A multiple join index for data warehouses

ADI-CRISTINA MITEA
Computer Science Department
"Lucian Blaga" University of Sibiu
Zaharia Boiu street No. 2-4, 2400 Sibiu
ROMANIA

Abstract: - The demands of today's software applications are more and more each day. They need to collect, manage and transform more and more data with a complex structure. These data are nowadays organized in data warehouses. To accelerate data mining searches that combine multiple restrictive queries indexing techniques are usually used. The paper proposes a new multiple join index structure, which is useful for multiple join operations between tables. The algorithm to build this index structure is presented and, also, some types of queries that can be accelerated with the index help.

Key-Words: - multiple join indexes, multiple join operations, data warehouses, relational databases

1 Introduction

A data warehouse is a complex information system used mainly for strategic decision making by using OLAP, data mining and knowledge discovery techniques. Data warehouses collect information from many different sources into a subject oriented, integrated, time variable and non-volatile data collection, used to support the decisional process of management ([1]). That information is normally extracted from operational business applications, is transformed and validated to verify some forms of data integrity and then is loaded into a specially designed database schema.

Data warehouses are usually placed on hardware platforms that claim high-performance query capability, both in terms of price-performance and response time, but this is not enough. It is important to use others techniques to improve its performance. One efficient way to attend this goal is to improve response time through indexes.

Data warehouses usually have a relational SQL interface. Indexing is becoming a common feature to accelerate data mining searches that combine multiple restrictive queries. New kinds of index structures were proposed over the time and some of them are already applied in databases. One of these new index structures is the bitmap index. Bitmap indexes create a vector of bits for each different value of the index key. The vector's bits correspond to the rows of the table and can be 1, if the index key value is present in that row, or 0 if it's not. Sparse bitmaps can be compressed as appropriate. Bitmap indexes can be static or dynamic. In Oracle 9*i* static and dynamic bitmap indexes are implemented ([2]). DB2 implements only dynamic bitmap indexes ([3]). SQL Server 2000 do not offer bitmap indexes ([4]). Reverse-key B-trees index, is another type of new index structure, which create the B-trees index using a reversed index key. This type of index is implemented in Oracle *9i* ([2]). Domain indexes create an index for a particular application domain providing efficient access to customized complex data types. Join indexes ([5]) greatly speed up joins by applying restrictions on one table as restrictions on another. Join indexes are implemented in Oracle *9i* ([2]). There are in literature another interesting solutions for indexes, which remains only at the stage of theory. Some of them could be found in ([5], [6], [7]).

The paper proposes a new approach for a join index. This type of index can be very useful for data warehouses, because it improves the performance of join operations performed between multiple tables.

2 Multiple join operations

A data warehouse usually store a huge volume of data, which are organized in fact tables and dimension tables. The dimension tables are linked to fact tables using a referential integrity constraint.

A "join index" is an index structure, which spans multiple tables, and improves the performance of joins between those tables. Typically, one would create a join index on a fact table, where the indexed column(s) would belong to a dimension table. A join index is the result of joining the fact table with a dimension table on a join attribute, which is the primary key for the dimension table and the foreign

key for the fact table, and projecting the index key on the result. To join the two tables means to use the join index to fetch only the tuples, which satisfy the join criteria, from the tables followed by a join of those tuples.

In relational data warehouse systems, it is of interest to perform a multiple join (a star join) on the fact tables and their dimension tables. To speed up some kind of multiple joins, the procedure used is to build join indexes between fact tables and each of their dimension tables implied in the multiple join. If the join indexes are represented in bitmap matrices, a multiple join could be replaced by a sequence of bitwise operations, followed by a relatively small number of fetch and join operations.

My proposal touches exactly that field: the multiple join between fact tables and their dimension tables.

3. The multiple join index

The index structure I propose is useful when multiple join operations between tables are needed. In a data warehouse a lot of queries need multiple join operations between tables, most of them between fact tables and their dimension tables. Because these tables store large volumes of data a join operation is a time-consuming one. To reduce the cost of performing such queries, I propose an index structure, which can be used to eliminate the need to perform the join operation or to minimize the join total cost.

