Native Multidimensional Indexing in Relational Databases

David Hoksza, Tomáš Skopal

Department of Software Engineering
Charles University in Prague
Malostranské nám. 25, 118 00, Prague 1, Czech Republic
{david.hoksza, tomas.skopal}@mff.cuni.cz

Abstract

In existing database systems there is a strong need for searching data according to many attributes. In commercial database platforms, the standard search over multiple attributes is provided by B^+-tree (or it's variants) with compound keys. On the other hand, such systems provide also multidimensional indexing, however, just for spatial purposes (such as GIS or CAD applications) and use special data types and querying syntax. In this paper we propose a native multidimensional method for indexing tables with simple attributes, such that multi-attribute queries can be processed (with standard SQL queries) more efficiently than by simple B^+-tree with compound keys. For implementation we have used the PostgreSQL and R-tree-based index, though our method is applicable to any other multidimensional indexing method. With this combination we outperformed commercial platforms (Oracle, SQL Server) by an order of magnitude in the number of accesses to index. As a by-product, a framework for easy implementation of external indexing methods into PostgreSQL was designed.

1 Introduction

Together with growing database sizes it becomes necessary to use novel indexing structures for faster query processing, especially in the case when fetching just a small portion of the stored data. Originally, indexes were used for searching the data according to just one attribute, for example, the well-known B-tree [3] (B^+-tree [4], respectively). In it's original form, the B-tree was not designed to index multiple attributes, and so it was not able to answer queries concerning multiple attributes. However, such queries are widely used in database applications. For example, let's have the following *window query* (conjunctive range query):

```
SELECT * FROM Products WHERE
BrandId       BETWEEN (13 AND 14) AND
ProductTypeID BETWEEN (13 AND 24) AND
PeriodID      BETWEEN  (3 AND 11)
```

The result set of such a query will be (probably) very small, which should be a favorable situation for an index. When not having a single index for multiple attributes, we could use three indexes. Then, we get three result sets being intersected to get the final result (Fig. 1a). In practical applications, the user wants to fetch just a small number of records from the database (on average), regardless of the number of attributes involved in the *WHERE* clause. Hence, with growing number of attributes, the size of the result should remain (more or less) constant. To achieve this, the sizes of individual result sets should grow because the joining operation in window query is intersection (always reducing the size). If the sizes of the individual result sets remain the same, the size of the intersection decreases (Fig. 1b). Thus, to preserve the size of the overall result set, the user needs to increase the sizes of the partial result sets by specifying wider ranges in the window query. The partial sets/ranges must grow even exponentially (with respect to the number of attributes) to preserve the final result size, which is a consequence of the so-called *curse of dimensionality* [6][7]. Finally, with growing size of the partial results, the advantage of using multiple single-attribute indexes diminishes, and in such a case it is more efficient to perform a sequential scan.

As we can observe, a solution based on multiple single-attribute (one-dimensional) indexes is not the right way. A possibility is to extend B-tree to index multiple keys in its nodes. This is the standard way of indexing multiple attributes in existing commercial database systems (more closely described in section 2.1). The B-tree with compound keys is more effective than using multiple one-dimensional indexes, but still is not very effective for indexing multi-attribute

International Conference on Management of Data
COMAD 2008, Mumbai, India, December 17–19, 2008

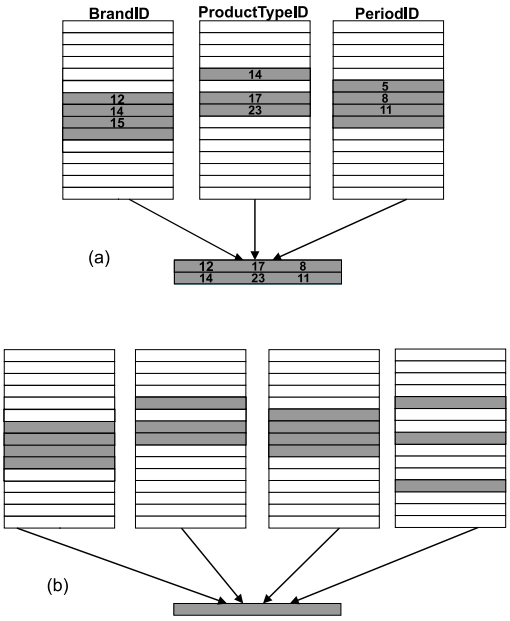

Figure 1: Querying over 3 (a) and 4 (b) attributes

(multidimensional) data.

