On Index Structures for Star Query Processing in Data Warehouses

Artur Wojciechowski and Robert Wrembel(✉)

Institute of Computing Science, Poznan University of Technology, Poznań, Poland
{Artur.Wojciechowski,Robert.Wrembel}@cs.put.poznan.pl

Abstract. One of the important research and technological issues in data warehouse performance is the optimization of analytical queries. Most of the research have been focusing on optimizing such queries by means of materialized views, data and index partitioning, as well as various index structures including: join indexes, bitmap join indexes, multidimensional indexes or index-based multidimensional clusters. These structures neither well support navigation along dimension hierarchies nor optimize joins with the *Time* dimension, which in practice is used in the majority of analytical queries. In this chapter we overview the basic index structures, namely: a bitmap index, a join index, and a bitmap join index. Based on these indexes, we show how to build another index, called *Time-HOBI*, for optimizing queries that address the *Time* dimension and compute aggregates along dimension hierarchies. We further discuss the extension of the index with additional data structure for storing aggregate values along the hierarchical structure of the index. The aggregates are used for speeding up aggregate queries along dimension hierarchies. Furthermore, we show how the index is used for answering queries in an example data warehouse. Finally, we discuss its performance-related characteristics, based on experiments.

Keywords: Data warehouse · Query optimization · Star query · Hierarchical index · Bitmap index · Join index · Bitmap join index · Time-HOBI

1 Introduction

A traditional data warehouse architecture has been developed in order to analyze heterogeneous and distributed data managed by an enterprise. A core component of this architecture is a database, called a *data warehouse* (DW) that stores the integrated data, both current and historical ones. The content of a DW is analyzed by various analytical queries for the purpose of discovering trends (e.g., demand and sales of products), discovering patterns of behavior (e.g., customer habits, credit repayment history) and anomalies (e.g., credit card usage) as well as for finding dependencies between data (e.g., market basket analysis, suggested buying, insurance fee assessment). These techniques are commonly referred to as On-Line Analytical Processing (OLAP).

E. Zimányi (Ed.): eBISS 2013, LNBIP 172, pp. 182–217, 2014.
DOI: 10.1007/978-3-319-05461-2_6,

Analytical queries, commonly known as *star queries* process large volumes of data. The queries join a central table with multiple reference tables (called dimension tables) that define the context of the analyses. The queries next aggregate data at various levels of granularity, from fine grained to coarse - by means of roll-up operations and from coarse to fine grained - by means of drill-down operations. Since a query response time is one of the key factors of a DW performance, providing means for reducing the time is one of the research and technological challenges. In this area, different mechanisms have been proposed in the research literature, e.g., [27] and applied in commercial data warehouse management systems (DWMSs), i.e., materialized views and query rewriting, e.g., [20], data partitioning and parallel processing, e.g., [17,45,55] as well as advanced indexing, e.g, [6]. The research on indexing resulted in multiple index structures. From these structures, the successfully applied ones in commercial DWMSs include: join indexes, e.g., [57], bitmap indexes, e.g., [38,53], bitmap join indexes, e.g., [9,39], various multidimensional indexes, like for example R-tree [21], Quad-tree [15], and K-d-b-tree [46], as well as index-based multidimensional clusters [42].

We argue that the flat structure of a bitmap index can still be better optimized to match a hierarchical structure of dimensions, in order to facilitate the roll-up and drill-down operations. Moreover, the existing implementations of the aforementioned indexes do not exploit the fact that most of the analytical queries analyze data in time and thus require costly operations of joining a central table with a dimension table, which stores time data.

Although bitmap indexes, join indexes, and bitmap join indexes substantially decrease execution times of analytical queries, not all commercially available database management systems support them. For example, Oracle implements the bitmap index and the bitmap join index. IBM DB2 and SQL Server support implicitly created temporary bitmap indexes only, which are used to optimize joins.

Chapter contribution. In this chapter we overview the basic index structures, namely: a bitmap index, a join index, and a bitmap join index. Based on these indexes, we show how to build another index, called *Time-HOBI*, for optimizing queries that compute aggregates along dimension hierarchies and that analyze data in time. The index was originally presented in [12,36]. In this chapter, we introduce the following additional contributions:

- the extension of *Time-HOBI* with additional data structure for storing aggregate values along the index hierarchy (the aggregates are used for speeding up aggregate queries along dimension hierarchies),
- the analysis of how the index is used for answering queries in an example data warehouse,
- the experimental evaluation of the extended *Time-HOBI*.

Chapter content. This chapter is organized as follows. Section 2 presents basic concepts on data warehousing used in this chapter. Section 3 outlines basic index

structures applied in data warehouses. Section 4 discusses the components of the *Time-HOBI* index and shows how the index is used in a query execution plan. Section 5 discusses performance characteristics of *Time-HOBI* obtained from multiple experimental evaluations. Section 6 presents related work in the area of indexing DW data. Finally, Sect. 7 summarizes the chapter.

2 Data Warehouse Basics

In this section we present the basic concepts and definitions in the area of data warehousing, i.e., a multidimensional data model and its relational implementations, as well as star queries.

2.1 DW Model and Schema

In order to support various analyses, data stored in a DW are represented in the *multidimensional data model* [22,24]. In this model an elementary information being the subject of analysis is called a *fact*. It contains numerical features, called *measures* that quantify the fact. Values of measures are analyzed in the context of *dimensions*. Dimensions often have a hierarchical structure composed of levels, such that $L_i \rightarrow L_j$, where $\rightarrow$ denotes hierarchical assignment between a lower level L_i and upper level L_j, also known as a roll-up or an aggregation path [32]. Following the aggregation path, data can be aggregated along a dimension hierarchy. Level data are called *level instances*. Hierarchically connected level instances form a *dimension instance*.

The multidimensional model is often implemented in relational databases (ROLAP) [11], where fact data are stored in a *fact table*, and level instances are stored in *dimension level tables*. In a ROLAP implementation two basic types of conceptual schemas are used, i.e. a star schema and a snowflake schema [11]. In the *star schema*, each dimension is composed of only one (typically denormalized) level table. In the *snowflake schema*, a dimension is composed of multiple normalized level tables connected by foreign key - primary key relationships. The two basic DW conceptual schemas can be used for creating a *starflake schema* [25]. In this schema some dimensions are composed of normalized and some of denormalized level tables. Star schemas store redundant data and are generally more efficient for queries that join upper levels of dimensions with a fact table. Conversely, for such queries snowflake schemas offer worse performance but there is no data redundancy.

The example "Auctions" DW snowflake schema is shown in Fig. 1. It includes the fact table *Auctions* that stores data about finished Internet auctions. The schema allows to analyze auctions and to aggregate values of measures *price* and *quantity*, with respect to three dimensions, namely: *Time*, *Location*, and *Product*. To this end, the *Auctions* fact table is connected to the dimensions via foreign keys: *dateID*, *cityID*, and *prodID*, respectively. The dimensions have hierarchical structures. For example, dimension *Product* is composed of two

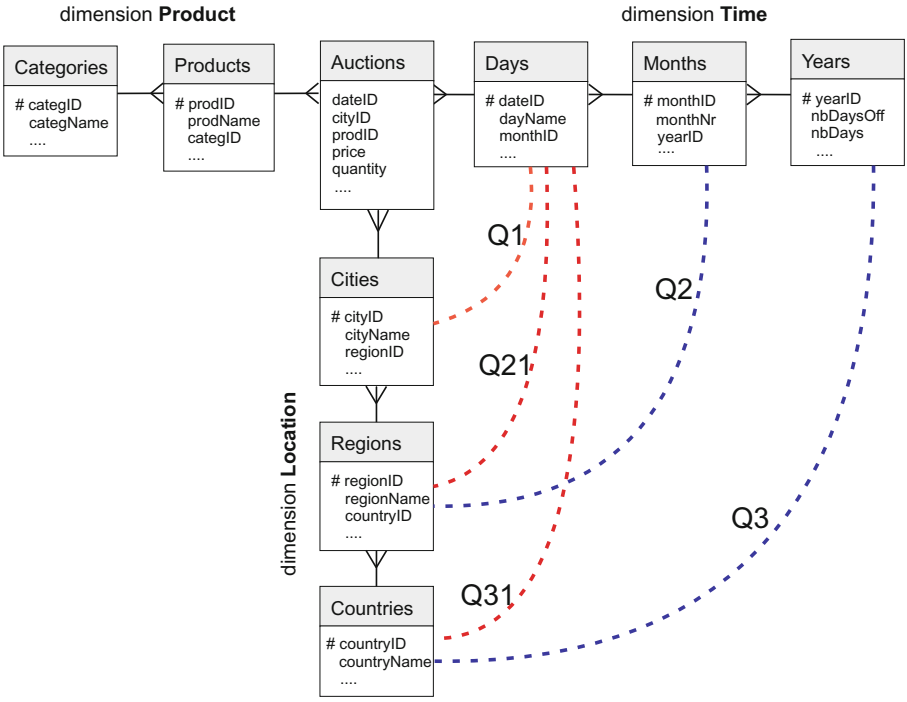


Fig. 1. The example data warehouse snowflake schema

level tables, namely *Products* and *Categories*, such that *Products* → *Categories*. For simplicity reasons only the most important tables' attributes are shown. Notice that the *price* measure represents the current price that was paid for the set of identical items in a given auction, whereas *quantity* is the number of the sold items.

The example "Auctions" DW star schema is shown in Fig. 2. The *Product*, *Location*, and *Time* dimensions were denormalized. Thus, each of them is implemented as a single table.

Figure 3 shows the instance of dimension *Product*. It includes the instances of level *Categories* and level *Products*. Level *Categories* include 2 instances, namely 'Ultrabook' and 'Tablet'. Level *Products* include 7 instances, namely 'Asus Zenbook', 'Dell XPS Duo', 'Toshiba Portege Z930-14T', etc. 'iPad mini', 'Samsung Galaxy Note', and 'Asus Vivio Tab' belong to category 'Tablet' and the others belong to category 'Ultrabook'.

Notice that throughout the paper we use the snowflake schema for illustration purposes only. The *Time-HOBI* index, discussed in Sect. 4 is applicable to the star, snowflake, and starflake schemas. Moreover, in Sect. 5 we evaluated the index for both the star and snowflake schemas.

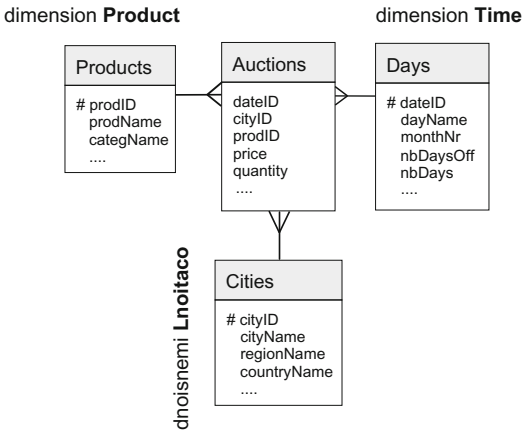


Fig. 2. The example data warehouse star schema

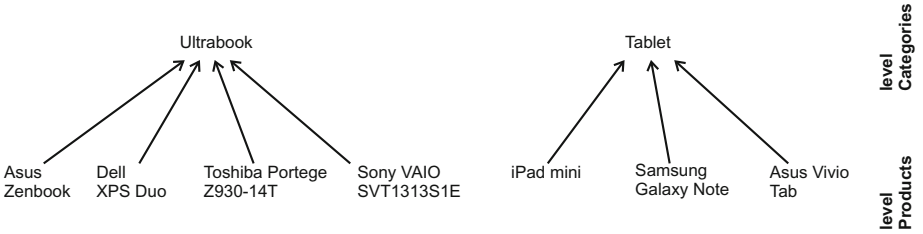


Fig. 3. The example of the *Product* dimension instance

2.2 Star Queries

Star queries, executed on any of the aforementioned DW schemas, join a fact table with multiple level tables. In Fig. 1, we marked (by means of dashed lines) the tables joined by various star queries. For example, Q1 joins tables *Auctions*, *Cities*, and *Days*, whereas Q3 joins *Auctions*, *Cities*, *Regions*, *Countries*, *Days*, *Months*, and *Years*. As an example let us consider the star query Q2 that computes monthly auction sales per region, as shown in Fig. 4.

