Multi-Table Joins Through Bitmapped Join Indices

Patrick O'Neil, Goetz Graefe
Microsoft Corp.[1]

Abstract

This technical note shows how to combine some well-known techniques to create a method that will efficiently execute common multi-table joins. We concentrate on a commonly occurring type of join known as a *star-join*, although the method presented will generalize to any type of multi-table join. A star-join consists of a central *detail* table with large cardinality, such as an orders table (where an order row contains a single purchase) with foreign keys that join to *descriptive* tables, such as customers, products, and (sales) agents. The method presented in this note uses *join indices* with *compressed bitmap* representations, which allow predicates restricting columns of descriptive tables to determine an answer set (or *foundset*) in the central detail table; the method uses different predicates on different descriptive tables in combination to restrict the detail table through compressed bitmap representations of join indices, and easily completes the join of the fully restricted detail table rows back to the descriptive tables. We outline realistic examples where the combination of these techniques yields substantial performance improvements over alternative, more traditional query evaluation plans.

1. Introduction

In recent years, decision support and on-line analytical processing have created a substantial resurgence of interest in query optimization and query evaluation techniques. In the present technical note, we present a method that improves performance for many queries in these environments.

Star-Joins

In this paper, the term *star-joins* is used to mean a join of a main table, also called the *detail* table, with multiple *descriptive* tables. The detail table is often very large, e.g., a row for each order for any product, by any customer, through any sales agent, etc. The descriptive tables are typically much smaller, e.g., one row per customer or one row per product. Many decision support queries place restrictions on the descriptive tables that translate to restrictions on detail rows, e.g., find all orders for products costing over $50.00 from customers with less than $1 Million dollar yearly billing through agents in Chicago. Sometimes there are multiple levels of descriptive tables, e.g., customers referring to a regions table, and there may be restrictions connecting these levels of descriptive tables, e.g., a query on the orders table restricts regions by a transitive join through customers.

2. The Basic Techniques

Before attempting to combine the techniques used in our method, we will review each of the techniques individually. None of the techniques is new; they are discussed here only in order to make the paper self-contained and to give credit to earlier inventors and adopters.

Join Indices and Domain Indices

While traditional table indices map column values to containing rows in a single table, usually by reference to a row identifier, *join indices* [VALD87] typically associate column values and rows of two tables. In this way, the join index represents the fully precomputed join. It is a special form of a materialized view. Typical organizations for join indices include B-trees or hash indices

[1] The authors are on leave from UMass/Boston and Portland State University, respectively.

organized in any of the following ways: lookup by common join column value listing record identifiers (RIDs) of rows in both tables that join with that value; lookup by RID for each row of one table giving a list of RIDs of a second table for rows that join with the first row; lookup by (non-join) column value of one table giving a list of RIDs of a second table for rows that join with the rows in the first table with that column value; or by variations of these, for example where single column values are extended to multiple columns. Using the precomputed join, it is also possible to access rows of one table by arbitrary column values in a second, through a process that determines rows with given column value in the second table and then transitively relates those rows through the join index with joining rows of the first table.

Of course, join indices can be generalized from two tables to multiple tables. They are called *domain indices* when they associate the values of a domain (say social security numbers) with all columns of tables in the database where such values occur. Typically, there is only one table for which the indexed column is a key; for the remaining tables, the column will be a foreign key, and many rows can contain the same column value. We will come back to this point later when we consider representations of sets and bitmap indices.

It is our understanding that join indices are being used in some current proprietary commercial database systems, although there is little public acknowledgment of this. The original research describing join indices is well known in the research community.

Semi-Joins for Data Reduction

If join indices are not available, it may be useful to scan the first table, say T_0, extract the relevant join column, say $T_0.A$, possibly remove duplicates from the list of column values, and then join this list with the second table, say T_1, followed by a join of the first join's result with the first table. While this method seems rather roundabout, it has been shown useful in distributed database systems with high communication costs, as published for the SDD-1 research prototype, as well as some other cases.

