations representing the xml data. Upon receiving an xml query against the view, it composes the query with the view, rewrites the composed query to a query in a (rich) intermediate language supported by middleware, and answers the query by using the computing power of both the middleware and the underlying rdbms. However, this approach is tempered by the following. First, it is nontrivial to define a (recursive) xml view of the relational data without loss of the original information (see, e.g., [7,20] for detailed discussion). Second, it requires middleware support and incurs communication overhead between the middleware and the rdbms. Third, as observed by no algorithms have been developed for handling recursive queries over xml views with a recursive dtd. Another approach is by providing an algorithm for rewriting xml queries into sql (possibly extended with a recursion operator). This has been studied in two settings: for schema based xml storage that chooses relational schema by making use of xml schema, and for schema-oblivious xml storage that stores xml data in relations of a fixed schema regardless of xml schema. The schema-based approach allows one to derive efficient relational storage for xml data, retaining the semantic and structural information of the xml data. This is important for, among other things, query optimization, data exchange (see, e.g., [18] for a recent survey), xml access control (e.g., [11]) and xml view updates (e.g., [13]). However, as observed by [10], with the exception of [12] and this work, we are aware of no algorithm published for translating recursive xml queries over recursive dtds to sql for *schema-based* xml *storage*.

Closest to our work is [19], which proposed the first technique to rewrite recursive path queries over recursive dtds to sql for schema-based xml storage. The translation consists of two phases. First, by representing the input dtd D and input xpath query Q as finite state automata, it constructs the product automaton of the two that captures xpath recursion and dtd recursion in a uniform framework. Second, it translates the product automata into a sequence of sql queries with the sql'99 recursion operator. This approach has a low polynomial bound $O(|D|2 * |Q|4)$ on the product automata generated, which is comparable to our bound on extended xpath queries given in Theorem 4.2. Furthermore, several optimization techniques have also been developed, to eliminate duplicate paths by making part of the automata deterministic, and to optimize sql queries by leveraging integrity constraints during the translation [11,12]. As remarked earlier, this work differs from [19] in that we use the simple lfp operator, a low-end recursion functionality already supported by many rdbms, instead of the sql'99 recursion operator. In addition, we use extended xpath instead of automata. This allows us to handle rich qualifiers and selection paths in an xpath query uniformly rather than treating them separately. Furthermore, the approach presented here can also be used to answer xml queries over certain xml views. On the other hand, as mentioned earlier, the optimization techniques developed for [9], e.g., [1, 2], are also applicable to our approach.

For schema-oblivious xml storage, a number of translation and optimization techniques have been proposed. These include path-based techniques that leverage index structures to store root-to-node paths [6,15,14], and interval-based approach, e.g., region encoding [11] and Dewey encoding [12], that maintains structural relationships among elements and their ordering [11,16,20,15,12,12]. MonetDB [10], for example, stores xml data in a "node" relation and associates each node with a pair of preorder traversal and post order traversal ranks. Leveraging these, xpath recursion ('//') can be efficiently processed in terms of range comparisons, without requiring the support of recursive operators by sql. This approach is hampered by the following problems. First, most of these techniques are developed for schema-oblivious xml storage and adopt relations of a fixed schema independent of the xml schema. As mentioned earlier, this makes it difficult for, among other things, data exchange, secure xml queries, and update xml views. Second, the indexes and structural coding introduce additional overhead when storing and querying the data. Worse still, the cost of the maintenance of the indexes and coding may become prohibitive expensive when the data is frequently changed (see, e.g., [20], for lower bounds on the maintenance cost). In contrast, our approach does not incur extra cost in the dynamic context. Third, in many applications one would prefer a lightweight tool that provides the capability of answering xpath queries within the immediate reach of commercial rdbms, instead of using a heavy-duty system. Finally, one cannot use the encoding and indexing approaches to answer xml queries over xml views.

VI. CONCLUSION

We have proposed a new approach to translating a practical lass of xpath queries over (possibly recursive) dtds to sql queries with a simple lfp operator found in many commercial rdbms. The novelty of the approach consists in (1) a notion of extended xpath expressions capable of capturing dtd recursion and xpath recursion in a uniform framework; (2) an efficient algorithm for translating an

xpath query over a recursive dtd to an equivalent extended xpath expression that characterizes all matching paths, without incurring exponential blowup and better still, optimizing the query by filtering unnecessary computation based on the structural properties of the dtd during the translation; and (3) an efficient algorithm for rewriting an extended xpath expression into an equivalent sequence of sql queries. These provide not only the capability of answering important xpath queries *within the immediate reach* of most commercial rdbms, but also the query answering ability for certain xml views. Several extensions are targeted for future work. First, we recognize that several factors affect the efficiency of the sql queries produced by our translation algorithms, and we are currently developing a cost model in order to provide better guidance for xpath query rewriting. Second, we are also exploring techniques for multi-query and recursive-query optimization to simplify the sql queries produced. Moreover, we intend to incorporate optimization by means of semantic information such as integrity constraints.

REFERENCES

1. Afanasiev, L., Grust, T., Marx, M., Rittinger, J., Teubner, J.: An inflationary fixed point in XQuery. In: Proc. of ICDE (2008)
2. Agrawal, R., Devanbu, P.: Moving selections into linear least fixpoint queries. In: Proc. of ICDE (1988)
3. Aho, A., Ullman, J.: Universality of data retrieval languages. In: Proc. of POPL (1979)
4. Amer-Yahia, S., Cho, S., Lakshmanan, L., Srivistava, D.: Minimizationof tree pattern queries. In: Proc. of SIGMOD (2001)
5. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic sets and other strangeways to implement logic programs. In: Proc. of PODS (1986)
6. Bancilhon, F., Ramakrishnan, R.: An amateur's introduction to recursive query processing strategies. In: Proc. of SIGMOD (1986)
7. Barbosa, D., Freire, J.,Mendelzon, A.:Designing information-preservingmapping schemes for XML. In: Prof. of VLDB (2005)
8. Beeri, C., Ramakrishnan, R.: On the power of magic. J. Log. Program10 (1991)
9. Benedikt, M., Fan, W., Geerts, F.: XPath satisfiability in the presenceof DTDs. J. ACM **55**(2) (2008)
10. Jan, W., Bohannon, P.: Information preserving network. TODS 33(1) (2007)
11. Boncz, P.A., Grust, T., van Keulen, M.,Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In: Proc. of SIGMOD (2004)
12. Choi, B.: What are real DTDs like. In: Proc. of WebDB (2002)
13. Choi, B., Cong, G., Fan, W., Viglas, S.: Updating recursive XML views of relations. In: Prof. of ICDE (2007)
14. Christophides,V., Cluet, S., Moerkotte,G.: Evaluating queries with generalized path expressions. In: Proc. of SIGMOD (1996)
15. Clark, J., DeRose, S.: XML path language (XPath). W3C Recommendation,Nov 1999
16. DeHaan, D., Toman, D., Consens, M., Ozsu, T.: Comprehensive XQuery to SQL translation using dynamic interval encoding. In: Proc. of SIGMOD (2003)
17 Deutsch, A., Tannen, V.: MARS: A system for publishing XML from mixed and redundant storage. In: Proc. of VLDB (2003)
18. Ehrenfeucht, A., Zeiger, P.: Complexity measures for regular expressions. In: Proc. of STC'74 (1974)
19. EXSLT.: http://www.exslt.org/dyn/functions/closure/index.html
20. Fan, W., Bohannon, P.: Information preserving XML schema embedding. TODS 33(1) (2008)