3 Index Data Structures

Star queries can profit from applying some indexes in the process of retrieving data. In this section we outline three indexes that inspired us while developing *Time-HOBI*. They include: a join index, a bitmap index, and a bitmap join index. We also outline how the oracle implementation of the bitmap join index supports star queries exemplified by query pattern Q3.

```
SELECT r.regionName, m.monthID, m.yearID, sum(a.price), sum(a.quantity)
FROM
   Auctions a,
   Days d, Months m,
   Cities c, Regions r
WHERE
   a.dateID=d.dateID
   AND d.monthID=m.monthID
   AND a.cityID=c.cityID
   AND c.regionID=r.regionID
GROUP BY
   r.regionName, m.monthID, m.yearID
```

Fig. 4. The example query Q2

3.1 Join Index

A *join index* represent the materialized join of two tables, say R and S. As defined in [31,57], a join index is a table composed of two attributes. It stores the set of pairs (r_i, s_j) where r_i and s_j denote identifiers of tuples from R and S, respectively, that join on a given predicate. For the purpose of searching the join index faster, it is physically ordered (clustered) by one of the attributes. Alternatively, the access to the join index can be organized by means of a B-tree or a hash index [39]. Typically, in a DW the index joins a dimension table and a fact table. The index is created either on a join attribute (typically a primary key) or on another attribute (typically storing unique values) of a dimension level table. In order to illustrate the idea behind the join index let us consider the Example 1.

Example 1. Let us consider the *Products* and *Auctions* tables from the DW schema shown in Fig. 1. Their content is shown in Table 1. For explanatory reasons, both tables include also explicit column *ROWID* that stores physical addresses of records. *ROWID*s also play the role of row identifiers. The join index defined on column *ProdID* is shown in Table 2.

As one can observe from the above example, the join index stores a materialized (precomputed) join of tables *Products* and *Auctions*. Thus, it will optimize queries like:

```
select ...
from Auctions a, Products p
where a.prodID=p.prodID ...
```

Table 1. Example tables in the "Auctions" data warehouse (from Fig. 1)

table Auctions			
ROWID	price	cityID	prodID
0AA0	...	POZ	100
0AA1	...	WRO	230
0AA2	...	POZ	100
0AA3	...	WAW	300
0AA4	...	WAW	300
0AA5	...	WRO	230

table Products			
ROWID	prodID	prodName	categID
BFF1	100	HP Pavilion	ELE
BFF2	230	Dell Inspiron	ELE
BFF3	300	Acer Ferrari	ELE

Table 2. Example join index on *Products.prodID*

Products.ROWID	Auctions.ROWID
BFF1	0AA0
BFF1	0AA2
BFF2	0AA1
BFF2	0AA5
BFF3	0AA3
BFF3	0AA4

3.2 Bitmap Index

Analytical queries not only join data, but also filter data by means of query predicates. Efficient filtering of large data volumes may be supported by *bitmap indexes* [13,38,53,60]. Conceptually, a bitmap index created on an attribute a_m of table T is organized as the collection of bitmaps. For each value val_i in the domain of a_m a separate bitmap is created. A bitmap is a vector of bits, where the number of bits is equal to the number of records in table T. The values of bits in bitmap for val_i are set as follows. The n-th bit is set to 1 if the value of attribute a_m for the n-th record is equal to val_i. Otherwise the bit is set to 0. At the implementation level, access to bitmaps can be realized either by means of a B-tree whose leaves store pointers to bitmaps [38] or as simple arrays in a binary file [48].

Example 2. In order to illustrate the idea behind the bitmap index let us review the fact table *Auctions*, shown in Table 3. The table contains attribute *prodID*, whose values are from the set {100, 230, 300}. A bitmap index created on this attribute will be composed of three bitmaps, denoted as *Bm100*, *Bm230*, and *Bm300*, as shown in Table 3.

Bitmap *Bm100* describes rows whose value of attribute *prodID* is equal to *100*, i.e., the first bit in this bitmap is equal to 1 since the value of *prodID* of the first row in table *Auctions* is equal to *100*. The second bit in *Bm100* is equal to 0 since the value of *prodID* of the second row in *Auctions* does not equal *100*, etc. In exactly the same way the bits are set in *Bm230* and *Bm300*. Such a bitmap index will offer a good response time for a query selecting for example data on auctions concerning products identified by *100* or by *300*.

Table 3. The example table *Auctions* and the bitmap index created on attribute *Auctions.prodID*

table Auctions				bitmap index on Auctions.prodID		
				Bm100	Bm230	Bm300
price	prodID	...		prodID=100	prodID=230	prodID=300
...	100	...	⟵	1	0	0
...	230	...	⟵	0	1	0
...	100	...	⟵	1	0	0
...	300	...	⟵	0	0	1
...	300	...	⟵	0	0	1
...	230	...	⟵	0	1	0

In order to find auction rows fulfilling this criterion, it is sufficient to OR bitmaps *Bm100* and *Bm300* to construct the final result bitmap. Then, records pointed to by bits equal to '1' in the result bitmap are fetched from the *Auctions* table.

Bitmap indexes allow to answer queries with the `count` function without accessing tables, since answers to such queries can be computed by simply counting bits equal to '1' in a result bitmap.

The size of a bitmap index strongly depends on the cardinality (domain width) of an indexed attribute, i.e., the index size increases when the cardinality of an indexed attribute increases. Thus, for attributes of high cardinalities (wide domains) bitmap indexes become very large. In order to reduce the size of bitmap indexes defined on attributes of high cardinalities, the two following approaches have been proposed in the research literature, namely: (1) extensions to the structure of the basic bitmap index, e.g., [10,29,41,54,62,63], and (2) bitmap index compression techniques, e.g., [4,14,37,52,59,61]. Discussing these techniques is out of scope of this chapter and their overviews can be found in [53,58].

3.3 Bitmap Join Index

A *bitmap join index* [5,39,41] combines concepts of the join index and the bitmap index. Thus, the bitmap join index takes the advantage of the join index since it allows to materialize a join of tables. It also takes the advantage of the bitmap index with respect to efficient data filtering by means of AND, OR, and NOT operations on bitmaps. Conceptually, this index is organized as the join index, but instead of ROWIDs of a fact table's rows the index stores bitmaps that point to the appropriate fact table's rows. A lookup entry to the bitmap is by the ROWID of a row from a dimension level table (or an attribute uniquely identifying a row in that table). Similarly as for the ordinary join index, the access to the bitmap join index lookup column can be organized by means of a B-tree or a hash index. In order to illustrate the idea behind the bitmap join index let us consider the Example 3.

Example 3. Let us return to Example 1 and let us define the bitmap join index on attribute *prodID* of level table *Products*. Conceptually, the entries of this index are shown in Table 4. The lookup attribute of the index is *prodID*. A bitmap is associated with every value of this attribute. For example, the bitmap for *prodID*=100 points to the rows from table *Auctions* that concern this product, i.e., the 1st and 3rd row in table *Auctions* concern a product of *prodID*=100.

3.4 Indexes in Star Query Processing

In order to asses how star queries utilize the indexes discussed above, we executed in Oracle11g queries Q1, Q2, Q21, Q3, and Q31, as shown in Fig. 1. We selected Oracle as it supports user-managed both bitmap indexes and bitmap join indexes, whereas other systems, including IBM DB2 and Microsoft SQL Server,

Table 4. Example bitmap join index organized as a lookup by attribute *Products.prodID*

Products.prodID	bitmap
100	1
	0
	1
	0
	0
	0
230	0
	1
	0
	0
	0
	1
300	0
	0
	0
	1
	1
	0

support only system defined temporal bitmap indexes [58]. In this section we outline the execution of query Q3, expressed by means of the SQL code shown in Fig. 5. Notice that Q3 allows to parameterize its selectivity by means of the WHERE clause. We run the query for selectivities from 5 to 60 %.

In order to provide a query optimizer the means for optimizing the query, we defined the following indexes: (1) the bitmap index on attribute *year*, (2) the bitmap index on attribute *countryName*, (3) the concatenated bitmap join index on *year* and *countryName*, using the SQL command shown in Fig. 6.

```
SELECT y.yearID, co.countryName, sum(a.price), sum(a.quantity)
FROM
   Auctions a,
   Days d, Months m, Years y,
   Cities ci, Regions r, Countries co
WHERE
   y.yearID in (year1, ..., yearN)
   AND co.countryName in (country1, ..., countryN)
   AND a.cityID=ci.cityID
   AND ci.regionID=r.regionID
   AND r.countryID=co.countryID
   AND a.dateID=d.dateID
   AND d.monthID=m.monthID
   AND m.yearID=y.yearID
GROUP BY
   y.yearID, co.countryName
```

Fig. 5. The example query Q3

```
CREATE BITMAP INDEX bmi_a_years_countries
ON auctions(y.yearID, co.countryName)
FROM
   Auctions a,
   Days d, Months m, Years y,
   Cities ci, Regions r, Countries co
WHERE
   a.cityID=ci.cityID
   AND r.regionID=ci.regionID
   AND r.countryID=co.countryID
   AND a.dateID=d.dateID
   AND d.monthID=m.monthID
   AND m.yearID=y.yearID
```

Fig. 6. The example concatenated bitmap join index

The query execution plan for selectivity equal 7 % is shown in Fig. 7. It was constructed by the Oracle11g (Enterprise Edition Release 11.2.0.1.0 - 64bit Production) cost query optimizer with full statistics available. We can notice that the plan is quite complex. Even with the defined bitmap join index, 6 joins were executed. For other tested query selectivities, their respective execution plans were complex and expensive as well.

The analysis of execution plans of other star queries, including Q1, Q2, Q21, and Q31 reveals that also these queries could be better optimized, i.e., the costly join operations with hierarchical dimensions, including *Time*, could be

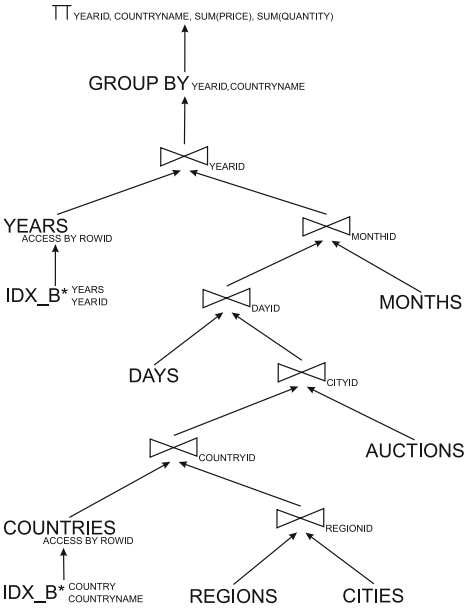


Fig. 7. Execution plan obtained from Oracle for star query Q3 with the selectivity equal 7 %

eliminated or minimized. This observation led us to the development of the index called *Time-HOBI*, originally proposed in [36].

4 Index Time-HOBI

The *Time-HOBI* index is build of three components, namely:

- Hierarchically Organized Bitmap Index (HOBI), where one bitmap index is maintained for one dimension level [12],
- Time Index (TI) that implicitly encodes time in every dimension [36],
- Partial Aggregates (PA) - that store precomputed aggregates along dimension hierarchies that is a new contribution introduced in this chapter.