Joining Indices instead of Base Tables

A special case of such semi-joins is to join a base table, say T_0, with the index of another table, say $T_1.A$; the result contains all columns from T_0 plus record identifiers for table T_1, permitting completion of the join by fetch operations based on those record identifiers. This idea can be expanded to join two indices, say $T_0.A$ and $T_1.A$, followed by fetch operations for both tables. Whether or not these techniques are advantageous depends on the join selectivity and the costs for scanning files and fetching records. Rooted in Kooi's thesis [KOOI80], the Ingres query optimizer and executor have used these techniques for many years.

Bitmap Indices

A traditional table index associates with each index keyvalue a list of row identifiers (RIDs) or primary keys for rows that have that value. It is well known that the list of rows associated with a given index keyvalue can be represented by a *bitmap* or *bit vector*. In a bitmap representation, each row in a table is associated with a bit in a long string, an N-bit string if there are N rows in the table, and the bit is set to 1 in the bitmap if the associated row is contained in the list represented; otherwise the bit is set to 0. This technique is particularly attractive when the set of possible keyvalues in the index is small, with a large number of rows, e.g. an index on a sex attribute, where sex = 'Male' or sex = 'Female'. In this example, there will be only two lists to be represented in an index, and the total number of bits stored will be 2N, while one out of two bits will (usually) be 1 in both bitmaps. (We can't be sure that these bitmaps will be complements of each other, since a deleted row will result in a bit that is zero in both.) When a large number of values exist in an index, each of the bitmaps is likely to be rather sparse, that is, very few bits will be 1 in the bitmaps, resulting in heavy storage requirements for storing a lot of zeros. In such cases, bitmap compression is used, e.g., run-length encoding (67 zeros turn into a prefix showing zeros follow, and a count of 67), or by changing representation from bitmap to RID list and back (as indicated in [O'NEI87]).

The point of using bitmap indices, of course, is the tremendous performance advantage to be gained. To start with there is reduced I/O when a large fraction of a large table is represented using a bitmap rather than by a RID list. In addition, a bitmap for a foundset on 10 million rows will require a maximum of only slightly more than a megabyte of storage (10 million bits = 1.25 million bytes) so bitmaps can commonly be pipelined or cached in memory, and the RIDs represented are automatically held in RID order, useful when combining predicates and when retrieving rows from disk. In addition, the most common operations used to combine predicates, AND and OR, can be performed using very efficient instructions that gain a lot of parallelism by executing 32 or 64 bits in parallel on most modern processors. See [O'NEI91] for benchmark results where 100-fold performance advantages of bitmap indexes are measured for some single-table DSS queries.

One stumbling block to using bitmap indices in database systems is that they require an effective mapping between integers (bit positions) and the rows indexed. This is typically done through a row identifier, or RID, composed of a page number and a slot number within a page where the row is stored. While it is quite reasonable to assign an equal number of bits to consecutive pages to represent the rows on those pages on successive slots, most database systems support variable-length records by permitting a variable number of records per page. The solution to this problem is to define a maximal number of records per page, and reserve bits according to this maximal numbers, as is done very effectively in CCA's Model 204 database system, for example (see [O'NEI87]).

In a join index supporting a star-join, a bitmap can be created for each row in a descriptive table (lookup by RID or unique Key value) to represent the set of rows in a detail table that join with that row. More generally, it is possible to define a join index that will look up any non-unique column value in the descriptive table likely to be restricted in a star join, to find the bitmap of rows in the detail table that correspond to rows in the descriptive table with that column value.

3. Query Plans for Star-Joins

Based on these techniques, one can define the following processing strategy for a star-join. First, since they will be joined very frequently, it makes sense to maintain join indices between the detail and descriptive tables. Since each descriptive table will typically join with the detail table via a different column, a domain index does not really help here. So we assume that T_0 is the detail table and that the descriptive tables T_i (for $i = 1, ..., N$ for some N) join with the detail table via the column A_i which is key for T_i and foreign key for T_0, i.e., joining through predicate $T_0.A_i = T_i.A_i$. We postulate a join index called $T_0T_i.A_i$ with entries for each RID of T_i containing compressed bitmaps for all related rows in T_0. We expect to see star-join queries of the following form:

