The Dimension-Join: A New Index for Data Warehouses

Pedro Bizarro and Henrique Madeira
University of Coimbra, Portugal
Dep. Engenharia Informática – CISUC
3030-397, Coimbra – Portugal
bizarro@dei.uc.pt, henrique@dei.uc.pt

Abstract

There are several auxiliary pre-computed access structures that allow faster answers by reading less base data. Examples are materialized views, join indexes, B-tree and bitmap indexes. This paper proposes dimension-join, a new type of index especially suited for data warehouses. The dimension-join borrows ideas from several concepts. It is a bitmap index, it is a multi-table join and when being used one of the tables is not read to improve performance. It is a multi-table join because it holds information belonging to two tables, which is similar to the join index proposed by Valduriez. However, instead of being composed by the tables' primary keys, the dimension-join index is a bitmap index over the fact table using values from a dimension column. The dimension-join index is very useful when selecting facts depending on dimension tables belonging to snowflakes. The dimension-join represents a direct connection between the fact table and a table in the snowflake that can avoid several joins and produce enormous performance improvements. This paper also evaluates experimentally the dimension-join indexes using the TPC-H benchmark and shows that this new index structure can dramatically improve the performance for some queries.

1. Introduction

Data warehouses (DWs) often grow to sizes of gigabytes or terabytes of information, which makes the performance of the queries issued by decision support tools one of the most important issues in Data Warehousing. In logical terms, a data warehouse is organized according to the multidimensional model. Each dimension of this model represents a different perspective for the analysis of the business. For instance, in the classical example of a chain of stores, some of the dimensions are products, stores, and time. Each cell within the multidimensional structure (a cube in this three dimension example) contains data (typically numerical facts). For example, a single cell may contain the total sales for a given product in a given store in a single day.

Although the data in a data warehouse could be stored in a multidimensional database server (e.g., Oracle Express server), most of the data warehouses and OLAP applications store the data in a relational database. That is, the multidimensional model is implemented as one or more star schemes. Each star scheme consists in a large central fact table surrounded by several dimensional tables, which are related to the fact table by foreign keys [1].

Fact tables store business measures (profit, units, etc) and have millions of records or more. These tables normally have few (6 to 15) attributes. Dimension tables store information characterizing the facts. Examples of typical dimensions are stores, suppliers, products, time, etc. Dimensions are often denormalized, have few rows (hundreds or thousands) and many columns (10 to 60). The central fact table is usually normalized, since it is the biggest table and any redundancy would lead to excessive space allocation.

Typical queries join facts and some of the dimensions and the results are limited normally by restrictions imposed in the dimensions. Because joining tables is such a heavy operation and since the fact table is enormous, several solutions have been proposed in the literature to optimize joins (see the related work section for references).

Some of the algorithms that implement joins use auxiliary structures and some do not. In general, the ones that do not use auxiliary structures are good choices for *ad hoc* queries but they normally present worse performances. Algorithms using auxiliary structures have better performances but the data warehouse administrator has to choose which auxiliary structures to use since the alternatives are plenty.

This paper reviews the most common joining algorithms and proposes a join using a new type of index, the dimension-join index based on the join index proposed by Valduriez [2] and resorting to function based-indexes, a feature available in Oracle DBMS. The Valduriez join index makes the joins faster while the proposed dimension-join index avoids run-time joins (like materialized views [3]) by storing the primary key of one table and a column of another table. Basically, the Valduriez method reads data from the first table, then reads data from the join-index and finally reads data from the second table. With the dimension-join, the first table is never read because the desired values are in the index. This improvement significantly improves the performance.

The remainder of this paper is organized as follows: section 2 presents related work regarding pre-computed access structures and algorithms for *ad hoc* joins. Section 3 reviews the currently used cached structures to aid joins. Section 4 presents some techniques that better use pre-computed structures. Section 5 presents the dimension-join along with a motivation example. Section 6 consists of experiment details and performance figures and section 7 ends the paper with the conclusions and future work.

2. Related work

There are access methods that do not require any auxiliary structure. These algorithms are especially suited for *ad hoc* joins because the administrator does not need to guess anything about what data is the user going to ask for. Examples are the nested block join [4], sort-merge join [4], simple hash join [4], Grace hash join [5], hybrid hash join, jive-join [6], slam-join [6] and diag-join [7]. Although users of decision support systems normally pose *ad hoc* queries, these algorithms alone are not enough to ensure an acceptable performance. Some papers refer to them as last resort algorithms [8] because they are slower then the algorithms that use auxiliary structures.

On the other hand, pre-computed or cached access structures return answers without reading (partially or totally) the base data in the tables providing a faster answer. However, these structures need to be maintained; i.e., if the base data changes, the structures must be updated or rebuilt. Examples are materialized views [3], join indexes [2], B-trees [9] and bitmap indexes. To the data warehouse administrator, one of the most difficult tasks regarding using pre-computed structures is to decide how to use them: which ones to use and where to use them (which joins, which columns, should every possibility be covered?, etc). The task is particularly difficult because the administrator cannot create materialized views for every possible query.

See [10] for a thorough analysis of several join algorithms. See [11] for a performance comparison between join indexes, materialized views and hybrid-hash joins. See [12] for a better cost model to compare *ad hoc* joins and for a performance comparison between nested

block join, sort-merge join, simple hash join, Grace hash join, and hybrid hash join. See [13] and [14] (page 144) for a comparison between B-tree and bitmap indexes.