In this section we present the three aforementioned components of *Time-HOBI*, show how a star query can be executed based on the *Time-HOBI* index, relate our index to a materialized view, and finally, we outline some alternative implementations of HOBI, TI, and PA.

4.1 Hierarchically Organized Bitmap Index

HOBI belongs to the class of bitmap join indexes as the index is defined on a dimension attribute and its bitmaps point to fact rows. *HOBI* is composed of bitmaps organized in a hierarchy that reflects the hierarchy of a dimension. Bitmaps on a lower level of a hierarchy are aggregated at an upper level.

Example 4. In order to illustrate the concept of *HOBI* let us consider dimension *Product*, such that *Products* → *Categories* and the dimension instance, as shown in Fig. 3. For this dimension, *HOBI* consists of two levels. At the lower level - *Products* there exist 7 bitmaps, each of which describes auction sales of one product, i.e., 'Asus Zenbook', 'Dell XPS Duo', etc. At the upper level - *Categories* there exist 2 bitmaps, i.e., 'Ultrabook' and 'Tablet', one bitmap for one category of sold products. The bitmaps are illustrated in Fig. reffig:TimeHobiExample in the box entitled "HOBI for dimension Product".

The upper level bitmap for 'Ultrabook', at level *Categories*, is computed by OR-ing the four bitmaps from level *Products*, i.e., 'Asus Zenbook', 'Dell XPS Duo', 'Toshiba Portege Z930-14T', and 'Sony VAIO SVT1313S1E'. Similarly, the 'Tablet' bitmap describes auction sales of products from this category and it is constructed by OR-ing bitmaps for 'iPad mini', 'Samsung Galaxy Note', and 'Asus Vivio Tab'.

4.2 Time Index

The *Time* dimension plays a special role as it is used in most of the star queries. In order to eliminate the frequent join operation of a fact table with the *Time* dimension, in [36] we proposed to implicitly encode the *Time* dimension in other dimensions. Similarly as in [1,19,33] we assume that data stored in a fact table

are sorted by a selected attribute, typically storing time. This assumption is realistic since a DW is loaded incrementally in time intervals. Moreover, data can be easily sorted by time in the ETL layer before being loaded into a DW. The *Time Index* (TI) takes advantage of data ordering by time. It is created on an attribute used to join a fact table with the *Time* dimension. *TI* stores ranges of bit numbers belonging to a given time interval. The time intervals in *TI* are identical as in the *Time* dimension.

The concept of *TI* is illustrated in Fig. 8. We assume that the *Time* dimension is composed of the following implicit hierarchy $Days \rightarrow Months \rightarrow Years$. Dimension D_i has only one denormalized level L with k instances. Thus, *HOBI* defined for D_i is composed of k bitmaps (denoted as $B_1, \ldots, B_k$) and they describe rows in a fact table. Let us assume that there are z such rows in the fact table. Therefore, every bitmap in *HOBI* is composed of z bits.

TI organizes bits in the bitmaps into intervals (segments) defined in the *Time* dimension. Thus, bits $b_1, \ldots, b_i$ point to fact rows that come from day_1, bits $b_{i+1}, \ldots, b_{i+x}$ point to fact rows that come from day_2, etc. Moreover, $day_1, \ldots, day_n$ aggregate to $month_1$. For this reason, bits $b_1, \ldots, b_{j+x}$ point to fact rows that come from $month_1$. Similarly, $month_1, month_2, \ldots, month_{12}$ aggregate to $year_1$ and bits b_1 to b_{o+x} point to fact rows that come from $year_1$.

Notice that: (1) all the bitmaps point to the same number of rows in a fact table, i.e., the length of every bitmap is identical, and (2) all the bitmaps in *HOBI* are divided into identical time intervals. For these reasons, *TI* is shared by all bitmaps in *HOBI*. *Time Index* eliminates the joins of a fact table with dimension *Time* as bit numbers representing fact rows that fulfill selection criteria on time may be easily retrieved with the support of *TI*.

Example 5. In order to illustrate the concept of *Time Index* let us consider 10 auctions (stored in the table *Auctions*) held on some days in months from February until July in the year 2010, as shown in Fig. 9. The *TI* maps the 9 distinct dates in the *Time* dimension into bit numbers. As the *Auctions* fact table stores 10 rows, every bitmap in *HOBI* includes 10 bits. Thus, the date '25-Feb-2010' maps to bit b_1, '2-Mar-2010' maps to the range of bits b_2–b_3, etc. At the level *Months*, 'February' maps to bit b_1, 'March' maps to the range of bits b_2–b_4, 'April' maps to b_5–b_6, etc. At the level *Year*, '2010' maps to b_1–b_{10}.

4.3 Partial Aggregates

Inspired by the concepts of Small Materialized Aggregates [33], Zone Maps [1], and Zone Filters [19] (Sect. 6), we augmented *HOBI* and *TI* with sets of aggregates that we call *Partial Aggregates* (PA). *PA* are computed for: (1) selected measures, (2) selected aggregation functions, (3) a given dimension, (4) a given dimension level, (5) a given dimension level instances, (6) a given time interval. The aggregates are associated with *HOBI* and they also have a hierarchical structure, which is identical to the structure of *HOBI*.

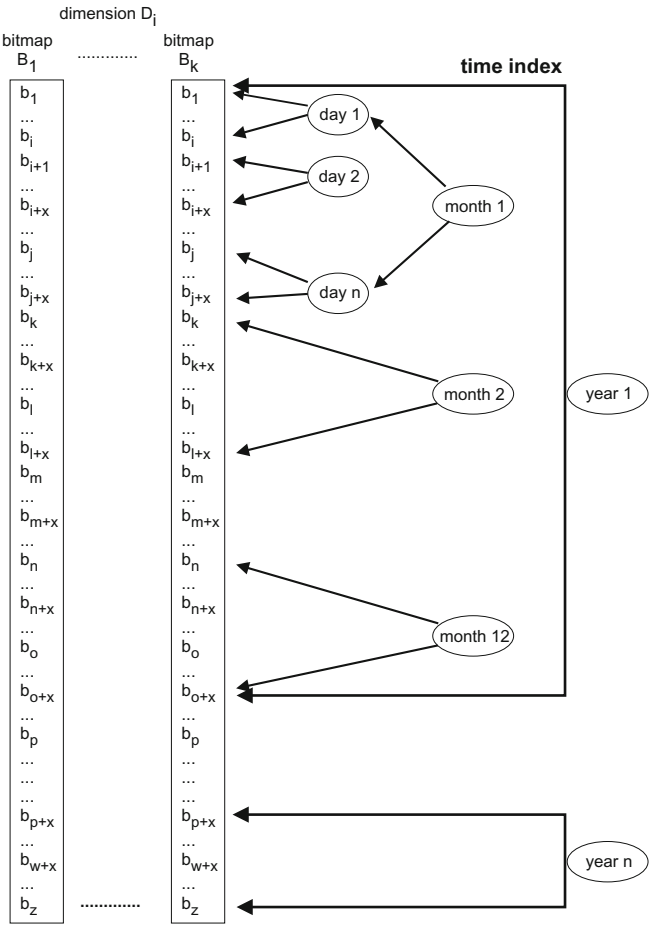


Fig. 8. The concept of Time Index

Let, in a given data warehouse schema:

- $\mathbb{M}$ denote the set of measures (e.g., *price*, *quantity*) and $m_i \in \mathbb{M}$,
- $\mathbb{AF}$ denote the set of aggregate functions (e.g., min, max, avg, sum, count) and $aggF_i \in \mathbb{AF}$,
- $\mathbb{D}$ denote the set of dimensions (e.g., *Time*, *Product*, *Location*) and $d_i \in \mathbb{D}$,
- $\mathbb{DL}$ denote the set of dimension levels (e.g., *Products*, *Categories*, *Cities*, *Regions*, *Countries*) and $dl_i \in \mathbb{DL}$,
- $\mathbb{LI}$ denote the set of dimension level instances (e.g., 'Ultrabook', 'Asus Zenbook', 'iTablet', 'iPad mini', 'Poznan', 'Warsaw') and $li_i \in \mathbb{LI}$,
- $\mathbb{TI}$ denote the set of time intervals in the *Time* dimension (e.g., 01-JAN-2013, JAN-2013, 2013) and $t_i \in \mathbb{TI}$.

Then, formally *PA* is a function $F : (m_i, aggF_i, d_i, dl_i, li_i, t_i) \mapsto v$ $(v \in \mathbb{R})$.

For example, F: (*price*, *sum*, *Location*, *Cities*, $Warsaw$, $MAR-2013$) maps to the aggregate - sum of sales prices in 'Warsaw' (in dimension *Location*, level *Cities*) in March 2013.

Partial Aggregates may be used:

- for selecting these segments of bitmaps that fulfill selection criteria based on aggregates and intersect the segments with segments selected by selection criteria defined by the intervals from the *Time* dimension, e.g.,

```
SELECT ...
WHERE sum(price) > 5000 AND yearID=2013
GROUP BY monthNr
```

- for computing aggregates on an upper level based on aggregates from a lower level of a dimension hierarchy (for distributive and algebraic aggregate functions), e.g., based on aggregate sales price per product in JAN 2013 computing aggregate sales per product category in the same time interval,
- in aggregate queries without selection predicates, e.g.,

```
SELECT sum(price), c.countryName
FROM Auctions a, Countries c, ...
GROUP BY c.countryName
```

- in aggregate queries with multiple selection predicates defined by means of dimensions, like for example Q3.

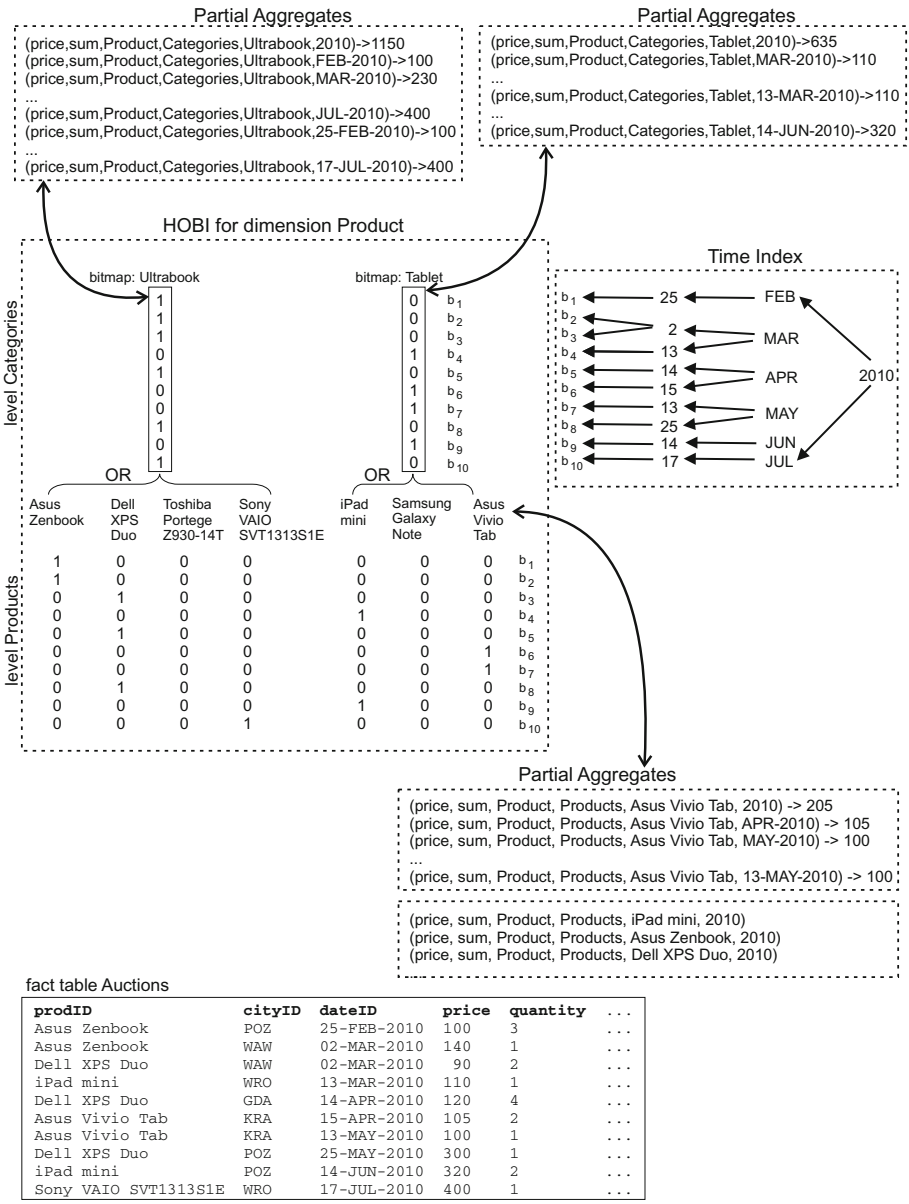
The application of *Time-HOBI* to index the tables in our example schema (Fig. 1) is shown in Fig. 9. We assume that fact table *Auctions* stores 10 rows. *HOBI* for dimension *Product* is composed of 7 bitmaps stored on level *Products* and 2 bitmaps on level *Categories*, as explained in Sect. 4.1. Since every bitmap is composed of 10 bits, *Time Index* points to 10 rows. Auction row from 25-FEB-2010 is represented by bit b_1, auction rows from 02-MAR-2010 are represented by bits b_2 and b_3. Auction rows from March 2010 are represented by bits $b_2, \ldots, b_4$, etc.