The native multidimensional indexing structures do exist, however, they have been designed for specific types of applications (considering just two or three dimensions)[1]. Here, the indexing was restricted to a special "geometry" type (i.e., again a single attribute). For these purposes, the R-tree [11] was proposed (KD-tree was proposed sooner, but it divides space much less effectively). Although designed for 2D/3D space, the R-tree is extensible to an arbitrary number of dimensions without a redesign of the original R-tree model.

As mentioned earlier, for multi-attribute indexing the current commercial database systems exploit structures based on B$^+$-trees. Furthermore, Oracle[2] and DB2[3] have extensions supporting spatial indexing with R-trees, but these are appointed to GIS/CAD applications. The possibilities of multi-attribute/spatial indexing types in platforms that appear in this paper can be seen in Tab. 1.

Database	*Multi-attribute indexes*	*Spatial extension*
Oracle 9.i	B*-tree[4]	YES (R-tree)
MS SQL Server 2000	B^{++}-tree[4] [5]	NO
PostgreSQL 8.1	B$^+$-tree[4]	YES (R-tree)
Transbase 6.4.1	UB-tree (multidim.)	n/a

Table 1: Multidimensional indexing in current DB systems

[1]In the first place geographic information systems (GIS) and computer-aided design systems (CAD).

[2]Oracle Spatial Data Cartridge

[3]DB2 Spatial Extender

[4]Compound keys.

[5]B$^+$-tree with connected nodes at inner levels.

One could ask why should be spatial extensions applicable to simple SELECTs like the one mentioned earlier? Well, records of the example vendor system can be visualized as points in three-dimensional space (Fig. 2a). Spatial indexes (like R-trees) allow to index arbitrary n-dimensional objects by bounding them into n-dimensional blocks and then indexing these blocks. A window query can be similarly visualized in the space as a block defined by ranges on the attributes involved in the *WHERE* clause (Fig. 2b).

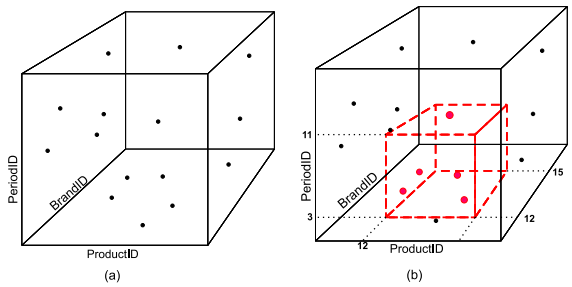

Figure 2: Multidimensional visualization of data (a) and window query (b)

With this approach we can natively index combinations of simple attributes with methods originally designed for usage in spatial databases. In such a way we could decrease the number of index accesses, while the users can use standard SQL queries – they are not forced to use any extending modules/language (otherwise needed when using a spatial extension). Note that not only numeric data types can be indexed by multidimensional indexes, there just must exist a linear ordering on the data type (i.e., date, time, string). It should be noticed that the only present-day commercial system utilizing native multidimensional indexing for combinations of simple attributes is the Transbase [12], employing the UB-tree (described in section 2.3).

2 Indexing Methods

In this paper, we consider tree-based indexes where the data is usually stored in (or referenced from) leaf nodes and are traced by traversing the inner nodes. The indexing structures differ in rules for traversing the inner nodes and in semantics of the leaf nodes. In the next sections we briefly describe indexing structures relevant to this paper.