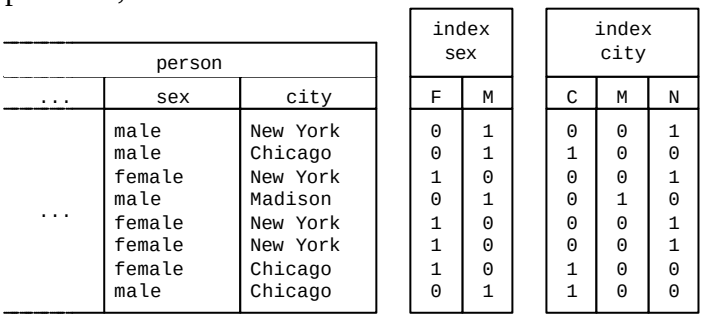
3. Pre-computed or cached access structures

3.1. B-tree index

The B-tree index is probably the most used pre-computed access structure in database systems. A B-tree is a balanced tree with each node having hundreds of connections to more nodes in the next level of the tree. The index is constructed and ordered with values of a column. The leaf nodes point to rows in the table where the base data is read. B-trees are very good in environments with mixed reading and writing operations, concurrency, exact searches and range searches. B-tree indexes are relatively expensive auxiliary structures to maintain since each one may take up as much space as 20% or more as the table. B-tree indexes present excellent performance figures when used to find few rows in big tables.

3.2. Bitmap index

A bitmap index is a special kind of index consisting in arrays of bits. Each array represents one of the values in the indexed column and the bit position in the array corresponds to the row position in the table. Bitmap indexes are especially good when used in columns with low cardinality and systems with low concurrency, few updates and searches with Boolean operations. Bitmap indexes are very frequent in data warehouses since all of these conditions are found there. A bitmap occupies much less space than a correspondent B-tree index over the same column if the column has low cardinality. Figure 1 depicts a table `person` and 2 bitmap indexes, one on column `sex` and the other on column `city`.



person

...	sex	city
...	male	New York
	male	Chicago
	female	New York
	male	Madison
	female	New York
	female	New York
	female	Chicago
	male	Chicago

index sex

F	M
0	1
0	1
1	0
0	1
1	0
1	0
1	0
0	1

index city

C	M	N
0	0	1
1	0	0
0	0	1
0	1	0
0	0	1
0	0	1
1	0	0
1	0	0

Figure 1: Bitmap indexes

Bitmap indexes require low cardinality columns because the more distinct values there are the biggest the bitmap becomes. Note that in Figure 1, the bitmap over `city` is 50% bigger than the bitmap over `sex` solely because there are more distinct values of `city` than there are of `sex`. There are already algorithms that use compressed bitmap when they became too sparse [15].

3.3. Materialized view

A materialized view is a very expensive yet very fast pre-computed access method. It is very expensive because it stores the result of a query, which can be quite big. The first time the query is asked the server stores its rows. Further calls to the same query retrieve the

already saved result. Of course, this procedure is only cost effective if between several calls the base data remains the same avoiding rebuilding the view. Storing just a few materialized views may take as much space as the all other data warehouse tables together. On the other hand, a materialized view is very fast because no join needs to be made at run time on successive calls; everything that is left to do is to read the pre-computed view. Thus, the great advantage of materialized views is that they allow the database system to avoid reading the base data. All the information needed to answer the query is already in the view.

3.4. Join index

The join index, proposed by Valduriez [2], is an index that helps to join rows from two relations, say R and S. By definition, the join index (JI) is created by joining R and S and projecting the result on R and S primary keys[1]. Figure 2 shows two relations, person and pet, and its respective join index.

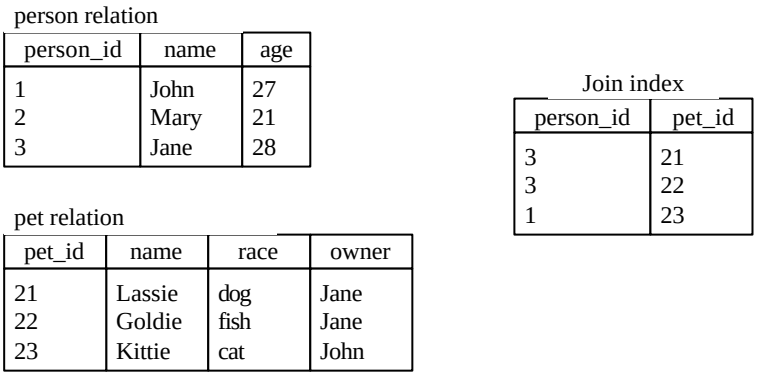


Figure 2: Join index

For performance reasons, Valduriez also proposed that the JI should be implemented as two B-tree indexes. Considering the example of Figure 3, one of the B-trees composing the JI is accessed by `person_id` with leaf nodes pointing to `pet_id` and the other B-tree is accessed by `pet_id` with leaf nodes pointing to `person_id`.