Following the definition of *Partial Aggregates*, they are stored for every bitmap and for every time interval in *Time Index*, i.e., day, month, and year. For example, at the level of bitmap 'Asus Vivio Tab', for 13-MAY-2010 there exist the following partial aggregate: *(price, sum, Product, Products, Asus Vivio Tab,13-MAY-2010)* $\rightarrow$ 100, for MAY-2010 there exist the following partial aggregate: *(price, sum, Product, Products, Asus Vivio Tab,MAY-2010)* $\rightarrow$ 100, and for the whole year 2010 there exist the following partial aggregate: *(price, sum, Product, Products, Asus Vivio Tab, 2010)* $\rightarrow$ 205. For simplicity reasons we assumed that only the *price* measure is aggregated by the SUM aggregate function.

Notice that *HOBI* (being the part of *Time-HOBI*) is applicable to any dimension but *Time*, since the *Time* dimension is used to organize bits in *HOBI*.

4.4 Star Query Optimization with the Support of Time-HOBI

As we have shown in Sect. 3.4, the optimization of a simple star query Q3 requires a complex execution plan with 6 joins. While applying *Time-HOBI*, we could optimize the query more efficiently since *Time-HOBI*:



fact table Auctions

prodID	cityID	dateID	price	quantity	...
Asus Zenbook	POZ	25-FEB-2010	100	3	...
Asus Zenbook	WAW	02-MAR-2010	140	1	...
Dell XPS Duo	WAW	02-MAR-2010	90	2	...
iPad mini	WRO	13-MAR-2010	110	1	...
Dell XPS Duo	GDA	14-APR-2010	120	4	...
Asus Vivio Tab	KRA	15-APR-2010	105	2	...
Asus Vivio Tab	KRA	13-MAY-2010	100	1	...
Dell XPS Duo	POZ	25-MAY-2010	300	1	...
iPad mini	POZ	14-JUN-2010	320	2	...
Sony VAIO SVT1313S1E	WRO	17-JUL-2010	400	1	...

Fig. 9. *Time-HOBI* for the *Product* dimension

– eliminates joins of a fact table with the *Time* by applying *TI*;
– allows efficient processing of bitmaps in *HOBI*;
– allows to fetch only the segments of bitmaps that are relevant to a time period selected in a query, by applying *TI* to *HOBI*;
– eliminates joins of a fact table with other dimensions by applying *HOBI*;
– eliminates or reduces the costs of computing aggregates of measures at various levels of a dimension hierarchy by applying *PA*.

The theoretical execution plan of Q3 with the support of *Time-HOBI* is shown in Fig. 10. The WHERE clause included the following selection criteria, resulting in 7 % query selectivity:

```
WHERE
   y.yearID in (2010, 2011)
   AND countryName in ('Poland', 'Germany')
```

As it can be noticed, no joins are needed in this plan. The query can be answered by accessing the following *PA*:

– $(price, sum, Location, Countries, Poland, 2010)$,
– $(price, sum, Location, Countries, Poland, 2011)$,
– $(price, sum, Location, Countries, Germany, 2010)$,
– $(price, sum, Location, Countries, Germany, 2011)$.

Addressing the partial aggregates is symbolized by:

– $PA\text{-}Address(SUM)^{YEARS}_{year=2010\ or\ year=2011}$,
– $PA\text{-}Address(SUM)^{COUNTRIES}_{countryName='Poland'\ or\ countryName='Germany'}$,

whereas fetching the aggregates is symbolized by *PA-Fetch*.

4.5 *Time-HOBI* vs. Materialized View

Time-HOBI takes advantage of materialized partial aggregates. With this respect, our index is similar to a materialized view as it can be applied to answering queries

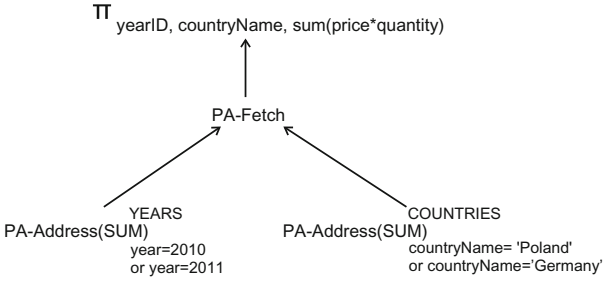


Fig. 10. Query execution plan for query Q3 constructed with the support of *Time-HOBI*

computing the aggregates being materialized in *PA*. It also associates the aggregates with their proper dimension levels and organizes the access to these aggregates by means of *PA*. Unlike a materialized view, *Time-HOBI* additionally can facilitate the execution of queries that filter fact data based on restrictions defined at various levels of dimensions. It is done by means of *HOBI* - being a kind of bitmap join index. In such cases, queries can be executed more efficiently with the support of *Time-HOBI* than with the support of bitmap, join, or bitmap join indexes, as it was shown in [36] and in Sect. 5. Finally, *Time-HOBI* eliminates the need to join a fact table with the *Time* dimension, especially in snowflake schemas. Such queries take advantage of *Time Index* in their execution plans. To this end, the following star query processing algorithm is applied.

Star query processing with the support of *Time-HOBI*

1. By means of the *Time Index* find the bit numbers that correspond to the given time interval $< t_k, t_{k+m} >$; let $[b_k : b_{k+m}]$ denote the corresponding range of bit numbers.
2. $\forall i = (j, \ldots, j+m)$ fetch fragments of bitmaps d_i pointed to by bit numbers $[b_k : b_{k+m}]$; let the fetched fragments be denoted as $f(d_j)$ for dimension instance d_j and $f(d_{j+m})$ for dimension instance d_{j+m}.
3. Compute the final bitmap fragment F from $f(d_i)$ $(i = (j, \ldots, j+m))$ by applying logical operators that were defined in the *where* clause of the star query.
4. $\forall b_i \in [b_k : b_{k+m}]$: if the bit value is equal to 1, then transform b_i into the physical addresses of the corresponding row and fetch the row.

To sum up, we believe that *Time-HOBI* offers slightly more flexibility than materialized views as it combines materialized aggregates at multiple levels of dimension hierarchies, bitmap join indexes, and encodes time on other dimensions.

4.6 Implementation Issues

There are three major implementation issues that impact query performance with the support of *Time-HOBI*, namely: (1) organizing access to bitmaps in *HOBI*, (2) implementing *Time Index*, and (3) organizing access to *Partial Aggregates*.

In the simplest case, when the domain of an indexed attribute is narrow, *HOBI* bitmaps can be accessed by means of an array or list sorted by the bitmap name (i.e., the value of the indexed attribute). In either of these implementations, an element of an array cell or a list stores: (1) the value of an indexed attribute and (2) a pointer to an appropriate bitmap. For wide domains, an access to the bitmaps may be organized either as a B-tree-like index or hash function.

Regarding *Time Index*, a crucial implementation issue is to organize ranges of bits for every time interval defined in the *Time* dimension. There are two straightforward methods of implementing *TI*. The first one is based on a record

structure and the second one is based on a tree structure. In a *record-based* implementation every instance of the *Time* dimension is stored as a record in a table. The record-based implementation can be altered in order to use a nested table (in the spirit of Oracle), where the days records are nested in months, which in turn are nested in a year record. A record-based and a nested table-based implementations can be further indexed in order to reduce access time to data. In the *tree-based* implementation, ranges of bits in the *Time Index* are organized in n trees, where n is equal to the number of years in the *Time* dimension. Each tree has a root that stores the year. A root points to the lower level time intervals, e.g., months, that in turn point to the lower level time intervals, e.g., days. This implementation is visualized in Fig. 9.

A straightforward storage implementation of *Partial Aggregates* is a table with columns $m_i, aggF_i, d_i, dl_i, li_i, t_i, v$ (Sect. 4.3). The access to the aggregates may be organized by an index either on all the columns but v or on the leading columns. Alternatively, *PA* may be accessed by a hash function on the same columns. *PA* may be implemented also as a n-dimensional array, dimensioned by $m_i, aggF_i, d_i, dl_i, li_i, t_i$, with the cells storing v. Thus, the values of the dimensions are used as indexes to the cells of interest.

4.7 Time-HOBI Limitations

As already mention, we assumed that data in a fact table must be sorted by the value of an attribute storing time. The values of the ordering time attribute are used to construct *Time Index*. For this reason, *Time-HOBI* can only be created on the ordering time attribute.

For fine grained instances of the *Time* dimension, like seconds, milliseconds, etc., *Time Index* would include millions or billions of items, resulting in a huge and inefficient size. Moreover, the number of *PA* would also contributed to the size of the whole *Time-HOBI*. These issues needs further investigation in the future.

5 Experimental Evaluation

Time-HOBI was implemented as an application layer in Oracle11g that stored all the data structures discussed in Sect. 4. In the experiments we use a sorted list for *HOBI*, the record-based storage for *TI*, and table storage for *PA*.

The Oracle instance used 3.4 GB of SGA (i.e., the main memory allocated to handle various instance buffers), 1.7 GB of which was allocated to a data cache. The experiments were conducted on a computer equipped with: processor - Intel Core i7-820QM 1.7 GHz, disk - Seagate ST9320423AS, and 8 GB RAM, under Windows7 Server. The DW schemas for the experimental evaluation are shown in Figs. 1 and 2. In the snowflake schema, the tables included the following number of rows: *Years*: 6, *Months*: 72, *Days*: 2190, *Cities*: 10 000, *Regions*: 200, *Countries*: 10, *Categories*: 230, *Products*: 48 000, and *Auctions*: 500 000 000, of the total size equal 30 GB. In the star schema, the tables included the following

```
SELECT ci.cityName, d.dayName, d.dayNr,
   sum(a.price), sum(a.quantity)
FROM
   Auctions a, Cities ci, Days d
WHERE
   ci.cityName like 'pattern'
   AND a.cityID=ci.cityID
   AND a.dateID=d.dateID
GROUP BY
   ci.cityName, d.dayName, d.dayNr
```

Fig. 11. The example query Q1 used in the experiments for the snowflake and star schemas

number of rows: *Days*: 2190, *Cities*: 10 000, *Products*: 48 000, and *Auctions*: 500 000 000, of the total size slightly over 30 GB, due to the redundancy in the dimension tables. The data used in the experiments were artificially generated but data distributions reflected real data that we used in [12].