2.1 B$^+$-tree with Compound Keys

The B$^+$-tree [4] is an extension of the (redundant) B-tree [3] (height-balanced structure with at least 50% node utilization) where leaves are linked together to enable sequential access in a given order of keys.

The original B$^+$-tree does not provide the possibility to index multidimensional data (keys). The simplest way how to support them is to interpret multiple

keys as a single compound (chained) value, concatenating keys of individual dimensions. When comparing query compound key against an indexed compound key, the key components are compared in lexicographical order. As far as we know, most of the database platforms use such a solution.

The asymmetry in the order is the biggest problem of compound keys. The main key component (attribute) is the first one and the records in the index are sorted according to it. Only if there are duplicate values in the first component, the second key component is used, and so on[4]. This may cause traversing many branches of the tree when processing a window query (if the most restrictive attribute is not the main key component).

2.2 R-tree

The R-tree [11] can be understood as a direct multidimensional extension of the B$^+$-tree. It introduces MBR (Minimal Bounding Rectangle) as a n-dimensional rectangular region covering the underlying spatial objects. Each leaf contains data objects and each inner node contains MBRs of its children MBRs (or data objects), see Fig. 3. In the searching phase, a node is entered if it has a non-empty intersection with the query rectangle/block.

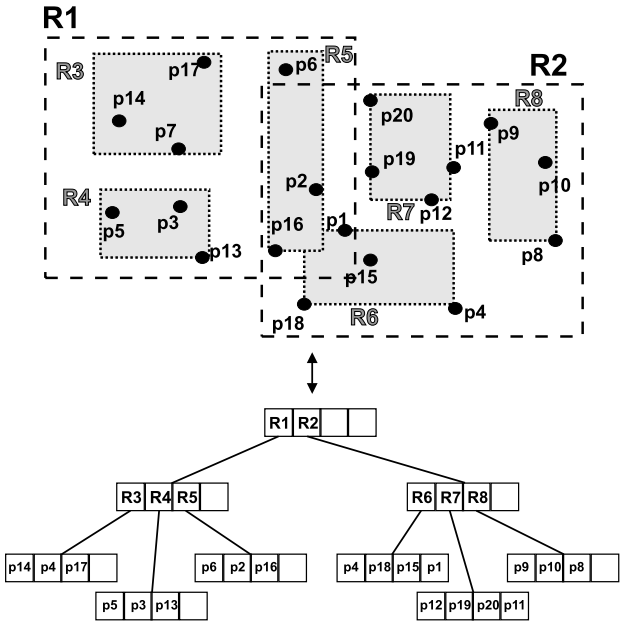

Figure 3: Space and tree representations of R-tree regions.

There also exist several well-known extensions to the R-tree such as R$^+$-tree [17] which has overlap-free MBRs at the same tree level; this minimizes the number of traversed branches when searching. Another well-known extension is the R*-tree [5] which takes margins of the MBRs into consideration.

[4]For this reason, it is recommended to use attributes with a small domain as the first components of the compound keys (many branches can be filtered out).

2.3 UB-tree

When the number of dimensions gets higher (>5–10), the R-tree and its alternatives become inefficient. In high dimensions, the overlap among MBRs rapidly grows, so many subtrees have to be traversed. Again, this is a consequence of the dimensionality curse.

One of the solutions is to transform points from n-dimensional into one-dimensional space and index these points with classical indexing structures. However, the transformation should preserve the original n-dimensional topology as much as possible. For example, the compound keys mentioned before represent such a transformation, but they do not preserve the topology very well. Hence, the success of such a transformation method lies in finding a suitable "curve" that fills the multidimensional space and preserves the distances among the points well. The Universal B-tree (UB-tree) [2] uses the *Z-curve* (Fig. 3a) for this purpose. Each point in the space has its *Z-address*. The Z-addresses are further clustered into *Z-regions* (Fig. 3b) whose lower and upper (one-dimensional) bounds play the role of B$^+$-tree's keys. An advantage of UB-tree is its linear ordering of Z-regions which do not overlap.

