A NEW INDEXING TECHNIQUE FOR DATA WAREHOUSES

Adi-Cristina Mitea

Department of Computer Science, "Lucian Blaga" University of Sibiu
Zaharia Boiu street No. 2 – 4, Sibiu, Romania

Abstract: Data warehouses are special-purpose databases to support decision-making. In data warehouses indexing is becoming a common feature to accelerate data mining searches that combine multiple restrictive queries. Different types of indexes had been proposed and some of them are already implemented. This paper proposes a new kind of join index structure, which can be useful for multiple join operations. The algorithm to build this type of index structure is presented. Some queries, which benefit from this kind of join index, are also presented.

Keywords: indexing techniques, data warehouse, multiple join operations, relational databases.

1. INTRODUCTION

Data warehouses are special-purpose databases to support decision-making. Data warehouses collect information from many sources into a single database. 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. One efficient way to improve response time is through indexing techniques.

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 kind of index structures was proposed and applied in data warehouses, such as bitmap index, reverse-key B-trees index, domain index, join index (Datta *et al.*, 1999), (O'Neil *et at.*, 1997). Bitmap indexing creates a vector of bits for each row of the table; sparse bitmaps can be compressed as appropriate. Reverse-key B-trees indexes create the B-trees index using a reversed index key. Domain indexes create an index for a particular application domain providing efficient access to customized complex data types. Join indexes (Valduriez, 1987) greatly speed up joins by applying restrictions on one table as restrictions on another.

2. THE PROBLEM TO BE SOLVED

A data warehouse usually store a huge volume of data, which is 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 of the 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 two tables on a join attribute and projecting the keys of the two tables. 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 PROPOSAL

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 one of its columns in the index key.

The index is created using a command like:

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

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

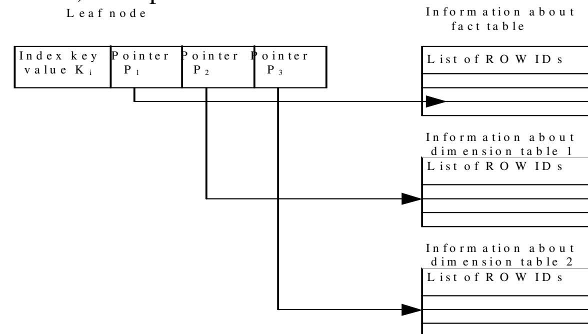

Figure 1. Index leaf node structure when ROWIDs are used..

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 low selectivity 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. This is illustrated in figure1. If the index key has high selectivity bitmaps are used. 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 (figure 2).

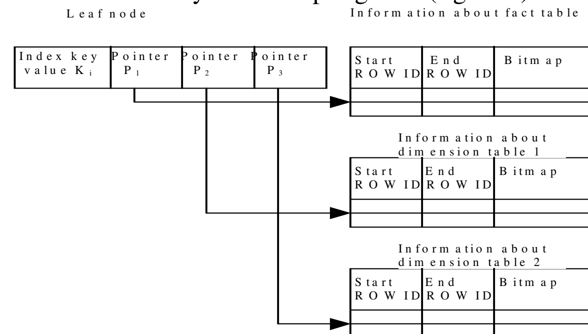

Figure 2. Index leaf node structure when bitmaps are used.

The algorithm for building the index structure in the case of lists of ROWIDs is as follow:

```
TEMP=

JOIN(FactTable,DimensionTable1,DimensionTable2,..
)
IndexKey=(Column1, Column2,..)

/* obtain the list of ROWIDs from the fact table for
each different value of the index key*/
```

```
List_{FactTable}={}
do while not eof()_{TEMP}
  if ((Columnl_{TEMP},Column2_{TEMP},..)=IndexKey)
        and (not eof()_{TEMP})
    List_{FactTable}=List_{FactTable}+ROWID_{FactTable}
    skip_{TEMP} 1
  else
    skip_{TEMP} 1
  endif
enddo
```

