Geoinformatica (2012) 16:165–205
DOI 10.1007/s10707-011-0128-5

The SB-index and the HSB-index: efficient indices for spatial data warehouses

Thiago Luís Lopes Siqueira · Cristina Dutra de Aguiar Ciferri · Valéria Cesário Times · Ricardo Rodrigues Ciferri

Received: 6 April 2010 / Revised: 18 February 2011
Accepted: 4 May 2011 / Published online: 14 June 2011

Abstract Spatial data warehouses (SDWs) allow for spatial analysis together with analytical multidimensional queries over huge volumes of data. The challenge is to retrieve data related to *ad hoc* spatial query windows according to spatial predicates, avoiding the high cost of joining large tables. Therefore, mechanisms to provide efficient query processing over SDWs are essential. In this paper, we propose two efficient indices for SDW: the SB-index and the HSB-index. The proposed indices share the following characteristics. They enable multidimensional queries with spatial predicate for SDW and also support predefined spatial hierarchies. Furthermore, they compute the spatial predicate and transform it into a conventional one, which can be evaluated together with other conventional predicates by accessing a star-join Bitmap index. While the SB-index has a sequential data structure, the HSB-index uses a hierarchical data structure to enable spatial

T. L. L. Siqueira
São Carlos Campus, São Paulo Federal Institute of Education, Science and Technology (IFSP),
Rodovia Washington Luis, Km 235, 13.565-905 São Carlos, SP, Brazil
e-mail: prof.thiago@cefetsp.br

T. L. L. Siqueira · R. R. Ciferri (✉)
Computer Science Department, Federal University of São Carlos (UFSCar),
Rodovia Washington Luis, Km 235, P.O. Box 676, 13.565-905 São Carlos, SP, Brazil
e-mail: ricardo@dc.ufscar.br

C. D. A. Ciferri
Computer Science Department, University of São Paulo at São Carlos (USP),
Av. do Trabalhador São-Carlense, 400, P.O. Box 668, 13.560-970 São Carlos, SP, Brazil
e-mail: cdac@icmc.usp.br

V. C. Times
Informatics Center, Federal University of Pernambuco (UFPE), Av. Jornalista Anibal Fernandes,
50.740-560 Recife, PE, Brazil
e-mail: vct@cin.ufpe.br

objects clustering and a specialized buffer-pool to decrease the number of disk accesses. The advantages of the SB-index and the HSB-index over the DBMS resources for SDW indexing (i.e. star-join computation and materialized views) were investigated through performance tests, which issued roll-up operations extended with containment and intersection range queries. The performance results showed that improvements ranged from 68% up to 99% over both the star-join computation and the materialized view. Furthermore, the proposed indices proved to be very compact, adding only less than 1% to the storage requirements. Therefore, both the SB-index and the HSB-index are excellent choices for SDW indexing. Choosing between the SB-index and the HSB-index mainly depends on the query selectivity of spatial predicates. While low query selectivity benefits the HSB-index, the SB-index provides better performance for higher query selectivity.

Keywords Spatial data warehouse · Indices · Bitmap index · Spatial on-line analytical processing · Spatial drill-down and roll-up operations

1 Introduction

A spatial data warehouse (SDW) is a multidimensional database that inherits and combines capabilities and characteristics from the data warehouse (DW), the geographic information system (GIS) and the on-line analytical processing (OLAP) to explore the decision-making process [1–4]. Like a conventional DW, a SDW is a subject-oriented, integrated, time-variant, voluminous and non-volatile database. Similar to a GIS, a SDW stores spatial data (e.g. vector geometries) and descriptive attributes, and also supports spatial analysis and *ad hoc* rectangular query windows in query processing. Furthermore, spatial OLAP (SOLAP) tools provide analytical multidimensional queries based on spatial predicates that mostly run over the SDW [5].

A conventional DW is often implemented in relational databases through a star schema [6], which is composed of fact and dimension tables. Fact tables store numeric measures of interest, while dimension tables contain attributes that contextualize these measures. Frequently, attributes of a dimension table are related with other attributes of the same dimension table through hierarchies, which specify different levels of granularity and data aggregation. For instance, suppose a DW of an application that represents historical data relating to orders and sales of a corporation [35]. Suppose also a multidimensional view "*revenue* by *customer* by *part* by *supplier* by *date*". In this DW, *Lineorder* is a fact table that contains the numeric measure of interest *revenue*, while *Customer*, *Part*, *Supplier* and *Date* are dimension tables. An example of hierarchy in the dimension table *Supplier* is *(region)* $\preceq$ *(nation)* $\preceq$ *(city)* $\preceq$ *(address)*, where *address* is the attribute of the lowest granularity level, and *region* is the attribute of the highest granularity level. The $\preceq$ operator imposes a partial ordering on the attributes, specifying that one aggregation of higher granularity can be determined using data from another aggregation of lower granularity [7]. For instance, (*nation*) can be determined from (*city*). In a conventional DW, the domain of the attributes *region*, *nation*, *city* and *address* are alphanumeric.

Differently from a conventional DW, a SDW stores spatial data as specific attributes in dimension tables or as measures in fact tables [1, 2, 4, 8, 9]. Therefore, instead of storing alphanumeric values for regions, nations, cities and addresses, a spatial dimension table *Supplier* represents the values of its geographic attributes *region_geo*, *nation_geo*, *city_geo*

and *s_address_geo* as points, lines and polygons. In a SDW, the suffix *_geo* refers to spatial attributes that store geometries. Furthermore, in a SDW, hierarchies may be defined also over spatial attributes of one or more spatial dimension tables. A *predefined spatial hierarchy* is a 1:N association among higher and lower granularity spatial attributes that is determined by a spatial relationship, such as *(region_geo)* $\preceq$ *(nation_geo)* $\preceq$ *(city_geo)* $\preceq$ *(s_address_geo)*, where the spatial relationship is containment [3, 12]. Regarding this example of predefined spatial hierarchy, it states that a given address is inside of only a specific city, a given city is inside of only a specific nation, and a given nation is inside of only a specific region. Therefore, for instance, the measure *lo_revenue* of a nation is calculated by summing the *lo_revenue* of each city inside of this nation.

Spatial hierarchies are a core aspect in the SDW design since they enable the processing of drill-down and roll-up operations extended with containment and intersection range queries [5–8, 12–15]. While drill-down operations analyze increasingly less aggregated data, roll-up operations, on the other hand, analyze increasingly more aggregated data. An example of a spatial and multidimensional query is "find out the total revenue earned by suppliers whose addresses are inside a rectangular window". This query defines a topological relationship and a spatial *ad hoc* spatial query window that was not previously stored in dimension tables. Another query may be issued to roll-up to the city granularity level by using a larger spatial query window that intersects the cities where the suppliers are located.

Regarding SOLAP query processing, the challenge in SDW is to retrieve data related to *ad hoc* spatial query windows, avoiding the high cost of joining large fact tables with dimension tables. Although several methods have been proposed in the literature to enhance the query processing performance in DW and SDW, such as view materialization [7, 9, 11, 16], vertical and horizontal fragmentation of dimension and fact tables [17–19], data partitioning aiming at parallel processing [20, 21] and also indices [10, 22–28], there is a lack of efficient SDW indices to support *predefined spatial hierarchies*, to deal with multidimensionality in SDW, and to perform spatial drill-down and roll-up operations. This issue motivates the development of new indices for SDW that enable these features.

In this paper, we propose two efficient indices for SDW: the Spatial Bitmap Index (SB-index) and the Hierarchical Spatial Bitmap Index (HSB-index). The SB-index has a sequential and compact data structure based on the Projection index [29], whose entries point to a star-join Bitmap index [25]. On the other hand, the HSB-index reuses the clustering technique of a tree-based spatial index in order to group spatial objects, thus enabling pruning.

Other major characteristics of the proposed SB-index and HSB-index are described as follows.

- They efficiently support analytical multidimensional queries based on spatial predicates. To comply with this goal, both the SB-index and the HSB-index compute the spatial predicate at first, and then transform the spatial predicate answer into a conventional predicate that can be solved by accessing a star-join Bitmap index. This proposed strategy avoids costly star-join operations between fact tables and dimension tables, providing better performance to SOLAP queries.
- They focus on predefined spatial hierarchies. A predefined spatial hierarchy in the SDW determines the existence of one SB-index (or HSB-index) per granularity level of the spatial hierarchy. This index organization takes advantage of the 1:N association among higher and lower granularity spatial attributes, benefiting the processing of drill-down

and roll-up operations extended with the spatial predicates intersection and containment range queries.

- They may be applied to process SOLAP queries based on different query selectivity of spatial predicates. On the one hand, the HSB-index must be used when the query requires that only few spatial objects be processed, since its hierarchical organization ensures that only a subset of the spatial objects will be analyzed in query processing. On the other hand, the SB-index should be used mainly when the query requires that a greater number of spatial objects be processed. In this situation, the sequential scan provided by the SB-index is the most appropriate technique for SOLAP query processing.

A preliminary version of the SB-index was presented in [22–24]. Here, we extend these works additionally describing the algorithms of the SB-index for building and query processing. We also evaluate the performance of this index over strictly *non-redundant* SDW schemas. Furthermore, we introduce the novel HSB-index and detail its data structure, buffering, building and query processing. Moreover, we carry out novel performance tests to evaluate the characteristics of the SB-index and the HSB-index, such as the impact of the increase in query selectivity, the benefits of embedding a buffer-pool in the HSB-index, the influence of the disk page size and of the buffer-pool size, the processing of uninterrupted roll-up operations with *overlapping* spatial query windows, the individual analysis on processing spatial and conventional predicates using the proposed indices, and the influence of different spatial data types on the performance of the SB-index and the HSB-index.

This paper is organized as follows. Section 2 describes concepts that are essential to understand the proposed indices. Sections 3 and 4 introduce the main contributions of this paper: the SB-index and the HSB-index, respectively. Each proposed index is described in terms of its data structure, building and query processing. Section 5 validates the proposed indices through performance tests, Section 6 surveys related work, and Section 7 concludes the paper.

2 Theoretical foundation

In this section, we describe concepts related to the SB-index and the HSB-index proposals. Section 2.1 details a hybrid SDW schema, Section 2.2 summarizes the R-tree, the R*-tree and the GiST spatial indices, and Section 2.3 surveys the Projection and the star-join Bitmap indices.

2.1 Spatial data warehouse

SDWs store spatial data as specific attributes in dimension tables or as measures in fact tables. Figure 1 illustrates a *hybrid SDW schema* [1, 22–24], which is a star schema extended to additionally store spatial dimension tables aiming at processing SOLAP queries. In Fig. 1, *Customer*, *Supplier*, *Part* and *Date* are *conventional dimension tables* that store only alphanumeric redundant data, while *C_Address*, *S_Address*, *City*, *Nation* and *Region* are *spatial dimension tables* that are stored separately according to the granularity level to avoid spatial data redundancy. Also, there are two *predefined spatial hierarchies*: (i) $(region\_geo) \preceq (nation\_geo) \preceq (city\_geo) \preceq (c\_address\_geo)$ for customers; and (ii) $(region\_geo) \preceq (nation\_geo) \preceq (city\_geo) \preceq (s\_address\_geo)$ for

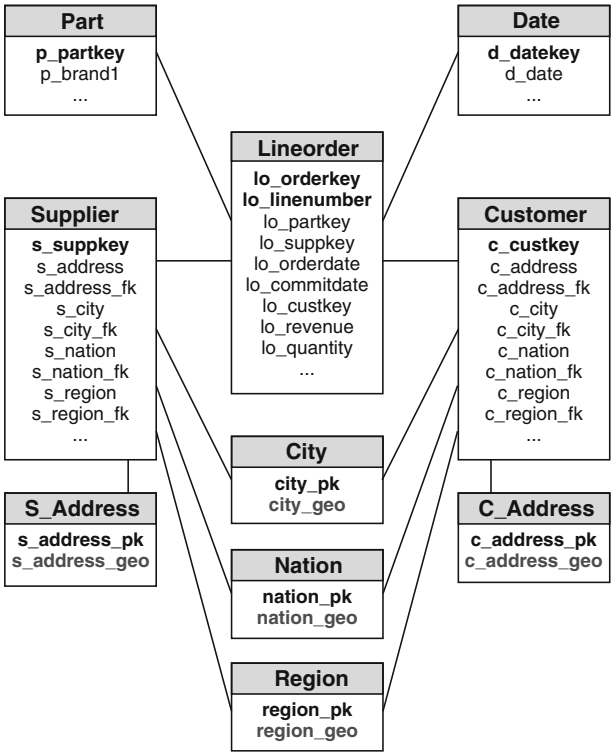


Fig. 1 The hybrid SDW schema

suppliers. As described in Section 1, attributes with the suffix *_geo* are spatial attributes that store geometries, and a predefined spatial hierarchy specifies that a given geometry in a lower spatial granularity attribute is contained by a single geometry in a higher spatial granularity attribute. Finally, the hybrid SDW schema allows the processing of *star-join operations*, which are operations that require several joins among the fact table and the dimension tables.

The hybrid SDW schema shown in Fig. 1 is considered as a running example throughout the paper. Note that the hybrid and the snowflake [6] schemas are different, since the former does not normalize the spatial hierarchies. Note also that our work is not based on redundant SDW schemas, which are schemas that store spatial data together with descriptive data in conventional dimensions. According to recent research results, although hybrid SDW schemas add new join costs to SOLAP queries, they require less storage space and determine lower SOLAP query response time than those observed in redundant SDW schemas [22, 23]. Therefore, hybrid schemas are more appropriate to SDW.

2.2 Spatial indices

The spatial indices related to our work are the R-tree, the R*-tree and the GiST. The R-tree [38] is a spatial access method that supports range queries in the Euclidean space and is commonly implemented by DBMSs (database management systems) to index spatial

objects stored in the secondary memory. The R-tree stores MBRs (minimum bounding rectangles) of the spatial objects instead of the original objects. Its hierarchical data structure, with non-leaf and leaf nodes, is responsible for pruning the index traversal. Also, its insertion algorithm allocates new spatial objects into leaf nodes, aiming at minimizing the total coverage of each non-leaf node entry. The goal is to reduce the possibility of intersection between non-leaf nodes' MBRs and the spatial query window during query processing, thus avoiding the undesirable ramification of the tree traversal during the search.

The R*-tree [40] improves the R-tree insertion algorithm according to the criteria of coverage, overlap, margin and storage, rather than only applying the criterion of reducing the coverage as the R-tree does. While the overlap criterion aims at minimizing the intersection area of the MBRs, the margin criterion is used to minimize the perimeter of the MBRs, and the storage criterion is used to maximize the occupation rate of the structure nodes. Analyzing these criteria during the insertion guarantees to the R*-tree a better space partitioning and, consequently, a better search performance than the R-tree.