In this schema we run five different query patterns, as shown in Fig. 1 and mentioned in Sect. 2.2 as well as the equivalents of the query patterns in the star schema. For example, the pattern of Q1 is shown in Fig. 11. Its selectivity is parameterized in the WHERE clause by means of attribute *Cities.cityName*. Notice that Q1 is identical for the snowflake and the star schema.

The patterns of Q3 for the snowflake and the star schema are shown in Figs. 12 and 13, respectively. Their selectivities are parameterized in the WHERE clause by means of attributes *countryName* and *yearID*. Since the filtering predicates are defined by means of the highest levels (i.e., *Countries*, and *Years*) of the *Location* and *Time* dimensions, the lowest possible selectivity for Q3 is 1.6 %.

In the snowflake schema, the Q2 query pattern computes the aggregates *sum(a.price)* and *sum(a.quantity)* at the levels of *Months* and *Regions* (cf. Fig. 1. It is parameterized by means of attributes *Regions.regionName* and *Months.*

```
SELECT y.yearID, co.countryName,
   sum(a.price), sum(a.quantity)
FROM
   Auctions a,
   Days d, Months m, Years r,
   Cities ci, Regions r, Countries co
WHERE
   NOT (co.countryName like 'pattern')
   AND (y.yearID > v_year)
   AND a.cityID=ci.cityID
   AND ci.regionID=r.regionID
   AND r.countryID=co.countryID
   AND a.dateID=d.dateID
   AND d.monthID=m.monthID
   AND m.yearID=y.yearID
GROUP BY
   y.yearID, co.countryName
```

Fig. 12. The pattern Q3 in the snowflake schema

```
SELECT d.yearID, ci.countryName,
   sum(a.price), sum(a.quantity)
FROM
   Auctions a, Days d, Cities ci
WHERE
   NOT (ci.countryName like 'pattern')
   AND (d.yearID > v_year)
   AND a.cityID=ci.cityID
   AND a.dateID=d.dateID
GROUP BY
   d.yearID, ci.countryName
```

Fig. 13. The pattern Q3 in the star schema

monthNr. Pattern Q21 computes the same aggregates at the levels of *Days* and *Regions* and it is parameterized by means of attributes *Regions.regionName* and *Days.dayNr*. Pattern Q31 computes the same aggregates as Q2 but at the levels of *Days* and *Countries*. It is parameterized by *Countries.countryName* and *Days.dayNo*.

In the star schema, the query patterns Q2, Q21, and Q3 compute the same aggregates as their equivalents in the snowflake schema, however, *Auctions* is joined with the denormalized dimension tables *Days* and *Cities*.

In the experiments we decided no to use any of the standard benchmarks like TPC-H, TPC-DS [3], SSB [40], and [2] for two reasons. First, because the benchmarks are typically designed to measure an overall system's response time and query processing efficiency for given query workloads. We found that it is inadequate to our setting where we aimed at comparing the performance of particular indexes with respect to parameterized selectivities of the queries and parameterized number of joins. Second, the patterns of queries that we applied in our experiments reflect real queries that were run in a real system [12] that we modeled with the snowflake and the star schema.

The experiments that we conducted aimed at comparing the following characteristics of *Time-HOBI* with respect to:

1. the performance of the aggregate query patterns Q1, Q2, Q21, Q3, and Q31,
2. index sizes,
3. index creation times.

We related the three characteristics to the competitors being:

- Oracle indexes in the snowflake schema,
- Oracle indexes in the star schema,
- Oracle materialized views in the snowflake schema,
- Oracle materialized views in the star schema.

In each of the experiments described below, for comparison, we defined the following Oracle indexes:

- the bitmap index on attribute *yearID* in both the snowflake and the star schema,
- the bitmap index on attribute *countryName* in both the snowflake and the star schema,
- the concatenated bitmap join index on *year* and *countryName* that joined tables *Years*, *Months*, *Days*, *Countries*, *Regions*, *Cities*, and *Auctions* in the snowflake schema,
- the bitmap index on attribute *monthNr* in both the snowflake and the star schema,
- the bitmap index on attribute *regionName* in both the snowflake and the star schema,
- the concatenated bitmap join index on *monthNr* and *regionName* that joined tables *Months*, *Days*, *Regions*, *Cities*, and *Auctions* in the snowflake schema,

- the bitmap index on attribute *dayNr* in both the snowflake and the star schema,
- the bitmap index on attribute *cityName* in both the snowflake and the star schema,
- the concatenated bitmap join index on *dayNr* and *cityName* that joined tables *Days*, *Cities*, and *Auctions* in the snowflake schema,
- the concatenated bitmap join index on *yearID*, *monthNr*, *dayNr*, *countryName*, *regionName* and *cityName* that joined tables *Locations*, *Time* and *Auctions* in the star schema.

Additionally, we created the set of materialized views in the snowflake schema. Their structures are shown in Fig. 14. The corresponding views in the star schema computed the same aggregates.

```
CREATE MATERIALIZED VIEW MV_A_CITY_DAY ... AS
SELECT ci.cityId, d.dateId, ci.cityName, d.dayName, d.dayNr,
  sum(a.price) as pricesum, sum(a.quantity) as quantitysum, count(*) as count
FROM Auctions a, Cities ci, Days d
WHERE a.cityId=ci.cityId AND a.dateId=d.dateId
GROUP BY ci.cityId, d.dateId, ci.cityName, d.dayName, d.dayNr

CREATE MATERIALIZED VIEW MV_A_REGION_MONTH ... AS
SELECT r.regionID, m.monthID, r.regionName, m.monthNr,
  sum(a.pricesum) as pricesum, sum(a.quantitysum) as quantitysum, sum(a.count) as count
FROM MV_A_CITY_DAY a, Cities ci, Regions r, Says d, Months m
WHERE a.cityID=ci.cityID AND a.dateID=d.dateID AND ci.regionID=r.regionID
  AND d.monthID=m.monthID
GROUP BY r.regionID, m.monthID, r.regionName, m.monthNr

CREATE MATERIALIZED VIEW MV_A_COUNTRY_YEAR ... AS
SELECT c.countryID, y.yearID, c.countryName,
  sum(a.pricesum) as pricesum, sum(a.quantitysum) as quantitysum, sum(a.count) as count
FROM MV_A_REGION_MONTH a, Regions r, Months m, Countries c, Years r
WHERE a.regionID=r.regionID AND a.monthID=m.monthID AND r.coutnryID=c.coutnryID
  AND m.yearID=y.yearID
GROUP BY c.coutnryID, y.yearID, c.countryName

CREATE MATERIALIZED VIEW MV_A_LOCATION_TIME ... AS
SELECT l.countryName, l.regionName, l.cityName, locationID,
  t.year, t.monthNr, t.dayNr, timeID,
  sum(a.price) as pricesum, sum(a.quantity) as quantitysum, count(*) as count
FROM Auctions a, Time t, Locations l
WHERE a.timeid=t.timeid AND a.locationID=l.locationID
GROUP BY l.countryName, l.regionName, l.cityName, locationID,
  t.year, t.monthNr, t.dayNr, timeID

CREATE MATERIALIZED VIEW MV_A_LOCATION_TIME_R2 ... AS
SELECT countryName, regionName, yearID, monthNr,
  sum(pricesum) as pricesum, sum(quantitysum) as quantitysum, sum(count) as count
FROM MV_A_LOCATION_TIME
GROUP BY countryName, regionName, yearID, monthNr

CREATE MATERIALIZED VIEW MV_A_LOCATION_TIME_R3 ... AS
SELECT countryName, yearID, sum(pricesum) as pricesum,
  sum(quantitysum) as quantitysum, sum(count) as count
FROM MV_A_LOCATION_TIME_R2
GROUP BY countryName, yearID
```

Fig. 14. The materialized views created in the snowflake schema

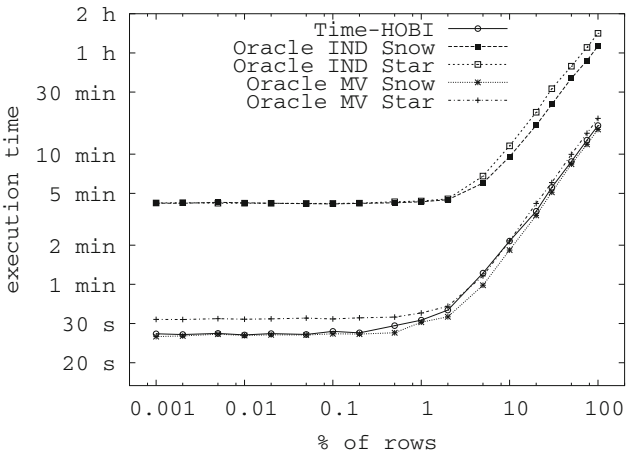


Fig. 15. Elapsed execution times of query pattern Q1 with the parameterized query selectivity

5.1 Query Performance

In these experiments we measured the performance characteristics of the indexes for query patterns Q1, Q2, Q21, Q3, and Q31. The selectivities of these queries were parameterized and ranged from 0.001 % to 100 % of rows in the *Auction* fact table. *Time-HOBI* indexes were defined in every dimension used in the queries, thus the indexes included the *Partial Aggregates* by (Product, Time) as well as (Location, Time), for all the levels in these dimensions. *PA* were created in the snowflake and the star schema.

The obtained results for query pattern Q1 are shown in Fig. 15. Notice that the query performance with the support of *Time-HOBI* is the same in the snowflake and star schema. It is because, the structure of the index is the same for both of the schemas. This observation is true also for the rest of the test queries.

From Fig. 15 we can observe that the elapsed execution times of the queries are much lower with the support of *Time-HOBI* than with the support of the set of Oracle indexes, for the whole range of the query selectivity. Moreover, the execution times of the queries are almost the same for *Time-HOBI* and the materialized views, for the same range of the selectivity.

The performance characteristics of query patterns Q2, Q21, Q3, and Q31 are shown in Figs. 16, 17, 18, and 19. From the figures we can observe that *Time-HOBI* also offers better performance than the Oracle indexes in both DW schemas and offers similar performance to the Oracle materialized views.