A multiple join implies at least two join operations between a fact table and its dimension tables. The basic idea is to build the index on an index key made by a combination of columns from the dimension tables involved in the multiple join. Each dimension table from the multiple join has at least one of its columns in the index key.

The index is created using a command like:

```
CREATE JOIN INDEX <IndexName>
ON
<FactTable+DimensionTable1+DimensionT
able2+…>
(<IndexKeyColumn1, IndexKeyColumn2,…>)
WHERE
   FactTable.Column1=
DimensionTable1.Column1
 AND
   FactTable.Column2=
DimensionTable2.Column2
 AND
   ….
where
```

IndexKeyColumn1 belongs to DimensionTable1
IndexKeyColumn2 belongs to DimensionTable2
Column1 is the primary key for DimensionTable1 and a foreign key for the FactTable
Column2 is the primary key for DimensionTable2 and a foreign key for the FactTable

The index is build like a B-tree using the index key values. A leaf node contains the index key value and *n*- pointers, one pointer for each table involved in the multiple join. Every pointer points to a contiguous area containing information about one of the tables from the multiple join. This information corresponds to the index key value. If the index key has many different values ROWIDs are used. The pointer points to a list of ROWIDs, which indicates the rows from the table which have the same value as the index key. The first pointer points to the list of ROWIDs from the fact table, the second pointer points to the list of ROWIDs from the first dimension

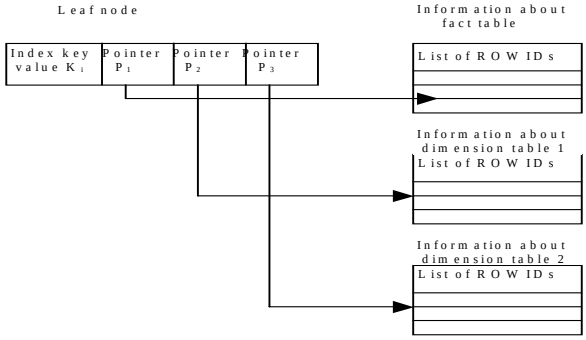

Fig.1. Index leaf node structure when ROWIDs are used.

table, the third pointer points to the list of ROWIDs from the second dimension table, and so on. This is illustrated in Fig.1. If the index key has few different values bitmaps are used. The leaf nodes contain, also, the index key value and *n*-pointers, one pointer for each table involved in the multiple join. The first pointer points to the bitmap of the fact table, the second pointer points to the bitmap of the first dimension table, the third pointer points to the bitmap of the second dimension table, and so on. The pointer points to a bitmap and also a start ROWID and an end ROWID are indicated. The start ROWID is the ROWID of the first row pointed to by the bitmap segment of the bitmap and the end ROWID is the ROWID of the last row in the table covered by the bitmap segment. This is illustrated in Fig.2.

The algorithm for building the index structure is composed from two major parts. The first one is concerned with finding the rows from the fact table, which satisfy the index key criteria, and the second

one is concerned with finding the rows from the dimension tables. The multiple join index is

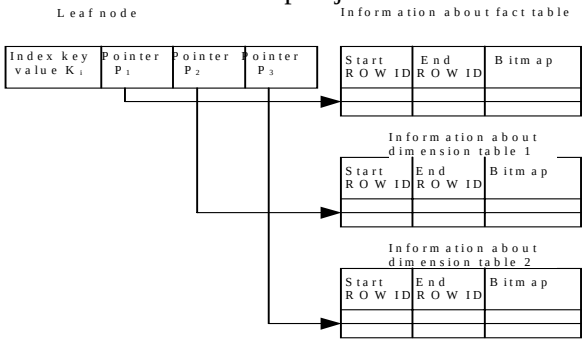

Fig. 2. Index leaf node structure when bitmaps are used.

constructed like a B-tree index using the values of the index key. The leaf nodes of the index contain pointers to different areas with information about rows from fact table and dimension tables, which correspond to the index key value. The algorithm for building the index structure in the case of lists of ROWIDs is partially presented below:

$List_{IndexKeys}$={} /*the list of index key values*/
$List_{FactTable}$={} /*the list of ROWIDs from Fact Table for a particular value of the index key*/
$List_{DimensionTable1}$={} /*the list of ROWIDs from DimensionTable1 for a particular value of the index key*/
$List_{DimensionTable2}$={} /*the list of ROWIDs from DimensionTable2 for a particular value of the index key*/
….
do while not eof ( ) $_{FactTable}$
fetch row from FactTable
fetch row from DimensionTable1 where
PrimaryKey $_{DimensionTable1}$ = ForeignKey1 $_{FactTable}$
fetch row from DimensionTable2 where
PrimaryKey $_{DimensionTable2}$ = ForeignKey2 $_{FactTable}$
…
n=ROWID $_{FactTable}$
IndexKey=(Column1$_{DimensionTable1}$,Column2$_{DimensionTable2}$,.)
if IndexKey is not in $List_{IndexKeys}$

/*there is a new value for the index key*/

do while not eof ( ) $_{FactTable}$
if
(Column1$_{DimensionTable1}$,Column2$_{DimensionTable2}$,.)=
IndexKey

/*the rows correspond to the index key value and are inserted in the lists*/

$List_{FactTable}$= $List_{FactTable}$ + ROWID$_{FactTable}$
$List_{DimensionTable1}$= $List_{DimensionTable1}$ + ROWID $_{DimensionTable1}$
$List_{DimensionTable2}$= $List_{DimensionTable2}$+ ROWID $_{DimensionTable2}$
….
skip$_{FactTable}$
if not eof ( ) $_{FactTable}$
fetch row from DimensionTable1 where
PrimaryKey$_{DimensionTable1}$=ForeignKey1$_{FactTable}$
fetch row from DimensionTable2 where
PrimaryKey$_{DimensionTable2}$=ForeignKey2$_{FactTable}$
…
endif
else

/*the index key value exist in $List_{IndexKeys}$ so the row is skiped*/
skip$_{FactTable}$
if not eof ( ) $_{FactTable}$
fetch row from DimensionTable1 where
PrimaryKey$_{DimensionTable1}$=ForeignKey1$_{FactTable}$
fetch row from DimensionTable2 where
PrimaryKey$_{DimensionTable2}$=ForeignKey2$_{FactTable}$
….
endif
endif
enddo

/*all the rows for that particular index key value are determined and they are written in their areas*/

write $List_{FactTable}$ in AreaFactTable
write $List_{DimensionTable1}$ in AreaDimensionTable1
write $List_{DimensionTable2}$ in AreaDimensionTable2
….

/*the lists are emptied*/

$List_{FactTable}$={}
$List_{DimensionTable1}$={}
$List_{DimensionTable2}$={}
goto row n
endif
skip$_{FactTable}$
enddo

If the bitmaps are used for the index structure the algorithm is a little bit different:

$List_{IndexKeys}$={} /*the list of index key values*/
$Bitmap_{FactTable}$=00…00 /*the bitmap has all the bits set to 0. The number of bits is equal with the number of rows from the FactTable*/

```
Bitmap_DimensionTable1=00…00   /*the bitmap has all the bits set to 0. The number of bits is equal with the number of rows from the DimensionTable1*/
Bitmap_DimensionTable2=00…00   /*the bitmap has all the bits set to 0. The number of bits is equal with the number of rows from the DimensionTable2*/
…

do while not eof ( ) FactTable
 fetch row from FactTable
 fetch row from DimensionTable1 where
    PrimaryKey DimensionTable1 = ForeignKey1 FactTable
 fetch row from DimensionTable2 where
    PrimaryKey DimensionTable2 = ForeignKey2 FactTable
 …
 n=ROWID FactTable

IndexKey=(Column1_DimensionTable1,Column2_DimensionTable2,.)
 if IndexKey is not in List_IndexKeys

/*there is a new value for the index key*/

   do while not eof ( ) FactTable
     if
(Column1_DimensionTable1,Column2_DimensionTable2,.)=
IndexKey

/*the rows correspond to the index key value, so the bits from the bitmaps are set to 1*/

      set bit corresponding to ROWID_FactTable from
         Bitmap_FactTable to 1
      set bit corresponding to ROWID_DimensionTable1
         from Bitmap_DimensionTable1 to 1
      set bit corresponding to ROWID_DimensionTable2
         from Bitmap_DimensionTable2 to 1
      ….
      skip_FactTable
      if not eof ( ) FactTable
        fetch row from DimensionTable1 where
        PrimaryKey_DimensionTable1=ForeignKey1_FactTable
        fetch row from DimensionTable2 where
        PrimaryKey_DimensionTable2=ForeignKey2_FactTable
        …
      endif
    else

/*the index key value exist in List_IndexKeys so the row is skiped*/

      skip_FactTable
      if not eof ( ) FactTable
        fetch row from DimensionTable1
where
PrimaryKey_DimensionTable1=ForeignKey1_FactTable
        fetch row from DimensionTable2
where
PrimaryKey_DimensionTable2=ForeignKey2_FactTable
                ….
      endif
     endif
   enddo

/*all the rows for that particular index key value are determined, so the bitmaps can be written in their areas*/

   write Bitmap_FactTable in AreaFactTable
   write Bitmap_DimensionTable1 in AreaDimensionTable1
   write Bitmap_DimensionTable2 in AreaDimensionTable2
   ….
/*the bitmaps are emptied*/

   Bitmap_FactTable=00…00
   Bitmap_DimensionTable1=00…00
   Bitmap_DimensionTable2=00…00
   goto row n
  endif
  skip_FactTable
enddo
```