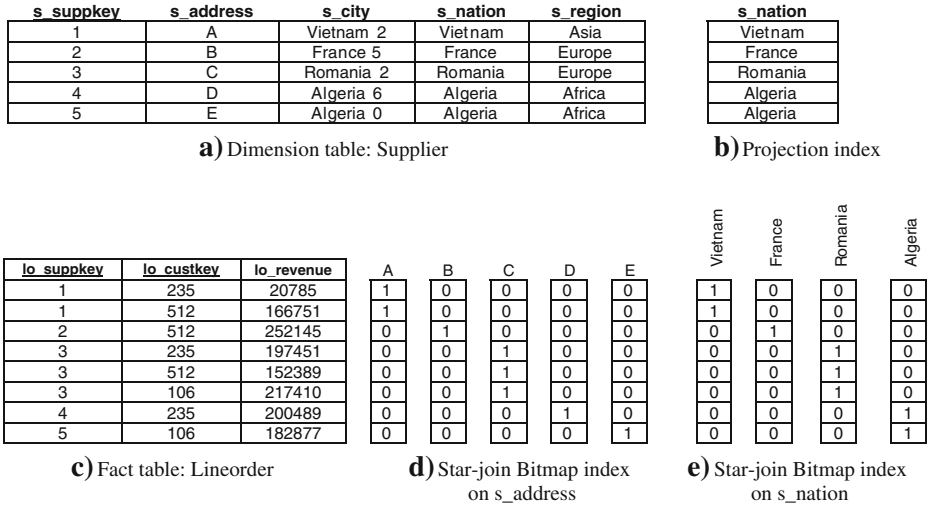
The Generalized Search Tree (GiST) [39] is aimed at supporting an extensible set of queries and data types that can unify and generalize the behavior of different search trees. Because of this flexibility, some DBMSs have implemented the GiST to efficiently index and retrieve conventional and complex data for different queries. Since the GiST is built on a spatial attribute, it has the same characteristics of the R-tree and the R*-tree.

In this paper, we use the R-tree and the GiST to index spatial attributes in order to improve SOLAP query performance over star-join computation and materialized views. Also, in the performance tests, we use the R*-tree to implement one of the components of the proposed HSB-index.

2.3 The projection and the star-join bitmap indices

Consider X as an attribute of a relation R. The Projection Index over X is a sequence of values for X extracted from R and sorted by the row number [29]. The basic Bitmap index [29, 30] associates one bit-vector to each distinct value v of the indexed attribute X. The bit-vectors maintain as many bits as the number of records found in the relation R. If for the k-th record of the relation R we have that $X = v$, then the k-th bit of the bit-vector associated to v has the value of one. Otherwise the k-th bit has the value of zero. The attribute cardinality, $|X|$, is the number of distinct values of X and determines the number of bit-vectors.

The Bitmap index is used in conventional DW to avoid the star-join computation. To this end, a star-join Bitmap index is built on the attribute Z of the dimension table to indicate the set of rows in the fact table to be joined with a given value of Z. For instance, Fig. 2 shows data from a conventional DW star schema (Fig. 2a,c), a Projection index defined on the attribute *s_nation* (Fig. 2b) and bit-vectors from a star-join Bitmap index defined on the attribute *s_address* (Fig. 2d). Although *s_address* is not involved in the star-join, it is possible to index this attribute by a star-join Bitmap index since there is a 1:1 relationship between *s_suppkey* and *s_address*, and *s_suppkey* is referenced by *lo_suppkey*. Consider that a query asking for *s_address* = 'D' has been issued. Instead of joining the fact table *Lineorder* with the dimension table *Supplier*, it is only necessary to verify the tuples that have the value of one in the bit-vector of the address 'D'.



s_suppkey	s_address	s_city	s_nation	s_region
1	A	Vietnam 2	Vietnam	Asia
2	B	France 5	France	Europe
3	C	Romania 2	Romania	Europe
4	D	Algeria 6	Algeria	Africa
5	E	Algeria 0	Algeria	Africa

a) Dimension table: Supplier

s_nation
Vietnam
France
Romania
Algeria
Algeria

b) Projection index

lo_suppkey	lo_custkey	lo_revenue
1	235	20785
1	512	166751
2	512	252145
3	235	197451
3	512	152389
3	106	217410
4	235	200489
5	106	182877

c) Fact table: Lineorder

A	B	C	D	E
1	0	0	0	0
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

d) Star-join Bitmap index on s_address

Vietnam	France	Romania	Algeria
1	0	0	0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	1	0
0	0	1	0
0	0	0	1
0	0	0	1

e) Star-join Bitmap index on s_nation

Fig. 2 Fragment of data, Projection and star-join Bitmap indices

Star-join Bitmap indices allow the quick processing of bit-wise logical operations. On the one hand, it is possible to index attributes of higher granularity from attributes of lower granularity using OR operations. For instance, executing a bit-wise OR with the bit-vectors for *s_address* = 'D' and *s_address* = 'E' produces the bit-vector for *s_nation* = 'Algeria', since *(s_nation)* $\preceq$ *(s_city)* $\preceq$ *(s_address)*. Figure 2e shows the star-join Bitmap index defined on the attribute *s_nation*. On the other hand, it is also possible to provide efficient query processing using AND operations. For instance, suppose that a query asking for *s_nation* = 'Algeria' and *s_address* = 'D' has been issued. In order to answer this query, a bit-wise AND operation is carried out involving the bit-vectors of the requested values for *s_nation* and *s_address*.

Bit-wise logical operations provide efficient query processing even when the number of involved bit-vectors is high. As a result, the performance of the Bitmap index is not drastically affected by the number of indexed dimensions. Therefore, the Bitmap index is frequently used to index warehouse data [30, 42, 43]. On the other hand, high cardinality has been seen as one of the Bitmap index' main drawback. However, even with very high cardinality, the Bitmap index can provide acceptable response time and storage utilization, since three techniques have been proposed to efficiently overcome this limitation: compression [32], binning [33] and encoding [34].

In this paper, the Projection index was used as a basis for the SB-index, while the star-join Bitmap index was used as a basis for both the SB-index and the HSB-index proposals.

3 The SB-index

In this section, we introduce our first proposal of index for SDW: the Spatial Bitmap Index (SB-index), which focuses on *predefined spatial hierarchies* and enables the Bitmap index

to be used in SDW. The SB-index is an adapted Projection index on the primary key of the spatial dimension table. A core aspect of the SB-index design is that it computes the spatial predicate and transforms it into a conventional one, which can be evaluated together with other conventional predicates. As a result, queries can be answered using a star-join Bitmap index, avoiding *star-join operations*. Therefore, the SB-index is able to process roll-up and drill-down operations extended with spatial predicates such as intersection and containment range queries.

This section is organized as follows. Section 3.1 describes the SB-index data structure, Section 3.2 details the operation of building the SB-index, and Section 3.3 focuses on SOLAP query processing using the SB-index.

3.1 Data structure

In the SB-index data structure, each entry is of the *sbitvector* (spatial bit-vector) data type, which is composed of a key value and a MBR. The key value references the spatial dimension table's primary key and also identifies the spatial object represented by the corresponding MBR. Furthermore, the key value is an integer, which represents an adequate choice for primary keys as DWs have surrogate keys. Four coordinates (i.e. four double precision numbers) represent the MBR: Xmin, Ymin, Xmax and Ymax. As a result, the size of one *sbitvector* entry, denoted as s, is given by the following expression: s = size of (integer) + 4 * size of (double).

A definition for the SB-index is given in Definition 1, assuming the existence of the *sbitvector* data type and the star-join Bitmap index.

Definition 1 (SB-Index): A SB-index is an array of sbitvector type, whose i-th entry points to the i-th bit-vector of the star-join Bitmap index built over the spatial dimension table's primary key. Also, the SB-index is persistently maintained on secondary storage (i.e. disk) together with the star-join Bitmap index. A predefined spatial hierarchy in the SDW determines the existence of one SB-index per granularity level of the spatial hierarchy.

In order to illustrate the SB-index data structure, consider the SDW schema given in Fig. 1 and the dataset shown in Fig. 3. City is a spatial dimension table containing the

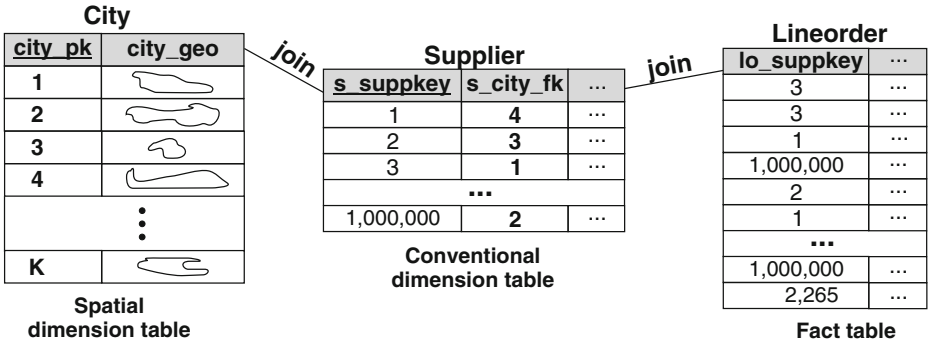


Fig. 3 The spatial dimension table City, the dimension table Supplier and the fact table Lineorder

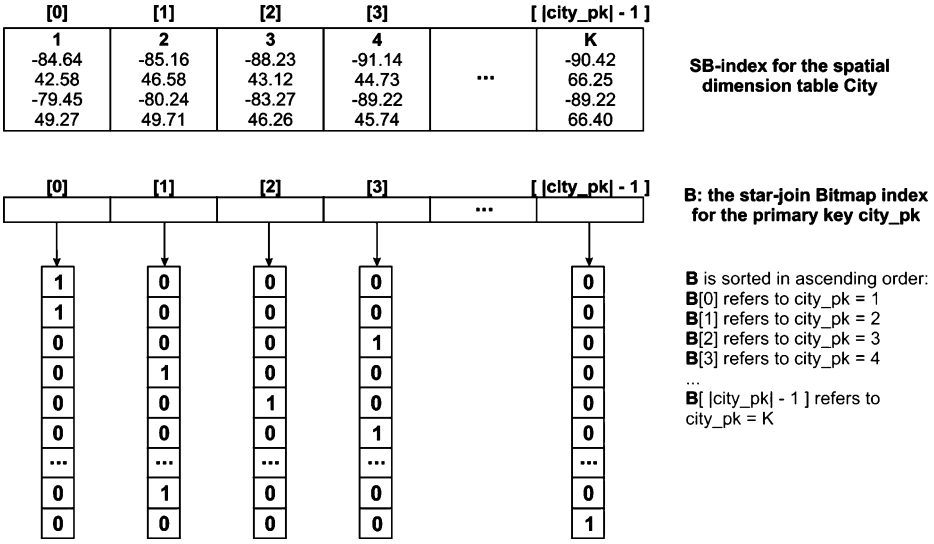


Fig. 4 The SB-index data structure

spatial attribute *city_geo* and the primary key *city_pk*. For instance, *city_pk* = 1 is associated to *s_suppkey* = 3 and *lo_suppkey* = 3. Therefore, SB-index[0] maintains the key value 1 and the MBR of the spatial object identified by *city_pk*=1, as shown in Fig. 4. Furthermore, *B* is the star-join Bitmap index defined over the attribute *city_pk* of the dimension table *City* to indicate the set of rows in the fact table to be joined with a given value of *city_pk.* The bit-vector pointed to by B[0] specifies the tuples in the fact table where *city_pk* = 1. Thus, accessing the bit-vector related to SB-index[0] just requires using the offset 0 to read *B*, i.e. B[0]. Consequently, SB-index[0] implicitly points to the bit-vector referenced by B[0].

3.2 The building operation

The building operation of the SB-index is performed as described in the *BuildSBindex* algorithm (Algorithm 1). In line 1, it is required a connection to the spatial database that maintains the SDW and issued a SQL query that: (i) selects the primary key and the spatial attributes from the spatial dimension table; (ii) sorts the query results in ascending order based on the primary key; and (iii) obtains the MBR of the spatial objects using proper DBMS functions. For each row retrieved from the spatial database, the key value and the corresponding MBR are copied to one entry of *sbitvector* type into an array in the main memory (lines 4 to 13). When this array becomes full, it is written to a disk page of the SB-index (line 15). Further, some unused bytes *U* are left in the SB-index (line 16). Since the entries are sorted by the spatial object identifiers, the SB-index is also sorted by the primary key. Therefore, SB-index[i] refers to the star-join Bitmap index B[i]. The actions in lines 4 to 17 are performed until there are no more retrieved rows to be copied. Finally, the star-join Bitmap index, whose entries refer to the SB-index entries, is built (lines 19 and 20).

The SB-index leaves some unused bytes U between different disk pages to avoid fragmented entries, thus not requiring two disk accesses to obtain a single entry. Furthermore, the size of U is usually very small. For instance, in the SB-index, every disk page has a fixed size and both the memory-resident array and each disk page have L entries, where L is the maximum number of *sbitvector* entries that can be stored in one disk page. Suppose that the size s of a *sbitvector* entry is equal to 36 bytes (one integer of 4 bytes and four double precision numbers of 8 bytes each) and that a disk page has 4 KB. Thus, L is equal to 113 entries (4096 div 36) and the array size is 4068 bytes (113 * 36). In this case, the size of U is only 28 bytes (4096–4068).

Algorithm 1: BuildSBindex (L, U, idx, T, pk, sa)

```
Input: parameters described in Table 1
Declarations: recordptr, page, array, i
Output: the SB-index file successfully created
 1  recordptr ← ExecuteDBMS (select pk, x_min(sa), y_min(sa),
                x_max(sa), y_max(sa) from T order by pk)
 2  i ← 0
 3  Open (idx)
 4  while (recordptr ≠ eof) do
 5        while (i ≤ L) and (recordptr ≠ eof) do
 6         array[i].pk ← recordptr.pk
 7         array[i].x_min ← recordptr.x_min
 8         array[i].y_min ← recordptr.y_min
 9         array[i].x_max ← recordptr.x_max
10         array[i].y_max ← recordptr.y_max
11         i ← i + 1
12         recordptr.next( )
13        end-while
14          i ← 0
15        Write (array, idx)
16        Forward (U, idx)
17  end-while
18  Close(idx)
19  recordptr ← Execute_dbms (select pk from T order by pk)
20  BuildStarJoinBitmapIndex(recordptr)
```

Table 1 Parameters of the SB-index building algorithm

Parameter	Description
L	The maximum number of *sbitvector* entries that a disk page can hold.
U	The unused amount of bytes of a disk page.
idx	The SB-index file.
T	A spatial dimension table.
pk	The primary key of the spatial dimension table *T*.
sa	A spatial attribute of the spatial dimension table *T* (i.e. the geometry).

3.3 Query processing

A typical SOLAP query has spatial and conventional predicates. The SB-index evaluates the spatial predicates in two phases, as shown in Fig. 5. The first phase *filters* candidates that are possible answers (steps 1 to 3), while the second phase *refines* these candidates by determining which ones are answers and by producing conventional predicates to them (steps 3 to 5). Then, the SB-index evaluates all the conventional predicates, i.e. the produced predicates plus the original conventional query predicates, by using a star-join Bitmap index.

The *SearchSBindex* algorithm (Algorithm 2) details query processing with the SB-index. In lines 3 to 11, all possible answers to the spatial predicate are collected as follows. During the scan on the SB-index file, one disk page is read at a time, and its content is copied into an array in the main memory (lines 4 and 5). After fulfilling this array, a sequential scan is performed on it. For each entry, the MBR is tested against the spatial predicate based on its query window (line 7). If this test evaluates to true, the corresponding primary key value is collected (line 8). After scanning every page of the file, a collection of candidates is found, i.e. a collection of primary key values whose spatial objects may satisfy the query spatial predicate.