For the above characteristics we computed ratio $\gamma = \frac{t^{IND}_{Oracle}}{t_{Time-HOBI}}$, where t^{IND}_{Oracle} and $t_{Time-HOBI}$ denote elapsed query execution times with the support of the set of Oracle indexes and *Time-HOBI*, respectively. Similarly, we computed

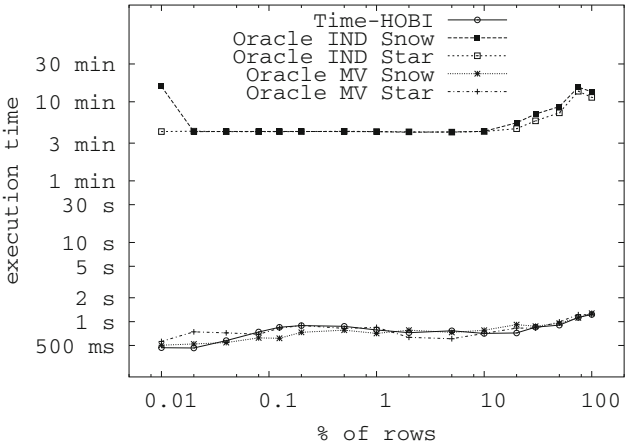


Fig. 16. Elapsed execution times of query pattern Q2 with the parameterized query selectivity

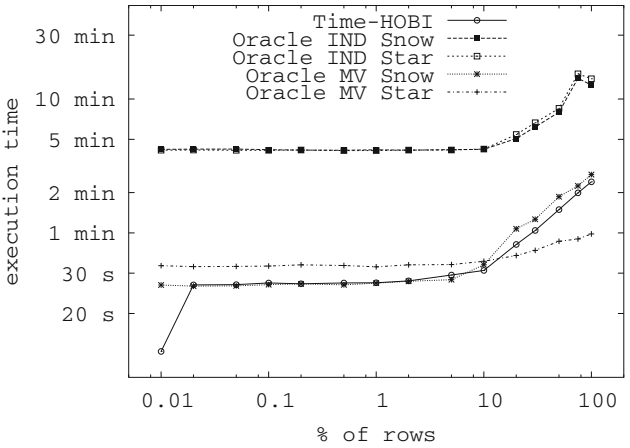


Fig. 17. Elapsed execution times of query pattern Q21 with the parameterized query selectivity

ratio $\lambda \frac{t^{MV}_{Oracle}}{t_{Time-HOBI}}$, where t^{MV}_{Oracle} denotes elapsed query execution times with the support of the set of Oracle materialized views.

The minimum and maximum values of the ratios γ and λ obtained for query patterns Q1, Q2, Q21, Q3, and Q31 are shown in Table 5. For every minimum and maximum value we indicated the query selectivity.

From the performance characteristics shown above we conclude that:

– *Time-HOBI* outperforms the Oracle indexes for aggregate queries, for the whole tested range of selectivities. As the experiments confirmed, this statement is true for the snowflake and the star schema.

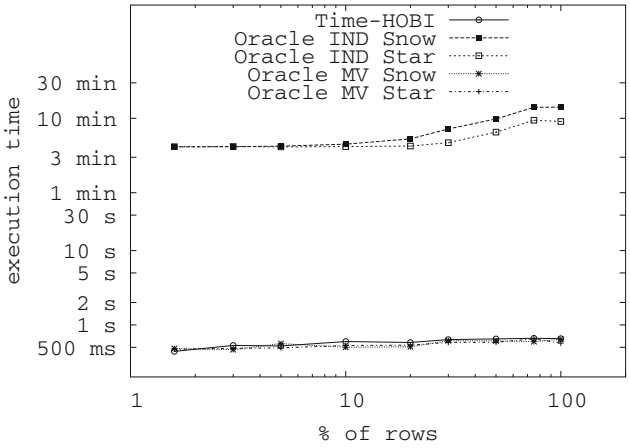


Fig. 18. Elapsed execution times of query pattern Q3 with the parameterized query selectivity

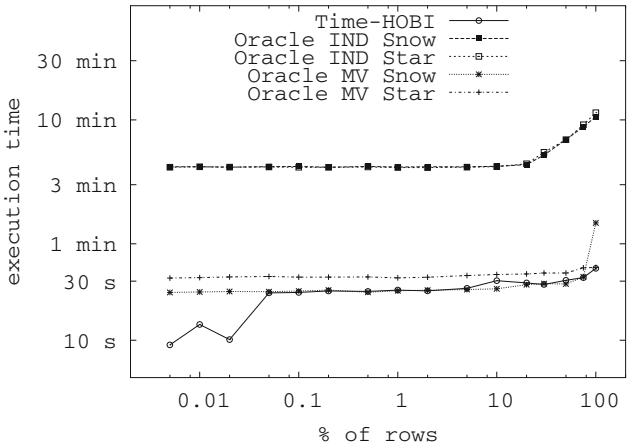


Fig. 19. Elapsed execution times of query pattern Q31 with the parameterized query selectivity

- *Time-HOBI* offers similar performance characteristics as the Oracle materialized views for aggregate queries, for the whole tested range of selectivities. As the experiments confirmed, this statement is true for the snowflake and the star schema.
- *Time-HOBI* offers the same query performance for the snowflake and the star DW schema as the structure of the index is the same for both of the schemas.

Table 5. The minimum and maximum values of the γ and λ ratios obtained for the test queries

	γ snow sch.		γ star sch.		λ snow sch.		λ star sch.	
	min	max	min	max	min	max	min	max
Q1	4.09	10.35	5.11	10.29	0.81	0.99	0.95	1.32
	sel.75–100 %	sel.0.01 %	sel.0.01 %	sel.0.01 %	sel.0.01 %	sel.0.005–0.05 %	sel.5 %	sel.0.05 %
Q2	283	1997	282	721	0.73	1.27	0.79	1.60
	sel.0.5 %	sel.0.01 %	sel.0.2 %	sel.75 %	sel.0.125 %	sel.20 %	sel.5 %	sel.0.02 %
Q21	5.29	32.37	5.71	31.90	0.92	3.14	0.41	4.37
	sel.100 %	sel.0.01 %	sel.50 %	sel.0.01 %	sel.5 %	sel.0.01 %	sel.100 %	sel.0.01 %
Q3	450	1292	417	853	0.84	1.09	0.87	1.07
	sel.50 %	sel.100 %	sel.10 %	sel.75 %	sel.10 %	sel.1.6 %	sel.100 %	sel.1.6 %
Q31	8.37	27.36	8.32	27.13	0.86	2.65	1.02	3.47
	sel.10 %	sel.0.005 %	sel.10 %	sel.0.005 %	sel.10 %	sel.0.005 %	sel.100 %	sel.0.005 %

5.2 Index Sizes

In these experiments we measured the sizes of *Time-HOBI* and compared them to the sizes of Oracle indexes and Oracle materialized views, for various data volumes being indexed (from 1 GB to 18 GB). The results are visualized in Fig. 20. As it can be observed from the figure, the size of *Time-HOBI* is larger than the size of the materialized views for the whole range of the fact table sizes and for both DW schema implementations. It is not surprising as *Time-HOBI* stores not only aggregates but also bitmaps. As compared to the Oracle indexes, in the snowflake schema our index is about 2.1 times larger in the whole tested range of DW size. In the star schema our index is about 1.5 times larger in the whole tested range of DW size. Table 6 shows the exact sizes of the tested data structures.

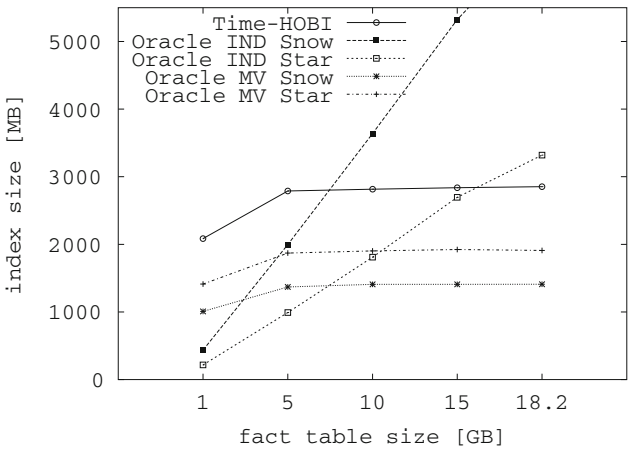


Fig. 20. The sizes of *Time-HOBI*, Oracle indexes, and Oracle materialized views for, a parameterized data volume

Table 6. The sizes of *Time-HOBI*, the Oracle indexes, and the Oracle materialized views, for the snowflake and the star schema

Data volume	PA [MB]	HOBI [MB]	Time-HOBI [MB]	Oracle IND snow [MB]	Oracle IND star [MB]	Oracle MV snow [MB]	Oracle MV star [MB]
1 [GB]	980	1105	2085	435	216	597	1412
5 [GB]	1310	1479	2789	1997	991	1644	1871
10 [GB]	1310	1505	2815	3638	1809	2508	1902
14 [GB]	1380	1458	2838	5324	2694	3356	1923
18 [GB]	1380	1472	2852	6644	3319	4026	1009

Recall that *Time-HOBI* stores three sets of data, i.e., time in *Time Index*, bitmaps in *HOBI*, and precomputed aggregated values of selected measures in *Partial Aggregates*. Table 6 shows the sizes of the aforementioned data structures, the Oracle indexes and materialized views, for five data volumes in the snowflake schema. The size of *TI* for 10 years time period is about 85 kB and it can be neglected, [36]. The size of *PA* should be constant for a given constant number of rows in dimension level tables. In our experiments, the size of *PA* changes slightly with the increase of the data volume, cf. Table 6. In our opinion it may be caused by a data allocation in database blocks (with different percentage free for different data volume sizes). Thus with the almost constant size of *PA* w.r.t. a data volume, for data volume sizes greater than 18 GB *Time-HOBI* will be smaller than the Oracle indexes, for both the snowflake and the star schema. Since the number of *PA* and the number of bitmaps in *HOBI* depends on the number of levels in dimensions and does not depend on the schema implementation, the size of *Time-HOBI* remains the same for the snowflake and star schema.

5.3 Index Creation Times

In these experiments we measured the creation times of *Time-HOBI* and compared them to creation times of Oracle indexes and Oracle materialized views, for various data volumes being indexed (from 1 GB to 18 GB). The results are shown in Fig. 21. The creation time characteristics are consistent with the index size characteristics, i.e., the larger the index is the longer it takes to create it. However, for data volumes larger than 5GB, *Time-HOBI* is created faster than the Oracle indexes. Moreover, its creation time is comparable to the creation time of the materialized views in the whole range of the test data volume.

As our index was implemented at an application layer, the procedure for creating *Time-HOBI* was not optimized and data volumes processed for creating *HOBI* were not shared while creating *PA*. In fact, the procedure executed joins on *Auctions* and its dimensions (*Time* and *Location*) twice (two independent queries). We believe that this procedure can be optimized at some extent, thus reducing the index creation time.

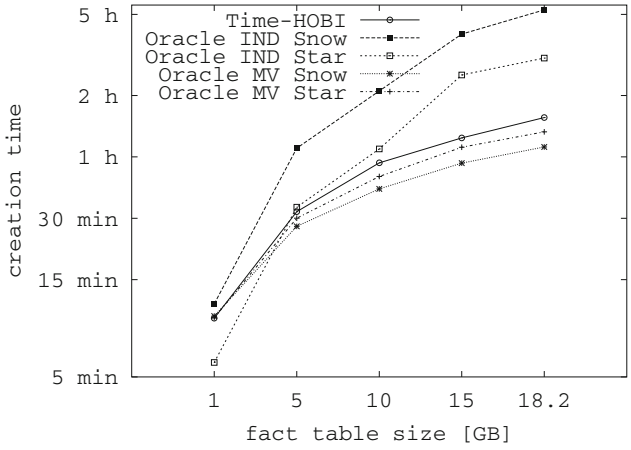


Fig. 21. Creation times of *Time-HOBI*, Oracle indexes, and Oracle materialized views, for a parameterized data volume

5.4 Experiments Summary

Query performance comparison of *Time-HOBI* to the Oracle indexes reveals that:

- the star queries that we tested in the snowflake DW schema were executed from 4 to 1292 times faster (depending on a query) with the support of *Time-HOBI*,
- the queries in the star DW schema were executed from 5 to 853 times faster with the support of our index.

Query performance comparison of *Time-HOBI* to the Oracle materialized views reveals that:

- the queries in the snowflake schema were executed in comparable times (λ ranges from 0.81 to 2.65),
- the queries in the star schema were executed also in comparable times (λ ranges from 0.79 to 4.37).

