2

Data Warehouse Performance: Selected Techniques and Data Structures

Robert Wrembel

Institute of Computing Science
Poznań University of Technology
Poznań, Poland
Robert.Wrembel@cs.put.poznan.pl

Summary. Data stored in a data warehouse (DW) are retrieved and analyzed by complex analytical applications, often expressed by means of star queries. Such queries often scan huge volumes of data and are computationally complex. For this reason, an acceptable (or good) DW performance is one of the important features that must be guaranteed for DW users. Good DW performance can be achieved in multiple components of a DW architecture, starting from hardware (e.g., parallel processing on multiple nodes, fast disks, huge main memory, fast multi-core processor), through physical storage schemes (e.g., row storage, column storage, multidimensional store, data and index compression algorithms), state of the art techniques of query optimization (e.g., cost models and size estimation techniques, parallel query optimization and execution, join algorithms), and additional data structures improving data searching efficiency (e.g., indexes, materialized views, clusters, partitions). In this chapter we aim at presenting only a narrow aspect of the aforementioned technologies. We discuss three types of data structures, namely indexes (bitmap, join, and bitmap join), materialized views, and partitioned tables. We show how they are being applied in the process of executing star queries in three commercial database/data warehouse management systems, i.e., Oracle, DB2, and SQL Server.

Keywords: data warehouse, star query, join index, bitmap index, bitmap join index, materialized view, query rewriting, data partitioning, Oracle, SQL Server, DB2.

2.1 Introduction

A *data warehouse architecture* has been developed in order to integrate and analyze data coming from multiple distributed, heterogeneous, and autonomous data sources (DSs), deployed throughout an enterprise. A core component of this architecture is a database, called a *data warehouse* (DW), that stores current and historical data, integrated from multiple DSs. The content of a DW is analyzed by various On-Line Analytical Processing (OLAP) applications 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

M.-A. Aufaure and E. Zimányi (Eds.): eBISS 2011, LNBIP 96, pp. 27–62, 2012.

dependencies between data (e.g., market basket analysis, suggested buying, insurance fee assessment). OLAP applications execute complex queries, some of them being predefined (e.g. reports), whereas others being executed ad-hoc by decision makers.

The queries are expressed with multiple join, filtering, aggregate, and sort operations and they process large (or extremely large) volumes of data. A special class of analytical queries includes *star queries* that join a central table with multiple referenced tables. The execution of analytical queries may take hours or even days. For this reason, providing means for increasing the performance of a data warehouse for analytical queries and other types of data processing (e.g., data mining) is one of the important research and technological areas.

2.1.1 Increasing DW Performance: Research and Technological Advances

A DW performance depends on multiple components of a DW architecture. The basic components include: (1) hardware on which a database/data warehouse management system (DB/DWMS) is installed and computational architectures, (2) physical storage schemes of data, (3) robustness of a query optimizer, (4) additional data structures supporting faster data access.

Hardware Architectures

Hardware and various processing architectures, e.g., shared memory, shared disk, shared nothing, have a substantial impact on the performance of a DW. Multiple nodes with their processing power allow to process data in parallel. Modern cluster, grid, and cloud architectures take advantage of parallel data access and processing. A lot of research efforts focus on this area, e.g., [1, 2, 3]. Recent research and technological trends concentrate also on using parallel processing of powerful graphic processing units, e.g,. [4, 5, 6, 7, 8] for processing data. Another hot topic research and technological issue concerns main memory databases and data warehouses, e.g., [9, 10, 11].

Physical Storage

In practice, a DW is implemented either in a relational server (the ROLAP implementation) or in a multidimensional server (the MOLAP implementation). The ROLAP implementation can be based either on a row storage (RS) (a typical one) or on a column storage (CS) (current trend). RS is more suitable for transactional processing (inserts, deletes, updates). CS offers better performance for applications that read and compute aggregates based on the subset of table columns, thus it is better suited for OLAP processing. An intensive research is conducted in the area of column store DWMSs, e.g.,

[12, 13, 14] in order to develop efficient join algorithms, query materialization techniques, index data structures, and compression techniques [15], to mention some of them. These and many other efforts resulted in commercially available and open source column storage DWMS, e.g., Sybase IQ, EMC Greenplum, C-Store/Vertica, MonetDB, Sadas, FastBit, Model 204. The MOLAP implementation is based on other storage structures, like multidimensional arrays and multidimensional indexes. In this technology research is focused among others on developing efficient storage implementations, index structures, and compression techniques, e.g., [16, 17].

Query Optimizer

The way queries are executed strongly impacts the performance of a DW. The process of optimizing query executions and building robust query optimizers has been receiving substantial focus from the research community. Huge research literature exists on this issue, cf., [18]. Recently, research in this area concentrates among others on join algorithms, e.g., [19], parallel query optimization and execution, e.g., [20, 21], designing robust and more intelligent query optimizers [22].

Querying

Typically, OLAP applications analyze all data in order to deliver reliable results. Nonetheless, if volumes of data are extremely large, some researchers propose to apply various techniques to computing approximate results, like for example (1) sampling, based on histograms or wavelets, e.g., [23, 24], (2) statistical properties of data, or (3) apply a probability density function, e.g., [25].

Data Structures

Query execution plans profit from additional data structures that make data searching faster and reduce the volume of data that has to be retrieved. Different data structures have been developed and applied in practice in commercial DWMSs. Some of them include: (1) various types of indexes, like join indexes, e.g., [26], bitmap indexes, e.g., [27, 28], and bitmap join indexes, e.g., [29, 30], (2) materialized views and query rewriting techniques, e.g., [31], (3) table partitioning, e.g., [32, 33, 34]. In these areas multiple research works focus on compressing bitmap indexes, e.g., [35, 36, 37, 28], algorithms for materialized view selection and fast refreshing, e.g., [38, 31, 39, 40], and finding the most efficient partitioning schemes, e.g., [41, 42].

Testing Performance

An important practical issue concerns the ability of evaluating the performance of a DWMS and compare multiple architectures with this respect.

To this end, multiple approaches to testing the performance and robustness of a DWMS have been developed, e.g., [43, 44, 45, 46] and are being further intensively investigated.

2.1.2 Chapter Focus

In this chapter we overview a narrow area only of the huge research and technological area devoted to increasing a data warehouse performance. We focus on the aforementioned data structures (indexes, materialized views, and partitions) in ROLAP servers and illustrate their basic functionality and usage in commercial DWMSs, i.e., Oracle, DB2, and SQL Server.

The chapter is organized as follows. In Section 2.2 we present basic data warehouse concepts and an example data warehouse used throughout this chapter. In Section 2.3 we present index data structures applied to the optimization of star queries. In Section 2.4 we discuss a technique of materializing query results and using the materialized results for query optimization. In Section 2.5 we present table partitioning techniques. Finally, Section 2.6 includes the chapter summary.

2.2 Basic Concepts

2.2.1 DW Model and Schema

In order to support various analyses, data stored in a DW are represented in a *multidimensional data model* [47, 48]. 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 [49]. Following the aggregation path, data can be aggregated along a dimension hierarchy.

The multidimensional model is often implemented in relational OLAP servers (ROLAP) [50], where fact data are stored in a *fact table*, and level data are stored in *dimension tables*. In a ROLAP implementation two basic types of conceptual schemas are used, i.e. a star schema and a snowflake schema [50]. In the *star schema* each dimension is composed of only one (typically denormalized) level table. In the *snowflake schema* each dimension is composed of multiple normalized level tables connected by foreign key - primary key relationships.

An example star schema that will be used throughout this chapter is shown in Figure 2.1. It is composed of the *Sales* fact table, and three dimension tables, namely *Products*, *Customers*, and *Time*. The *Sales* fact table is connected with its dimension table via three foreign keys, namely *ProductID*, *CustomerID*, and *TimeKey*. The fact table includes measure column *SalesPrice*.

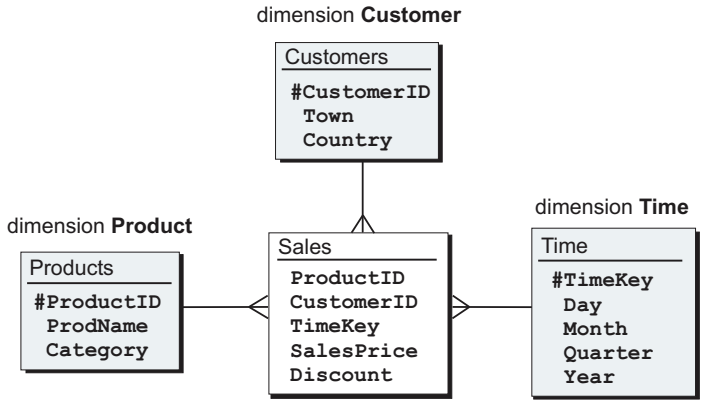


Fig. 2.1. Example data warehouse star schema on sales of products

2.2.2 Star Queries

Based on a DW schema, analytical queries are executed in a DW. As mentioned earlier, such queries typically join multiple tables, filter and sort data, as well as aggregate data at different levels of dimension hierarchies. Typically, the queries join a fact table with multiple dimension tables and are called *star queries*.

An example star query computing the yearly sales (in years 2009 and 2010) of products belonging to category 'electronic' in countries of sales, is shown below.

```
select sum(SalesPrice), ProdName, Country, Year
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
and s.CustomerID=c.CustomerID
and s.TimeKey=t.TimeKey
and p.Category in ('electronic')
and t.Year in (2009, 2010)
group by ProdName, Country, Year;
```

2.3 Index Data Structures

Star queries can profit from applying some indexes in the process of retrieving data. Three indexes, the most frequently used in practice include: a join index, a bitmap index, and a bitmap join index, which are outlined in this section.

2.3.1 Join Index

A *join index* represent the materialized join of two tables, say R and S. As defined in [51, 26], 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 tuple identifiers from R and S, respectively, that join on a given predicate. In order to make searching the join index faster, it is assumed that the join index 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 [30].