In lines 13 to 16, the *SearchSBindex* algorithm checks which candidates can really be seen as answers. This refinement requires the access to the spatial dimension table using each candidate primary key value to fetch the whole geometry of the spatial object. Then, these objects are tested against the spatial predicate using proper DBMS functions (line 14). If this test evaluates to true, the corresponding primary key value is used to compose a new conventional predicate, i.e. it is concatenated to a new conventional predicate string (line 15). Here, the SB-index transforms the spatial predicate into a conventional predicate. When the new conventional predicate string becomes ready, it is concatenated to the

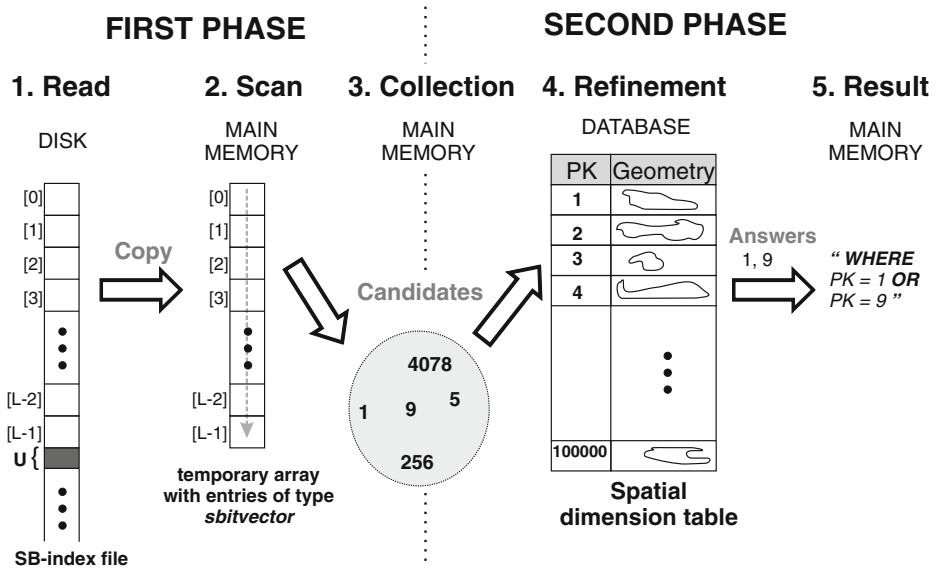


Fig. 5 The SB-index query processing

SOLAP query string (line 17). Both the new conventional predicate and the SOLAP query are memory-resident strings. The result is a rewritten query, which is executed by accessing a star-join Bitmap index to produce the final query answer (line 18).

```
Algorithm 2: SearchSBindex(idx, L, R, QW, T, pk, sa, query, answer)
Input: parameters described in Table 2
Output: the SOLAP query answer
Declarations: page, array, candidates, str_conventional_predicate
 1  Open (idx)
 2  n ← 0
 3  while not (eof(idx))
 4     Read (idx, page)
 5     Copy (page, array)
 6     for i ← 0 to L do
 7        if R(array[i], QW) then
 8           candidates[n] ← array[i].pk
 9           n ← n + 1
10     end-for
11  end-while
12  Close(idx)
13  for i ← 0 to n do
14     if (ExecuteDBMS(select R(QW, sa) from T where pk=candidates[i]))
15        then Concatenate (str_conventional_predicate, candidates[i])
16  end-for
17  Concatenate (query, str_conventional_predicate)
18  answer ← ExecuteStarJoinBitmap (query)
```

In order to illustrate the SB-index query processing, suppose that the query of Fig. 6a is executed on the SDW given in Fig. 1. The bold spatial predicate in Fig. 6a evaluates the intersection between the cities and an *ad hoc* spatial query window QW. The SB-index processes this query as follows. The cities identified by the primary key values 1, 5, 9, 256 and 4078 are considered as candidates (first phase of Fig. 5). Then, only the cities 1 and 9 are considered as answers, and the new conventional predicate is "*where city_pk= 1 OR city_pk= 9*" (second phase of Fig. 5). This predicate is appended to the query as shown in Fig. 6b. Several clauses of the rewritten query have been omitted in Fig. 6b, since they are not processed by the star-join Bitmap index, which is responsible for solving the rewritten query.

Table 2 Parameters of the SB-index query processing algorithm

Parameter	Description
idx	The SB-index file.
L	The maximum number of *sbitvector* entries that a disk page can hold.
R	A spatial relationship.
QW	An *ad hoc* spatial query window.
T	A spatial dimension table.
pk	The primary key for the spatial dimension table *T*.
sa	A spatial attribute of the spatial dimension table *T* (i.e. the geometry).
query	The query to be rewritten.
answer	The final set of query answers.

```
SELECT SUM (lo_revenue), d_year, p_brand1
FROM lineorder, date, part, supplier, city
WHERE lo_orderdate = d_datekey
        AND lo_partkey  = p_partkey
        AND lo_suppkey = s_suppkey
        AND p_brand1 = 'MFGR#2239'
        AND intersects (city_geo, QW)
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1
```

a) SQL query issued to the SDW.

```
SELECT SUM (lo_revenue ), d_year, p_brand1
WHERE p_brand1  = 'MFGR#2239'
        AND (city_pk = 1 OR city_pk = 9)
```

b) Query rewritten by the SB-index.

Fig. 6 An example of spatial predicate replacement made by the SB-index

4 The HSB-index

In this section, we introduce our second index proposal for SDW: the Hierarchical Spatial Bitmap Index (HSB-index). The motivations for the HSB-index proposal are given as follows. The previous SB-index enables the Bitmap index in SDW and, as will be shown in Section 5, it produces much better SOLAP query response times than the star-join computation and materialized views available in current database technologies. However, it is important to recognize that:

- Since the SB-index is conceptually an array stored on disk, a sequential scan is necessary to test every entry of the SB-index against a given *ad hoc* spatial query window.
- Every query of a certain granularity level requires a fixed number of disk accesses to traverse the SB-index, since the sequential scan must visit all disk pages, leading to a linear complexity denoted by O(n).
- There is not an efficient method to reuse the SB-index entries that were fetched during the processing of previous queries to avoid unnecessary disk accesses.
- The SB-index is not able to cluster spatial objects. Consequently, for example, if the *ad hoc* spatial query window stands on the west, the entries that represent spatial objects from the west, and also from the east, south and north are tested against the spatial query window. Ideally, only the entries from the west and nearby should be tested.

The aforementioned issues have motivated us to improve the SB-index data structure to enhance its capabilities, mainly when SOLAP queries require fetching only few spatial objects.

For this purpose, we propose the new HSB-index, whose advantages rely on reducing disk accesses by hierarchically clustering spatial objects. The HSB-index provides the same functionalities as the SB-index. However, the HSB-index replaces the SB-index' sequential scan by using hierarchical pruning techniques, ensuring that only a subset of the entries are analyzed in SOLAP query processing. This results in a smaller number of disk accesses. Furthermore, the HSB-index also manages a buffer-pool to temporarily store disk pages in the main memory, decreasing even more the number of disk accesses during SOLAP query processing.

This section is organized as follows. The HSB-index is described in Sections 4.1 to 4.4, which focus on its data structure, buffering, building and query processing, respectively.

4.1 Data structure

The first component of the HSB-index is a disk resident *spatial index* that has a tree organization and hierarchically clusters spatial objects based on their MBRs. The leaf nodes of the spatial index extend the *sbitvector* data type (Section 3.1), enabling the access to the star-join Bitmap index. No further adaption in the internal nodes of the spatial index is required and, as a result, the HSB-index can use any tree-based spatial index with its corresponding clustering technique. The second component of the HSB-index is a disk resident *star-join Bitmap index* defined over the primary key attribute of the spatial dimension table. A definition for the HSB-index is given in Definition 2.

Definition 2 (HSB-index): A HSB-index has a tree-based spatial index, such that each entry of a leaf node points to a given bit-vector of a star-join Bitmap index. Also, we have that:

(i) A spatial dimension table maintains all the spatial objects being indexed by a HSB-index.
(ii) In the HSB-index, each entry of a leaf node is a triple < **mbr**, **pk**, **ptr** > where **mbr** is the n-dimensional MBR of the spatial object identified by the primary key value **pk** and **ptr** points to a bit-vector that refers to **pk**, where $n=2$ represents two-dimensional spatial objects, $n=3$ represents three-dimensional spatial objects, and so on.
(iii) The star-join Bitmap index is defined over the primary key attribute of the spatial dimension table.
(iv) Both the spatial index and the star-join Bitmap index are persistently stored on disk.
(v) Each internal and leaf node of the HSB-index occupies one disk page.
(vi) A predefined spatial hierarchy in the SDW determines the existence of one HSB-index per granularity level of the spatial hierarchy.

Figure 7 shows the design of the HSB-index for the dataset shown in Fig. 3. Spatial objects of the dimension table City (Fig. 7a) are represented by their MBRs (Fig. 7b), which compose the *spatial index* data structure and reuse its clustering algorithm (Fig. 7c). Instead of just reusing the spatial index, the HSB-index also contains a pointer to a *bit-vector* in each leaf node entry (Fig. 7d). Therefore, a spatial object is identified by a key value, represented by a MBR, and is also associated with a set of tuples through a bit-vector. For example, the polygon identified by the key value 1 (Fig. 7a) is approximated to R1 (Fig. 7b) and clustered by region M1 in the spatial index (Fig. 7c). Furthermore, R1 is held in an entry of a leaf node and associated with a specific bit-vector (Fig. 7d), whose first and second bits (i.e. value 1) indicate that the first and the second tuples of the fact table reference the polygon with key value 1.

4.2 Buffering the HSB-index

Aiming at reusing previous index entries already fetched in spatial roll-up and drill-down operations and therefore avoiding unnecessary disk accesses, we have adapted the HSB-index to encompass a specialized buffer-pool, which is described in Definition 3.

Definition 3 (buffer-pool): The buffer-pool of the HSB-index consists of an array used to temporarily store disk pages in the main memory. It is a finite set of fixed-size disk

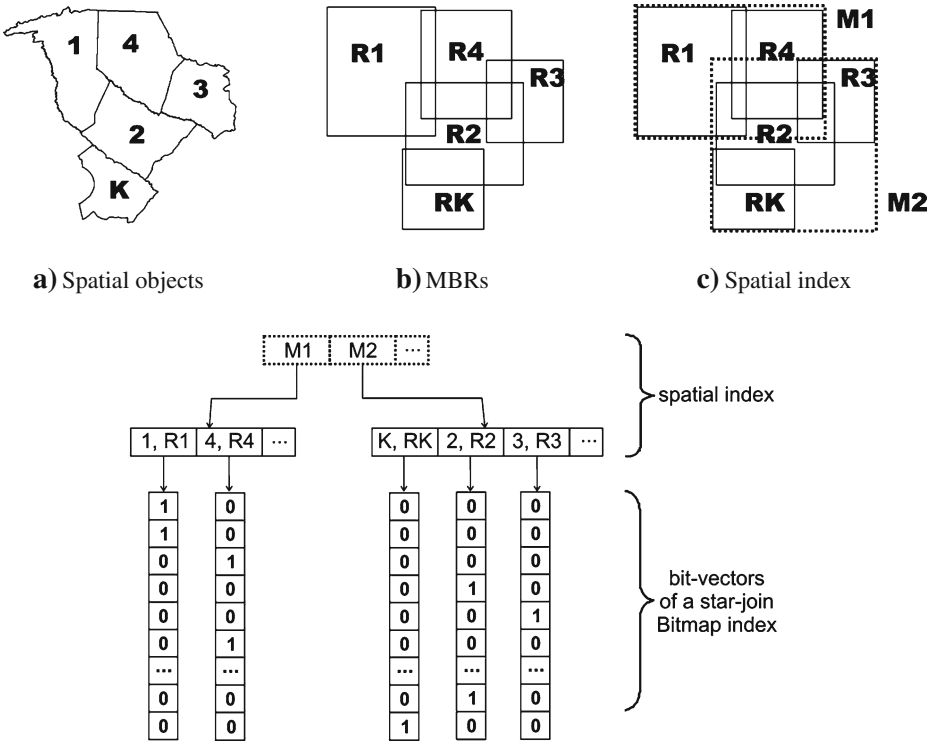


Fig. 7 The HSB-index design

pages, having each of them the same size and the same structure as the HSB-index disk pages.

The tree-based spatial index of the HSB-index, called *diskSpatialIndex*, persistently stores all the nodes of the HSB-index and contains P disk pages of size c. In order to comply with Definition 3, it is necessary to use a strategy to allocate the *diskSpatialIndex* nodes in the main memory. For this purpose, the HSB-index also makes use of the following components (Fig. 8):

- The *partial* tree-based spatial index that is stored in a buffer-pool in the main memory, and is called *bufferSpatialIndex*. It stores temporarily a subset of the *diskSpatialIndex* nodes and contains p disk pages of size c, where $p < P$.
- A *buffer manager* resident in the main memory.

Additional characteristics of the *bufferSpatialIndex* are as follows. It can be adjusted to occupy a certain amount of the main memory, according to the number of disk pages used by the *diskSpatialIndex*: 5% to 30% are values that usually provide good performance to process SOLAP queries as will be shown in Section 5. Particularly in the HSB-index design, the *bufferSpatialIndex* size refers to a percentage of the estimated

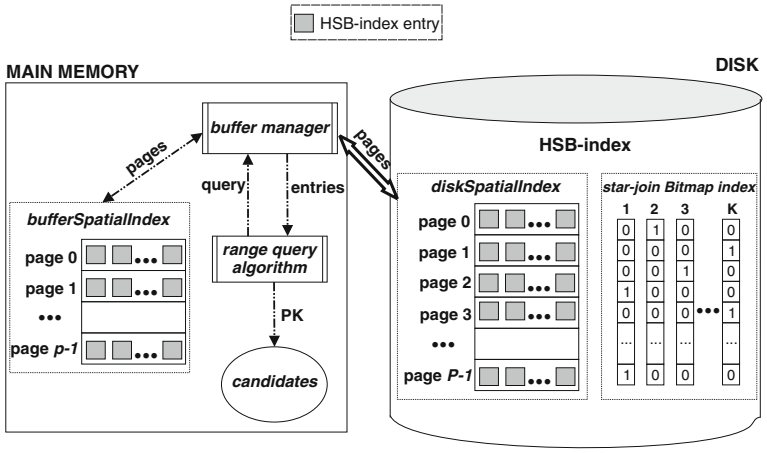


Fig. 8 Components of the HSB-index

number of disk pages that the *diskSpatialIndex* should have. For instance, if the *diskSpatialIndex* stores 100 disk pages, the *bufferSpatialIndex* set to 30% will maintain exactly 30 disk pages.

Note that the *bufferSpatialIndex* is a specialized buffer-pool and, therefore, differs from the operating system buffer-pool and from the DBMS buffer-pool because it stores only disk pages of the HSB-index. Conversely, the operating system and DBMS buffer-pools may store disk pages from different indices and programs, which compete to use the buffer. Another differential introduced by using a specialized buffer-pool is that it may be designed to use a hierarchical replacement policy based on the disk page level (e.g. hierarchical tree-based LRU). In this case, the replacement policy may choose to replace a leaf node, instead of replacing nodes from upper levels.