Size comparison of *Time-HOBI* to the Oracle indexes reveals that:

- in the snowflake schema our index is up to 4.8 times larger for a DW volume 1–7.5 GB, and it is up to 2.3 times smaller for a DW data volume 8–18 GB,
- in the star schema our index is up to 9.6 times larger for a DW volume 1–16 GB, and it is up to 1.2 times smaller for a DW data volume over 16 GB.

Size comparison of *Time-HOBI* to the Oracle materialized views reveals that:

- in the snowflake schema our index is about 2.1 times larger in the whole tested range of DW size,

– in the star schema our index is about 1.5 times larger in the whole tested range of DW size.

Creation time comparison of *Time-HOBI* to the Oracle indexes reveals that:

– in the snowflake schema our index is created from 1.18 to 3.37 times faster in the whole tested range of DW size,
– in the star schema our index is created from 1.17 to 2 times faster for a DW data volume greater than 5 GB.

Creation time comparison of *Time-HOBI* to the Oracle materialized views reveals that:

– in the snowflake schema our index is created from 1.2 to 1.4 times slower in the whole tested range of DW size,
– in the star schema our index is created from 1.1 to 1.5 times slower in the whole tested range of DW size.

6 Related Work

Assuring an efficient access to large volumes of data is an important research problem. Various physical structures have been proposed in the research literature to solve the problem. In this section we outline the physical structures that inspired the development of *Time-HOBI*. The structures include indexes and materialized aggregates.

6.1 Traditional Indexes

Various indexes have been proposed for different application domains. In relational databases and data warehouses, most widely applied index structures in practice include: B-tree like indexes, bitmap indexes, and join indexes. They gained popularity due to their relatively simple structures and maintenance algorithms. In geographical databases, various multi-dimensional indexes have been developed. For advanced data analysis in statistical databases, for data mining, as well as for object databases hierarchical indexes have been developed.

The indexes from the B-tree family [26] are efficient only in indexing data of high cardinalities (i.e., wide domains) and they well support queries of high selectivities (i.e., when few records fulfill query criteria). However, for OLAP queries that are often expressed on attributes of low cardinalities (i.e., narrow domains), B-tree indexes do not provide an acceptable performance. For this reason, for indexing data of low cardinalities, for efficient filtering large data volumes, and for supporting OLAP queries of low cardinalities, bitmap indexes have been developed (Sect. 3.2). A drawback of a bitmap index is that its size increases when the cardinality of an indexed attribute increases. As a consequence, bitmap indexes defined on attributes of high cardinalities become very large or too large to be efficiently processed in main memory [63]. In order to improve the efficiency of accessing data with the support of bitmap indexes

defined on attributes of high cardinalities, either different kinds of bitmap encodings have been proposed, e.g., [10,29,47,54,62] or compression techniques have been developed, e.g., [4,52,53,61].

For efficient executions of star queries, a join index was developed [31,39,57]. It can be perceived as the materialized join of a level table and a fact table. The index is created on a join attribute of a level table. The index is typically organized as a B-tree. It differs from a traditional B-tree with respect to the content of its leaves. The leaves of the join index store physical addresses of records from all the joined tables. An extension to the join index was proposed in [39] where the authors represented precomputed joins by means of bitmaps. The join index whose leaves store bitmaps rather than ROWIDs is called a bitmap join index. Bitmap join indexes provide an efficient optimization mechanism for star queries not only in traditional data warehouses but also in spatial data warehouses [8,51].

6.2 Multi-level Indexes

Concepts similar to the join index were developed for object databases for the purpose of optimizing queries that follow the chain of references from one object to another ($o_i \rightarrow o_{i+1} \rightarrow \ldots o_{i+n}$). Persistent (precomputed) chains of object references are stored either in an access support relation [28] or in a join index hierarchy [23]. In [66] two index structures for indexing hierarchies of classes were described. Both of them are based on tree-like structures.

In [34,35,49,50] indexes of multi-level structures have been proposed. A multi-resolution bitmap index was presented in [49,50] for the purpose of indexing scientific data. The index is composed of multiple levels. Lower levels are implemented as standard bitmap indexes offering exact data look-ups, while upper levels, are implemented as binned bitmaps, offering data look-ups with false positives. An upper level index (the binned one) is used for retrieving a dataset that totally fulfills query search criteria. A lower level index is used for fetching data from boundary ranges in the case when only some data from bins fulfill query criteria.

In [34,35], a hierarchical bitmap index was proposed for set-valued attributes for the purpose of optimizing subset, superset, and similarity queries. The index, being defined on a given attribute, consists of the set of index keys, where every key represents a single set of values. Every index key comprises signature S. The length of the signature, i.e. the number of bits, is equal to the size of the domain of the indexed attribute. S is divided into n-bit chunks (called index key leaves) and the set of inner nodes. Index key leaves and inner nodes are organized into a tree structure. Every element from the indexed set is represented once in the signature by assigning value '1' to an appropriate bit in an appropriate index key leaf. The next level of the index key stores information only about these index key leaves that contain '1' on at least one position. A single bit in an inner node represents a single index key leaf. If the bit is set to '1' then the corresponding index key leaf contains at least one bit set to '1'. The i-th index key leaf is represented by j-th position in the k-th inner node, where $k = \lceil i/l \rceil$

and $j = i - (\lceil i/l \rceil - 1) * l$. Every upper level of the inner nodes represents the lower level in an analogous way. This procedure repeats recursively up to the root of the tree.

6.3 Multidimensional Indexes

Since more than 40 years of research in this domain a few dozens of multidimensional indexes have been proposed, including the most frequently applied families of R-trees [43,44,64] grid files, and K-D-trees. Excellent overviews of existing multidimensional indexes have been proposed in [7,18]. Most of the indexes have been designed for the support of access methods to spatial data in geographical databases, mostly in a two dimensional space. Multidimensional indexes are also applied to supporting top-k (e.g., [65]), k-NN queries (e.g., [67]), and data mining (e.g., [30]).

Most of the multi-dimensional indexes are very complex data structures. They are difficult to implement and to maintain and sometimes their complexity penalizes the performance. For these reasons, some research efforts focus on mapping the multidimensional indexes into relational DBMSs [7,16].

6.4 Materialized Aggregates

The second data storage structure that inspired our work on *Time-HOBI* are materialized aggregates. Two types of such aggregates can be distinguished, namely: (1) materialized views and (2) summary data. A *materialized view* is a precomputed query whose result is stored in a database. Typically, various data warehouse queries take advantage of materialized views in the process of query rewriting. An overview of multiple research problems on materialized views can be found in [20].

Summary data are materialized sets of aggregates, typically associated with storage units of data, e.g., segments, extents (like in Oracle), buckets [33], or zones [1,19]. The first concept, called *Small Materialized Aggregates* (SMA) was proposed in [33]. The concept of SMA assumes that data are ordered on disk by the value of a selected attribute, typically date. Physically, a disk is divided into logical storage units called buckets. Every bucket stores at most n rows. SMA is associated with each bucket. For each bucket, SMA typically include aggregates like minimum and maximum value of the ordering attribute as well as the number of rows in the bucket. The min and max values allow to check whether the bucket fulfills selection criteria on an ordering attribute. If so, the bucket is fetched from disk, otherwise it is skipped.

A concept similar to SMA, called *Zone Maps*, was implemented in Netezza [1]. Netezza organizes disk space in zones, each of which has its own zone map. The zone map includes minimum and maximum values of every attribute of a table stored in the zone. Zone maps are used for verifying whether a given zone qualifies to be accessed by a query.

Finally, [19] extends the two aforementioned concepts by means of *zone filters* and *zone indexes*. The first mechanism generalize SMA and zone maps. It is

similar to zone maps but it stores n minimum and n maximum values of every attribute in a zone (where $n>2$). An separate index - a zone index is dedicated to every zone to facilitate searching data within the zone. All the aforementioned structures assume that the summary data are maintained within by a process that loads a data warehouse.

Although SMA, zone maps, and [19] inspired the development of *PA*, it differs from these three concept as *PA* store aggregated values of measures at all levels of dimension hierarchies, whereas in the three concepts minimum and maximum values of attribute values are stored for every bucket/zone. Moreover, *PA* organizes the aggregates in hierarchies, whereas the three concepts not.

6.5 The Missing Functionality

From the index structures discussed above none was proposed for indexing hierarchical dimensional data in a data warehouse. Moreover, none of them reflects the hierarchy of dimensions. Such a feature may be useful for computing aggregates in an upper level of a dimension based on data computed for a lower level. Furthermore, none of them exploits the fact that the *Time* dimension is used in most of the star queries in predicates and is used for aggregating measures.

From commercial DBMSs, IBM DB2 supports a cluster index and a multidimensional cluster. Both data structures use B-tree based indexes to order data by the values of selected attributes. Oracle11g supports a sorted hash cluster used for the same purpose. Nonetheless, none of these data structures support indexing hierarchical dimensions.

7 Summary

In this chapter we gave an overview of the indexes typically applied to the optimization of star queries in a data warehouse, i.e., the bitmap, join, and bitmap join indexes. We showed how an example star query is executed in Oracle, which motivated us to develop an alternative index structure. We also presented the previously developed *Time-HOBI* index. This chapter introduced as additional contributions: (1) an extension to *Time-HOBI* by means of *Partial Aggregates* and (2) experimental evaluation of the extended *Time-HOBI*. The performance of the extended *Time-HOBI* was compared to the Oracle bitmap and bitmap join indexes as well as to materialized views, for the snowflake and the star DW schema.

In summary, *Time-HOBI* offers the following functionality:

- it eliminates joins of a fact table with the *Time* dimension as *Time Index* stores bit ranges for all the time intervals in the *Time* dimension;
- it eliminates joins of a fact table with other dimensions as *HOBI* stores bitmaps at all levels of a given dimension and the bitmaps point to rows in a fact table;

– it eliminates or reduces the costs of computing aggregates of measures at various levels of a dimension hierarchy as *Partial Aggregates* provide access to precomputed aggregates;
– it offers the same query performance for the snowflake and the star schema as the structure of *Time-HOBI* is the same for both of the schemas.

The experimental evaluation of *Time-HOBI* shows that:

– the index outperforms the Oracle indexes for aggregate queries, for the whole tested range of selectivities, which is true for the snowflake and the star schema;
– the index offers similar performance characteristics as the Oracle materialized views for aggregate queries, for the whole tested range of selectivities, which is true for the snowflake and the star schema;
– for the DW data volume over 7.5 GB - for the snowflake and over 16 GB - for the star schema the size of our index is smaller than the size of the Oracle indexes;
– *Time-HOBI* is larger from 2 to 3.4 times than the set of materialized views;
– our index was created from 1.1 to 1.5 times slower than the set of materialized views (as the size impacts the creation time), however, *Time-HOBI* was created from 1.17 to 3.37 times faster than the Oracle indexes.

Notice that for the experiments *Time-HOBI* was implemented as an application on top of DBMS Oracle, resulting in some additional time overheads. One may expect yet better performance while building the index in the DBMS so that it could be efficiently managed by a system and used by a query optimizer. Our ongoing work is focused on developing efficient algorithms for maintaining all the components of *Time-HOBI* as well as on embedding the index and the maintenance algorithms in the Oracle DBMS by means of the data cartridge technology [56]. Next, we plan to evaluate the performance of the implementation w.r.t. query processing and maintenance. Our research in the future will concentrate also on analyzing the impact of fine grained *Time* dimensions on the size and performance of the index.

Acknowledgment. This work was supported from the Polish National Science Center (NCN), grant No. 2011/01/B/ST6/05169. The authors express their gratitude to the anonymous Reviewers whose very thorough comments greatly improved the quality of this paper.