A join index can be created either on a join attribute or on a non-join attribute of a table. In the second case, the index is organized in such a way that it allows lookups by the value of the non-joined attribute in order to retrieve the ROWIDs of rows from the joined tables that join with the non-joined attribute value.

In the context of a data warehouse, a join index is applied to joining a dimension table and a fact table. The index is created either on a join attribute of a dimension table or on another attribute (typically storing unique values) of a dimension table. In order to illustrate the idea behind the join index let us consider the example below.

Example 1. Let us consider tables *Products* and *Sales* from the DW schema shown in Figure 2.1. Their content is shown in Table 2.1. For clarity purpose, both tables include also explicit column *ROWID* that stores physical addresses of records and that serve as row identifiers.

Table 2.1. Example tables in the star schema on sales of products (from Fig.2.1)

Sales			
ROWID	SalesPrice	Discount	ProductID
0AA0	...	5	100
0AA1	...	15	230
0AA2	...	5	100
0AA3	...	10	300
0AA4	...	10	300
0AA5	...	15	230

Products			
ROWID	ProductID	ProdName	Category
BFF1	100	HP Pavilion	electronic
BFF2	230	Dell Inspiron	electronic
BFF3	300	Acer Ferrari	electronic

The join index defined on column *ProductID* is shown in Table 2.2.

Table 2.2. Example join index on ProductID

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

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

```
select ...
from Sales s, Products p
where s.ProductID=p.ProductID ...
```

2.3.2 Bitmap Index

OLAP queries not only join data, but also filter data by means of query predicates. Efficient filtering of large data volumes is well supported by the so-called bitmap indexes [52, 27, 28, 53]. Conceptually, a *bitmap index* created on 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 equals 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 equals to val_i. Otherwise the bit is set to 0.

In order to illustrate the idea behind the bitmap index let us consider the example below.

Example 2. Let us review the *Sales* fact table, shown in Table 2.3. The table contains attribute *Discount* whose domain includes three values, namely *5*, *10*, and *15*, denoting a percent value of discount. A bitmap index created on this attribute will be composed of three bitmaps, noted as *Bm5perc*, *Bm10perc*, and *Bm15perc*, respectively, as shown in Table 2.3.

The first bit in bitmap *Bm15perc* equals to 0 since the *Discount* value of the first record in table *Sales* does not equal *15*. The second bit in bitmap *Bm15perc* equals to 1 since the *Discount* value of the second record in table *Sales* equals to *15*, etc.

Such bitmap index will offer a good response time for a query selecting for example data on sales with 5% or 15% discounts. In order to find sales records fulfilling this criterion, it is sufficient to OR bitmaps *Bm5perc* and *Bm15perc* in order to construct a result bitmap. Then, records pointed to by bits equal to '1' in the result bitmap are fetched from the *Sales* table. □

At the implementation level, bitmap indexes are organized either as B-trees [27] or as simple arrays in a binary file [54]. In the first case, B-tree leaves store bitmaps and a B-tree is a way to organize indexed values and bitmaps in files.

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 equaled to '1' a result bitmap. Moreover, such indexes offer a good query performance for attributes of narrow domains. For such attributes, a bitmap index will be much smaller than a traditional B-tree

Table 2.3. Example bitmap index created on the *Discount* attribute

Sales			bitmap index on *Discount*		
SalesPrice	Discount	...	Bm5perc	Bm10perc	Bm15perc
...	5	...	1	0	0
...	15	...	0	0	1
...	5	...	1	0	0
...	10	...	0	1	0
...	10	...	0	1	0
...	15	...	0	0	1

index. Additionally, while evaluating queries with multiple predicates with equality and inequality operators, a system processes bitmaps very fast by AND-ing/OR-ing them.

The size of a bitmap index strongly depends on the cardinality (domain width) of an indexed attribute, i.e., the size of a bitmap index increases when the cardinality of an indexed attribute increases. Thus, for attributes of high cardinalities (wide domains) bitmap indexes become very large (much larger than a B-tree index). As a consequence, they cannot fit into main memory and the efficiency of accessing data with the support of such indexes deteriorates [55].

In order to reduce the size of bitmap indexes defined on attributes of high cardinalities, two following approaches have been proposed in the research literature, namely: (1) extensions to the structure of the basic bitmap index, and (2) bitmap index compression techniques. In the first approach, two main techniques, generally called binning as well as bit slicing, can be distinguished.

Extensions to the Structure of the Basic Bitmap Index

In [56] (called *range-based bitmap indexing*) and in [57, 58, 59] (called *binning*), values of an indexed attribute are partitioned into ranges. A bitmap is constructed for representing a given range of values, rather than a distinct value. Bits in a single bitmap indicate whether the value of a given attribute of a row is within a specific range. This technique can also be applied when values of an indexed attribute are partitioned into sets.

The technique proposed in [60] can be classified as a more general form of binning. In [60] sets of attribute values are represented together in a bitmap index. Such a technique reduces storage space for attributes of high cardinalities. The selection of attribute values represented in this kind of an index is based on query patterns and their frequencies, as well as on the distribution of attribute values.

Another form of binning was proposed in [52]. This technique, called *property maps*, focuses on managing the total number of bins assigned to all indexed attributes. A property map defines properties on each attribute, such

as the set of queries using the attribute, distribution of values for the attribute, or encoded values of the attribute. The properties are represented as vectors of bits. A query processor needs extension in order to use property maps. Property maps support multi-attribute queries, inequality queries or high selectivity queries, and they are much smaller than bitmap indexes.

The second technique is based on the so-called *bit-sliced index* [61, 62, 55]. It is defined as an ordered list of bitmaps, $B^n, B^{n-1}, \ldots, B^1, B^0$, that are used for representing values of a given attribute A, i.e., B^0 represents the 2^0 bit, B^1 represents the 2^1 bit, etc. Every value of an indexed attribute is represented on the same number of n bits. As a result, the encoded values in a table form n bitmaps. The bitmaps are called bit-slices. Data retrieval and computation are supported either by the bit-sliced index arithmetic [63] or by means of a dedicated retrieval function [55]. Additionally, a mapping data structure is required for mapping the encoded values into their real values [55].

Compression Techniques

The second approach that allows to reduce the size of a bitmap index defined on an attribute of high cardinality is based on compression. Four main loss-less techniques can be distinguished in the research literature, namely: (1) *Byte-aligned Bitmap Compression* (BBC) [64], (2) *Word-Aligned Hybrid* (WAH) [28, 65, 53, 66], (3) Position List WAH (PLWAH) [67, 35], and (4) RL-Huffman [36] and RLH [68, 37].

All the aforementioned compression techniques apply the *run-length* encoding. The basic idea of the run-length encoding consists in encoding continuous vectors of bits having the same value (either "0" or "1") into: (1) a common value of all bits in the vector (i.e., either "0" for a vector composed of zeros or "1" for a vector composed of ones) and (2) the length of the vector (i.e., the number of bits having the same value). Before encoding, a bitmap is divided into words. Next, words are grouped into the so-called runs. The *run* is composed of words that can be either a fill or a tail. The *fill* represents series of words that are composed of bits of the same value. The *tail* represents the series of words that are composed of both "0" and "1" bits. Fills are compressed because of their homogeneous content, whereas tails are not.

BBC divides bit vectors into 8-bit words, WAH and PLWAH divide them into 31-bit words, RLH uses words of a parameterized length, whereas RL-Huffman does not divide a bitmap into words. PLWAH is the modification of WAH. PLWAH improves compression if tail T that follows fill F differs from F on few bits only. In such a case, the fill word encodes the difference between T and F on some dedicated bits. Moreover, BBC uses four different types of runs, depending on the length of a fill and the structure of a tail, whereas the other compression techniques use only one type of a run. The overall idea of the WAH compression is illustrated with the example below taken from [28].

Example 3. For the sake of simplicity let us assume that a 32-bit processor is used. A bitmap being compressed is composed of 5456 bits, as shown in Figure 2.2a. The WAH compression of the bitmap is executed in three following steps.

In the first step, the bitmap is divided into groups composed of 31 bits, as shown in Figure 2.2b. In the example, 176 such groups are created. In the second step, adjacent groups containing identical bits are merged into one group, as shown in Figure 2.2c. Since *group 1* is heterogeneous, i.e., it is composed of "0" and "1" bits, it is not merged with a group following it. Groups 2 to 175 are homogeneous (composed of "0" bits) and they are merged into one large group, denoted in Figure 2.2c as *group 2-175*. This group includes 174*31 bits. The last *group 176*, similarly as *group 1*, is heterogeneous and it cannot be merged with a group preceding it. As the result of group merging, three final groups are created, as shown in Figure 2.2c.