To illustrate in a proper manner the way the multiple join index bill be constructed, lets take an example. Suppose there is a data warehouse, which has a fact table SALES and three dimension tables PRODUCTS, CUSTOMERS and TIMES (Fig.3).

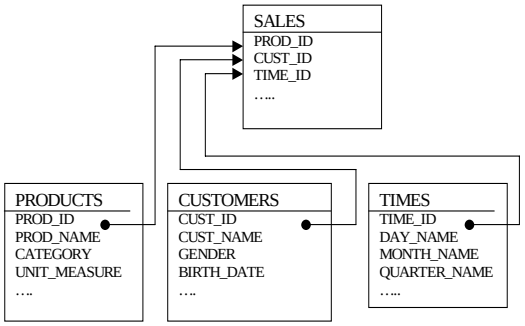

Fig. 3. The fact table and its dimension tables.

A short example of data for these tables is presented in Fig.4. A multiple join index can be constructed on the SALES, PRODUCTS and CUSTOMERS tables using the index key (CATEGORY, GENDER).

Relation R = PRODUCTS

ROWID	PROD_ID	PROD_NAME	CATEGORY	UNIT_MEASURE	…
R1	P1		CAT1		

R2	P2		CAT2		
R3	P3		CAT3		
R4	P4		CAT1		
R5	P5		CAT2		
R6	P6		CAT3		
R7	P7		CAT2		

Relation S = CUSTOMERS

ROWID	CUST_ID	CUST_NAME	GENDER	BIRTH_DATE	…
S1	C1		M		
S2	C2		M		
S3	C3		F		
S4	C4		M		
S5	C5		F		
S6	C6		F		
S7	C7		M		

Relation T = SALES

ROWID	PROD_ID	CUST_ID	TIME_ID	….
T1	P1	C1		
T2	P2	C3		
T3	P1	C4		
T4	P1	C1		
T5	P4	C3		
T6	P3	C4		
T7	P6	C5		
T8	P7	C5		
T9	P7	C6		
T10	P1	C3		

Fig. 4. Data examples.

The index will be created using the command:

```
create join index INDEX01
on SALES+PRODUCTS+CUSTOMERS
(CATEGORY, GENDER)
where SALES.PROD_ID=PRODUCTS.PROD_ID
and SALES.CUST_ID=CUSTOMERS.CUST_ID
```

If the index key selectivity is low the index structure will contain bitmaps. For the given example, the bitmaps will be as in Fig.5.

Index key value	T Start ROWID	T End ROWID	T Bitmap	R Start ROWID	R End ROWID	R Bitmap	S Start ROWID	S End ROWID	S Bitmap
CAT1F	T5	T10	0000100001	R1	R4	1001000	S3	S3	0010000
CAT1M	T1	T4	1011000000	R1	R1	1000000	S1	S4	1001000
CAT2F	T2	T9	1000001100	R2	R7	0100001	S3	S6	0010110
CAT3F	T7	T7	0000001000	R6	R6	0000010	S5	S5	0000100
CAT3M	T6	T6	0000010000	R3	R3	0010000	S4	S4	0001000

Fig. 5. The bitmaps from the index.