Figure 8 shows the components of the HSB-index and the interaction among them. In this figure, besides the aforementioned components of the HSB-index, there are two additional components to support query processing. The first is a collection of possible answers to the spatial predicate of a query, i.e. a collection of *candidates*, which is stored in the main memory. The second is the *range query algorithm* used to evaluate the spatial predicate.

4.3 The building operation

The *BuildHSBindex* algorithm (Algorithm 3) works as follows. In line 1, the DBMS extracts the primary key values and the MBRs of the corresponding spatial objects from a spatial dimension table. Also, in line 2, the index file is created according to the page size (in bytes) and the *bufferSpatialIndex* is set to a given capacity of disk pages (e.g. 5% of the *diskSpatialIndex*). Then, the *InsertionAlgorithm* inserts the extracted input data in lines 3 to 6. After the insertion, the index file is closed in line 7. Finally, lines 8 and 9 build the star-join Bitmap index whose entries refer to the spatial index entries.

Note that the *InsertionAlgorithm* is not a specific algorithm provided by the HSB-index. Conversely, it is the insertion algorithm of the spatial index used to hierarchically cluster the spatial objects. For instance, if the HSB-index uses the R*-tree to cluster the objects, then the insertion algorithm of the R*-tree is used to insert the extracted data. The only

difference refers to the fact that, to be used by the HSB-index, the algorithm must reuse the *sbitvector* data type in the leaf nodes entries.

```
Algorithm 3: BuildHSBindex (idx, T, pk, sa, pagesize, bufferpoolsize)
Input: parameters described in Table 3
Output: the HSB-index index file successfully created
Declarations: recordptr
 1  recordptr ← ExecuteDBMS (select pk, x_min(sa), y_min(sa),
                  x_max(sa), y_max(sa) from T order by pk)
 2  Open (idx, pagesize, bufferpoolsize)
 3  while (recordptr ≠ eof) do
 4        InsertionAlgorithm (idx, recordptr)
 5        recordptr.next( )
 6  end-while
 7  Close (idx)
 8  recordptr ← Execute_dbms (select pk from T order by pk)
 9  BuildStarJoinBitmapIndex(recordptr)
```

4.4 Query processing

Query processing with the HSB-index is similar to query processing with the SB-index. One difference is that the latter evaluates the spatial predicate in the filter phase through a full sequential scan in the index, while the former prunes index entries by clustering spatial objects using a hierarchical spatial index. Another difference refers to the fact that the HSB-index also decreases the query processing cost by additionally using a specialized buffer-pool.

The *SearchHSBindex* algorithm (Algorithm 4) works as follows. In line 1, the *SearchTree* routine executes a *range query algorithm* to determine the entries whose MBRs satisfy the spatial relationship with a given *ad hoc* spatial query window. First, the *buffer manager* fetches MBRs for a given disk page in the *bufferSpatialIndex*, aiming at avoiding unnecessary disk accesses. Only if the disk page is not found, the *buffer manager* fetches the *diskSpatialIndex* and transparently exchanges pages between the *bufferSpatialIndex* and the *diskSpatialIndex* using any page replacement policy, such as the LRU policy or a hierarchical tree-based LRU. As a result, the *SearchTree* routine generates a set of entries from the leaf nodes whose MBRs satisfy the spatial relationship. Each primary key value of this set is added to the collection of *candidates*, which is stored in the main memory (lines 3 to 5). The remaining actions performed by the *SearchHSBindex*

Table 3 Parameters of the HSB-index building algorithm

Parameter	Description
idx	The *diskSpatialIndex* file.
T	A spatial dimension table.
pk	The primary key for the spatial dimension table *T*.
sa	A spatial attribute of the spatial dimension table *T* (i.e. the geometry).
pagesize	The disk page size in bytes.
bufferpoolsize	The *bufferSpatialIndex* size in bytes.

algorithm (lines 6 to 13) are similar to those performed by the *SearchSBindex* algorithm in lines 13 to 18, as described in Section 3.3.

Note that, similarly to the discussion regarding the *InsertionAlgorithm* in Section 4.3, the *SearchTree* algorithm is not a specific algorithm provided by the HSB-index. In fact, it is the search algorithm of the spatial index used to prune the index traversal.

Algorithm 4: SearchHSBindex (bufferSpatialIndex, diskSpatialIndex, R, QW, T, pk, sa, query, answer)

```
Input: parameters described in Table 4
Output: the SOLAP query answer
Declarations: i, entries, candidates, str_conventional_predicate
 1  entries ← SearchTree (diskSpatialIndex, bufferSpatialIndex, QW)
 2  if (entries <> NULL) then
 3     for i ← 0 to sizeof(entries)-1 do
 4       candidates[i] ← entries[i].pk
 5     end-for
 6     for i ← 0 to sizeof(entries)-1 do
 7       if ExecuteDBMS(select R(QW, sa)from T where
            pk = candidates[i]) then
 8          Concatenate (str_conventional_predicate, candidates[i])
 9       end-if
10     end-for
11     Concatenate (query, str_conventional_predicate)
12  end-if
13  answer ← ExecuteStarJoinBitmap (query)
```

5 Performance evaluation

In this section, we present and discuss the results that point out the remarkable performance of the SB-index and the HSB-index to process SOLAP queries. We focus on roll-up and drill-down operations extended with containment and intersection spatial predicates, which use *ad hoc* spatial query windows to retrieve spatial objects that are organized through predefined spatial hierarchies. In the performance tests, we investigate several issues, as described as follows.

Table 4 Parameters of the HSB-index search algorithm

Parameters	Description
bufferSpatialIndex	The memory-resident buffer-pool of the HSB-index.
diskSpatialIndex	The disk-resident HSB-index.
R	A spatial relationship.
QW	An *ad hoc* spatial query window.
T	A spatial dimension table.
pk	The primary key for the spatial dimension table *T*.
sa	A spatial attribute of the spatial dimension table *T*.
query	The query to be rewritten.
answer	The final set of query answers.

- We compare the performance of the proposed indices against the current technology of DBMS and also investigate the SB-index and the HSB-index performance on processing spatial and conventional predicates separately. We address both the processing of a single spatial query window and the processing of two spatial query windows. The obtained results are discussed in Section 5.2.
- We investigate the advantages of introducing a specialized buffer-pool into the HSB-index, as described in Section 5.3.
- We evaluate the influence of increasing the HSB-index' buffer-pool size and the SB-index' and the HSB-index' disk page size, as described in Section 5.4.
- We analyze the impact of increasing the selectivity of the spatial predicate on the performance of the SB-index and the HSB-index. To this end, we issued SOLAP queries that retrieved increasing number of spatial objects. The obtained results are discussed in Section 5.5.
- We evaluate the influence of different spatial data types on the performance of the SB-index and the HSB-index. In this sense, we used datasets storing more complex objects, such as lines and multipolygons, as described in Section 5.6.

Before presenting the results, we provide in Section 5.1 details on the experimental setup used in our performance evaluation.

5.1 Experimental setup

In this section, we describe the experimental setup that was used to evaluate the performance of the proposed SB-index and HSB-index indices. We detail the characteristics of the datasets and the materialized views, introduce the concept of query selectivity, define *disjoint* and *overlapping* spatial query windows, describe the workload, and describe the hardware and software platforms.

Regarding the datasets, we adapted the Star Schema Benchmark (SSB) [35], which is derived from the TPC-H benchmark (http://www.tpc.org/tpch), to support SDW analysis, since the spatial information of the SSB is strictly alphanumeric and maintained in the dimension tables *Supplier* and *Customer*. The DS10 dataset was created based on the SDW schema shown in Fig. 1. Our adaptions preserved alphanumeric data and created two predefined spatial hierarchies based on the conventional ones: (i) *(region_geo)* $\preceq$ *(nation_geo)* $\preceq$ *(city_geo)* $\preceq$ (*c_address_geo*) for customers; and (ii) (*region_geo*) $\preceq$ (*nation_geo*) $\preceq$ (*city_geo*) $\preceq$ (*s_address_geo*) for suppliers. According to [9], *Supplier* and *Customer* are spatial-to-spatial dimensions since the primitive level and all of its high-level generalized data are spatial, i.e. all of them store vector geometries. Furthermore, as stated in [1, 22, 24], SDW spatial data must not be redundant and should be shared whenever is possible. Thus, *Supplier* and *Customer* shared city, nation and region locations but had individual addresses. Also, attributes *c_address_geo* and *s_address_geo* were disjoint.

The DS10 dataset data generation was based on the SSB scale factor 10 and produced 599,862,140 tuples in the fact table, 50 distinct regions, 250 nations, 2,500 cities, 1,000,000 supplier addresses and 3,000,000 customer addresses. We generated 5 nations per region, 10 cities per nation and a certain quantity of addresses per city that ranged from 349 to 455. Cities, nations and regions were represented by polygons, while addresses were represented by points. The polygons were collected from the Tiger/Line (http://www.census.gov/geo/www/tiger), while the points were synthetic data. The DS10 dataset occupied 113 GB. Figure 9 exemplifies its spatial data distribution for city, nation and region.

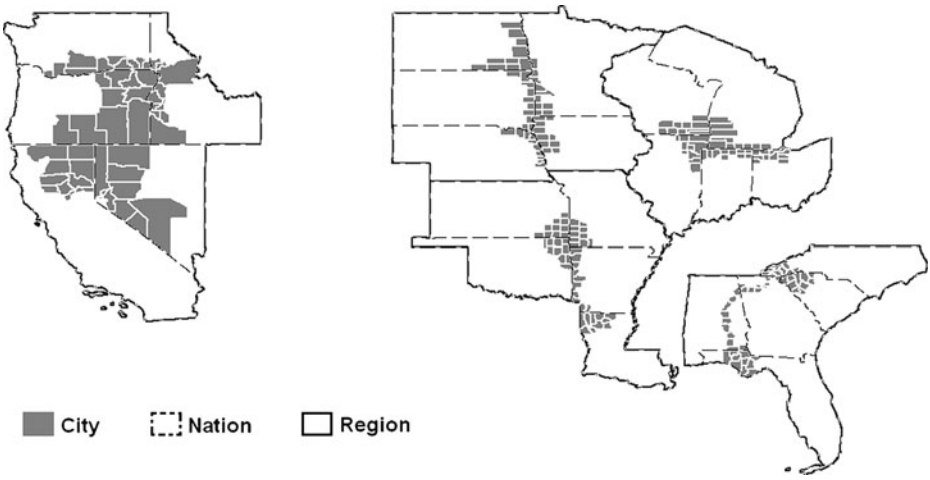


Fig. 9 The DS10 dataset spatial distribution

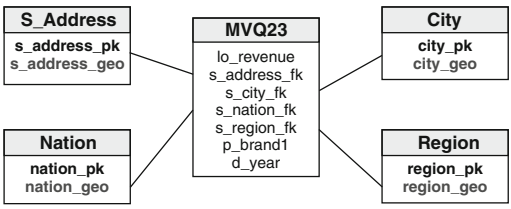
We also created two materialized views, the MVQ23 and the MVQ33, to answer specific SSB queries adapted with spatial predicate. Figure 10 shows how the MVQ23 was built and its schema. Similarly, Fig. 11 shows how the MVQ33 was built and its schema. While the MVQ23 referenced each spatial dimension table once, the MVQ33 did it twice at the City, Nation and Region levels since it referred to both customers and suppliers locations. Both materialized views avoided spatial data redundancy and join operations involving conventional dimension tables. MVQ23's data volume was 65 GB, while MVQ33's volume was 53.5 GB. Considering that the MVQ23 has nearly six hundred million tuples, in order to efficiently fetch p_brand1 = 'MFGR#2239', we built a B-tree on its attribute *p_brand1*. We also analyzed the need to build such index for the MVQ33 on the attribute *d_year*, but concluded that this index did not improve the performance of this materialized view.

Regarding the queries, we define in this paper that the selectivity is the percentage of the number of spatial objects that are retrieved by a query. A *low selectivity* means that few spatial objects are retrieved, while a *high selectivity* means that more spatial objects are retrieved by the query. Another aspects related to queries in our tests are spatial query windows and query types.

Aiming at evaluating the spatial predicate of a SOLAP query and the SB-index and HSB-index capabilities, we designed spatial query windows, which were quadratic,

```
CREATE TABLE MVQ23 AS
SELECT sum(lo_revenue), s_address_fk,
s_city_fk, s_nation_fk, s_region_fk, d_year,
p_brand1
FROM lineorder, part, supplier , date
WHERE lo_suppkey= s_suppkey AND
      lo_orderdate = d_datekey AND
      lo_partkey = p_partkey
GROUP BY s_address_fk, s_city_fk,
s_nation_fk, s_region_fk, d_year, p_brand1
```

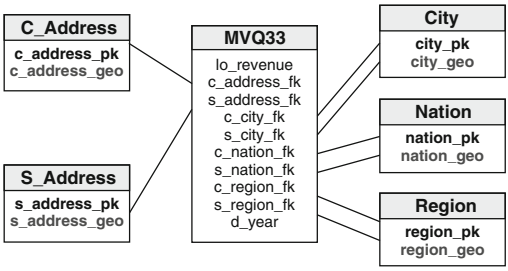
a) Building the MVQ23



b) The MVQ23 schema

Fig. 10 The MVQ23 materialized view

```
CREATE TABLE MVQ33 AS
SELECT SUM(lo_revenue), c_address_fk,
        s_address_fk, c_city_fk, s_city_fk,
        c_nation_fk, s_nation_fk,
        s_region_fk, c_region_fk, d_year
FROM lineorder, customer, supplier, date
WHERE  lo_custkey = c_custkey AND
        lo_suppkey = s_suppkey AND
        lo_orderdate = d_datekey
GROUP BY c_address_fk,  s_address_fk,
        c_city_fk, s_city_fk, c_nation_fk,
        s_nation_fk, s_region_fk,
        c_region_fk, d_year
```



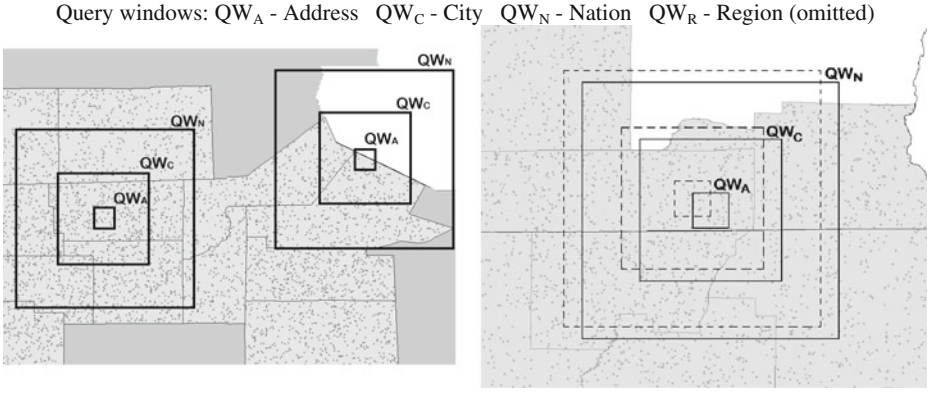
a) Building the MVQ33 **b)** The MVQ33 schema

Fig. 11 The MVQ33 materialized view

correlated with the spatial data, and considered *ad hoc* because their rectangles were not previously stored in any spatial dimension table. A spatial roll-up operation required a set of four spatial query windows, each one associated to a given granularity level (i.e. the Address, City, Nation or Region granularity level) and had a specific size (i.e. the lower the granularity, the smaller the spatial query window). We defined two different types of spatial query windows: disjoint and overlapping. The difference between these two types of query windows is related to the fact that *overlapping* spatial query windows allow for reusing cached data during query processing and therefore can be used to evaluate the efficiency of a specialized buffer-pool in the HSB-index.