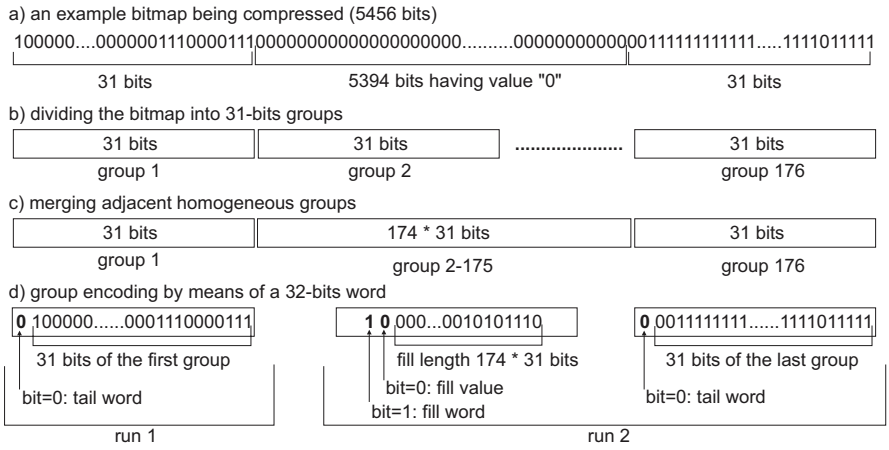


Fig. 2.2. The steps of the WAH compression

In the third step, the three final groups are encoded on 32-bit words as follows (cf. Figure 2.2d). The first group represents the tail of the first run. The most significant bit (the leftmost one) has value "0" denoting a tail. Next 31 bits are original bits of *group 1*. The second group (*group 2-175*) represents the fill of the second run. The most significant bit (at position ^{31}) is set to "1" denoting a fill. The bit at position 2^{30} is set to "0" denoting that all bits in original *group 2-175* have value "0", i.e., the fill is used for compressing groups whose all bits have value "0". The remaining 30 bits are used for encoding the number of homogeneous groups filled with "0." In the example, there are 174 such groups. The number of homogeneous groups is represented by the binary value equaled to 00000000000000000000000010101110, stored on the remaining 30 bits. The last 31-bits group, denoted as *group 176*, represents

the tail of the second run. The most significant bit in this group has value "0" denoting a tail. The remaining 31 bits are original bits of *group 176*. □

The compression techniques proposed in [36] and [37] additionally apply the Huffman compression [69] to the run-length encoded bitmaps. The main differences between [36] and [37] are as follows. First, in [36] only some bits in a bit vector are of interest, the others, called 'don't cares' can be replaced either by zeros or ones, depending on the values of neighbor bits. In RLH all bits are of interest and have their exact values. Second, in [36] the lengths of homogeneous subvectors of bits are counted and become the symbols that are encoded by the Huffman compression. RLH uses run-length encoding for representing distances between bits having value 1. Next, the distances are encoded by the Huffman compression.

In practice, the commercial systems (Oracle DBMS) and prototype systems (FastBit) [70, 71, 54] apply bitmap compression techniques. Oracle uses the BBC compression whereas FastBit uses the WAH compression.

Bitmap Indexes in Oracle

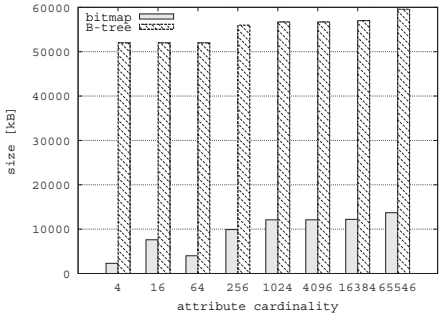
The Oracle DBMS [72] supports explicitly created bitmap indexes. A bitmap index is created by means of the below command.

```
create bitmap index BIName on table(column);
```

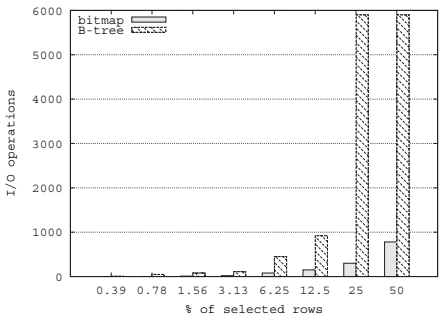
As mentioned in Section 2.3.2, the size of a bitmap index increases with the increase of the cardinality of the indexed attribute. In order to reduce the size of such a bitmap index, Oracle DBMS applies a compression based on BBC. When the sparsity of bitmaps exceeds a given threshold, Oracle automatically compresses bitmaps. Figure 3(a) shows how the size of a bitmap index depends on the cardinality of the indexed attribute. The cardinality was parameterized from 4 to 65564 unique values. The experiment was conducted on Oracle10g R2. The indexed table included 320 000 000 of rows. As a reference, the size of the Oracle B*-tree is also included in the chart.

As we can observe from chart 3(a), with the increase of cardinality from 4 to 16 the size of the bitmap index increases. Next, for cardinality equal to 64, the size of the BI decreases that is caused by compressing the BI. Next, with the increase of the cardinality above 64 we observe further increase in the size of the compressed BI.

Bitmap indexes support fast computations of function `count`. Figure 3(b) shows the computation efficiency of `count(1)` for various selectivities of a query. The query selected from 0.39% to 50% of rows based on the values of the indexed attribute. As a reference, the figure includes also the performance of the Oracle B*-tree index. The characteristics of the indexes were tested on identical attributes. It must be noticed that when `count` is replaced by other aggregate functions, like for example `sum`, `avg`, the bitmap index performs much worse, but still better than B*-tree. It is because, in order to compute these aggregate functions data must be fetched from disk.



(a) The size of a bitmap index and a B*-tree index



(b) The computation efficiency of `count(1)` with the support of a bitmap index and a B*-tree index

Fig. 2.3. Comparison of some features of a bitmap index and a B*-tree index in Oracle (# of rows in an indexed table: 320 000 000)

Bitmap Indexes in DB2

IN the DB2 DBMS, queries with predicates on multiple indexed attributes can be optimized by means of the *index AND-ing* technique. In general, this technique is based on determining the intersection of the sets of ROWIDs, which are retrieved with the support of individual indexes on attributes used in query predicates. The intersection is determined with the support of bloom filters [73, 74]. The bloom filters transform each set of ROWIDs into a separate temporal bitmap. Multiple bitmaps can then be further processed by AND-ing or OR-ing them, depending on query predicates. The bitmaps are called *dynamic bitmap indexes*.

As an example illustrating the application of a dynamic bitmap index to the optimization of a query on one table, let us consider table *Sales* with two B-tree indexes, one defined on attribute *ProductID* and one on *CustomerID*. An example query with predicate `ProductID in ('ThE1', 'SoV1', 'DeV1') and CustomerID = 100` could be answered by applying the index AND-ing technique with the support of a dynamic bitmap index as follows. First, by using index on *ProductID* a bitmap is created that describes rows fulfilling the predicate `ProductID in ('ThE1', 'SoV1', 'DeV1')`. Let us denote the bitmap as B_{Prod}. Similarly, by using index on *CustomerID* a bitmap, say B_{Cust} is created that describes rows fulfilling the predicate `CustomerID = 100`. Second, the final bitmap is computed by AND-ing B_{Prod} and B_{Cust}. Based on the final bitmap that is transformed back to ROWIDs, rows fulfilling the predicates are retrieved.

As an example illustrating the application of dynamic bitmap indexes to optimizing star queries, let us consider star query $Q1$ from Section 2.3.3. In order to use the technique, a B-tree index has to be created on each of the foreign keys. In the first step, every dimension table is semi-joined with the *Sales* table in order to determine the set of *Sales* rows that join with rows form each of the dimensions. Let the result of $Sales \ltimes Products$, i.e., the set of ROWIDs fulfilling the predicate on *ProdName* is denoted as S_{Prod}. Let the result $Sales \ltimes Customers$, i.e., the set of ROWIDs fulfilling the predicate on *Town* is denoted as S_{Cust}. Finally, let $Sales \ltimes Time$, i.e., the set of ROWIDs fulfilling the predicate on *Year* is denoted as S_{Time}. In the second step, S_{Prod}, S_{Cust}, and S_{Time} are transformed to three bitmaps, denoted as B_{Prod}, B_{Cust}, and B_{Time}, respectively. In the third step, bitmaps B_{Prod}, B_{Cust}, B_{Time} are AND-ed. The final bitmap describes rows fulfilling the whole query predicate. It is then transformed back to ROWIDs. Based on the ROWIDs records are fetched from the *Sales* table. If some columns from dimension tables are needed in the query result, then the final set of sales rows is joined with appropriate rows from the dimensions.

Bitmap Indexes in SQL Server

Similarly as DB2, SQL Server [75] does not support explicitly created bitmap indexes. Instead, it supports bitmap filters (this mechanism is available from version 2005). However, bitmap filters are not bitmap indexes. The *bitmap filter* represents (in a compact format) the set of values from one table being joined, typically a dimension table. Based on the bitmap filter, rows from the second joined table, typically a fact table, are filtered. Thus, the bitmap filter is applied as a semi-join reduction technique but only in parallel execution plans. A bitmap filter is automatically created by the SQL Server query optimizer when the optimizer estimates that such a filter is selective. A bitmap filter is a main memory data structure.

In order to illustrate the application of bitmap filters to the optimization of a star query, let us consider star query $Q1$ from Section 2.3.3. It could be

processed as follows. In the first step, bitmap filters are created. Bitmap filter BF_{Prod} is created, based on values ('ThinkPad Edge', 'Sony Vaio', 'Dell Vostro') from the *Products* dimension table. In parallel, bitmap filters on dimension tables *Customers* and *Time* are created, based on the predicates `Town='London'` and `Year=2009`, respectively. Let us denote the filters as BF_{Cust} and BF_{Year}, respectively. In the second step, BF_{Prod} is used for semi-join reduction with the *Sales* table (let denote the intermediate result as temporary table $Sales^{reduced}_{Prod}$). In the third step, bitmap filter BF_{Cust} is applied as semi-join reduction operator to temporary table $Sales^{reduced}_{Prod}$. Let us denote its result as $Sales^{reduced}_{Prod\_Cust}$. In the fourth step bitmap filter BF_{Year} is applied to $Sales^{reduced}_{Prod\_Cust}$, resulting in temporary table $Sales^{reduced}_{Prod\_Cust\_Year}$. Finally, rows fulfilling the whole predicate of the query are fetched by means of $Sales^{reduced}_{Prod\_Cust\_Year}$. The order in which the bitmap filters are applied depends on the selectivity of the filters. The most selective one should be applied first.

2.3.3 Bitmap Join Index

The advantages of the join index and the bitmap index have been combined in a *bitmap join index* [76, 30, 62]. This index, conceptually is organized as the join index but the entries to the bitmap join index are organized as a lookup by the value of a dimension attribute. Each entry to the bitmap join index is composed of the ROWID of a row from a dimension table (or an attribute uniquely identifying a row in a dimension table) and a bitmap (possibly compressed) describing rows from a fact table that join with this value. Similarly as for the 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 below.