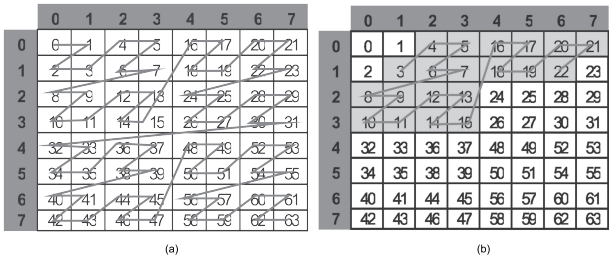

Figure 4: Z-curve (a) and Z-region (b).

3 Indexing in PostgreSQL

The PostgreSQL [16][10] is an object-relational database system which pays special attention to extensibility. As a part of the extensibility mechanism, users can implement their own access methods.

Each database platform that wants to allow to implement user-defined access methods has to provide interface for the methods to allow:

1. a communication with the optimizer concerning the estimated time required by the indexing method, so that it can decide about its possible usage

2. a synchronization with the content of the indexed relation (table), so that the DB gives an opportunity to reflect modifications in the relation

3. a searching in the relation. The functions of the interface should allow:

   - search initialization
   - searching for the next item in an ongoing search
   - finishing a search
   - finalizing a search (cleaning)

The purpose of the outlined behavior is to abstract the programmer from the physical details of the given platform. Therefore, the list does not contain fetching the retrieved records from the disk etc. Such low-level procedures are hidden in the system and the implemented access method works just with records' IDs.

3.1 Heap and Index Relations

The PostgreSQL makes use of two types of relations – *heap* relations (*HRs*) and *index* relations (*IRs*). The HRs maintain all the user relations (together with the system catalog) which are stored in an undefined order. The heap itself consists of blocks of constant size and each of the blocks contains zero or more records. On the other hand, IRs contain pairs <`key; value`> that allow fast retrieval of a record out of the heap following its key.

3.1.1 Tuple Identifiers

The PostgreSQL uses *tuple identifiers* (*TIDs*) (stored in IRs) as unique identifiers for records in each relation. The TID is a pair <`block; offset`> where blocks are defined as integer values starting from 0 for each relation. The blocks are of constant size, while the offsets allow to address variable sized records within a block. If a record is updated, a new version of the record is created and new TID is allocated. The references chaining individual versions of a record are stored in an additional structure attached to TID and thus older version can be tracked. The chaining of record versions is visualized in Fig. 5.

3.2 Implementation of an Indexing Method

Building of a user-defined access method in PostgreSQL requires:

1. to implement a set of functions defining the interface that PostgreSQL core uses for communication with the access method. These functions specify behavior of the method.

2. to register the functions (located in libraries) from the previous step.

3. to define an index by connecting the registered functions with a newly created index.

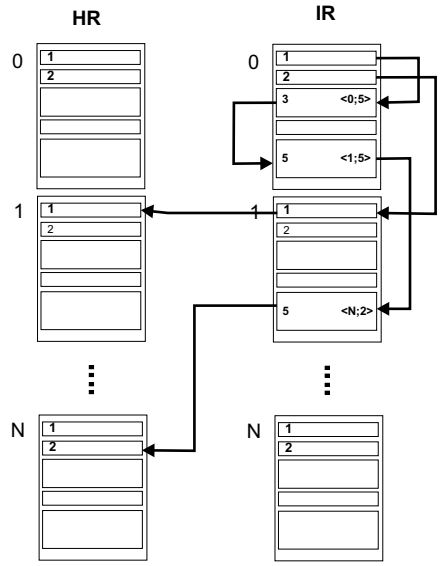

Figure 5: Chained IRs referring to HRs.

4. to establish a class of operators defining SQL operators and types supported by the registered access method.

5. to use the index over columns of a table containing types supported by the operators' classes.