[1] select distinct $T_0.K$, $T_1.A_1$, $T_2.A_2$, . . . , $T_N.A_N$ from T_0, T_1, T_2, . . . , T_N
where $T_0.A_1 = T_1.A_1$ and $T_0.A_2 = T_2.A_2$ and . . . $T_0.A_N = T_N.A_N$
and $T_1.B_1 = C_1$ and $T_2.B_2 = C_2$ and . . . $T_N.B_N = C_N$;

where the columns B_i are non-key values of the tables T_i, and the C_i represent constant values. Let us further presume that for all likely selection predicates on $T_i.B_i = C_i$, there are indices for the columns $T_i.B_i$, and that there are also table indices for key columns, $T_i.A_i$. Join indices from $T_i.B_i$ to bitmaps of related rows of T_0 would be particularly valuable, but are not absolutely required.

Given these assumptions, a number of query execution strategies become possible. One of them seems particularly promising: First, for each descriptive table T_i, employ the index $T_i.B_i$ to find suitable rows for the predicate $T_i.B_i = C_i$; loop on all these rows and OR all bitmaps found through the join index $T_0T_i.A_i$ to create a bitmap for all related rows of table T_0. After doing this for each T_i, there will be N bitmaps on T_0, corresponding to each of the predicates $T_i.B_i = C_i$. Now, intersect the bitmaps so determined, and this will be the set of rows in T_0 satisfying all restrictions on joining tables T_i. In many cases the remaining set or rows in T_0 will be so small that those rows in T_0 can be joined inexpensively through foreign keys with all rows in the indexed descriptive tables T_i to obtain the final query result. Alternative strategies exist in cases where the set of rows in T_0 is large enough so that unordered references in the descriptive tables would be wasteful; in this case, foundsets on each of the T_i tables can be inexpensively restricted through the join index and then sorted in join order to derive the correct result.

A Performance Example

A common goal in decision support queries is to avoid scanning the very large detail table, or fetching a large number of rows from that table. If each of the predicates on the descriptive tables has a selectivity of 1%, or 10^{-2}, and if all those predicates are statistically independent of each other, *N* such predicates have a selectivity of 10^{-2N}. Even if the detail table is rather large, say 10^8 rows, and the number of descriptive tables with predicates rather small, say $N = 3$, the result after performing the bitmap technique given in the last Section will be that only 100 rows have to be fetched from the detail table. Clearly fetching 100 rows is faster than scanning 10^8 rows by a large margin, large enough to hide the surprisingly small overhead of searching the join indices of the descriptive tables and fetching descriptive table rows for the final assembly of the result.

Special circumstances can arise when the predicates that restrict the descriptive table do not significantly restrict the detail table, or when the number of rows retrieved in a detail table is so large that it is not cost effective to OR bitmaps of the join index T_i. To ensure the best decision support performance, the query optimizer has to make cost-based decisions among alternative execution plans. The point here is not to show that the execution techniques outlined are always superior to all alternatives, but to demonstrate that they commonly result in efficient executions so they are worthwhile to be taken into consideration by an optimizer.

Summary and Conclusions

In this brief technical note, we have outlined a query execution method for multi-table joins and exemplified it for so-called star-joins. The challenge facing database systems developers is not to implement any special, new, or complex query execution technique but to include suitable building blocks into their query execution engines and to guarantee that their optimizers will consider them when they are promising. To the best of our knowledge, no query optimizers with non-privileged strategies currently consider the plans outlined above.

References

[KOOI80] Robert Kooi, *The Optimization of Queries in Relational Databases,* Ph.D. thesis, Case Western Reserve University, Cleveland, OH, 1980.

[O'NEI87] Patrick O'Neil, *Model 204 Architecture and Performance*, Springer-Verlag Lecture Notes in Computer Science 359, 2nd International Workshop on High Performance Transactions Systems, Asilomar, CA, September 1987.

[O'NEI91] Patrick O'Neil, The Set Query Benchmark, The Benchmark Handbook for Database and Transaction Processing Systems, Jim Gray (Editor), 2nd Edition, 1993.

[VALD87] Patrick Valduriez, *Join Indices*, ACM TODS, Vol. 12, No. 2, June 1987, Pages 218-246.