Example 4. Let us return to Example 1 and let us define the bitmap join index on attribute *ProductID* of table *Products*. Conceptually, the entries of this index are shown in Table 2.4. The lookup of the index is organized by the values of attribute *ProductID*. Assuming that the leftmost bit in each of the bitmaps represents the first row in the *Sales* table, one can see that the first and third sales row join with product identified by value 100, for example.

Table 2.4. Example bitmap join index organized as a lookup by attribute *Products.ProductID*

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

Every indexed value (e.g. 100) in dimension table *Products* has associated a bitmap describing records from fact table *Sales* that join with the dimension record. □

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 AND, OR, NOT operations on bitmaps.

Bitmap Join Indexes in Oracle

Oracle also supports the *bitmap join index* that offers a very good performance of star queries. As an example let us consider the below star query $Q1$, that joins the *Sales* fact table with all of its dimension tables.

```
/* query Q1 */
select sum(SalesPrice)
from Sales, Products, Customers, Time
where Sales.ProductID=Products.ProductID
and Sales.CustomerID=Customers.CustomerID
and Sales.TimeKey=Time.TimeKey
and ProdName in ('ThinkPad Edge', 'Sony Vaio', 'Dell Vostro')
and Town='London'
and Year=2009;
```

In order to reduce the execution time of $Q1$, three bitmap indexes can be created, as shown below.

```
create bitmap index BI_Pr_Sales
on Sales(Products.ProdName)
from Sales, Products
where Sales.ProductID=Products.ProductID;

create bitmap index BI_Cu_Sales
on Sales(Customers.Town)
from Sales, Customers
where Sales.CustomerID=Customers.CustomerID;

create bitmap index BI_Ti_Sales
on Sales(Time.Year)
from Sales, Time
where Sales.TimeKey=Time.TimeKey;
```

The query execution plan, shown below, reveals how the bitmap join indexes are used. First, bitmaps for 'ThinkPad Edge', 'Sony Vaio', and 'Dell Vostro' are retrieved and OR-ed (cf. lines 10-12), creating a temporary bitmap. Next, the bitmap for 'London' is retrieved (cf. line 8) and it is AND-ed with the temporary bitmap (cf. line 7). The result bitmap is then converted to ROWIDs of records fulfilling the criteria (cf. line 6). In this query plan table *Time* was accessed with the support of a B-tree index defined on its primary key, rather than by means of the bitmap join index.

```
-------------------------------------------------------------------------------------
|Id  | Operation                      |Name          |Rows |Bytes |Cost(%CPU)|Time    |
-------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT                |              |    1|   58 |   13  (8)|00:00:01|
| 1 |  SORT AGGREGATE                 |              |    1|   58 |          |        |
| 2 |   NESTED LOOPS                  |              |   21| 1218 |   13  (8)|00:00:01|
| 3 |    HASH JOIN                    |              |   22| 1012 |   12  (9)|00:00:01|
| 4 |     TABLE ACCESS FULL           |PRODUCTS      |    3|   51 |    3  (0)|00:00:01|
| 5 |     TABLE ACCESS BY INDEX ROWID |SALES         | 1155|33495 |    8  (0)|00:00:01|
| 6 |      BITMAP CONVERSION TO ROWIDS|              |     |      |          |        |
| 7 |       BITMAP AND                |              |     |      |          |        |
| 8 |        BITMAP INDEX SINGLE VALUE|BI_CU_SALES   |     |      |          |        |
| 9 |        BITMAP OR                |              |     |      |          |        |
|10 |         BITMAP INDEX SINGLE VALUE|BI_PR_SALES   |     |      |          |        |
|11 |         BITMAP INDEX SINGLE VALUE|BI_PR_SALES   |     |      |          |        |
|12 |         BITMAP INDEX SINGLE VALUE|BI_PR_SALES   |     |      |          |        |
|13 |    TABLE ACCESS BY INDEX ROWID  |TIME          |    1|   12 |    1  (0)|00:00:01|
|14 |     INDEX UNIQUE SCAN           |PK_TIME       |    1|      |    0  (0)|00:00:01|
-------------------------------------------------------------------------------------
```

Alternatively, it is possible to create one bitmap join index on all of the dimension attributes. The example below command creates the bitmap join index on three attributes of the dimension tables, namely: *Products.prodName*, *Customers.Town*, and *Time.Year*.

```
create bitmap index BI_Pr_Cu_Ti_Sales
on Sales(Products.ProdName, Customers.Town, Time.Year)
from Sales, Products, Customers, Time
where Sales.ProductID=Products.ProductID
and Sales.CustomerID=Customers.CustomerID
and Sales.TimeKey=Time.TimeKey;
```

With the support of this index, the aforementioned star query $Q1$ is executed as shown below. As we can observe, its execution plan is much simpler than before. The bitmap join index is accessed only once (cf. line 5).

```
-----------------------------------------------------------------------------------------
|Id | Operation                     |Name               |Rows |Bytes|Cost(%CPU)|Time     |
-----------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT              |                   |    1|   29|    7  (0)|00:00:01 |
| 1 |  SORT AGGREGATE               |                   |    1|   29|          |         |
| 2 |   INLIST ITERATOR             |                   |     |     |          |         |
| 3 |    TABLE ACCESS BY INDEX ROWID |SALES              |   22|  638|    7  (0)|00:00:01 |
| 4 |     BITMAP CONVERSION TO ROWIDS|                   |     |     |          |         |
| 5 |      BITMAP INDEX SINGLE VALUE |BI_PR_CU_TI_SALES  |     |     |          |         |
-----------------------------------------------------------------------------------------
```

Unfortunately, this bitmap join index cannot be used for answering queries with predicates on one or two dimensions, like for example

```
ProdName in ('ThinkPad Edge', 'Sony Vaio', 'Dell Vostro') and Town=
'London'
```

or

```
Town='London' and Year=2009
```

or

```
Year=2009
```

On the contrary, the three independent bitmap join indexes defined earlier, i.e., *BI_Pr_Sales*, *BI_Cu_Sales*, *BI_Ti_Sales* offer much flexible indexing scheme as they can be used answering queries with predicates on some of the dimensions.

2.4 Materialized Views and Query Rewriting

Execution time of complex, time consuming star queries can be reduced by physically storing and re-using their previously computed results. This way, a precomputed result is perceived by a query optimizer as another source of data that can be queried. The precomputed result is commonly called a *materialized view* (MV). If a user executes query Q that computes values that have already been stored in materialized view MV_i, then a query optimizer will rewrite original query Q into another query Q' so that Q' is executed on MV_i and Q' returns the same result as original query Q. This technique of automatic, in-background rewriting of users' queries is called a *query rewriting*. Notice that a user's query must not necessarily be exactly the same as the query whose result was stored in a materialized view. In this case, a query optimizer may join a materialized view with other MVs or tables, may further aggregate and filter the content of a MV and may project columns of a MV, in order to answer a user's query [77]. In order to illustrate the idea behind the query rewriting mechanism let us consider the example below.

Example 5. Let us return to the DW logical schema shown in Figure 2.1 and let us assume that in this DW users execute the below queries. The first one computes quarterly sales of product categories in countries. The second one computes monthly and quarterly sales of products belonging to category 'electronic'.

```
select Category, Country, Quarter,
       sum(SalesPrice) as SumSales
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
and s.CustomerID=c.CustomerID
and s.TimeKey=t.TimeKey
group by Category, Country, Quarter;

select ProdName, Country, Month,
       sum(SalesPrice) as SumSales
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
and s.CustomerID=c.CustomerID
and s.TimeKey=t.TimeKey
and p.Category='electronic'
group by ProdName, Country, Month
```

Both queries could be optimized by materialized view *SalesMV1*, shown below, whose query makes available monthly and quarterly sales values of

products and their categories for each country. Both of the above queries could be answered by means of the materialized view by further aggregating its content.

```
create materialized view SalesMV1
...
select ProdName, Category, Country, Month, Quarter,
       sum(SalesPrice) as SumSales
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
and s.CustomerID=c.CustomerID
and s.TimeKey=t.TimeKey
group by ProdName, Category, Country, Month, Quarter
```
□

As we can see from the example, the definition of a materialized view includes a query. Tables referenced in the query are called *base tables*.

Although in the above example one materialized view is used for rewriting two queries, in practice, multiple materialized views need to be created in order to optimize a certain workload of queries. For this reason, a challenging research issue concerns the selection of such a set of materialized views that: (1) will be used for optimizing the greatest possible number of the most expensive queries and (2) whose maintenance will not be costly. Several research works have addressed this problem and they have proposed multiple algorithms for selecting optimal sets of materialized views for a given query workload, e.g. [78, 79, 80, 81]. Commercial DB/DWMSs, like, Oracle, DB2, and SQL Server provide tools that analyze workloads and based on them, propose sets of database objects, typically materialized views and indexes, for optimizing the workloads.

2.4.1 Materialized Views in Oracle

In Oracle, a materialized view is created with the `create materialized view` command. Its definition includes: (1) the moment when the view is filled in with data, (2) its refreshing method (fast, complete, force), (3) whether the materialized view is refreshed automatically or on demand, (4) row identification method (required for incremental refreshing), (5) the view automatic refreshing interval, and (6) a query computing the content of the materialized view.

A command creating an example materialized view *YearlySalesMV* is shown below.

```
create materialized view YearlySalesMV1
build immediate
refresh force
with rowid
as
select ProdName, Category, Quarter, Year,
```

```
       sum(SalesPrice) as SumSales
from Sales s, Products p, Time t
where s.ProductID=p.ProductID
and s.TimeKey=t.TimeKey
group by ProdName, Category, Quarter, Year;
```

The `build immediate` clause denotes that the view will be filled in with data during the execution of the command. The `refresh force` clause denotes that the system automatically selects the refreshing mode (either fast, i.e., incremental or complete). If the system is able to refresh the MV fast, then this method is used. Otherwise the complete refreshing is used. The `with ROWID` clause defines the method of identifying rows in the MV and its base tables. In this example, rows will be identified based on their physical addresses (ROWIDs). Rows can also be identified based on their primary keys. To this end, the `with primary key` clause is applied. Row identification is required for incremental refreshing.