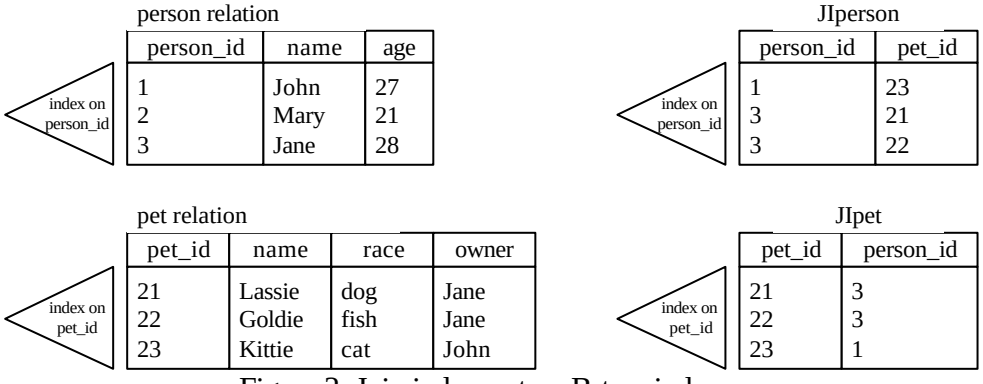


Figure 3: Join index as two B-tree indexes

Each B-tree of the join index (`JIperson` and `JIpet` in the example) assists for a join order. If the DBMS is reading the `person` relation first and than the `pet`, it will: 1) read the

[1] Actually, Valduriez identified rows using surrogates instead of primary keys. Surrogates are system unique identifiers while primary keys are table unique identifiers.

person relation and find which rows to take to the next step; 2) read JIperson to find which pets belong to those persons in 1); 3) and finally, with the pet_ids from JIperson, the DBMS will read just the right rows in the relation pet. If the DBMS is joining the relations in the reverse order it will use JIpet instead.

In short, a join index is a structure where the search is made using values from a table and the results point to rows of another table.

4. Techniques using pre-computed structures

4.1. Reading only the index

Sometimes, the optimizer can decide that after reading the index it is not necessary to read the indexed table. This happens if all the information needed is available already at the index. Consider the tables showed in Figure 1. The query that finds the persons with pets is:

```
SELECT person.name
  FROM person, pet
 WHERE pet.owner = person.name;
```

If the optimizer knows about the join index, all it needs to do is read every row in person and than search in JIperson a row matching that person_id. It won't need to search for anything more in pets because the SELECT clause does not specify any information regarding pets. The dimension-join uses such improvement. In fact, the whole point of a dimension-join is to use just one table and the index instead of reading and joining two tables.

4.2. Star Joins

One very interesting technique to optimize joins in star schemes is the star join described in [16]. Basically, there is a bitmapped join index between the fact table and every dimension table. When a search is executed, the DBMS reads the dimensions first, then it finds their correspondent entries in its bitmapped join index. Finally, it processes (ORs and ANDs) all bitmapped join indexes to find the rows in the fact table. The gain produced by this approach comes from processing several indexes before reading any data in the fact table, thus restricting the amount of rows retrieved in intermediate steps.

4.3. Star transformation

Oracle implements its own version of the star join. They call star join to a technique that consists in producing a Cartesian join between the selected rows of each dimension table and then, using it to find the desired rows in the fact table. This version is worse than the star join of [16] because the intermediate Cartesian join can became large if the restrictions are not selective enough. With the original star join, the join is made with bitmap indexes instead of tables, and *joining* two bitmaps still produces a bitmap with the same size. Recognizing that their star join could lead to large Cartesian products ([17], Chapter 20, section Optimizing "Star" Queries, sub-section Star Transformation) Oracle implemented another join method to star schemes resembling the original star join definition. They call it star transformation: "The star transformation is based on combining bitmap indexes on individual fact table columns".

However, star transformation rewrites the queries in order to find the entries in each bitmap join index. A query like:

```
SELECT …
  FROM fact, dim1, dim2, …, dimN
 WHERE fact.fk_dim1 = dim1.pk            /* joins       */
   …   …
   AND fact.fk_dimN = dimN.pk            /* joins       */
   AND dim1.attribute = value1           /* restriction */
   …   …
   AND dimN.attribute = valueN           /* restriction */
```

Is rewritten into:

```
SELECT …
  FROM fact, dim1, dim2, …, dimN
 WHERE fact.fk_dim1 IN (SELECT pk        /* joins       */
                          FROM dim1
                         WHERE = value1) /* restriction */
   …   …
   AND fact.fk_dimN IN (SELECT pk        /* joins       */
                          FROM dimN
                         WHERE = valueN) /* restriction */
```

5. The Dimension-join

The dimension-join borrows ideas from several concepts. It is a bitmap index, it has information belonging to two tables like a join index and when being used one of the tables is not read. It can also be used like the star-join in the sense that several indexes can be processed before reading any table. The dimension-join also requires the optimizer to rewrite queries as with star-transformation.

5.1. TPC-H, a motivation example

As a motivating example consider the TPC benchmark H scheme [18] presented in Figure 4. The TPC Benchmark H (TPC-H for short) is a decision support benchmark produced by the TPC Council [19]. It consists of a set of tables and business oriented *ah hoc* queries. Both queries and data have been chosen to have broad industry relevance. The goal is to provide performance results to industry users regarding decision support implementations.

The TPC-H scheme consists in eight tables and their relations. Data can be scaled up to model businesses with different data sizes. The scheme and the number of rows for each table at scale 1 are presented in Figure 4.

TPC-H represents a retail business. Customers order products, which can be bought from more than one supplier. Every customer and supplier belongs to a nation, which is located in a region. The central fact table is LINEITEM although PARSUPP can also be considered a fact table. The dimensions are PART, SUPPLIER and ORDERS. There are two snowflakes, ORDERS? CUSTOMER? NATION? REGION and SUPPLIER? NATION? REGION. To find out to which region some LINEITEM was sold it is necessary to read data from all the tables in the first snowflake.