Disjoint spatial query windows were created according to the following description. Their centroids were distinct supplier addresses. To create them, initially, one supplier address was randomly chosen to be the centroid of the address query window. Then, city, nation and region query windows were produced according to Fig. 12a. The next centroid for the address query window was chosen assuring that the new query windows would not overlap any one of the previously created query windows.

On the other hand, *overlapping spatial query windows* were built as follows. Firstly, one supplier address was chosen to be the centroid of an address query window. Then, city,



a) Disjoint query windows **b)** Overlapping query windows

Fig. 12 Disjoint and overlapping *ad hoc* spatial query windows for distinct granularity levels

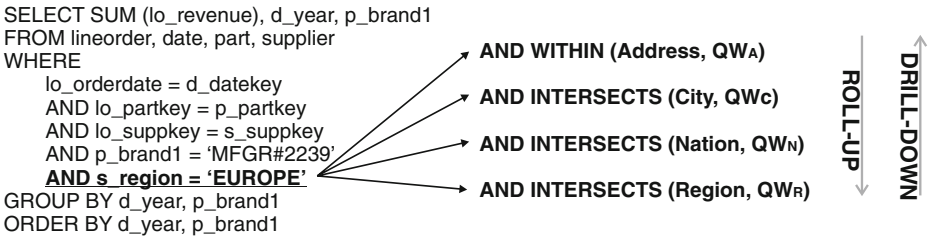


Fig. 13 Roll-up and drill-down operations for the DS10 dataset

nation and region query windows were built similarly to the address query windows (continuous-line query windows in Fig. 12b). The next address query window centroid was not a supplier address, but any point inside the previous address query window. Further, city, nation and region query windows were generated similarly (dashed-line query windows in Fig. 12b). As a result, new address query windows overlapped the previous ones, as well as the city, nation and region query windows did.

For both *disjoint* and *overlapping* spatial query windows, addresses were evaluated with containment range queries and their spatial query windows covered 0.001% of the extent. The City, Nation and Region levels were evaluated with intersection range queries and their spatial query windows covered 0.05%, 0.1% and 1% of the extent, respectively. All queries issued over the DS10 dataset and over the MVQ23 and the MVQ33 materialized views presented low query selectivity for the spatial predicate. For instance, less than 200 addresses were retrieved per spatial query window at the Address level, considering that there were 1 million supplier addresses and 3 million customer addresses.

The workload for the DS10 dataset was based on the query Q2.3 of the SSB, as shown in Fig. 13. We replaced the underlined conventional predicate with spatial predicates involving the *ad hoc* spatial query windows represented by QW_A, QW_C, QW_N and QW_R, as illustrated in Fig. 12. The windows' extents allowed the aggregation of data in different spatial granularity levels, i.e. the application of spatial roll-up and drill-down operations. In fact, a complete spatial roll-up operation started with the query with containment spatial predicate at the Address level, and continued with other three queries that comprised intersection spatial predicate at the City, Nation and Region levels, exactly in this order. When issued over the DS10 dataset, each query required three join operations among the conventional dimension tables and the fact table, one join operation with a spatial dimension table, one conventional predicate computation and one spatial predicate processing.

The workload for the MVQ23 was designed according to Fig. 14, which shows how a complete spatial roll-up operation was performed. These queries avoided joining the conventional dimension tables, but they required one join operation related to the corresponding spatial dimension table, which prevented the spatial data redundancy and the high cost of storing redundant geometries.

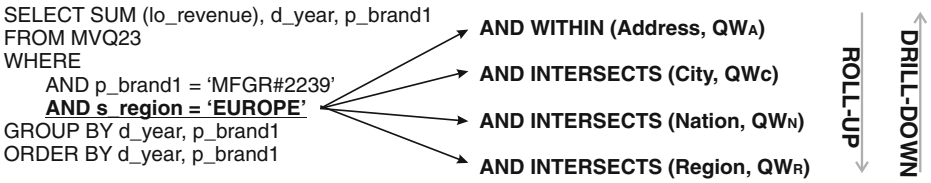


Fig. 14 Roll-up and drill-down operations for the MVQ23 materialized view

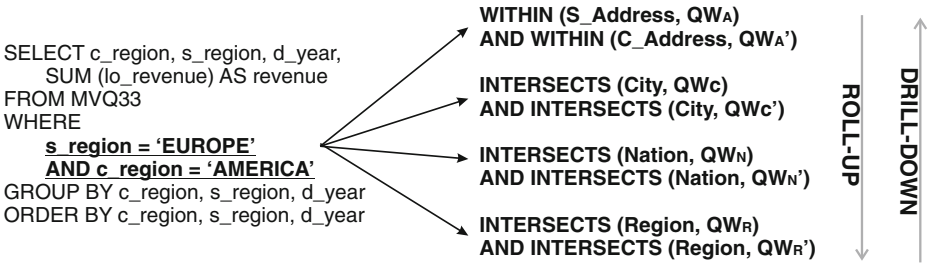


Fig. 15 Roll-up and drill-down operations for the MVQ33 materialized view

On the other hand, the workload for the MVQ33 was based on the query Q3.3 of the SSB, as shown in Fig. 15, and consisted of spatial roll-up and spatial drill-down operations with two *ad hoc* spatial query windows, which added one more spatial predicate to be processed and an extra high join cost. Basically, this query retrieved "the revenue per year per brand for suppliers of an area A_1 to the customers of an area A_2". The same granularity level was used for both customers and suppliers simultaneously. The containment spatial predicate was used at the Address level while the intersection predicate was used at the City, Nation and Region levels. Two joins were necessary to fetch the spatial objects required by the two query windows. MVQ33 also avoided joining conventional dimension tables.

The performance tests were carried out on a computer with a 3.2 GHz Pentium D processor, 8 GB of main memory, a 7200 RPM SATA 750 GB hard disk with 32 MB of cache, Linux CentOS 5.2, PostgreSQL 8.2.5 and PostGIS 1.3.3. We chose the PostgreSQL/PostGIS DBMS because it is an efficient open source software that follows standard specifications for spatial data. We employed FastBit version 0.9.2b as the Bitmap software to implement the star-join Bitmap index and to process the conventional predicates, since it has proven to efficiently implement Bitmap indices, and it is a Free Software as well [36, 37]. Specifically in this paper, FastBit used only the WAH compression method [32].

Both the SB-index and the HSB-index were implemented in C/C++. Regarding the HSB-index, we employed the R*-tree [40] as the *diskSpatialIndex*, the R*-tree's insertion algorithm as the *BuildHSBindex* algorithm, and also the R*-tree's search algorithm as the *SearchHSBindex* algorithm. We chose the R*-tree because it improves the R-tree insertion algorithm with the coverage, overlap, margin and storage criteria, providing a better space partitioning and consequently a better search performance. The R*-tree has a maximum and a minimum number of entries, denoted by M (i.e. disk page size) and given by $m = 40\%$ *of* M, respectively. The percentage used in the m parameter was chosen according to the results described in [40]. Furthermore, we applied the close reinsert strategy in the insertion algorithm of the R*-tree to avoid overlapping. For exchanging pages between the *bufferSpatialIndex* and the *diskSpatialIndex*, we adopted the LRU page replacement policy. The disk page sizes of the SB-index and the HSB-index were set to 4 KB, and the *bufferSpatialIndex* was set to 5%, except for the sections that specifically investigate these parameters.

5.2 Comparing the DBMS resources and the proposed indices

In this section, we compare our proposed indices with current DBMS resources for answering SOLAP queries. Section 5.2.1 focuses on the processing of a single *disjoint* spatial query window, while Section 5.2.2 investigates more complex queries, which require the processing of two *disjoint* spatial query windows.

5.2.1 Using a single disjoint spatial query window

This section compares the query processing performance of the SB-index and the HSB-index with the resources currently available in DBMSs for SDW indexing. Regarding the DBMS resources, we implemented four configurations, as described as follows.

- Configuration C1: the star-join computation using the R-tree on the spatial attributes over the DS10 dataset.
- Configuration C2: query processing over the MVQ23 materialized view using the R-tree on the spatial attributes.
- Configuration C3: the star-join computation using the GiST on the spatial attributes over the DS10 dataset.
- Configuration C4: query processing over the MVQ23 materialized view using the GiST on the spatial attributes.

Both the R-tree and the GiST are implemented by the DBMS, and were used in our performance evaluation to improve the spatial predicate processing performance. We issued 5 spatial roll-up operations, taking the average of the measurements for each granularity level. The system cache was flushed at the end of each operation.

Figure 16 shows the elapsed times collected while running the experiments for configurations C1 to C4. According to the performance results, the attribute granularity affected the query processing over the DS10 dataset. For instance, the Region level queries took 1,186 seconds in configuration C1, while the Address level queries took 21,763 seconds for the same configuration. There was a high increase of almost 1,735%. This trend was also observed for the MVQ23 materialized view. For instance, the increase was of 305% in configuration C4.

As expected, using the MVQ23 materialized view instead of the DS10 dataset drastically decreased the query response time, since the MVQ23 avoids joins and uses a B-tree on the attribute *p_brand1*. The time reduction ranged from 85% at the Region level to 96% at the Address level, independently of using the R-tree or the GiST. The experiments also showed that there was an insignificant difference between the use of the R-tree and the GiST, for both the DS10 dataset and the MVQ23.

Finally, independently of the attribute granularity, the performance results showed that the aforementioned configurations were very expensive and provided unacceptable query response times.

Regarding the experiments using the SB-index and the HSB-index, we issued queries only over the DS10 dataset. Table 5 shows the total elapsed time spent by the HSB-index

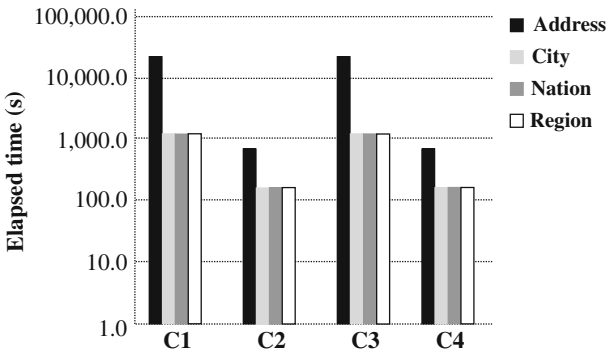


Fig. 16 Performance results of configurations C1, C2, C3 and C4. The performance results are shown in log scale for better visualization

Table 5 Total elapsed time for the SB-index, HSB-index and comparisons

	SB-index (s)	HSB-index (s)	HSB-index Time Reduction		
			C1 or C3	C2 or C4	SB-index
Address	40.832978	40.555110	99.82%	94.02%	0.68%
City	52.379771	52.167719	95.59%	68.96%	0.40%
Nation	38.467938	38.009189	96.79%	77.21%	1.19%
Region	32.791074	32.513888	97.25%	80.49%	0.85%

and the SB-index, and compares the performance of the HSB-index with the SB-index and the best result obtained by configurations C1, C2, C3 or C4. The time reduction columns report these comparisons.

The SB-index and the HSB-index greatly overcame the best results obtained by configurations C1 and C3 (i.e. star-join computation), since the time reduction ranged from 95% to 99%. The time reduction of the proposed indices over the best results achieved by configurations C2 and C4 (i.e. materialized view) were also impressive, ranging from 68% up to 93%. Therefore, we conclude that using the SB-index or the HSB-index instead of computing the star-join or creating materialized views provided much lower query response time.

Comparing the SB-index with the HSB-index, Table 5 also shows that the total elapsed time spent by the HSB-index was almost the same as that of the SB-index with a very small performance gain ranging from 0.68% to 1.19%. The similarity between the results of the proposed indices is due to the fact that the cost to manipulate the conventional predicate was much higher than the cost to process the spatial predicate, i.e. two orders of magnitude greater, as shown in the column FastBit in Table 6.

Regarding only the cost to process the spatial predicate, our results demonstrated that the HSB-index greatly overcame the SB-index, as shown in Table 6. The low query selectivity for the spatial predicate benefited the HSB-index: the reduction provided by the HSB-index over the SB-index ranged from 27% up to 70%. For instance, as stated in Section 5.1, this low query selectivity retrieved less than 200 addresses over 1 million addresses. Therefore, we conclude that the hierarchical data structure of the HSB-index and its capability of pruning index entries required less time to process the spatial predicate than the sequential scan of the SB-index when few spatial objects were processed.

Figure 17 compares the performance results to build the SB-index and the HSB-index. As both the SB-index and the HSB-index required a star-join Bitmap index that was constructed under FastBit software and took 3,532.57 seconds, we show in Fig. 17 only the

Table 6 Performance results for processing the spatial and the conventional predicates

	Spatial predicate			Conventional predicate
	SB-index (s)	HSB-index (s)	Time Reduction	FastBit (s)
Address	0.393210	0.115342	70.67%	40.439768
City	0.774435	0.562383	27.38%	51.605336
Nation	1.115122	0.656373	41.14%	37.352816
Region	0.760018	0.482831	37.47%	32.031057

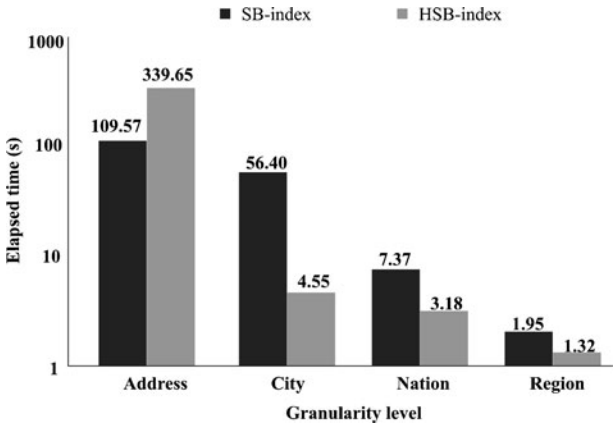


Fig. 17 Elapsed time to build the SB-index and the HSB-index. The performance results are shown in log scale for better visualization

elapsed time to build the sequential file for the SB-index and the elapsed time to build the *diskSpatialIndex* for the HSB-index. We measured the performance for all the granularity levels. Except for the Address level, the elapsed time to build the HSB-index was clearly shorter than the time to build the SB-index. In fact, the higher cardinality of the Address granularity level benefited the construction of the sequential file used by the SB-index and impaired the routines to build the HSB-index.

Considering that the star-join Bitmap index for the DS10 dataset occupied 64 GB, the inclusion of our indices required low additions to the storage requirements (Table 7). Although the HSB-index occupied about 50% more space on disk than the SB-index, it added only at most 0.077% to storage requirements. The small sizes of the SB-index and the HSB-index are another advantage that reinforces their utilization in SDWs.