For some versions of Oracle (e.g., 10g) clause `with rowid` clashes with clause `enable query rewrite`. The latter makes available a MV for query rewriting. Normally, this clause is part of the MV definition, placed between `with rowid|primary key` and `as`. In cases these clauses clash, one has to make the view available for query rewriting by executing additional command shown below.

```
alter materialized view MVName enable query rewrite;
```

Oracle materialized views will be used in query rewriting provided that: (1) the cost-based query optimizer is used, (2) materialized views have been enabled for query rewriting, (3) the system works in the query rewriting mode, and (4) a user executing a query has appropriate privileges.

In order to illustrate the query rewriting process, let us consider the below query. We assume that MV *YearlySalesMV1* was created and made available for query rewriting. The query execution plan reveals that it was automatically rewritten on the *YearlySalesMV*, cf. row number 2 in the below plan. This row denotes that the MV was sequentially scanned and its content grouped (cf. row number 1) in order to compute the more coarse aggregate.

```
select ProdName, Year, sum(SalesPrice) as SumSales
from Sales s, Products p, Time t
where s.ProductID=p.ProductID
and s.TimeKey=t.TimeKey
and t.Year=2009
group by ProdName, Year;
```

```
-------------------------------------------------------------------------------
|Id |Operation                    |Name          |Rows |Bytes|Cost(%CPU)|Time    |
-------------------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              | 159 | 3180|   4  (25)|00:00:01|
| 1 | HASH GROUP BY               |              | 159 | 3180|   4  (25)|00:00:01|
| 2 |  MAT_VIEW REWRITE ACCESS FULL|YEARLYSALESMV | 159 | 3180|   3   (0)|00:00:01|
-------------------------------------------------------------------------------
```

As mentioned before, a MV can be refreshed fast, provided two conditions are fulfilled. First, the MV query definition includes only the allowed constructs. The set of these constructs is extended from version to version of the Oracle DBMS (cf. [72]). Second, each of the MV base tables has associated its own *materialized view log*. The log records DML operations applied to the base table. The content of the log is used for fast refreshing the MV.

The materialized view log is created by the below command (only its basic form is shown). For a MV created with the `with primary key` clause the MV log must include the same clause. The same applies to the `with rowid` clause. In the first case the log will store the values of columns that constitute the primary key of the table, whereas in the second case it will store the values of ROWID. A MV log may include both clauses if it serves for both types of MV. Apart from the primary key and ROWID column, a materialized view log can store values of other columns whose changing values have impact on the content of the MV. The list of these columns is denoted as `col1, ..., coln`.

```
create materialized view log on BaseTable
with primary key|ROWID (col1, ..., coln),
      sequence
      including new values;
```

Including the `sequence` clause in a materialized view log definition results in the creation of a column *SEQUENCE$$* in the log. Its values represent the order in which DML operations are executed on the base table for which the log was created. Oracle advises to create this column for fast refreshing after mixed insert, update, delete operations on the base table. The `including new values` clause forces Oracle to store in the MV log old values of columns listed in `col1, ..., coln` as well as their new values. Omitting this clause results in storing only the old values of columns. This clause must be included in the MV log definition in order to support fast refreshing MVs computing aggregates.

2.4.2 Materialized Views in DB2

In DB2 a materialized view, called a *materialized query table* (MQT) or a *summary table*, is created with the `create table` command with additional clauses defining its maintenance. The clauses include: (1) `maintained by {system | user}`, (2) `refresh {immediate | deferred}`, (3) `data initially {immediate | deferred}`, and (4) `enable query optimization`.

A MQT can be maintained either by a system or by a user. In the first case, the `maintained by system` clause is used. This is a default clause. For a *system-maintained* MQT, one can define either automatic or non-automatic refreshing mode. In the automatic mode, a MQT is refreshed automatically as the result of changes in the content of its base tables. To this end, the

definition of a MQT must include the `refresh immediate` clause. This refreshing mode requires that a unique key from each base table is included in the `select` command defining the MQT. `refresh deferred` denotes that a MQT has to be explicitly refreshed by explicit execution of the `refresh table` command, with the syntax shown below.

```
refresh table TableName {incremental|not incremental}
```

The `data initially immediate` clause causes that the content of a MQT is computed as part of the command creating the MQT. As a result, as soon as the command is finished, the MQT is filled in with its materialized data. The `data initially deferred` clause denotes that the content of the MQT is not computed as part of the command creating the MQT. After being created, the MQT is in the *check pending* state. In this state, the MQT cannot be queried until the `set integrity` command has been executed on the MQT.

The `enable query optimization` clause, similarly as in Oracle, makes a MQT available for query rewriting. It is the default clause.

An example system-maintained MQT, called *YearlySalesMV2*, is shown below. The MQT is filled in with data during its creation process (`data initially immediate`), is refreshed immediately after changes have been applied to its base tables (`refresh immediate`), and is made available for query optimization (`enable query optimization`).

```
create table YearlySalesMV2
as
(select ProdID, ProdName, Year, sum(salesPrice) as SumSales
 from Sales s, Products p, Time t
 where s.ProductID=p.ProductID
 and s.TimeKey=t.TimeKey
 and t.Year=2009
 group by ProdID, ProdName, Year)
data initially immediate
refresh immediate
maintained by system
enable query optimization;
```

In order to create a user-defined MQT, the `maintained by user` clause has to be included in the definition of a MQT. This type of a MQT can be refreshed only in the deferred mode and the `refresh table` command cannot be applied to such a MQT. Thus, a user is responsible for implementing the procedure of refreshing the MQT (by means of inserts, imports, triggers, etc.). An example definition of a user-maintained MQT is shown below. After being created, the MQT is in an inconsistent state and must be made consistent by executing the `set integrity` command.

```
create table YearlySalesMV3
as
(select Year, sum(salesPrice) as SumSales
```

```
 from Sales s, Time t
 where and s.TimeKey=t.TimeKey
 group by Year)
data initially deferred
refresh deferred
maintained by user

set integrity for YearlySalesMV3 materialized query
immediate unchecked
```

A MQT created with the `refreshed deferred` clause can be incrementally refreshed provided that a system maintains a log of changes that are used for refreshing the MQT. This log is called a *staging table*. A staging table is created by the `create table` command. After being created, a staging table is in an inconsistent state and must be made consistent by executing the `set integrity` command. The simple example below illustrates how to create a staging table for MQT *YearlySalesMV3*.

```
create table YearlySalesMV3_ST for YearlySalesMV3 propagate immediate
set integrity for YearlySalesMV3 staging immediate unchecked
```

2.4.3 Materialized Views in SQL Server

In SQL Server, a materialized view, called *indexed view*, is created by creating a unique clustered index on a view (cf. [75]). The index causes that the view is materialized, i.e., the whole result set of a query defining a view is persistently stored in a database. The definition of an indexed view has to fulfill the following conditions. First, an indexed view requires a column whose value is unique. A unique index is then created on this column that results in clustering the data by this column. Second, an indexed view has to be created with the `schemabinding` clause that prevents from modifying the base tables of a view as long as the view exists. Third, all the view base tables must be referenced by `schemaname.tablename`. Fourth, the query defining a materialized view may not contain the following SQL constructs, among others: `exists`, `not exists`, `count(*)`, `min`, `max`, `top`, `union`, `outer join`, non deterministic functions (e.g., `GetDate`), and subqueries.

The below example illustrates commands creating indexed view *YearlySalesMV2*. The `create view` command defines the view with a query. The second command creates a unique cluster index on columns *ProdID*, *ProdName*, and *Year*. As a consequence, the whole result set of the query is materialized.

```
create view YearlySalesMV2
with schemabinding
as
select ProdID, ProdName, Year, sum(salesPrice) as SumSales
from Sales s, Products p, Time t
where s.ProductID=p.ProductID
```

```
and s.TimeKey=t.TimeKey
and t.Year=2009
group by ProdID, ProdName, Year

create unique clustered index Indx_ProdID
on YearlySalesMV(ProdID, ProdName, Year)
```

Similarly as in Oracle and DB2, in the Developer and Enterprise editions of SQL Server, an indexed view can be used by a query optimizer for query rewriting. In other editions of SQL Server, in order to force a query optimizer to rewrite a query based on an indexed view, the query must explicitly reference the indexed view and must include hint `noexpand`. In order to force a query optimizer to use base tables, rather than an indexed view, a query must include option `expand views`. Both of them are used in the below two query templates.

```
select Column1, Column2, ...
from Table, IndexedView with (noexpand)
where ...

select Column1, Column2, ...
from Table, IndexedView
where ...
option (expand views)
```

Indexed views, similarly as traditional indexes are automatically refreshed by a system. The refreshing mode is immediate and incremental, as for traditional indexes. Notice that, non-clustered indexes can also be created on a materialized view in order to support quicker access to the view content.

2.5 Partitioning

Partitioning is a mechanism of dividing a table or index into smaller parts, called *partitions*. Due to space limitations, in this chapter we will focus only on table partitioning.

The most benefit from partitioning is achieved if every partition is stored on a separate disc. This way, multiple partitions can be accessed in parallel without disc contest.

There are two types of partitioning, namely horizontal and vertical. When a table, say T, is partitioned *horizontally*, its content is divided into disjunctive subsets of rows. Every partition includes identical schema, that is the schema of T. When table T is partitioned *vertically*, it is divided into disjunctive subsets of columns (except a primary key that exists in every partition) and includes all rows from the initial table.

Partitioning is guided by three following rules, namely completeness, disjointness, and reconstruction. *Completeness* states that when table T was partitioned into P_1, P_2, ..., P_n then every row from T or its fragment must

be stored in one of these partition. This criterion guarantees that after partitioning no data will disappear. *Disjointness* states that when table T was partitioned into P_1, P_2, ..., P_n then every row or its fragment from T must be stored in exactly one partition. An exception to this rule is vertical partitioning where ever vertical partition stores a primary key of T. This criterion guarantees that partitioning does not introduce data redundancy. *Reconstruction* states that there mus be a mechanism of reconstructing original table T from its partitions. In horizontal partitioning the reconstruction of T is done by unions of the partitions, whereas in vertical partitioning it is done by joining the partitions.