TPC-H query number 7, called Volume Shipping, finds, for two given nations, the gross discount revenues derived from LINEITEMs in which Parts where shipped from a Supplier in either Nation to a Customer in the other Nation during 1995 and 1996. Two nations are given as input parameters. The full query 7 text in Oracle's SQL dialect is:

```
 1: SELECT
 2:   supp_nation,
 3:   cust_nation,
 4:   l_year,
 5:   sum(volume) revenue
 6: FROM
 7:   (
 8:     SELECT
 9:       n1.n_name supp_nation,
10:       n2.n_name cust_nation,
11:       to_char(l_shipdate, 'YYYY') l_year,
12:       l_extendedprice * (1 - l_discount) volume
13:     FROM
14:       supplier,
15:       lineitem,
16:       orders,
17:       customer,
18:       nation n1,
19:       nation n2
20:     WHERE
21:           s_suppkey   = l_suppkey
22:       AND o_orderkey  = l_orderkey
23:       AND c_custkey   = o_custkey
24:       AND s_nationkey = n1.n_nationkey
25:       AND c_nationkey = n2.n_nationkey
26:       AND (
27:               (    n1.n_name = 'FRANCE'
28:                AND n2.n_name = 'GERMANY')
29:            OR (    n1.n_name = 'GERMANY'
30:                AND n2.n_name = 'FRANCE'))
31:       AND l_shipdate
32:           BETWEEN to_date('1995-01-01', 'yyyy-mm-dd')
33:               AND to_date('1996-12-31', 'yyyy-mm-dd')
34:   ) shipping
35: GROUP BY
36:   supp_nation,
37:   cust_nation,
38:   l_year
39: ORDER BY
40:   supp_nation,
41:   cust_nation,
42:   l_year;
```

Listing 1: TPC-H query number 7 code – Volume Shipping

Figure 5 represents a visual representation of query 7. The lighter attributes are the ones that are used either as input or output. The darker attributes are needed only to reach the data, to perform the joins along the snowflakes. Note that the query reads `N_NAME` twice, one

represents the SUPPLIER's nation and the other is the CUSTOMER's nation and each one is connected by LINEITEM through a different snowflake.

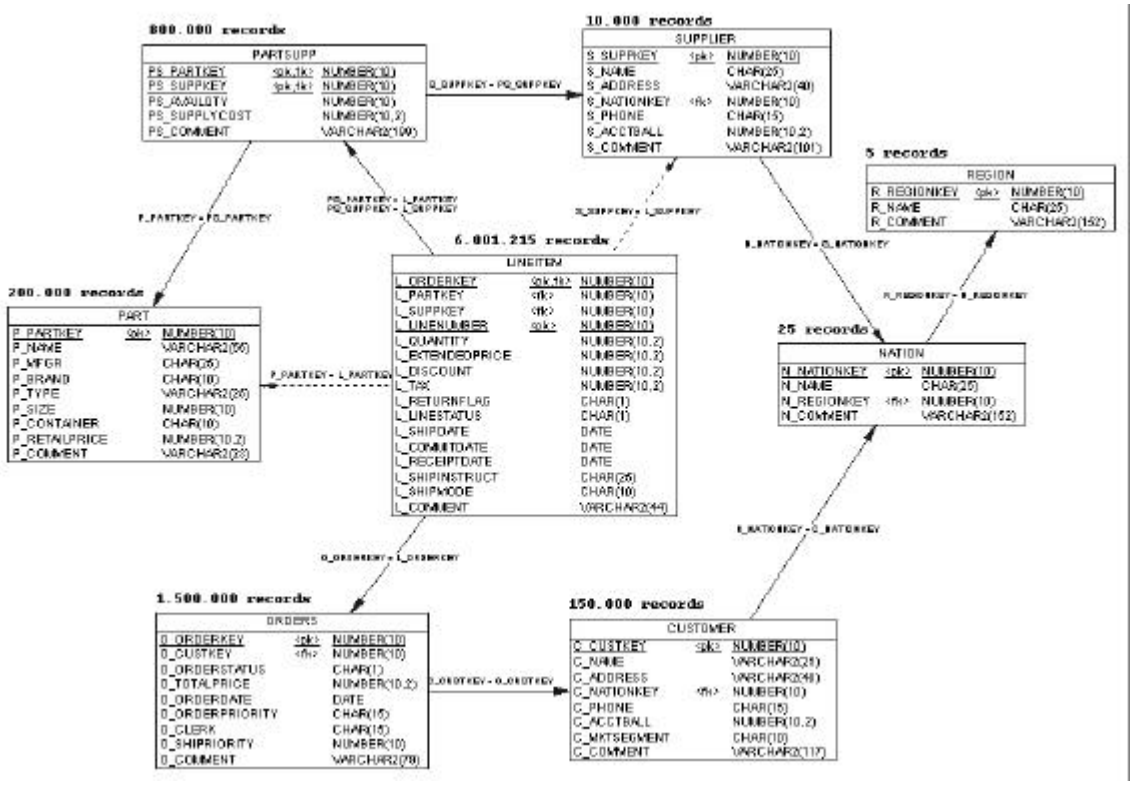


Figure 4: The TPC-H scheme

As Figure 5 reveals, the performance problem with this query is that the desired attributes, from NATION and LINEITEM, are too further apart from each other. The query will have to read data from ORDERS, CUSTOMER and SUPPLIER only to reach NATION. The dimension-join will bring NATION closer to LINEITEM.