5.2.2 Using two disjoint spatial query windows

An interesting issue arises when the user requires more than one spatial query window to fetch spatial objects, as it may demand more join operations and spatial predicates to be computed. Therefore, we investigate in this section the performance of the SB-index and the HSB-index considering these more complex queries. We compare the proposed indices with the MVQ33 materialized view. Regarding this view, we indexed each of its spatial attributes with an R-tree to speed up the spatial predicate processing. Note that, from now on, we do not consider the star-join computation in our evaluation since it provided lower performance than materialized views to process SOLAP queries, as described in Section 5.2.1. In the tests, we performed five roll-up operations, and collected the average elapsed time.

Table 8 shows the elapsed times spent by the SB-index, the HSB-index and the MVQ33 materialized view. It also shows the time reduction provided by the HSB-index over the

Table 7 Sizes of the SB-index and the HSB-index for the DS10 dataset

	Address	City	Nation	Region
SB-index	34.5 MB	96 KB	16 KB	8 KB
HSB-index	50.6 MB	144 KB	24 KB	8 KB

Table 8 Comparing the HSB-index, the SB-index and the MVQ33 materialized view

	HSB-index (s)	SB-index (s)	MVQ33 (s)	Time Reduction (%)		
				HSB-index *vs.* SB-index	HSB-index *vs.* MVQ33	SB-index *vs.* MVQ33
Address	2.653380	4.941398	2176.796584	46.30%	99.88%	99.77%
City	147.564586	147.699792	2180.571846	0.09%	93.23%	93.23%
Nation	154.552928	155.097352	2180.241769	0.35%	92.91%	92.89%
Region	158.141973	158.444147	2166.572934	0.19%	92.70%	92.69%

SB-index, as well as the performance gain of the proposed indices over the MVQ33 materialized view. Compared with the MVQ33, our indices obtained outstanding results, with a time reduction varying from 92% to 99%. We conclude that our indices greatly outperformed the current DBMS resources also while processing more complex queries, such as those that require the processing of two spatial query windows.

Comparing the SB-index with the HSB-index, the performance results showed that the HSB-index overcame the SB-index by providing a significant time reduction of 46% at the Address level, which imposed a very low selectivity query. On the other hand, at the City, Nation and Region levels the difference between the SB-index and the HSB-index was unexpressive. This is due the fact that the query selectivity increased as the cardinality became smaller.

Considering that a SOLAP query is composed of spatial and conventional predicates, a deeper analysis on the elapsed time spent by the proposed indices to compute each one of them is given in Table 9. Regarding each granularity level, each predicate was examined according to its percentage of the total elapsed time. Both the SB-index and the HSB-index processed the spatial predicates in less than 1% of the total elapsed time at the City, Nation and Region levels. Furthermore, the obtained results were very similar. Conversely, these interesting findings were not observed at the Address level, which has the highest cardinality. At this level, the cost to process the spatial predicate using the HSB-index was 13% of the total elapsed time, while using the SB-index this cost was 55% of the total elapsed time. We conclude that the hierarchical structure of the HSB-index benefited the spatial predicate computation at the Address level when compared with the sequential scan performed by the SB-index.

Table 9 The cost to process spatial and conventional predicates using our proposed indices

	HSB-index		SB-index	
	Spatial predicates	Conventional predicates	Spatial predicates	Conventional predicates
Address	13.66%	86.34%	55.33%	44.67%
City	0.58%	99.42%	0.60%	99.40%
Nation	0.93%	99.07%	0.93%	99.07%
Region	0.77%	99.23%	0.77%	99.23%

5.3 Investigating the advantages of the specialized buffer-pool

In this section, we investigate the advantages of introducing a specialized buffer-pool (i.e. the *bufferSpatialIndex*) into the HSB-index. Section 5.3.1 compares the HSB-index with current DBMS resources for answering SOLAP queries, while Section 5.3.2 compares the buffered HSB-index with the bufferless SB-index.

5.3.1 Comparing the HSB-index with materialized views

In this section, we investigate the performance of the HSB-index against the MVQ23 materialized view, and also analyze if the system cache (i.e. buffers of the operating system, the DBMS and the hard disk) aids spatial roll-up operation processing as well as the *bufferSpatialIndex* does for the HSB-index. We focus on *uninterrupted* spatial roll-up operations processing with *overlapping* spatial query windows. The term uninterrupted refers to the fact that the system cache *is not flushed* after each roll-up operation, so a subsequent operation may use cached data from a previous operation. As for the HSB-index, the term *overlapping* spatial query windows refers to the fact that, before the first query is processed, the *bufferSpatialIndex* is empty, but in the remaining queries this component tends to contain relevant entries that are used to answer subsequent queries, thus reducing disk accesses.

In the tests, we issued 10 uninterrupted spatial roll-up operations. We gathered the elapsed time to process only the first spatial roll-up operation and the average elapsed time to process the subsequent nine spatial roll-up operations. Each roll-up operation encompassed queries issued over the Address, City, Nation and Region levels.

The performance results are reported in Table 10, where the time reduction column shows how much faster uninterrupted spatial roll-up operations using the HSB-index over the DS10 dataset were than uninterrupted spatial roll-up operations using the DBMS over the MVQ23 materialized view. The HSB-index impressively outperformed the DBMS, although the DBMS used the MVQ23 materialized view, whose size is almost half of the size of the DS10 dataset. The first spatial roll-up operation using the HSB-index executed quite faster and provided a time reduction of 85%, while the second to the tenth spatial roll-up operations provided a great time reduction of 91%.

The system cache aided the DBMS using the MVQ23 materialized view to execute the spatial roll-up operations, since the time reduction was of 43% between the processing of the first roll-up operation and the processing of the nine subsequent ones. However, embedding a small size specialized buffer-pool in the HSB-index produced a more impressive time reduction, of 65%, between the first roll-up operation and the remaining ones. This is an improvement of 22% over the system cache performance.

Based on the performance tests described in this section, we conclude that the buffered HSB-index is an excellent choice for SDW indexing, specifically when executing *uninterrupted* spatial roll-up operations with *overlapping* spatial query windows.

Table 10 DBMS and the HSB-index executing ten roll-up operations sequentially

Roll-up	MVQ23 with R-tree (s)	HSB-index (s)	Time reduction (%)
1st	1,186.734604	170.369645	85.64
2nd to 10th	671.893342	58.958536	91.23

Table 11 Results for the SB-index and the HSB-index with buffer-pool of 7.5% for the first and the subsequent nine queries, according to different granularity levels

Level	Query	SB-index (s)	HSB-index (s)	Time reduction (%)
Address	1st	0.584209	0.128196	78.06
	2nd to 10th	0.487470	0.002988	99.39
City	1st	0.228349	0.004655	97.96
	2nd to 10th	0.228309	0.000083	99.96
Nation	1st	0.499640	0.000053	99.99
	2nd to 10th	0.505050	0.000056	99.99
Region	1st	0.284220	0.000020	99.99
	2nd to 10th	0.319578	0.000023	99.99

5.3.2 Comparing the buffered HSB-index with the bufferless SB-index

In this section, we compare the performance of the *bufferless* SB-index and the *buffered* HSB-index specifically in the spatial filter phase. Note that, differently from Sections 5.2 and 5.3.1, in which we analyzed the *complete query* processing performance (i.e. processing of both spatial and conventional predicates), from now on we investigate only the performance of the *spatial filter phase*, aiming at assessing our indices for their efficiency to process the spatial predicate of SOLAP queries. In the tests, we issued 10 *uninterrupted* spatial roll-up operations using *overlapping* spatial query windows.

Table 11 compares the performance of the bufferless SB-index and the buffered HSB-index. It shows the elapsed time for the first query, as well as the average elapsed time spent in the execution of the other nine queries. The absence of a buffer-pool for the SB-index indicated that the time spent by the first query was similar to that spent by the subsequent nine queries. As for the HSB-index, the first query was not aided by the *bufferSpatialIndex*, while the subsequent queries were. Therefore, the HSB-index showed a remarkable improvement in the processing of the 2nd to 10th queries when compared with the first query at the Address and City levels, which have higher cardinalities. At the Nation and Region levels, which have lower cardinalities, the overhead of the *bufferSpatialIndex* impaired the HSB-index performance. Table 11 also shows how faster the HSB-index was than the SB-index in the Time reduction column.

Based on the results described in this section, we concluded that introducing a specialized buffer-pool into the HSB-index is a positive design characteristic.

5.4 Investigating the indices parameters

In this section, we evaluate the influence of modifying the values of the indices parameters on their performance. Section 5.4.1 investigates the influence of the *bufferSpatialIndex* size in the performance of the HSB-index, while Section 5.4.2 focuses on the impact of different disk page sizes in the performance of both the SB-index and the HSB-index.

5.4.1 Investigating the influence of the buffer-pool size

In this section, we evaluate the HSB-index by investigating the influence of increasing *bufferSpatialIndex* sizes, as follows: 7.5%, 15%, 30%, 45% and 60%. We focused on verifying at which point a larger *bufferSpatialIndex* minimizes disk accesses so that the

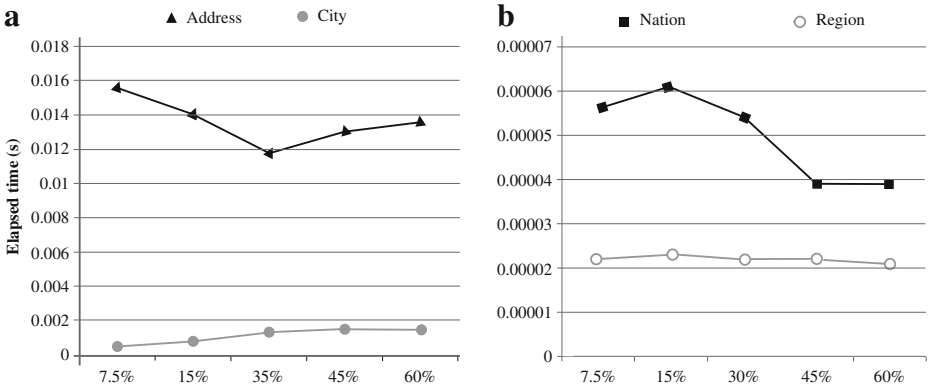


Fig. 18 Elapsed times in seconds for increasing buffer-pool sizes at different granularity levels

spatial filter becomes faster. In the tests, we issued 10 *uninterrupted* spatial roll-up operations using *overlapping* spatial query windows.

Figure 18 shows the performance results for each granularity level. As there were few spatial objects at the Region level, the increase in the size of the *bufferSpatialIndex* did not introduce any performance gain (Fig. 18b). The result obtained with the smallest size was very close to the result gathered with the largest size. The Nation level has also a small cardinality, but the *bufferSpatialIndex* sizes of 45% and 60% outperformed the smaller *bufferSpatialIndex* sizes (Fig. 18b). For instance, the size of 45% imposed a time reduction of 30.35% over the size of 7.5%. On the other hand, both the sizes of 45% and of 60% provided the same elapsed time. Therefore, the *bufferSpatialIndex* size of 45% limited the performance gains at the Nation level. At the City level, there were 10 times more polygons than at the Nation level. However, according to Fig. 18a, the smallest *bufferSpatialIndex* size was enough to provide the shorter elapsed time. At the Address level, which has the highest cardinality and stores points, the best performance was obtained with the intermediate *bufferSpatialIndex* size of 30%. Also, using larger sizes than 30% did not improve the query processing performance at the Address level, as shown in Fig. 18a.

Based on the results described in this section, we conclude that the increase in the *bufferSpatialIndex* size influenced the HSB-index query processing when indexing attributes with higher cardinalities. A remarkable result obtained with our tests is that the HSB-index had a good performance even with the smallest *bufferSpatialIndex* size. Therefore, we also conclude that the HSB-index in general did not require a specialized buffer-pool larger than 30% to efficiently process SOLAP queries.

5.4.2 Investigating the influence of disk page size

In this section, we investigate the influence of the disk page size in the spatial filter phase for both the SB-index and the HSB-index. There are three properties of our indices that motivate this investigation. Firstly, larger page sizes assure that each page holds more entries, and therefore fewer pages must be accessed on disk and copied to the main memory. Furthermore, for the HSB-index, there is a threshold that limits query processing improvements, as these improvements just occur if few disk accesses are performed and overcome the cost of transferring false hits from disk to the main memory. Finally, varying

the disk page size influences the clustering of objects in the HSB-index and, consequently, the node occupation rate.

In the performance tests described in this section, the experimental setup was almost the same as that described in Section 5.1. The difference is that we used five increasing disk pages sizes for both the SB-index and the HSB-index, as follows: 512 bytes, 1 KB, 4 KB, 8 KB and 16 KB. Recall that we also used *overlapping* spatial query windows in this experiment.

Figure 19 shows the average elapsed time spent by the SB-index and the HSB-index at the Address, City, Nation and Region granularity levels. Comparing the SB-index and the HSB-index, the results showed that the HSB-index performed better than the SB-index at all the granularity levels and for all the disk page sizes.

Regarding the SB-index, the increase in the disk page size did not greatly improve its performance at the Address level, since the evaluation of points for containment predicate was very fast despite the high cardinality of this level. For all the remaining levels, which store polygons and have lower cardinality than the Address level, the impact was very high. At the Nation and Region levels, the results showed that the larger the disk page size, the better the SB-index performance. For these levels, one disk page of 16 KB stored the entire

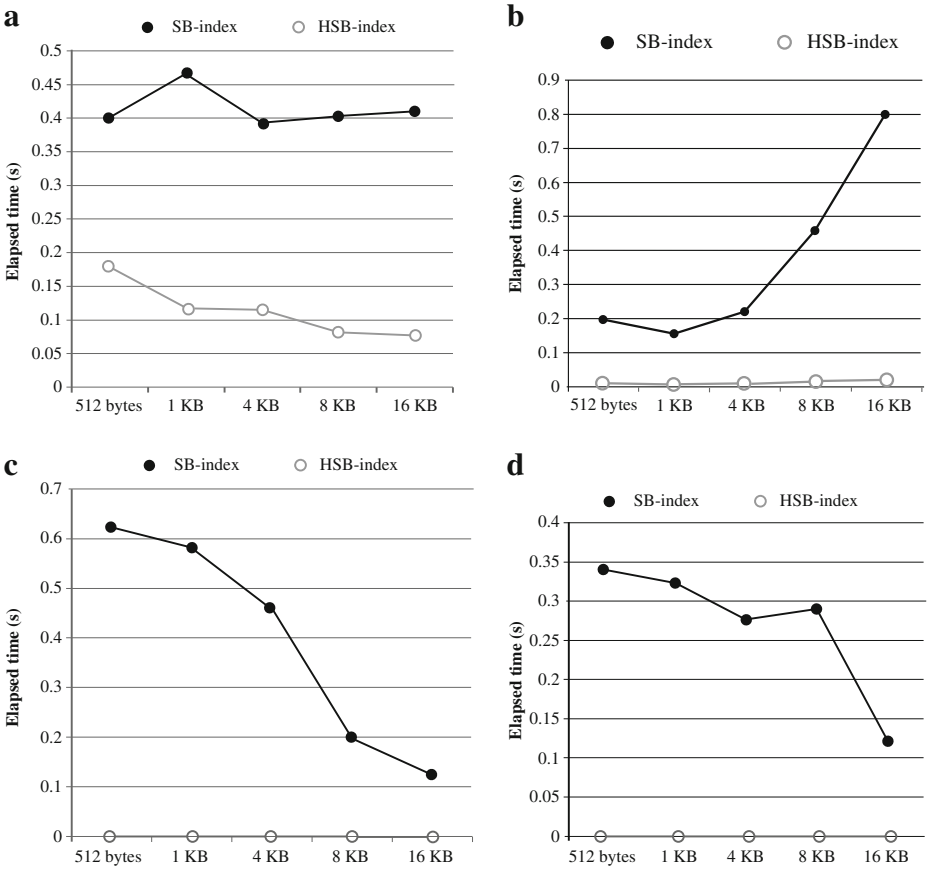


Fig. 19 Performance results for increasing disk page sizes at different granularity levels

Table 12 Performance results for the HSB-index using increasing disk page sizes

Disk page size	Address	City	Nation	Region
512 bytes	0.179943	0.010388	0.000042	0.000027
1 KB	0.118366	0.005256	0.000033	0.000024
4 KB	0.115342	0.008621	0.000156	0.000019
8 KB	0.082581	0.015967	0.000049	0.000030
16 KB	0.077466	0.019425	0.000063	0.000051
Difference between the shortest and the longest elapsed time	132%	269%	372%	168%

index, which was exchanged from disk to the main memory only once. On the other hand, the larger the disk page size, the worst the SB-index performance at the City level. For this level, transferring false hits from disk to the main memory was costly for larger disk pages.