In horizontal partitioning rows from an original table are placed in partitions based on partitioning criteria. In this type of partitioning rows are divided into subsets based on the value of a selected attribute (or attributes), called a *partitioning attribute*. With this respect, there are several techniques of partitioning rows based on a partitioning attribute. The first one is *hash-based*. In this technique, rows are placed in partitions based on a hash function that for each row takes as its argument the value of a partitioning attribute and returns the number of a partition where the row is to be stored. The second one is *range-based*. In this technique, every partition has defined the range of values of a partitioning attribute it can store. The third technique is *value-based*. In this technique, every partition has defined the set of values it can store. The fourth technique is based on the *round robin* algorithm.

Inserting, updating, and deleting data from partitioned tables are managed by a system. If a row is inserted, it is a system that select the right partition where the row will be stored. If a row is deleted, then the system finds a right partition where the row was stored. If the value of a partitioning attribute of an existing row is updated, then depending on a system and system parameters, a system may move an updated row from one to another partition.

In this chapter we focus on horizontal partitioning, which is natively supported by Oracle, DB2, and SQL Server. Vertical partitioning must be simulated in these three DB/DWMSs. Typically, vertical partitioning of table T is simulated by creating n separate tables T_i, each of which contains the subset of columns of T. Additionally, each of T_i must contain a primary key column(s) used for the reconstruction of the original table T. The reconstruction can be implemented by a view on top of tables T_i.

Vertical partitioning is supported in commercial and open source systems that use column storage, e.g., Sybase IQ, EMC Greenplum, C-Store/Vertica, MonetDB, Sadas, FastBit, Model 204.

2.5.1 Partitioning in Oracle

Oracle supports multiple partitioning techniques, namely: range, interval, list, hash, virtual column, system, reference, and composite. In this section, we will briefly overview the partitioning techniques.

Range Partitioning

In the range partitioning, each partition has defined its own ranges of values it can store. The ranges are applicable to a partitioning attribute. For example, the below table *Sales_Range_TKey* includes five partitions named *Sales_1Q_2009*, ..., *Sales_Others*. Partition *Sales_1Q_2009* accepts rows whose values of partitioning attribute *TimeKey* are lower than '01-04-2009'. Partition *Sales_2Q_2009* accepts rows whose values of the partitioning attribute fulfill the condition '01-04-2009' $\leq$ TimeKey $<$ '01-07-2009', etc. Notice, that in range partitioning partitions must be ordered by the ranges. Partition *Sales_Others* is defined with the `MAXVALUE` keyword. It allows to store all the other records having the value of *TimeKey* greater or equal to '01-01-2010'. Clause `tablespace` allows to point a tablespace where a partition is to be physically stored (an Oracle tablespace is a database object that allows to logically organize multiple files under one name).

```
create table Sales_Range_TKey
(ProductID varchar2(8) not null references Products(ProductID),
 TimeKey date not null references time(TimeKey),
 CustomerID varchar2(10) not null references Customers(CustomerID),
 SalesPrice number(6,2))
PARTITION by RANGE (TimeKey)
  (partition Sales_1Q_2009
      values less than (TO_DATE('01-04-2009', 'DD-MM-YYYY'))
         tablespace Data01,
   partition Sales_2Q_2009
      values less than (TO_DATE('01-07-2009', 'DD-MM-YYYY'))
         tablespace Data02,
   partition Sales_3Q_2009
      values less than (TO_DATE('01-10-2009', 'DD-MM-YYYY'))
         tablespace Data03,
   partition Sales_4Q_2009
      values less than (TO_DATE('01-01-2010', 'DD-MM-YYYY'))
         tablespace Data04,
   partition Sales_Others
      values less than (MAXVALUE) tablespace Data05);
```

Interval Partitioning

A special type of a range partition is an interval partition whose partitioning attribute is typically of type `date`. The difference between these two partitions is that interval partitions are automatically created by a system when needed. If data not fitting into existing partitions are to be inserted, then appropriate new partitions are created to store the data. Interval partitions are created by using the `interval` keyword followed by the definition of the interval. The interval is defined by means of a system function `NumToYMInterval` whose fist argument is the value (length) of an interval and the second one is its measurement unit. A fragment of an SQL command defining interval

partitioning is shown below. In this example, an interval is equal to three months.

```
...
PARTITION by RANGE (TimeKey)
INTERVAL (NumToYMInterval(3, 'MONTH'))
  (partition Sales_1Q_2009
     values less than (TO_DATE('01-04-2009', 'DD-MM-YYYY')),
   partition Sales_2Q_2009
     values less than (TO_DATE('01-07-2009', 'DD-MM-YYYY')));
```

List Partitioning

In the list partitioning each partition has assigned the set of values of a partitioning attribute that the partition accepts. The example below table *Sales_List_PayType* is divided into three partitions. Partition *Sales_Credit_Debit* stores sales record paid with a credit card ('Cr') or a debit card ('De'). Partition *Sales_Cash* stores sales records paid with cash. The last partition stores records having the value of *PaymentType* other than the three aforementioned. To this end, the `DEFAULT` keyword is used.

```
create table Sales_List_PayType
(ProductID varchar2(8) not null references Products(ProductID),
 TimeKey date not null references time(TimeKey),
 CustomerID varchar2(10) not null references Customers(CustomerID),
 SalesPrice number(6,2),
 PaymentType varchar(2))
PARTITION by LIST (PaymentType)
  (partition Sales_Credit_Debit values ('Cr','De') tablespace Data01,
   partition Sales_Cash values ('Ca') tablespace Data02,
   partition Sales_Others values (DEFAULT) tablespace Data05
);
```

Hash Partitioning

In the hash partitioning, data are placed in partitions by an internal Oracle hash function. As an example, let us consider table *Sales_Hash_CustID* that is composed of two partitions. Both of them have names assigned by a system and are stored in a default tablespace.

```
create table Sales_Hash_CustID
(ProductID varchar2(8) not null references Products(ProductID),
 TimeKey date not null references time(TimeKey),
 CustomerID varchar2(10) not null references Customers(CustomerID),
 SalesPrice number(6,2),
 PaymentType char(1))
PARTITION by HASH (CustomerID) partitions 2;
```

A user can explicitly assign names and storage locations for hash partitions. To this end, the `PARTITION by` clause has to be modified, as shown below.

```
...
PARTITION by HASH (CustomerID)
  (partition Cust1 tablespace Data01,
   partition Cust2 tablespace Data02));
```

Virtual Column Partitioning

Virtual column partitioning (available from Oracle11g) requires a virtual column in the definition of a table. A virtual column is a column whose value is computed either by a formula or a stored deterministic function (keyword `deterministic` in a function definition). Next, this column is used as a partitioning attribute, but when a virtual column is used as a partitioning attribute its values cannot be returned by a stored function. Typically, this type of partitioning is applicable either to range or list partitioning.

As an example, we show below a fragment of the command creating partitions based on virtual column *Gross*.

```
create table Products_Virt1
(...
 SellPrice number(6,2),
 Tax number(4,2),
 Gross as (SellPrice*Tax))
 PARTITION by range(Gross)
   (partition Prod1 values less than (1000),
    partition Prod2 values less than (2000));
```

Queries that use in their predicates either virtual column *Gross* or formula *SellPrice*Tax* can profit from the above partitioned table. For example, the simple query below

```
select * from Products_Virt1
where SellPrice*Tax<1000
```

will be answered by accessing only partition *Prod1*.

System Partitioning

In a system partitioning (available from Oracle11g), partitions do not have assigned any constraints and any row can be inserted into any partition. In this case, the DBMS does not control the placement of rows, i.e., it is a user (or application logic) that is responsible for inserting rows into the required partitions. In the system partitioning, every `insert` command must explicitly include the name of a partition where a row is to be inserted. A fragment of a command defining a system partitioned table is shown below.

```
create table Customers_Sys
(CustomerID varchar2(10) CONSTRAINT pk_Customers PRIMARY KEY,
```

```
 ...)
 PARTITION by SYSTEM
 (partition Cust_Europe, partition Cust_America, partition Cust_Asia);
```

Reference Partitioning

A reference partitioning (available from Oracle11g) is applicable to partitioning tables related to each other by primary key - foreign key relationships. A table with a primary key has defined explicitly a partition schema whereas a table with a foreign key inherits partitioning attribute and schema from the parent table. As an example illustrating this type of partitioning let us consider range partitioned table *Products_List_Cat*, as shown below.

```
create table Products_List_Cat
(ProductID varchar2(8) PRIMARY KEY,
 ProdName  varchar2(30),
 Category  varchar2(15),
 SellPrice number (6,2),
 Manufacturer varchar2(20))
 PARTITION by LIST(Category)
   (partition Prod_Elect values ('electronic'),
    partition Prod_Clo values ('clothes'));
```

Table *Sales_List_Cat* will inherit partitioning attribute and partition definitions from *Products_List_Cat*.

```
create table Sales_List_PayType
(ProductID varchar2(8) not null
      constraint ProdID_FK references Products_List_Cat(ProductID),
 ...)
 PARTITION by REFERENCE (ProdID_FK);
```

When tables are partitioned by reference, a query optimizer joins the tables by the *partition wise join*. In this join, only those partitions are joined that produce non empty set.