5.2. Dimension-join index

The dimension-join is a bitmap index over a fact table with values from a dimension table. The tables must be connected with one or more (the case of a snowflake) *many to one* relationships. In the example of Figure 5, one ideal dimension-join index should be constructed over LINEITEM rows mapping values of N_NAME from NATION. In fact, considering query 7, two dimension-join indexes are in order: one mapping the CUSTOMER's nation and the other mapping SUPPLIER's nation. Both indexes should be constructed over LINEITEM, i.e., the bits should point to rows in LINEITEM.

The dimension-join index occupies little space because it is a bitmap index. It can, however lead to big performance improvements because it allows avoiding several joins. Figure 6 depicts a dimension-join index over rows of LINEITEM with information regarding each row CUSTOMER's NATION.

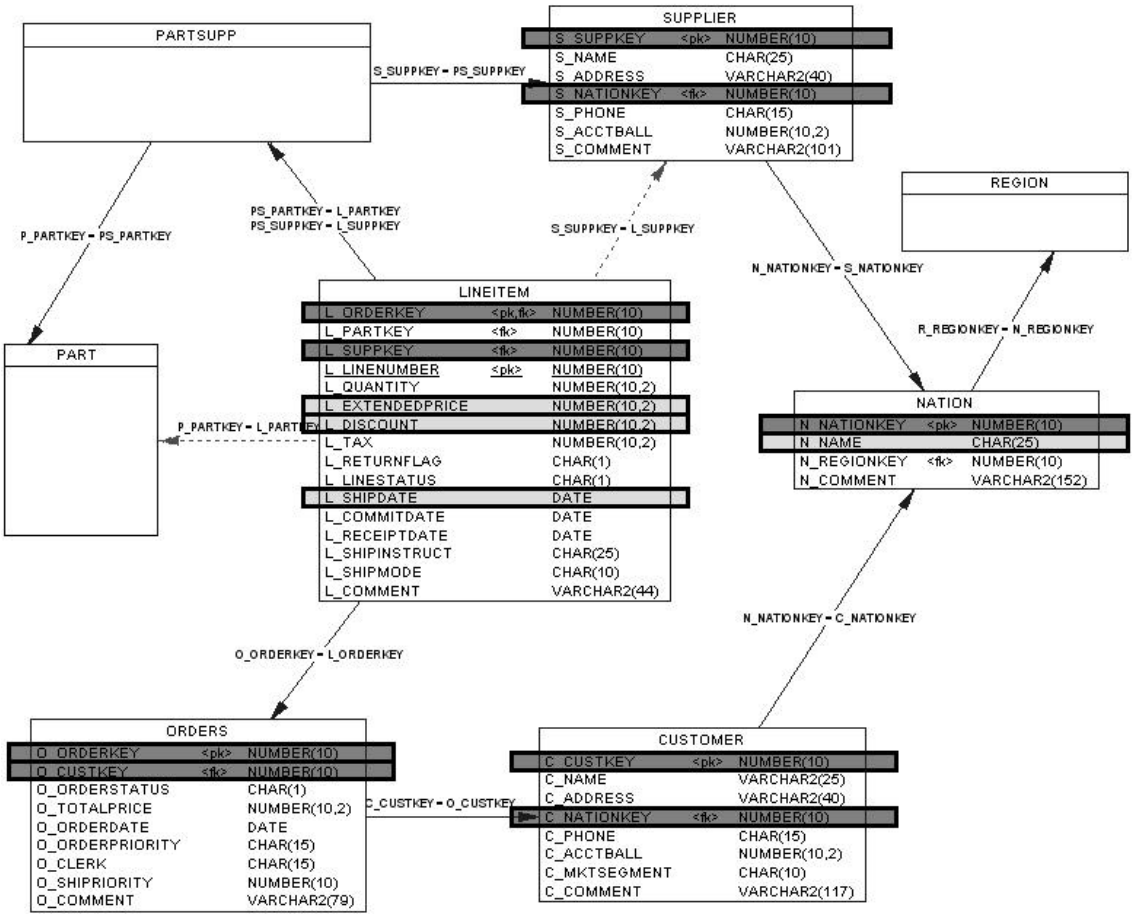


Figure 5: TPC-H query number 7 – Volume Shipping

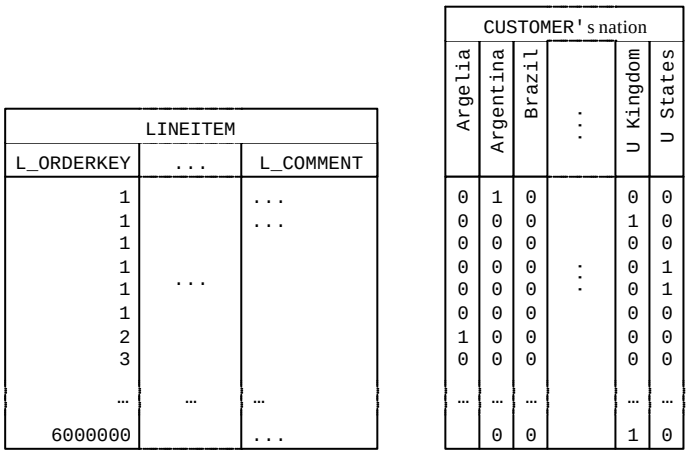


Figure 6: `LINEITEM` and dimension-join index on `CUSTOMER`'s nation

6. Results

To test query 7 we built two dimension-join indexes: IDX_CUST, which relates customers' nations with LINEITEM's tuples and IDX_SUPP, which relates suppliers' nations to LINEITEM's tuples. In our implementation query 7 uses both indexes instead of

performing several run-time joins. At first, we wanted to use only TPC-H queries but eventually we had to devise new queries to simulate different usage profiles. Essentially we wanted to have queries using only one or the other dimension-join indexes. We construct two new queries 7cust and 7supp, which use only the IDX_CUST or the IDX_SUPP dimension-join index.