Concerning the HSB-index, Table 12 provides an accurate visualization of the elapsed time spent by this index. As can be noted in this table, the disk page size drastically affected the HSB-index performance at all the granularity levels. The best performance at the Address level, which stores points and has the highest cardinality, was obtained with the largest disk page size. The results showed that as the disk page size increased, the elapsed time decreased. For the remaining three levels, which store polygons and have lower cardinality than the Address level, the HSB-index achieved better results using smaller disk page sizes, especially 1 KB and 4 KB. As the HSB-index clusters MBRs to enable pruning, smaller page sizes determined a better performance.

The results described in this section evidenced that attribute cardinality and geometry type (i.e. point or polygon) are important issues to be taken into account when determining the disk page size of both the SB-index and the HSB-index.

5.5 Investigating the influence of the query selectivity

Differently from all previous performance tests that considered spatial query windows with a fixed size, in this section we analyze how spatial query windows with increasing sizes influence the performance of the SB-index and the HSB-index in the spatial filter phase, specifically at the Address level that has the highest cardinality.

The spatial query windows previously used contained less than 200 objects, which lead to a very low query selectivity. On the other hand, in this section, spatial query windows were designed to contain more objects and to provide a higher selectivity, i.e. they were designed to increase the number of spatial objects retrieved. In the tests, we issued five *ad hoc disjoint* spatial query windows that retrieved increasing number of spatial objects, which varied from less than 200 objects up to 10,000 objects. After each query, the system cache was properly flushed. Figure 20 details the results.

Considering that the SB-index performs a sequential scan to filter spatial objects, it was expected a constant elapsed time. However, it is possible to note that the greater the number of retrieved objects, the longer the elapsed time spent by the SB-index to process the spatial filter. The main factor that impaired the sequential scan of the SB-index was the task of collecting primary key values. The greater the number of candidates, the greater the number of primary key values collected in the main memory, which caused an overhead. On the other hand, the elapsed time did not show an abrupt increase rate.

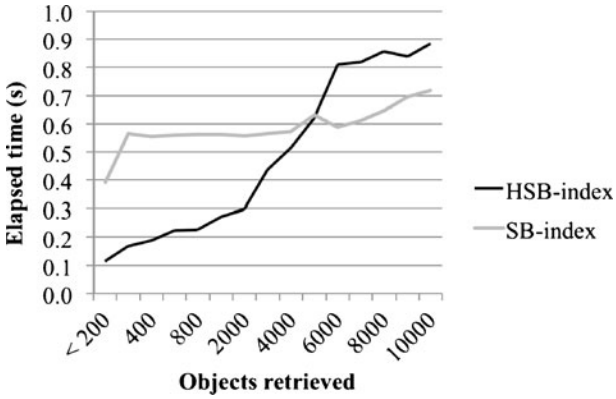


Fig. 20 The performance of the SB-index and the HSB-index according to increasing query selectivity in the spatial filter phase

The same factor that affected the SB-index performance should also impair the HSB-index spatial filter processing. But, there was a more relevant issue related to the HSB-index performance losses. Since the HSB-index has a tree-based structure, the search algorithm prunes the tree traversal whenever possible. However, when the traversal includes ramifications, the number of disk accesses increases, thus introducing additional overhead. In Fig. 20, it is possible to observe that these factors drastically decreased the HSB-index performance in the spatial filter phase. In fact, the SB-index overcame the HSB-index for queries that retrieved more than 5,000 objects.

Comparing the performance results described in this section with those described in previous sections, we conclude that queries that required only a few objects to be processed benefitted the HSB-index against the SB-index. On the other hand, queries that required a higher number of spatial objects to be processed benefitted the SB-index.

5.6 Investigating the influence of the spatial data type

In this section, we investigate the influence of the spatial data type on the performance of the SB-index and the HSB-index in the spatial filter phase. We evaluated the Address level, which has 1,000,000 objects, using points, lines and multipolygons. Similarly to the data generation described in Section 5.1, here we also used synthetic points and obtained the geometries of lines and multipolygons from the Tiger/Line real dataset (http://www.census.gov/geo/www/tiger). We issued five queries with exactly the same selectivity for each spatial data type, i.e. each query retrieved less than 200 objects, and used the spatial predicate containment considering *disjoint* spatial query windows. After each query, we flushed the system cache.

Figure 21 shows the average elapsed time spent by the SB-index and the HSB-index to process each data type. Using points as spatial objects, the HSB-index outperformed the SB-index. On the other hand, the SB-index outperformed the HSB-index using lines or multipolygons as spatial objects. Although the tree-based HSB-index is able to prune the tree traversal, this strategy was not efficient for lines and multipolygons, whose geometries introduced dead space in the MBR representation and lead to an overlapping of the HSB-index intermediate nodes. Therefore, the sequential scan provided by the SB-index was more advantageous than the pruning of the HSB-index for these two data types.

Analyzing each proposed index separately, the results show that the time spent by the SB-index was almost the same for points, lines and multipolygons. Therefore, the spatial data type did not affect this index. This is due the fact that the SB-index performs sequential scan. Conversely, the spatial data type did impair the performance of the HSB-index, as previously discussed.

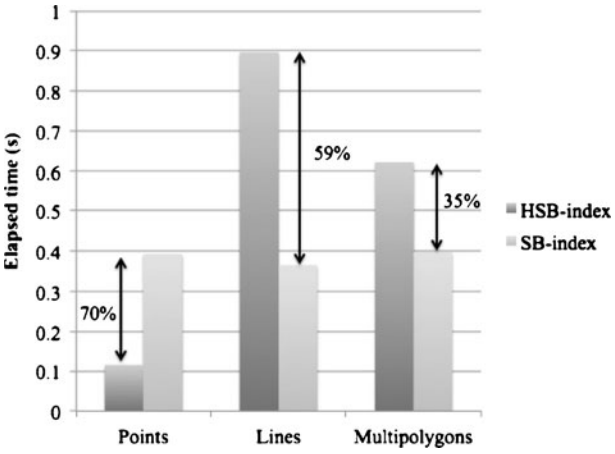


Fig. 21 Performance results for using different spatial data types

6 Related work

The Bitmap index has been applied successfully in conventional DWs to improve OLAP query processing, as it avoids join operations among the fact table and the dimension tables and because multidimensionality is not an obstacle. Therefore, it has been a focus of interest for both researchers and commercial DBMSs [25, 29–31, 41, 42]. Furthermore, new functionalities have been proposed for it, such as binning, encoding and compression techniques [32–34, 36, 37]. Other related proposals focus on organizing hierarchically Bitmap indices for indexing dimensional data [43, 44]. However, these proposals do not investigate how the Bitmap index should handle spatial data, which is the objective of the indices proposed in this paper. In fact, the Bitmap index is not designed to support spatial hierarchies, spatial predicates and SOLAP queries such as spatial roll-up and drill-down operations.

In [27], the authors state the need for indices to efficiently answer spatial statistical queries that require repeated computation of neighborhood relationships. For this purpose, they introduce the SJELI and the SJALI indices, which are self-join indices that speed up queries that aim at discovering spatial instance clusters and hotspots for different granularity levels. However, the authors do not discuss the need for joining tables and computing spatial and conventional predicates in drill-down and roll-up operations over SDWs. On the other hand, the proposed SB-index and HSB-index focus on predefined spatial hierarchies, taking advantage of the 1:N association among higher and lower granularity spatial attributes and, therefore, benefiting the processing of drill-down and roll-up operations extended with the spatial predicates intersection and containment.

Other indices for spatio-temporal DW are the R-MVB, the STCAT, and a Bitmap index for distributed environments [20]. They are appropriate for parallel processing and depend on the use of a specific load-balancing algorithm that defines the participation of every node in query processing. These indices differ from the indices proposed in this paper since they do not specify how spatial hierarchies should be handled and, therefore, how spatial drill-down and roll-up operations should be carried out. That is, similarly to the SJELI and the SJALI indices, the indices proposed in [20] do not offer functionalities to deal with predefined spatial hierarchies, which is one of the main characteristics of the SB-index and the HSB-index.

There is a specific index in the literature addressed for multidimensional queries with spatial predicate in SDW, called the aR-tree [10], which reuses the R-tree's partitioning method. Each entry of the aR-tree maintains a MBR, the aggregation function value for all the spatial objects enclosed by this MBR, a pointer to the node that maintain these objects, and an array that stores the aggregation function values for attributes defined in conventional dimensions. Visiting higher or lower levels of the tree, progressively, provide spatial roll-up and drill-down operations by ascending or descending the tree levels, respectively. But the aR-tree differs from our proposed indices on its purpose. While the SB-index and the HSB-index focus on *predefined spatial hierarchies*, the aR-tree is aimed at another type of hierarchy, which is called *ad hoc spatial hierarchy* in this paper.

In a predefined spatial hierarchy, the association among higher and lower granularity spatial attributes is well known at design time, and the SDW schema is properly organized to support this hierarchy, as described in Section 1. For instance, suppose that the cities illustrated in Fig. 22a report the quantity of an arbitrary part sold by a certain supplier. In this figure, each color in gray scale indicates that the city is located in a specific nation. Considering a predefined spatial hierarchy *(nation_geo)* $\preceq$ *(city_geo)*, a roll-up operation would aggregate the measures as shown in Fig. 22b, that is, considering that each city is *inside* one nation.

On the other hand, in an *ad hoc* spatial hierarchy, the spatial objects are stored such that the spatial hierarchies are not known at design time nor the spatial dimension tables are

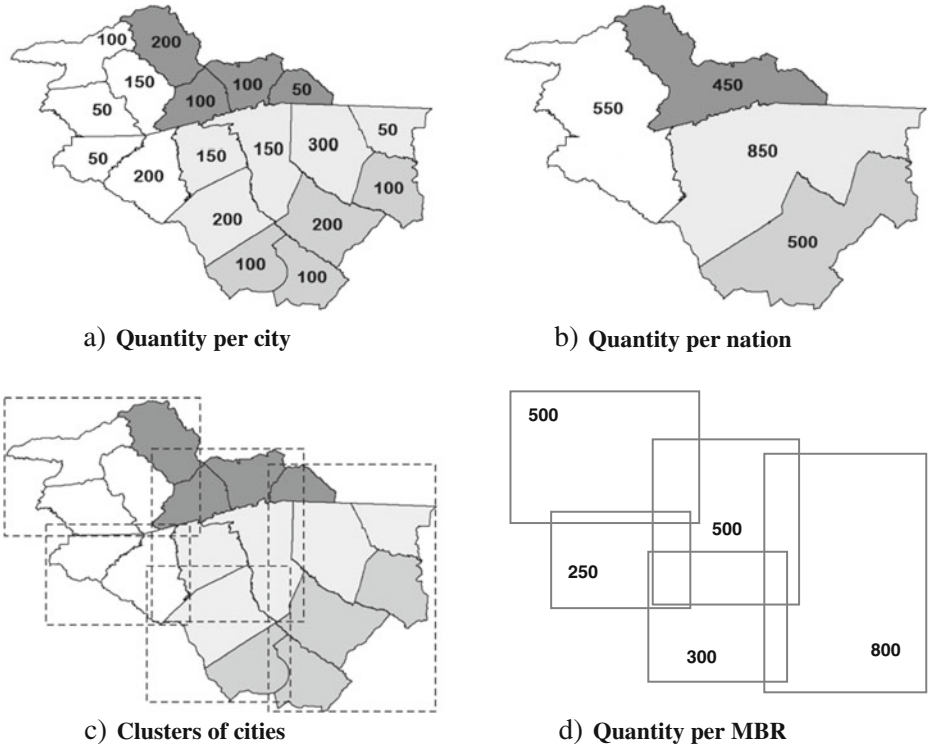


Fig. 22 Spatial roll-up operations according to predefined and *ad hoc* spatial hierarchies

organized to support them. As a result, the hierarchy is obtained through building a spatial index that clusters the spatial objects maintained by the lowest granularity level attribute. The spatial objects referenced by the leaf nodes of the index are considered at the lowest level of granularity, while the parent nodes are considered at a higher level. Note that the entries in leaf nodes reference spatial objects that are really stored in the SDW, while the entries in internal nodes contain MBRs that are created by the spatial index and are not stored in the SDW.

For instance, Fig. 22c,d show a roll-up operation that aggregates the lowest granularity level attribute that corresponds to the leaf nodes into non-leaf nodes. Although the cities remain shaded in Fig. 22c, the *ad hoc* spatial hierarchy does not consider the existence of nations. There is still a relationship of 1:N among each MBR of Fig. 22d and the spatial objects, but these MBRs were built by applying the clustering algorithm of a spatial index, such as the R-tree (as the aR-tree does). Clearly, the aggregated measure values in Fig. 22d differ from those values in Fig. 22b. Also, the areas of the MBRs are different from the areas of the nations. Furthermore, the number of MBRs is not the same number of nations, and these MBRs do not exist in the SDW. As a result, measures of cities from different nations may be aggregated considering a given MBR, but this aggregation will not consider and respect the hierarchies between nations and cities.

Based on the aforementioned discussion, it is possible to note that there are substantial differences between predefined and *ad hoc* spatial hierarchies and also between the roll-up and drill-down operations that they define. These hierarchies should not be misused or mixed, since they are strongly linked to distinct domains of SDW. Therefore, index structures designed for *ad hoc* spatial hierarchies are not able to perform the roll-up and drill-down operations outlined by predefined spatial hierarchies. This is an important difference between the aR-tree and the SB-index and HSB-index.

An extension of the aR-tree is the aRB-tree [28], which is a spatio-temporal DW index that comprises an aR-tree where each entry points to a B-tree that maintains the historical aggregation values for the MBR in the corresponding entry. The spatial predicates are answered by the aR-tree, while temporal predicates are answered by the B-tree. Therefore, comparing the SB-index and the HSB-index with the aRB-tree, the same differences previously discussed for the aR-tree also apply to the aRB-tree.