This type of index eliminate the need to perform join operations between tables or to make bitwise operations between bitmaps from different join indexes, for queries like:

"How many products from category X were sold to customers with gender Y?"
"How many products from category X were sold to customers?
"How many products were sold to customers with gender Y?"
"How many products were sold?"
"How many customers with gender Y bought products from category X?
"How many customers with gender Y bought products?"
"How many customers bought products from category X?"
"How many customers bought products?"
"How many sales were made from products with category X to customers with gender Y?"
"How many sales were made from products with category X?
"How many sales were made to customers with gender Y?"

In such cases, the answer can be obtained directly from the index. For example, for the first query the answer is equivalent with number of bits set to "1" from the R bitmap.

The index also eliminate the need to perform join operations between tables and reduce the cost of query processing for queries like:

"Which are the products from category X sold to customers who has Y gender?"
"Which are the products from category X sold to customers?"
"Which are the products sold to customers who has Y gender?"
"Which are the products which were sold?"
"Who are the customers with gender Y who bought products from category X?"
"Who are the customers who bought products from category X?"
"Who are the customers with gender Y who bought products?"
"Who are the customers who bought products?"
"When products with category X were sold to customers with gender Y?"
"When products with category X were sold?"
"When were made the sales to customers with gender Y?"

In these cases, the index is used to determine the rows which satisfy the searching criteria and after that the rows are fetched from their tables.

For example, for the first query the answer is obtained from the R bitmap of index key value *XY*. The query results are the rows fetched from table R who has the bit set to 1 in the bitmap.

There are, also, other queries, which can benefit from this new type of join index.

Now, I am in the implementation phase with this new kind of multiple join index. When the tests will be finished the final conclusions can be made. For now, is clear that for some kind of queries that type of index provide better performance than simple join indexes, which need to be combined to obtain the result. The time needed to create this new multiple join index structure is longer than the time needed for a simple join index, but this is not crucial because in data warehouse environments the insert, update and delete operations are not performed very often, so the index structure is not affected. The selects are the basic operations and they can benefit from this new kind of join index.

4. Conclusion

This paper presents a new type of join index, which can be used for multiple join operations between tables in a relational database. For some queries the index will eliminate the need to perform the join operation, saving in this manner a lot of time and improving system performance. In some cases the query result can be obtained directly from the index and the improvements are substantial. In other cases, the query result is obtained faster because bitwise operations between join indexes and join operations between tables are eliminated providing a performance improvement. For other kind of queries the index will reduce the time needed to perform the join operation, obtaining also a system performance improvement.

This kind of index structure can be interesting because it brings together, under one single roof, information from different tables which are searched together using the same search criteria. This index is bigger than a traditional join index, but it eliminates the need to combine the results of several indexes to obtain the final result. System performance can be improved in this manner. The number of queries, which can find answers directly from index, is greater, also.

The process of testing this new type of join index had to be continued, to analyze all advantages or disadvantages of this new kind of join index.

References:

[1] S. Anahory, D. Murray, *Warehousing in the real world: A practical guide for building decision support systems* (Harlow: Addison Wesley Longman, 1997).

[2] *Oracle9i Concepts.* Oracle Corporation, Release 9.1.2, 2002.

[3] *DB2 Concepts.* IBM Corporation, Release UDB V8.1, 2002.

[4] *SQL Server 2000 Concepts.* Microsoft Corporation, 2000.

[5] P. Valduriez, Join indices, *ACM Transactions on Database Systems, 12*(2), 1987, 218-246.

[6] A. Datta, K. Ramamritham, H. Thomas, Curio: A novel solution for efficient storage and indexing in data warehouses, *Proceedings of the International Conference on Very Large Databases*, 1999, 730-733.

[7] P. O'Neil, D. Quass, Improved query performance with variant indexes, *Proceeding of ACM SIGMOD International Conference on Management of Data*, 1997, 38-49.