Query 7cust was

```
SELECT
  to_char(l_shipdate, 'YYYY') l_year,
  sum(l_extendedprice * (1 - l_discount)) volume
FROM
  lineitem,
  orders,
  customer,
  nation
WHERE
      o_orderkey  = l_orderkey
  AND c_custkey   = o_custkey
  AND c_nationkey = n_nationkey
  AND n_name = 'GERMANY'
GROUP BY
  to_char(l_shipdate, 'YYYY');
```

Query 7supp was:

```
SELECT
  count(*) qtd
FROM
  lineitem,
  supplier,
  nation
WHERE
      s_suppkey   = l_suppkey
  AND s_nationkey = n_nationkey
  AND n_name = 'FRANCE';
```

To access how well dimension-indexes perform we measured total response time, space required by the structures and the time lost in data loads.

6.1. Total response time

For query 7, and without the dimension-join, every row in `LINEITEM` had to be joined with `ORDERS`, `CUSTOMER` and `NATION` to find its `CUSTOMER`'s nation. Similarly, every row in `LINEITEM` had to be joined with `SUPPLIER` and `NATION` to find its `SUPPLIER`'s nation. With the dimension-join indexes, the optimizer can rewrite query 7, showed in Listing 1, and remove the joins (lines 15 to 19 and 21 to 25).

Without dimension-join indexes, TPC-H query number 7 took 385 seconds to completion. With the two dimension-join indexes the query took only 123 seconds representing an improvement of over 273%. Figure 7 represents the optimized visual version of TPC-H query 7. Another important result was that it was much easier to optimize the query with the dimension-join, i.e., at first, query number 7 took 809 seconds without the

dimension-join indexes and only after some tuning effort could we reduce the time to the 385 seconds.

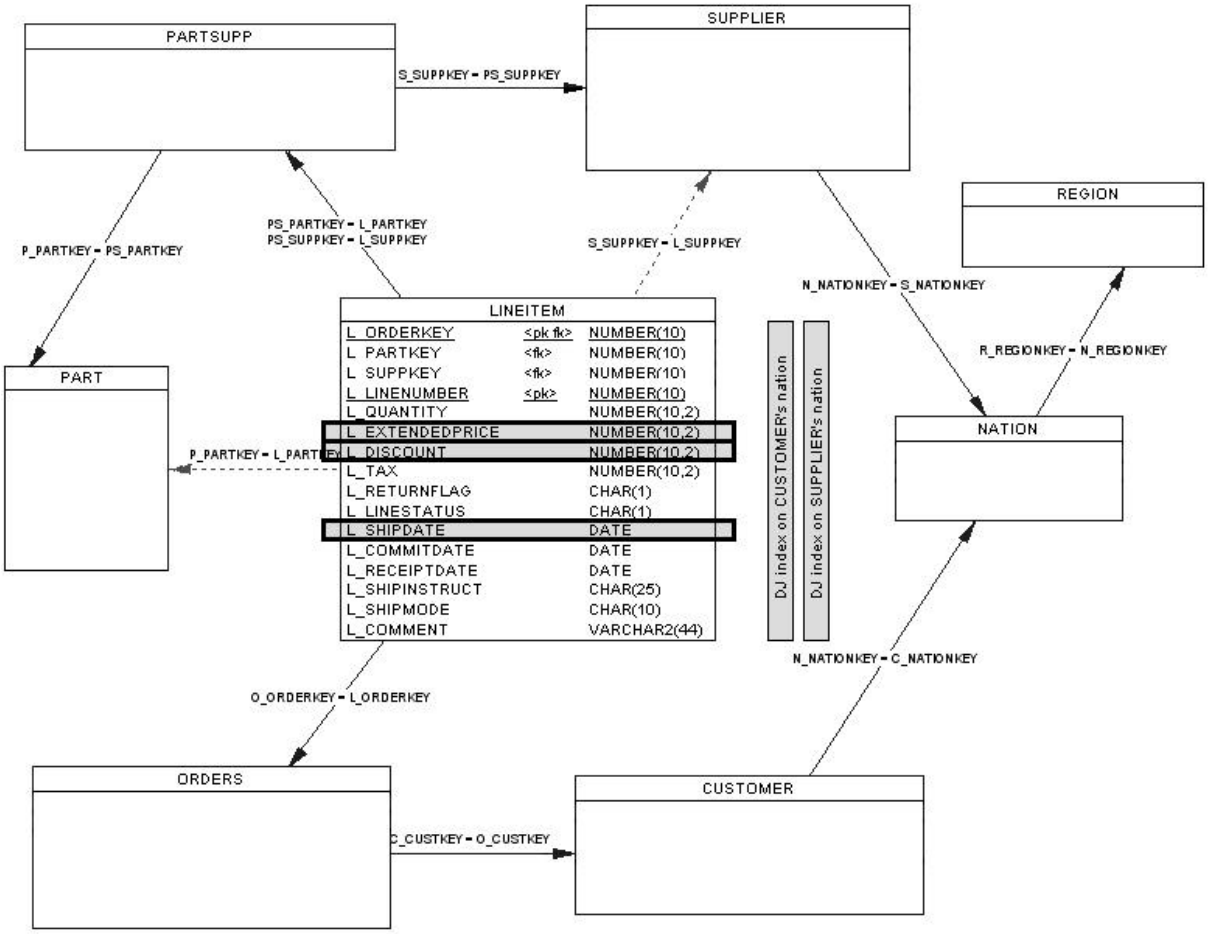


Figure 7: Optimized TPC-H query number 7

Query 7cust took 618 seconds without the dimension-join indexes and only 82 seconds with them accounting for an improvement of over 750%. Query 7supp, which is very simple, took 0.5 seconds without the dimension-join (IDX_SUPP) and only 0.15 seconds with it, returning the answer 3.3 times faster.