Finally, our approach is also related to reusing consolidated resources offered by commercial DBMSs to efficiently support analytical multidimensional queries based on spatial predicates. In detail, the SB-index and the HSB-index have two layers, a spatial filter and the star-join Bitmap index. The spatial filter layer is used to handle the spatial predicate, and is a sequential file in the SB-index and any tree-based spatial index with its corresponding clustering technique in the HSB-index. Although commercial DBMSs provide resources to process queries over DWs and geographic data, they do not consider them in the same set. Conversely, in this paper, we propose two data structures based on the star-join Bitmap index and on a spatial index and show how these resources can be coupled to provide good SOLAP query performance. Furthermore, the proposed indices are aimed at offering support to drill-down and roll-up operations over spatial data, which is a feature not supported by commercial DBMSs.

7 Conclusions and future work

In this paper, we proposed two indices for spatial data warehouses (SDWs): the SB-index and the HSB-index. Both of them share the following characteristics: (i) they

enable multidimensional queries with spatial predicate for SDW; (ii) they support *predefined spatial hierarchies*; (iii) they introduce the use of the Bitmap index in SDW, inheriting all the Bitmap index' advantages; (iv) they compute the spatial predicate and transform it into a conventional one, which can be evaluated together with other conventional predicates by accessing a star-join Bitmap index; and (v) they require a reduced amount of storage space.

Our indices corroborated the idea that an efficient spatial filter followed by processing a star-join Bitmap index is a good choice to speed up SOLAP queries in SDW. The SB-index is an adapted Projection index on the primary key of the spatial dimension table, which was designed as a sequential file whose entries store a key value and a MBR. It performs a sequential scan to filter the spatial objects. On the other hand, the HSB-index replaces the SB-index' sequential scan by a hierarchical data structure to cluster spatial objects, thus pruning index entries and reducing the number of disk accesses. The HSB-index also manages a specialized buffer-pool to decrease even more the number of disk accesses.

The advantages introduced by the SB-index and the HSB-index were analyzed through performance tests aimed at investigating their efficiency, characteristics and applicability. The results showed that both the proposed indices were very efficient compared with typical DBMS resources to process spatial drill-down and roll-up operations. Comparisons among the SB-index, the HSB-index, the star-join computation and the use of materialized views demonstrated that the performance gain of our indices ranged from 95% to 99% over the star-join computation, and was of at least 68% over materialized views. These results were obtained for queries with different complexities, i.e. we investigated the processing of a single spatial query window, the processing of two spatial query windows, and the processing of uninterrupted spatial roll-up operations.

Regarding the characteristics of the SB-index and the HSB-index, they were evaluated considering using a specialized buffer-pool, increasing buffer-pool sizes and different disk pages sizes. The performance results demonstrated that introducing a specialized buffer-pool into the HSB-index is a positive design characteristic. The results also showed that increasing the buffer-pool size influenced the performance of the HSB-index when indexing attributes with high cardinality, and evidenced that the HSB-index in general did not require a buffer-pool larger than 30% to efficiently process SOLAP queries. Furthermore, the results demonstrated that the attribute cardinality and the geometry type of the objects are important issues to be taken into account when choosing the disk page size of the proposed indices

Choosing between the SB-index and the HSB-index in terms of their applicability strongly depends on the query selectivity of the spatial predicates. The HSB-index should be used when the query requires that only few spatial objects be processed. On the other hand, the SB-index should be used mainly when the query requires that a greater number of spatial objects be processed.

We are currently extending the SB-index and the HSB-index to perform spatial drill-across operations, i.e. we are extending the proposed indices to support analytical multidimensional queries involving both drill-across and drill-down/roll-up operations based on spatial predicates. We are also investigating the applicability of the proposed indices over spatial joins. Furthermore, both the SB-index and the HSB-index need update policies that are beyond the scope of this paper. These policies consist of an important future work to enhance the scalability of the proposed indices. Another future work refers to the use of our approach to solve multidimensional spatial queries based on the computation of spatial measures.

Acknowledgments This work has been supported by the following Brazilian research agencies: FAPESP, CAPES, CNPq, INEP, and FINEP. The first and the last authors thank the support of the Web-PIDE Project in the context of the Observatory of the Education of the Brazilian Government. The second author's work has been funded by FAPESP under the Grant 2009/06052-7. The work carried by the third author was supported by funds from the CNPq under the Grant 479018/2009-0.

References

1. Mateus RC, Times VC, Siqueira TL, Ciferri RR, Ciferri CDA (2010) How does the spatial data redundancy affect query performance in geographic data warehouses? Journal of Information and Data Management 1:519–534
2. Sampaio MC, Sousa AG, Baptista CS (2006) Towards a logical multidimensional model for spatial data warehousing and OLAP, Proceedings of the 9th ACM International Workshop on Data warehousing and OLAP, New York, NY, USA: ACM, pp. 83–90
3. Malinowski E, Zimányi E (2007) Logical representation of a conceptual model for spatial data warehouses. Geoinformatica 11:431–457
4. Bimonte S, Tchounikine A, Miquel M (2005) Towards a spatial multidimensional model. Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP, New York, NY, USA: ACM, pp. 39–46
5. Rivest S, Bédard Y, Proulx M, Nadeau M, Hubert F, Pastor J (2005) SOLAP technology: merging business intelligence with geospatial technology for interactive spatio-temporal exploration and analysis of data. ISPRS J Photogramm Remote Sens 60:17–33
6. Kimball R, Ross M (2002) The data warehouse toolkit: the complete guide to dimensional modeling. Wiley, New York
7. Harinarayan V, Rajaraman A, Ullman JD (1996) Implementing data cubes efficiently. SIGMOD Rec 25:205–216
8. Malinowski E, Zimányi E (2008) Advanced data warehouse design: from conventional to spatial and temporal applications (Data-centric systems and applications). Springer
9. Stefanovic N, Han J, Koperski K (2000) Object-Based Selective Materialization for Efficient Implementation of Spatial Data Cubes. IEEE Trans Knowl Data Eng 12:938–958
10. Papadias D, Kalnis P, Zhang J, Tao Y (2001) Efficient OLAP Operations in spatial data warehouses. Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases. Springer-Verlag, London, pp 443–459
11. Rao F, Zhang L, Yu XL, Li Y, Chen Y (2003) Spatial hierarchy and OLAP-favored search in spatial data warehouse. Proceedings of the 6th ACM International Workshop on Data Warehousing and OLAP. ACM, New York, pp 48–55
12. Malinowski E, Zimányi E (2005) Spatial Hierarchies and Topological Relationships in the Spatial MultiDimER Model. In: Jackson M, Nelson D, Stirk S (eds) British National Conference on Databases. Springer, Sunderland, UK, pp. 17–28
13. Ruiz CV, Times VC (2009) A Taxonomy of SOLAP Operators, Proceedings of the 24th Brazilian Symposium on Database. SBC, Porto Alegre, pp 151–165
14. Gaede V, Günther O (1998) Multidimensional access methods. ACM Comput Surv 30:170–231
15. Pourabbas E, Rafanelli M (1999) Characterization of hierarchies and some operators in OLAP environment, Proceedings of the 2nd ACM International Workshop on Data Warehousing and OLAP. ACM, New York, pp 54–59
16. Baikousi E, Vassiliadis P (2009) View usability and safety for the answering of top-k queries via materialized views, Proceeding of the ACM 12th International Workshop on Data Warehousing and OLAP. ACM, New York, pp 97–104
17. Golfarelli M, Maniezzo V, Rizzi S (2004) Materialization of fragmented views in multidimensional databases. Data Knowl Eng 49:325–351
18. Ciferri CD, Ciferri RR, Forlani DT, Traina AJ, Souza FF (2007) Horizontal fragmentation as a technique to improve the performance of drill-down and roll-up queries, Proceedings of the 2007 ACM Symposium on Applied computing. ACM, New York, pp 494–499
19. Bellatreche L, Woameno KY (2009) Dimension table driven approach to referential partition relational data warehouses, Proceeding of the ACM 12th International Workshop on Data Warehousing and OLAP. ACM, New York, pp 9–16
20. Gorawski M, Gorawski M (2006) Balanced Spatio-Temporal Data Warehouse with R-MVB, STCAT and BITMAP Indexes", Proceedings of the 5th International Symposium on Parallel Computing in Electrical Engineering. Washington, IEEE Computer Society, pp 43–48

21. Wehrle P, Miquel M, Tchounikine A (2007) A grid services-oriented architecture for efficient operation of distributed data warehouses on globus, Proceedings of the 21st International Conference on Advanced Networking and Applications. IEEE Computer Society, Washington, pp 994–999
22. Siqueira TL, Ciferri RR, Ciferri CD, Times VC (2008) Investigating the effects of spatial data redundancy in query performance over geographical data warehouses, Proceedings of the 10th Brazilian Symposium on Geoinformatics. SBC, Rio de Janeiro, pp 1–12
23. Siqueira TL, Ciferri RR, Times VC, Ciferri CD (2009) A spatial bitmap-based index for geographical data warehouses, Proceedings of the 24th ACM Symposium on Applied Computing. ACM, New York, pp 1336–1342
24. Siqueira TL, Ciferri CD, Times VC, Oliveira AG, Ciferri RR (2009) The impact of spatial data redundancy on SOLAP query performance. J Braz Comput Soc 15:19–34
25. O'Neil P, Graefe G (1995) Multi-table joins through bitmapped join indices. SIGMOD Rec 24:8–11
26. Jürgens M, Lenz H (1999) Tree based indexes vs. Bitmap indexes: a performance study, Proceedings of the 1st International Workshop on Design and Management of Data Warehouses. CEUR-WS.org, Heidelberg, pp. 14–15
27. Mohan P, Wilson RE, Shekhar S, George B, Levine N, Celik M (2008) Should SDBMS support a join index?: a case study from CrimeStat, Proceedings of the 16th International Conference on Advances in Geographic Information Systems. ACM, New York, pp 1–10
28. Papadias D, Tao Y, Kalnis P, Zhang J (2002) Indexing spatio-temporal data warehouses, Proceedings of the 18th International Conference on Data Engineering. IEEE Computer Society, Washington, pp 166–175
29. O'Neil P, Quass D (1997) Improved query performance with variant indexes, Proceedings of the 1997 ACM SIGMOD International Conference on Management of data. New York, ACM, pp 38–49
30. Stockinger K, Wu K (2006) Bitmap Indices for Data Warehouses. In: Wrembel R, Koncilia C (eds) Data Warehouses and OLAP. IRM Press, pp. 157–178
31. Chen L (2009) Curse of dimensionality. Encyclopedia of database systems. Springer, pp. 545–546
32. Wu K, Otoo EJ, Shoshani A (2006) Optimizing bitmap indices with efficient compression. ACM Trans Database Syst 31:1–38
33. Wu K, Stockinger K, Shoshani A (2008) Breaking the curse of cardinality on bitmap indexes. In: Ludäscher B, Mamoulis N (eds) Scientific and statistical database management conference. Springer, Hong Kong, pp. 348–365
34. Chan C, Ioannidis YE (1999) An efficient bitmap encoding scheme for selection queries, Proceedings of the 1999 ACM SIGMOD International Conference on Management of data. ACM, New York, pp 215–226
35. O'Neil P, O'Neil E, Chen X, Revilak S (2009) The Star Schema Benchmark and Augmented Fact Table Indexing, Proceedings of the 1st Transaction Processing Performance Council Technology Conference. Springer LNCS 5895, Lyon, pp. 237–252
36. Wu K, Ahern S, Bethel EW, Chen J, Childs H, Cormier-Michel E, Geddes C, Gu J, Hagen H, Hamann B, Koegler W, Lauret J, Meredith J, Messmer P, Otoo E, Perevoztchikov V, Poskanzer A, Prabhat, Rübel O, Shoshani A, Sim A, Stockinger K, Weber G, Zhang W (2009) FastBit: interactively searching massive data. J Phys Conf Ser 180:12053
37. O'Neil E, O'Neil P, Wu K (2007) Bitmap index design choices and their performance implications, Proceedings of the 11th International Database Engineering and Applications Symposium. IEEE Computer Society, Washington, pp 72–84
38. Guttman A (1984) R-trees: a dynamic index structure for spatial searching, Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. ACM, New York, pp 47–57
39. Aoki PM (1997) Generalizing "Search" in generalized search trees, Proceedings of the 14th International Conference on Data Engineering. IEE Computer Society, Washington, pp 380–389
40. Beckmann N, Kriegel H, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. SIGMOD Rec 19:322–331
41. Bellatreche L, Boukhalfa K (2010) Yet Another Algorithms for Selecting Bitmap Join Indexes, Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery, Springer LNCS 6263, Bilbao, pp 105–116
42. Morales T, Rich K (2009) Oracle database reference, 11 g Release 1, Oracle Corporation. Available at http://www.oracle.com/pls/db111/homepage
43. Chmiel J, Morzy T, Wrembel R (2009) HOBI: Hierarchically Organized Bitmap Index for Indexing Dimensional Data, Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery, Springer LNCS 5691, pp. 87–98
44. Morzy M, Morzy T, Nanopoulos A, Manolopoulos Y (2003) Hierarchical Bitmap Index: an efficient and scalable indexing technique for set-valued attributes, Proceedings of the 7th East European Conference on Advances in Databases and Information Systems, Springer LNCS 2798, pp. 236–252

Thiago Luís Lopes Siqueira received the BSc degree in computer science from the State University of São Paulo, Brazil, in 2006. In 2009, he received the MSc degree in computer science from the Federal University of São Carlos. He is currently a PhD student at the Federal University of São Carlos, and also an assistant professor in the São Paulo Federal Institute of Education, Science and Technology in São Carlos, Brazil. His research interests include data warehousing, geographical information systems and spatial databases.

Cristina Dutra de Aguiar Ciferri received the BSc degree in computer science from the Federal University of São Carlos, Brazil, in 1992. In 1995, she received the MSc degree in computer science from the State University of Campinas, Brazil. She obtained her PhD degree in 2002 in computer science from the Federal University of Pernambuco, Brazil. She is currently an assistant professor in the Computer Science Department at the University of São Paulo in São Carlos, Brazil. Her research interests include data warehousing, geographical information systems, spatial databases, heterogeneous and distributed databases, data provenance and bioinformatics.

Valéria Cesário Times received the BSc degree in statistics from the Catholic University of Pernambuco, Brazil, in 1991. In 1994, she received the MSc degree in computer science from the Federal University of Pernambuco, Brazil. She obtained her PhD degree in computer science in 1999 from University of Leeds, United Kingdom. She is currently an assistant professor in the Informatics Center at the Federal University of Pernambuco, Brazil. Her research interests include data warehousing, geographical information systems, spatial databases, mobile object databases, autonomous databases and advanced database applications.

Ricardo Rodrigues Ciferri received the BSc degree in computer science from the Federal University of São Carlos, Brazil, in 1992. In 1995, he received the MSc degree in computer science from the State University of Campinas, Brazil. He obtained his PhD degree in 2002 in computer science from the Federal University of Pernambuco, Brazil. He is currently an assistant professor in the Computer Science Department at the Federal University of São Carlos, Brazil. His research interests include data warehousing, geographical information systems, spatial databases, bioinformatics and biological databases.