References

1. Netezza underground: zone maps and data power. www.ibm.com/developerworks/community/blogs/Netezza/entry/zone_maps_and_data_power20?lang=en. Accessed 27 June 2013
2. OLAP council APB-1 OLAP benchmark release 2. www.olapcouncil.org/research/APB1R2_spec.pdf. Accessed 20 Dec 2012

3. Transaction processing performance council. www.tpc.org/information/benchmarks.asp
4. Antoshenkov, G., Ziauddin, M.: Query processing and optimization in Oracle RDB. Int. J. Very Larg. Data Bases **5**(4), 229–237 (1996)
5. Aouiche, K., Darmont, J., Boussaïd, O., Bentayeb, F.: Automatic selection of bitmap join indexes in data warehouses. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 64–73. Springer, Heidelberg (2005)
6. Bellatreche, L., Missaoui, R., Necir, H., Drias, H.: A data mining approach for selecting bitmap join indices. J. Comput. Sci. Eng. **1**(2), 177–194 (2007)
7. Böhm, C., Berchtold, S., Kriegel, H., Urs, M.: Multidimensional index structures in relational databases. J. Intell. Inf. Syst. **15**(1), 51–70 (2000)
8. Brito, J.J., Siqueira, T.L.L., Times, V.C., Ciferri, R.R., de Ciferri, C.D.: Efficient processing of drill-across queries over geographic data warehouses. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2011. LNCS, vol. 6862, pp. 152–166. Springer, Heidelberg (2011)
9. Bryla, B., Loney, K.: Oracle Database 11g DBA Handbook. McGraw-Hill Osborne Media, New York (2007). ISBN 0071496637
10. Chan, C., Ioannidis, Y.: Bitmap index design and evaluation. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 355–366 (1998)
11. Chaudhuri, S., Dayal, U.: An overview of data warehousing and OLAP technology. SIGMOD Rec. **26**(1), 65–74 (1997)
12. Chmiel, J., Morzy, T., Wrembel, R.: Time-HOBI: indexing dimension hierarchies by means of hierarchically organized bitmaps. In: Proceedings of ACM International Workshop on Data Warehousing and OLAP (DOLAP), pp. 69–76 (2010)
13. Davis, K.C., Gupta, A.: Indexing in data warehouses: bitmaps and beyond. In: Wrembel, R., Koncilia, C. (eds.) Data Warehouses and OLAP: Concepts, Architectures and Solutions, pp. 179–202. Idea Group Inc., London (2007). ISBN 1-59904-364-5
14. Deliège, F., Pedersen, T.B.: Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In: Proceedings of International Conference on Extending Database Technology (EDBT), pp. 228–239. ACM (2010)
15. Finkel, R.A., Bentley, J.L.: Quad trees: a data structure for retrieval on composite keys. Acta Informatica **4**, 1–9 (1974)
16. Fonseca, M.J., Jorge, J.A.: Indexing high-dimensional data for content-based retrieval in large databases. In: Proceedings of International Conference on Database Systems for Advanced Applications (DASFAA) (2003)
17. Furtado, P.: Workload-based placement and join processing in node-partitioned data warehouses. In: Kambayashi, Y., Mohania, M., Wöß, W. (eds.) DaWaK 2004. LNCS, vol. 3181, pp. 38–47. Springer, Heidelberg (2004)
18. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2), 170–231 (1998)
19. Graefe, G.: Fast loads and fast queries. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 111–124. Springer, Heidelberg (2009)
20. Gupta, A., Mumick, I.S. (eds.): Materialized Views: Techniques, Implementations, and Applications. MIT Press, Cambridge (1999)
21. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 47–57. ACM (1984)

22. Gyssens, M., Lakshmanan, L.V.S.: A foundation for multi-dimensional databases. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 106–115 (1997)
23. Han, J., Xie, Z., Fu, Y.: Join index hierarchy: an indexing structure for efficient navigation in object-oriented databases. IEEE Trans. Knowl. Data Eng. **11**(2), 321–337 (1999)
24. Jarke, M., Lenzerini, M., Vassiliou, Y., Vassiliadis, P.: Fundamentals of Data Warehouses. Springer, Heidelberg (2003). ISBN 3-540-42089-4
25. Jensen, C.S., Pedersen, T.B., Tomsen, C.: Multidimensional Databases and Data Warehousing. Morgan & Claypool Publishers, San Rafael (2010). ISBN 978-1-60845-537-9
26. Johnson, T., Sasha, D.: The performance of current B-tree algorithms. ACM Trans. Database Syst. (TODS) **18**(1), 51–101 (1993)
27. Karayannidis, N., Tsois, A., Sellis, T.: Advanced ad hoc star query processing. In: Wrembel, R., Koncilia, C. (eds.) Data Warehouses and OLAP: Concepts, Architectures and Solutions, pp. 136–156. Idea Group Inc., London (2007). ISBN 1-59904-364-5
28. Kemper, A., Moerkotte, G.: Access support in object bases. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 364–374 (1989)
29. Koudas, N.: Space efficient bitmap indexing. In: Proceedings of ACM Conference on Information and Knowledge Management (CIKM), pp. 194–201 (2000)
30. Li, C., Tang, C., Yu, Z., Liu, Y., Zhang, T., Liu, Q., Zhu, M., Jiang, Y.: Mining multi-dimensional frequent patterns without data cube construction. In: Yang, Q., Webb, G. (eds.) PRICAI 2006. LNCS (LNAI), vol. 4099, pp. 251–260. Springer, Heidelberg (2006)
31. Li, K.A., Ross, Z.: Fast joins using join indices. Int. J. Very Larg. Data Bases **8**(1), 1–24 (1999)
32. Malinowski, E., Zimányi, E.: Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications. Springer, Heidelberg (2008). ISBN 9783540744047
33. Moerkotte, G.: Small materialized aggregates: a light weight index structure for data warehousing. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 476–487 (1998)
34. Morzy, M.: Advanced database structure for efficient association rule mining. Ph.D. thesis, Poznan University of Technology, Institute of Computing Science (2004)
35. Morzy, M., Morzy, T., Nanopoulos, A., Manolopoulos, Y.: Hierarchical bitmap index: an efficient and scalable indexing technique for set-valued attributes. In: Kalinichenko, L.A., Manthey, R., Thalheim, B., Wloka, U. (eds.) ADBIS 2003. LNCS, vol. 2798, pp. 236–252. Springer, Heidelberg (2003)
36. Morzy, T., Wrembel, R., Chmiel, J., Wojciechowski, A.: Time-HOBI: index for optimizing star queries. Inf. Syst. **37**(5), 412–429 (2012)
37. Nourani, M., Tehranipour, M.H.: RL-Huffman encoding for test compression and power reduction in scan applications. ACM Trans. Des. Autom. Electron. Syst. (TODAES) **10**(1), 91–115 (2005)
38. O'Neil, P.E.: Model 204 architecture and performance. HPTS 1989. LNCS, vol. 359, pp. 39–59. Springer, Heidelberg (1989)
39. O'Neil, P., Graefe, G.: Multi-table joins through bitmapped join indices. SIGMOD Rec. **24**(3), 8–11 (1995)

40. O'Neil, P., O'Neil, E., Chen, X., Revilak, S.: The star schema benchmark and augmented fact table indexing. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 237–252. Springer, Heidelberg (2009)
41. O'Neil, P., Quass, D.: Improved query performance with variant indexes. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 38–49 (1997)
42. Padmanabhan, S., Bhattacharjee, B., Malkemus, T., Cranston, L., Huras, M.: Multi-dimensional clustering: a new data layout scheme in DB2. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 637–641 (2003)
43. Papadias, D., Kalnis, P., Zhang, J., Tao, Y.: Efficient OLAP operations in spatial data warehouses. In: Jensen, C.S., Schneider, M., Seeger, B., Tsotras, V.J. (eds.) SSTD 2001. LNCS, vol. 2121, pp. 443–459. Springer, Heidelberg (2001)
44. Papadias, D., Tao, Y., Kalnis, P., Zhang, J.: Indexing spatio-temporal data warehouses. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 166–175. IEEE Computer Society (2002)
45. Rao, J., Zhang, C., Megiddo, N., Lohman, G.: Automating physical database design in a parallel database. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 558–569 (2002)
46. Robinson, J.T.: The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 10–18. ACM (1981)
47. Rotem, D., Stockinger, K., Wu, K.: Optimizing candidate check costs for bitmap indices. In: Proceedings of ACM Conference on Information and Knowledge Management (CIKM), pp. 648–655 (2005)
48. Scientific Data Management Research Group: FastBit: an efficient compressed bitmap index technology. http://sdm.lbl.gov/fastbit/. Accessed 10 Nov 2006
49. Sinha, R.R., Mitra, S., Winslett, M.: Bitmap indexes for large scientific data sets: a case study. In: Proceedings of Parallel and Distributed Processing Symposium. IEEE (2006)
50. Sinha, R.R., Winslett, M.: Multi-resolution bitmap indexes for scientific data. ACM Trans. Database Syst. (TODS) **32**(3), 1–38 (2007)
51. Siqueira, T.L., Ciferri, C.D.D., Times, V.C., Ciferri, R.R.: The sb-index and the hsb-index: efficient indices for spatial data warehouses. Geoinformatica **16**(1), 165–205 (2012)
52. Stabno, M., Wrembel, R.: RLH: bitmap compression technique based on run-length and Huffman encoding. Inf. Syst. **34**(4–5), 400–414 (2009)
53. Stockinger, K., Wu, K.: Bitmap indices for data warehouses. In: Wrembel, R., Koncilia, C. (eds.) Data Warehouses and OLAP: Concepts, Architectures and Solutions, pp. 157–178. Idea Group Inc., London (2007). ISBN 1-59904-364-5
54. Stockinger, K., Wu, K., Shoshani, A.: Evaluation strategies for bitmap indices with binning. In: Galindo, F., Takizawa, M., Traunmüller, R. (eds.) DEXA 2004. LNCS, vol. 3180, pp. 120–129. Springer, Heidelberg (2004)
55. Stöhr, T., Rahm, E.: Warlock: a data allocation tool for parallel warehouses. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 721–722 (2001)
56. Technical Documentation. Oracle Database Data Cartridge Developer's Guide 11g Release 1(11.1)
57. Valduriez, P.: Join indices. ACM Trans. Database Syst. (TODS) **12**(2), 218–246 (1987)

58. Wrembel, R.: Data warehouse performance: selected techniques and data structures. In: Aufaure, M.-A., Zimányi, E. (eds.) eBISS 2011. LNBIP, vol. 96, pp. 27–62. Springer, Heidelberg (2012)
59. Wu, K., Otoo, E.J., Shoshani, A.: An efficient compression scheme for bitmap indices. Research report, Lawrence Berkeley National Laboratory (2004)
60. Wu, K., Otoo, E.J., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 24–35 (2004)
61. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. (TODS) **31**(1), 1–38 (2006)
62. Wu, K., Yu, P.: Range-based bitmap indexing for high cardinality attributes with skew. In: International Computer Software and Applications Conference (COMPSAC), pp. 61–67 (1998)
63. Wu, M., Buchmann, A.: Encoded bitmap indexing for data warehouses. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 220–230 (1998)
64. Xu, X., Han, J., Lu, W.: RT-tree: an improved R-tree index structure for spatiotemporal databases. In: International Symposium on Spatial Data Handling, pp. 1040–1049 (1990)
65. Yiu, M.L., Mamoulis, N.: Efficient processing of top-k dominating queries on multidimensional data. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 483–494 (2007)
66. Yu, T.C., Meng, W.: Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann, San Francisco (1998). ISBN 1-55860-434-0
67. Zhuang, Y., Zhuang, Y., Li, Q., Chen, L., Yu, Y.: Indexing high-dimensional data in dual distance spaces: a symmetrical encoding approach. In: Proceedings of International Conference on Extending Database Technology (EDBT), pp. 241–251 (2008)