Composite Partitioning

Composite partitioning allows to divide main partitions into subpartitions. In Oracle11g main partitions can be either range or list, whereas subpartitions can be range, list, or hash.

```
create table Sales_Comp_RH
(ProductID varchar2(8) not null references Products(ProductID),
 TimeKey date not null references time(TimeKey),
 CustomerID varchar2(10) not null references Customers(CustomerID),
 SalesPrice number(6,2))
PARTITION by RANGE (TimeKey)
SUBPARTITION by HASH (ProductID) subpartitions 2
  (partition Sales_1Q_2009
```

```
    values less than (TO_DATE('01-04-2009', 'DD-MM-YYYY'))
       tablespace Users,
  partition Sales_2Q_2009
    values less than (TO_DATE('01-07-2009', 'DD-MM-YYYY'))
       tablespace Users,
  partition Sales_3Q_2009
    values less than (TO_DATE('01-10-2009', 'DD-MM-YYYY'))
       tablespace Users,
  partition Sales_4Q_2009
    values less than (TO_DATE('01-01-2010', 'DD-MM-YYYY'))
       tablespace Users);
```

DDL and Select on Partitioned Tables

When the value of a partitioning attribute of a given row is updated and this row no longer qualifies for its partition then the row can be automatically migrated into another partition. Automatic migration is available when for a partitioned table the `alter table TableName enable row movement` command is executed. Otherwise, such an update will not be possible.

Partitions can be explicitly addressed by queries, delete commands, and insert commands (the latter is possible only for system partitions). To this end, the table name must be followed by keyword `partition(PartitionName)`. If however, a query does not address a partition explicitly, a query optimizer, based on partition definitions and query predicates, will optimize the query and will address only these partitions that contribute to the query result.

2.5.2 Partitioning in DB2

DB2 supports range partitioning of tables and indexes. The mechanism is similar to the one described for Oracle. An example fragment of a command creating a range-partitioned table is shown below. For every partition, the range of values of a partitioning attribute (i.e., *TimeKey*) is defined, similarly as in Oracle. The below table is composed of four partitions, each of which stores sales from one quarter of year 2009.

```
create table Sales_Range_TKey
(iProductID varchar2(8) , ...)
PARTITION BY RANGE(TimeKey)
(partition Sales_1Q_2009 starting '01-01-2009',
 partition Sales_2Q_2009 starting '01-04-2009',
 partition Sales_3Q_2009 starting '01-07-2009',
 partition Sales_4Q_2009 starting '01-10-2009' ending '31-12-2009')
```

2.5.3 Partitioning in SQL Server

SQL Server (from version 2005) provides horizontal range partitioning, similar to Oracle and DB2. Partitioning applies to tables in indexes.

Partitioning is defined with the support of two database object, namely a partition function and a partition scheme. The *partition function* defines the number of partitions for a table and ranges of values for every partition. The *partition scheme* defines storage locations for table partitions. The definition of a partition scheme is based on the partition function.

An example partition function *Sales_Range_TKey*, defined for attribute of type `datetime`, is shown below. Applying this function to partitioning a table will result in five partitions having the following ranges of dates: date<'2009-04-01', '2009-04-01'≤ date <'2009-07-01', '2009-07-01'≤ date <'2009-10-01', '2009-10-01' ≤ date <'2010-01-01', and date≥'2010-01-01'. The `range right` or `range left` keywords define the policy of inclusion of border values. If the `range left` keyword is specified then the ranges for partitions would be defined as follows: date≤'2009-04-01', '2009-04-01'< date ≤ '2009-07-01', ..., and date>'2010-01-01'.

```
create PARTITION FUNCTION [Sales_Range_TKey] (datetime)
as RANGE right for values
  ('20090401', '20090701', '20091001', '20100101');
```

A partition function may be used also for numeric and character columns in the same way as illustrated above.

An example partition scheme *PS_Sales_Range_TKey* based on the *Sales_Range_TKey* partition function is shown below. Each of the five partitions created by the partition function is placed in a separate filegroup, whose name is given in the `to` clause. Partition storing range date<'2009-04-01' is stored in file group *Data01*, partition storing range '2009-04-01'≤ date <'2009-07-01' is stored in *Data02*, etc. A SQL Server filegroup is a database object offering the mechanism similar to a *tablespace* in Oracle, i.e., it allows to combine multiple files under a given name.

```
create PARTITION SCHEME PS_Sales_Range_TKey
as partition Sales_Range_TKey
to (Data01, Data02, Data03, Data04, Data05);
```

Finally, based on the defined partitioning scheme, a partitioned table can be created, as shown below.

```
create table Sales_Range_TKey
(ProductID varchar(8),
 TimeKey datetime ...)
on PS_Sales_Range_TKey (TimeKey)
```

2.6 Summary

Multiple research and technological works in the area of data warehouses have been focusing on providing means for increasing the performance of a data warehouse for analytical queries and other types of data processing. As mentioned already, DW performance depends on multiple components of a DW

architecture that include among others: hardware and computational architectures, physical storage schemes of data, query optimization and execution, dedicated data structures supporting faster data access.

In this chapter we focused on just one component from the aforementioned list, i.e., on data structures. The most popular data structures used in practice in major commercial DWMSs include: various index structures, materialized views, as well as partitioning of tables and indexes. In this chapter, first we discussed basic index structures, including a bitmap index, a join index, and a bitmap join index. We outlined the functionality of explicitly created bitmap indexes and bitmap join indexes in Oracle and we showed how they are applied in a query execution process. We also showed how system-managed bitmap indexes are applied to star query executions in DB2 and SQL Server. Second, we discussed the concept of materialized views and their application to query rewriting. We analyzed the functionality of materialized views in Oracle, DB2, and SQL Server. We showed how star queries are rewritten based on Oracle materialized views. Third, we discussed table partitioning techniques in Oracle, DB2, and SQL Server.

From the research and technological point of view, open issues in the area of DW performance include among others: building DWs in a parallel computation environments (cloud, grid, clusters, GPUs), main memory DWs, efficient data storage schemes (the column store and storage in MOLAP servers), query optimization and processing (especially in parallel computation environments and in main memory architectures), novel data structures supporting faster access to data, data and index compression techniques, testing and assessing a DW performance.

New business domains of DW application require more advanced DW functionalities, often combining the transactional and analytical features [82]. For example, monitoring unauthorized credit card usage, monitoring telecommunication networks and predicting their failures, monitoring car traffic, analyzing and predicting share rates, require accurate and up to date analytical reports. In order to fulfill this demand, one has to assure that the content of a DW is synchronized with the content of data sources with a minimum delay, e.g., seconds or minutes, rather than hours. Moreover, queries have to be answered instantly analyzing new data that were just loaded into a DW. To this end, the technology of a real-time (near real-time, right-time) data warehouse (RTDW) has been developed, e.g., [83, 84]. Strong demand for up to date data at any moment opens a new areas of research on assuring high performance of RTDWs.

Acknowledgements. This chapter was prepared during a research visit at Universidad de Costa Rica (San Jose, Costa Rica), supported from the project *Engineer's era. Expansion potential of Poznań University of Technology* (financed by the European Union as part of the Human Capital Operational Programme).

References

1. d'Orazio, L., Bimonte, S.: Multidimensional Arrays for Warehousing Data on Clouds. In: Hameurlain, A., Morvan, F., Tjoa, A.M. (eds.) Globe 2010. LNCS, vol. 6265, pp. 26–37. Springer, Heidelberg (2010)
2. Furtado, P.: A survey of parallel and distributed data warehouses. International Journal of Data Warehousing and Mining 5(2), 57–77 (2009)
3. Jhingran, A., Jou, S., Lee, W., Pham, T., Saha, B.: IBM Business Analytics and Cloud Computing: Best Practices for Deploying Cognos Business Intelligence to the IBM Cloud. MC Press, LLC (2010)
4. Andrzejewski, W., Wrembel, R.: GPU-WAH: Applying Gpus to Compressing Bitmap Indexes with Word Aligned Hybrid. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010. LNCS, vol. 6262, pp. 315–329. Springer, Heidelberg (2010)
5. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proc. VLDB Endow. 3, 670–680 (2010)
6. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: Gputerasort: high performance graphics co-processor sorting for large database management. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 325–336 (2006)
7. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast computation of database operations using graphics processors. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 215–226 (2004)
8. Lauer, T., Datta, A., Khadikov, Z., Anselm, C.: Exploring graphics processing units as parallel coprocessors for online aggregation. In: DOLAP, pp. 77–84 (2010)
9. Kemper, A., Neumann, T.: Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In: ICDE, pp. 195–206 (2011)
10. Qiao, L., Raman, V., Reiss, F., Haas, P.J., Lohman, G.M.: Main-memory scan sharing for multi-core cpus. Proc. VLDB Endow. 1, 610–621 (2008)
11. Ross, K.A., Zaman, K.A.: Serving datacube tuples from main memory. In: Proc. of Int. Conf. on Scientific and Statistical Database Management (SSDBM), p. 182 (2000)
12. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 671–682 (2006)
13. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 967–980 (2008)
14. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented dbms. In: Proc. of Int. Conf. on Very Large Data Bases (VLDB), pp. 553–564 (2005)
15. Abadi, D.J., Boncz, P.A., Harizopoulos, S.: Column-oriented database systems. Proc. VLDB Endow. 2, 1664–1665 (2009)
16. Cuzzocrea, A.: Data cube compression techniques: A theoretical review. In: Encyclopedia of Data Warehousing and Mining. IGI Global (2009)
17. Hasan, K.M.A., Tsuji, T., Higuchi, K.: An Efficient Implementation for MOLAP Basic Data Structure and its Evaluation. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 288–299. Springer, Heidelberg (2007)