6.2. Space taken by the database structures

The new auxiliary indexes, IDX_CUST and IDX_SUPP, take up 11 Mbytes and 25 Mbytes, which correspond to about 2% of the total TPC-H scheme (tables, primary key indexes and foreign key indexes). Although IDX_CUST and IDX_SUPP index exactly the some number of tuples they have different sizes due to clustering. The DBMS compresses bitmap indexes with good clustering values.

Table 1 depicts the sizes of all major structures in the TPC-H scheme.

Tables	Mbytes	PK Indexes	Mbytes	FK Indexes	Mbytes	TOTALS
LINEITEM	757,813	PK_LINEITEM	126,563	FK_ORDERKEY_L FK_PARTKEY_L FK_SUPPKEY_L	126,563 126,563 126,563	**1264,065**
ORDERS	168,633	PK_ORDERS	23,633	FK_CUSTKEY_O	37,539	**229,805**
PARTSUPP	114,805	PK_PARTSUPP	23,516	FK_SUPPKEY_PS FK_PARTKEY_PS	16,719 16,719	**171,759**
PART	26,680	PK_PART	2,969	-	-	**29,649**
CUSTOMER	24,141	PK_CUSTOMER	2,266	FK_NATIONKEY_C	3,359	**29,766**
SUPPLIER	1,484	PK_SUPPLIER	0,156	FK_NATIONKEY_S	0,273	**1,913**
NATION	0,023	PK_NATION	0,016	FK_REGIONKEY_N	0,039	**0,078**
REGION	0,016	PK_REGION	0,016	-	-	**0,032**
TOTALS	**1093,595**		**179,135**		**454,337**	**1727,067**

Table 1: Space required by every major structure in the scheme

Table 2 shows the space taken by the TPC-H scheme with and without the dimension-join indexes.

	Every structure	Dimension-joins		TOTALS
TPC-H	? 1727	-	-	1727
TPC-H + Dimension-joins	? 1727	IDX_SUPP IDX_CUST	25 11	1763

Table 2: Space required by all structures in the scheme

6.3. Time lost in data loads

The time lost rebuilding the dimension-join indexes due to data loads represents the major drawback of this approach. However, we constructed the dimension-join indexes using an Oracle 8i database feature, the function based index which relies upon PL/SQL calls.

Since TPC-H does not define data load details it is not possible to measure precisely how much time is spent in this phase. However, even without loading any data we measure the time spent dropping and re-creating all other indexes. It took us about 9000 seconds to drop and create all primary key and foreign key indexes and it took us about 7300 seconds to drop and rebuild only the two dimension-join indexes. Thus, including the dimension-join indexes considerably increase the index recreation time.

We think that an implementation of the dimension-indexes support by the DBMS vendor would greatly diminish the creation time.

6.4. Implementation

The Oracle 8i database server provides features that allowed for a high-level implementation of dimension-join indexes. In version 8i, Oracle permits a special type of index called Function Based Index. A function-based index is a B-tree or bitmap index created from the result of a function applied to every row in a particular column. For instance, it is possible to have an index built on `UPPER(name)` on persons instead of just `name`. The SQL command that creates a B-tree index on `UPPER(name)` is:

```
CREATE INDEX ON person(UPPER(name));
```

Oracle also allows defining function-based indexes using user-defined functions. The user-defined functions should be deterministic and should not read any other database tables or variables. However, it is possible to work around these restrictions, basically by *lying* to the

Oracle's PL/SQL compiler. We defined two functions: `get_supp_nation` and `get_cust_nation` which receive a `LINEITEM` order number and return the `SUPPLIER`'s nation and the `CUSTOMER`'s nation respectively. The next step was just to define indexes on those functions over `LINEITEM`:

```
CREATE BITMAP INDEX idx_supp ON lineitem(get_supp_nation(l_orderkey);
CREATE BITMAP INDEX idx_cust ON lineitem(get_cust_nation(l_orderkey);
```

Finally, the queries were rewritten to force the usage of the new indexes. The new queries read fewer tables and perform no joins like showed in Figure 7. The complete implementation is outside the scope of this paper. It can be found at [20].

6.5. Comparison of pre-computed structures

The usage of pre-computed structures to perform joins involves three steps: 1) fetch and process some information in the first relation; 2) fetch and process some information in the structure; 3) fetch and process the remaining information in the other relation or table.