/* obtain the list of ROWIDs from the dimension table for each different value of the index key*/
/* this part of the algorithm is executed once for each dimension table participating in the multiple join*/

```
List_{DimensionTable}={}
for i=1 to NList_{FactTable}
  begin
    fetch row with ROWID_i from FactTable
    read ForeignKey_{FactTable}
    find row j from DimensionTable where
          PrimaryKey_{DimensionTable}=ForeignKey_{FactTable}
    List_{DimensionTable}=List_{DimensionTable}+ROWID_{DimensionTable}
  end
```

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

/* determine the bitmaps, the start ROWIDs and the end ROWIDs from the fact table for each different value of the index key*/

```
Bitmap_{FactTable}=''
StartROWID_{FactTable}=s
EndROWID_{FactTable}=e
Start=0
do while not eof()_{TEMP}
  if ((Columnl_{TEMP},Column2_{TEMP},..)=IndexKey)
        and (not eof()_{TEMP})
    if Start=0
      StartROWID_{FactTable}=ROWID_{FactTable}
      Start=1
    endif
    Bitmap_{FactTable}=Bitmap_{FactTable}+'1'
    EndROWID_{FactTable}=ROWID_{FactTable}
    skip_{TEMP} 1
  else
    Bitmap_{FactTable}=Bitmap_{FactTable}+'0'
    skip_{TEMP} 1
  endif
enddo
```

/* determine the bitmaps, the start ROWIDs and the end ROWIDs from the dimension table for each different value of the index key*/

/* this part of the algorithm is executed once for each dimension table participating in the multiple join*/

```
StartROWID_{DimensionTable}=s
EndROWID_{DimensionTable}=e
Bitmap_{DimensionTable}='000..00'
/*number of 0 bits is equal to number of rows of DimensionTable*/
for i=1 to NRow_{FactTable}
  if bit_{iBitmapFactTable}='1'
    fetch row with ROWID_i from FactTable
    read ForeignKey_{FactTable}
    find row j from DimensionTable where
          PrimaryKey_{DimensionTable}=ForeignKey_{FactTable}
    Bitmap_{DimensionTable}[j]='1'
    if StartROWID_{DimensionTable}> ROWID_{DimensionTable}
      StartROWID_{DimensionTable}= ROWID_{DimensionTable}
    endif
    if EndROWID_{DimensionTable}< ROWID_{DimensionTable}
      EndROWID_{DimensionTable}= ROWID_{DimensionTable}
    endif
  end
```

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 (figure 3).

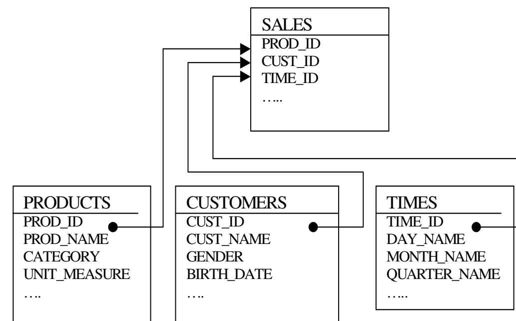

Figure 3. The fact table and its dimension tables.

A short example of data for these tables is presented in figure 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_	CUST_	GENDER	BIRTH_	…

	ID	NAME		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		

Figure 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 high the index structure will contain bitmaps. For the given example, the bitmaps will be as in figure 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
CAT1 M	T1	T4	1011000000	R1	R1	1000000	S1	S4	100100
CAT1F	T5	T10	0000100001	R1	R4	1001000	S3	S6	001011
CAT2F	T2	T9	0100000110	R2	R7	0100001	S3	S3	001000
CAT3 M	T6	T6	0000010000	R3	R3	0010000	S2	S2	001000
CAT3F	T7	T7	0000001000	R6	R6	0000010	S5	S5	000010

Figure 5. The bitmaps from the index.

This type of index eliminate the need to perform join operations between tables 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?"

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

The index also reduce the cost of join operations 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?"
"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?"

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.

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. 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

(1992) *Oracle7 Server Concepts Manual.* Oracle Corporation, Redwood City, CA.

(1999) *Oracle8i Concepts.* Oracle Corporation, Release 8.1.5.

(2002) *Oracle9i Concepts* .Oracle Corporation, Release 9.1.2.

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

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

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