18. Moerkotte, G.: Building query compilers, http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf (retrieved October 6, 2011)
19. Graefe, G.: A generalized join algorithm. In: BTW, pp. 267–286 (2011)
20. Graefe, G.: Parallel query execution algorithms. In: Encyclopedia of Database Systems, pp. 2030–2035. Springer, Heidelberg (2009)
21. Zeller, H., Graefe, G.: Parallel query optimization. In: Encyclopedia of Database Systems, pp. 2035–2038. Springer, Heidelberg (2009)
22. Chaudhuri, S.: Query optimizers: time to rethink the contract? In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 961–968 (2009)
23. Ioannidis, Y.E., Poosala, V.: Histogram-based approximation of set-valued query-answers. In: Proc. of Int. Conf. on Very Large Data Bases (VLDB), pp. 174–185 (1999)
24. Vitter, J.S., Wang, M.: Approximate computation of multidimensional aggregates of sparse data using wavelets. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 193–204 (1999)
25. Gramacki, A., Gramacki, J., Andrzejewski, W.: Probability density functions for calculating approximate aggregates. Foundations of Computing and Decision Sciences Journal 35 (2010)
26. Valduriez, P.: Join indices. ACM Transactions on Database Systems (TODS) 12(2), 218–246 (1987)
27. O'Neil, P.: Model 204 architecture and performance. In: Gawlick, D., Reuter, A., Haynie, M. (eds.) HPTS 1987. LNCS, vol. 359, pp. 40–59. Springer, Heidelberg (1989)
28. 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. (2007); ISBN 1-59904-364-5
29. Bryla, B., Loney, K.: Oracle Database 11g DBA Handbook. McGraw-Hill Osborne Media (2007)
30. O'Neil, P., Graefe, G.: Multi-table joins through bitmapped join indices. SIGMOD Record 24(3), 8–11 (1995)
31. Gupta, A., Mumick, I.S. (eds.): Materialized Views: Techniques, Implementations, and Applications. MIT Press (1999)
32. 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)
33. Rao, J., Zhang, C., Megiddo, N., Lohman, G.: Automating physical database design in a parallel database. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 558–569 (2002)
34. Stöhr, T., Rahm, E.: Warlock: A data allocation tool for parallel warehouses. In: Proc. of Int. Conf. on Very Large Data Bases (VLDB), pp. 721–722 (2001)
35. Deliège, F., Pedersen, T.B.: Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In: EDBT, pp. 228–239. ACM (2010)
36. Nourani, M., Tehranipour, M.H.: Rl-huffman encoding for test compression and power reduction in scan applications. ACM Trans. Design Autom. Electr. Syst. 10(1), 91–115 (2005)
37. Stabno, M., Wrembel, R.: RLH: Bitmap compression technique based on run-length and Huffman encoding. Information Systems 34(4-5), 400–414 (2009)
38. Aouiche, K., Darmont, J.: Data mining-based materialized view and index selection in data warehouses. J. Intell. Inf. Syst. 33, 65–93 (2009)

39. Lawrence, M., Rau-Chaplin, A.: Dynamic view selection for olap. In: Strategic Advancements in Utilizing Data Mining and Warehousing Technologies, pp. 91–106. IGI Global (2010)
40. Theodoratos, D., Xu, W., Simitsis, A.: Materialized view selection for data warehouse design. In: Encyclopedia of Data Warehousing and Mining, pp. 1182–1187. IGI Global (2009)
41. Chen, Y., Dehne, F.K.H.A., Eavis, T., Rau-Chaplin, A.: Improved data partitioning for building large rolap data cubes in parallel. IJDWM 2(1), 1–26 (2006)
42. Furtado, P.: Large Relations in Node-Partitioned Data Warehouses. In: Zhou, L.-z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, pp. 555–560. Springer, Heidelberg (2005)
43. Binnig, C., Kossmann, D., Kraska, T., Loesing, S.: How is the weather tomorrow?: towards a benchmark for the cloud. In: Proc. of Int. Workshop on Testing Database Systems, DBTest (2009)
44. Funke, F., Kemper, A., Krompass, S., Neumann, T., Seibold, M., Kuno, H., Nica, A., Poess, M.: Metrics for measuring the performance of the mixed workload ch-benchmark. In: Proc. of Technology Conference on Performance Evaluation and Benchmarking, TPCTC (2011)
45. Kersten, M.L., Kemper, A., Markl, V., Nica, A., Poess, M., Sattler, K.U.: Tractor pulling on data warehouses. In: Proc. of Int. Workshop on Testing Database Systems, DBTest (2011)
46. O'Neil, P.E., O'Neil, E.J., 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)
47. Gyssens, M., Lakshmanan, L.V.S.: A foundation for multi-dimensional databases. In: Proc. of Int. Conf. on Very Large Data Bases (VLDB), pp. 106–115 (1997)
48. Jarke, M., Lenzerini, M., Vassiliou, Y., Vassiliadis, P.: Fundamentals of Data Warehouses. Springer, Heidelberg (2003)
49. Malinowski, E., Zimányi, E.: Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications. Springer Publishing Company, Inc., Heidelberg (2008)
50. Chaudhuri, S., Dayal, U.: An overview of data warehousing and OLAP technology. SIGMOD Record 26(1), 65–74 (1997)
51. Li, Z., Ross, K.A.: Fast joins using join indices. VLDB Journal 8(1), 1–24 (1999)
52. 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. (2007); ISBN 1-59904-364-5
53. Wu, K., Otoo, E.J., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In: Proc. of Int. Conf. on Very Large Data Bases (VLDB), pp. 24–35 (2004)
54. Scientific Data Management Research Group: FastBit: An efficient compressed bitmap index technology, `http://sdm.lbl.gov/fastbit/` (retrieved October 24, 2011)
55. Wu, M., Buchmann, A.: Encoded bitmap indexing for data warehouses. In: Proc. of Int. Conf. on Data Engineering (ICDE), pp. 220–230 (1998)
56. Wu, K., Yu, P.: Range-based bitmap indexing for high cardinality attributes with skew. In: Int. Computer Software and Applications Conference (COMPSAC), pp. 61–67 (1998)

57. Rotem, D., Stockinger, K., Wu, K.: Optimizing candidate check costs for bitmap indices. In: Proc. of ACM Conf. on Information and Knowledge Management (CIKM), pp. 648–655 (2005)
58. Rotem, D., Stockinger, K., Wu, K.: Optimizing I/O Costs of Multi-Dimensional Queries using Bitmap Indices. In: Andersen, K.V., Debenham, J., Wagner, R. (eds.) DEXA 2005. LNCS, vol. 3588, pp. 220–229. Springer, Heidelberg (2005)
59. 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)
60. Koudas, N.: Space efficient bitmap indexing. In: Proc. of ACM Conf. on Information and Knowledge Management (CIKM), pp. 194–201 (2000)
61. Chan, C., Ioannidis, Y.: Bitmap index design and evaluation. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 355–366 (1998)
62. O'Neil, P., Quass, D.: Improved query performance with variant indexes. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 38–49 (1997)
63. Rinfret, D., O'Neil, P., O'Neil, E.: Bit-sliced index arithmetic. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 47–57 (2001)
64. Antoshenkov, G., Ziauddin, M.: Query processing and optimization in Oracle RDB. VLDB Journal 5(4), 229–237 (1996)
65. Wu, K., Otoo, E.J., Shoshani, A.: An efficient compression scheme for bitmap indices. Research report, Lawrence Berkeley National Laboratory (2004)
66. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Transactions on Database Systems (TODS) 31(1), 1–38 (2006)
67. Deliège, F.: Concepts and Techniques for Flexible and Effective Music Data Management. PhD thesis, Aalborg University, Denmark (2009)
68. Stabno, M., Wrembel, R.: RLH: Bitmap compression technique based on run-length and Huffman encoding. In: Proc. of ACM Int. Workshop on Data Warehousing and OLAP (DOLAP), pp. 41–48 (2007)
69. Huffman, D.A.: A method for the construction of minimum-redundancy codes. In: Proc. of the Institute of Radio Engineers, pp. 1098–1101 (1952)
70. O'Neil, E., O'Neil, P., Wu, K.: Bitmap index design choices and their performance implications. Research report, Lawrence Berkeley National Laboratory (2007)
71. Reiss, F., Stockinger, K., Wu, K., Shoshani, A., Hellerstein, J.M.: Efficient analysis of live and historical streaming data and its application to cybersecurity. Research report, Lawrence Berkeley National Laboratory (2006)
72. Oracle Corp.: Oracle Database Data Warehousing Guide, Rel. 11g, `http://www.oracle.com/technetwork/database/enterprise-edition/documentation/database-093888.html` (retrieved June 24, 2010)
73. Bhattacharjee, B., Padmanabhan, S., Malkemus, T., Lai, T., Cranston, L., Huras, M.: Efficient query processing for multi-dimensionally clustered tables in db2. In: VLDB, pp. 963–974 (2003)
74. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 422–426 (1970)
75. Microsoft Corp.: SQL Server (2008) R2, `http://msdn.microsoft.com/enus/library/ms191267.aspx` (retrieved June 24, 2010)
76. Aouiche, K., Darmont, J., Boussaïd, O., Bentayeb, F.: Automatic Selection of Bitmap Join Indexes in Data Warehouse. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 64–73. Springer, Heidelberg (2005)

77. Hobbs, L., Hillson, S., Lawande, S.: Oracle9iR2 Data Warehousing. Digital Press (2003)
78. de Sousa, M.F., Sampaio, M.C.: Efficient materialization and use of views in data warehouses. SIGMOD Record 28(1), 78–83 (1999)
79. Gupta, H.: Selection of Views to Materialise in a Data Warehouse. In: Afrati, F.N., Kolaitis, P.G. (eds.) ICDT 1997. LNCS, vol. 1186, pp. 98–112. Springer, Heidelberg (1996)
80. Lawrence, M., Rau-Chaplin, A.: Dynamic View Selection for OLAP. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2006. LNCS, vol. 4081, pp. 33–44. Springer, Heidelberg (2006)
81. Theodoratos, D., Xu, W.: Constructing search space for materialized view selection. In: Proc. of ACM Int. Workshop on Data Warehousing and OLAP (DOLAP), pp. 48–57 (2004)
82. Krueger, J., Tinnefeld, C., Grund, M., Zeier, A., Plattner, H.: A case for online mixed workload processing. In: Proc. of Int. Workshop on Testing Database Systems (DBTest). ACM (2010)
83. Castellanos, M., Dayal, U., Miller, R.J.: Enabling Real-Time Business Intelligence. LNBIP, vol. 41. Springer, Heidelberg (2010)
84. Thiele, M., Fischer, U., Lehner, W.: Partition-based workload scheduling in living data warehouse environments. Information Systems 34(4-5), 382–399 (2009)