Structure	Step 1	Step 2	Step 3
Bitmap	Reads the first table, gets some rows and attribute values.	With those values, finds which rows in the second table are joined with the rows from the first table. Finding the rows in the bitmap is extremely fast when the column has low cardinality.	Reads the rows in the second table by direct access indicated by the position the bits are found in the bitmap. It may remove some rows from the answer set based on the query conditions and on the values obtained.
Dimension-join	Does nothing.	Finds which rows in the fact table are joined with the rows from the dimension table. Finding the rows in the bitmap is extremely fast when the column has low cardinality.	Reads the rows in the fact table by direct access indicated by the position the values are found in the bitmap. It may remove some rows from the answer set based on the query conditions and on the values obtained.
B-tree	Reads the first table, gets some rows and attribute values.	With those values, finds which rows in the second table are joined with the rows from the first table. Finding the rows in the B-tree is fast if the column has high cardinality and if the query is joining few rows.	Reads the rows in the second table indicated by the leaf nodes in the B-tree. It may remove some rows from the answer set based on the query conditions and on the values obtained.
Join index	Reads the first table, gets some rows and attribute values.	Used the values from step 1) to find which rows in the second table are joined with these rows.	Reads the rows in the second table indicated by the leaf nodes in the B-tree. It may remove some rows from the answer set based on the query conditions and on the values obtained.
Materialized view	--	The whole answer is obtained using only the materialized view.	--

Table 3: Steps involved in joins using several pre-computed structures

Table 3 is ordered by space, from the structure that occupies less space to the one that occupies more.

The DBMS can also use the join index by inverting steps 1) and 2); i.e., reading first the information in the index and afterwards reading the rows in one table and than in the other.

7. Conclusions and future work

The dimension-join index allows for data away from the central fact table to be drawn closer to it. The dimension-join index is a bitmap index over the fact table with information regarding one of the dimension's columns and in this characteristic it resembles the join index from Patrick Valduriez. Like in materialized views, with the dimension-join, run-time joins are avoided. Also like in star-join, several indexes may be processed before a (fact) table is accessed. Finally, like in the star-transformation, the DBMS engine may have to re-write the query.

Since the dimension-join represents a shorter path from the fact table to a dimension, the further away from the fact table is that dimension, the better the dimension-join index performs. The dimension-join is a structure especially suited to data warehouses because the joins are big, the number of retrieved lines may be plenty, there are few updates to base data, and there is almost no concurrency. Also, since dimension-join indexes are much cheaper in terms of space than B-trees or materialized views, the data warehouse administrator can use more aggressive policies when deciding which columns to index.

The dimension-join index was tested using the TPC-H benchmark scheme and query number 7 along with two other queries providing constant improvement in performance. The creation phase is currently the drawback of the dimension-join indexes manly because it uses features in a way they were not meant for (the implementation used features of Oracle 8i database server, namely function based indexes with user-defined functions).

Future work is needed measuring more queries and different tables. Also an implementation of the dimension-join indexes at a lower level (instead of PL/SQL and function based indexes) can provide better creation times.

References

[1] Ralph Kimball. **The Data Warehouse Toolkit**. John Willey & Sons, Inc.; 1996.

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

[3] N. Roussopoulos. **Materialized Views and Data Warehouses**. ACM SIGMOD Record 27(1): 21-26 (1998)

[4] Raghu Ramakrishnan, Johannes Gehrke. **Database Management Systems, Second Edition**. McGraw-Hill International Editions, Computer Science Series. 2000.

[5] Masaru Kitsuregawa, H Tanaka, T Molo-oka. Relational Algebra Machine CRACE. Lecture Notes in Computer Science. Springer-Verlag. pp.l91-2.14 (1982)

[6] Zhe Li, Kenneth A. Ross. **Fast Joins Using Join Indices**. The VLDB Journal, volume 7, number 4, 1998.

[7] Sven Helmer, Till Westmann, Guido Moerkotte. **Diag-Join, An Opportunistic Join Algorithm for 1 to N Relationships**. Proc. VLDB1998 Conference, pags 98-109.

[8] Kjell Bratbergsengen. **Hashing Methods And Relational Algebra Operations**. In Proc. 10th VLDB Conference, pags 323-333, 1984.

[9] Rudolf Bayer, Edward M. McCreight. **Organization and Maintenance of Large Ordered Indices**. Acta Informatica 1, pags 173-189; 1972.

[10] Priti Mishra, Margaret H. Eich. **Join Processing in Relational Databases**. In ACM Computing Surveys, Vol 24, No 1, March 1992.

[11] José A. Blakeley, Nancy L. Martin. **Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis**. Proc. ICDE Conference 1990, pags 256-263.

[12] Laura M. Haas, Michael J. Carey, Miron Livny, Amit Shukla. **SEEKing the truth about *ad hoc* join costs**. The VLDB Journal, volume 6, number 3, pags 241-256, 1997.

[13] Marcus Jürgens, Hans-J. Lenz. **Tree Based Indexes vs. Bitmap Indexes - a Performance Study**. Proc. DMDW Conference 1999.

[14] Michael J. Corey, Michael Abbey. **Oracle Data Warehousing**. Osborne McGrawHill – Oracle Press. 1997.

[15] C. Y. Chan, Y. E. Ioannidis: **Bitmap Index Design and Evaluation**. Proc. SIGMOD Conference 1998: 355-366.

[16] P. O'Neil and G. Graefe. **Multi-Table Joins Through Bitmapped Join Indices**. SIGMOD Record, pages 8-l 1, September 1995.

[17] Oracle Corporation. **Oracle8 Concepts (Release 8.0 - A58227-01)**. Oracle Documentation Library. Available at http://technet.oracle.com/

[18] **Benchmark TPC-H - Decision Support for Ad Hoc Queries**. http://www.tpc.org/tpch.

[19] **Transaction Processing Council**. http://www.tpc.org.

[20] Pedro Bizarro. **Avoid costly joins with FBIs**. Oracle Professional Newsletter, Volume 7, number 9. September 2000. Available at www.oracleprofessionalnewsletter.com.