Except for the first, a description in the PostgreSQL documentation can be found for each step. Thus, the only step worth further discussion is the first one. Here PostgreSQL requires the following functions to be implemented:

- *index_build*
  - Usually creates a structure for the index (e.g., tree).
- *index_insert*
  - Inserts a record into the index.
- *index_beginscan*
  - Starts a new search.
- *index_gettuple*
  - Gets a record meeting searching conditions.
- *index_getmulti*
  - Gets a set of records of specified size.
- *index_endscan*
  - Finishes a search.
- *index_markpos*
  - Marks the actual position in a scan.

- *index_restrpos*
  - Returns to the marked position.
- *index_rescan*
  - Repeats a search with the same structure of keys but possibly different values (can be used in joins).
- *index_bulkdelete*
  - Removes a set of records from the index.
- *index_costestimate*
  - Estimates cost of the search.

We just mentioned methods that form the core of an access method. They can be implemented in virtually every language supported by PostgreSQL, but in practice, they are written in C (as the PostgreSQL core itself). The crucial function is *index_gettuple* which is called repeatedly as long as there are records to be fetched from the index.

3.3 Searching for a Record

When the access method is implemented, the process of searching and fetching a record from the DB storage goes as follows:

a User enters an SQL command (Fig. 6a).

b PostgreSQL searches for all indexes created upon the table (Fig. 6b).

c For each index, time needed for finding the result is estimated - *index_costestimate*.

d *WHERE* clause is parsed and a set of records representing the restriction is returned - *scan_key* structure (Fig. 6c).

e The optimal index (according to c) is used for the search.

f Function of the access method returning desired set of records is called repeatedly (Fig. 6d) to get the result (Fig. 6e).

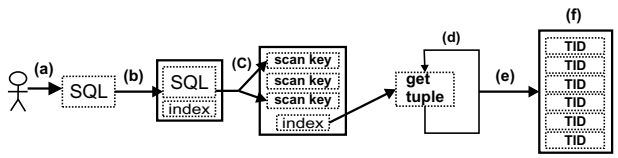

Figure 6: Process of searching and fetching a record.

3.4 External Indexing

Each access method provided by PostgreSQL implements the required methods and uses HRs as its repository to store IRs for the indexed relations. Thus, it passes IR's TIDs to the core and the core picks up the relevant records out of the heap. We have a problem here – when somebody wants to use his/her external (3^{rd} party) framework for indexing in PostgreSQL, it is necessary to store IRs in this framework. This causes an overhead since, in the optimal case, the framework should refer directly to the HRs (Fig. 7). In the current version, our framework passes IR's TID to the core which fetches HR's TID which, in turn, fetches the resulting record.

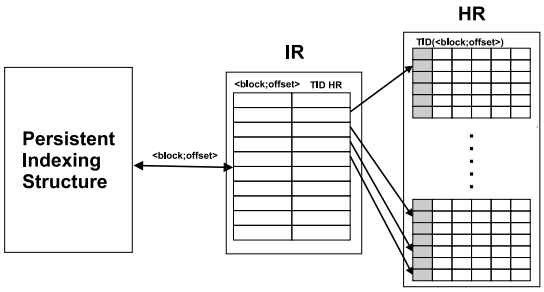

Figure 7: Storing IR in an external indexing structure.

The PostgreSQL's interface to access methods allows user not to bother with physical details (fetching records, etc.) but it is still necessary to work with IRs and know (relatively thoroughly) the implementation details of PostgreSQL. We have developed a framework that makes implementation of user-defined access methods even easier. Actually, it is a framework on the database side which communicates with the external indexing framework on one side, and *index_** methods on the other. The framework requires the following functions to be implemented:

- void *FW_CreateStructure*(Relation index_relation);
  - On the basis of the *index_relation*, the index's structure is set (types and number of attributes).
- void **FW_PrepareInsert*(Relation index_relation);
  - In this function, the index is initialized (opening a persistent storage, setting global variables, ...).
- void *FW_InsertTuple*(void *fw_data, Relation index_relation, IndexTuple index_tuple, BlockNumber block_number, OffsetNumber offset);
  - Inserts record into the index. The input parameter *fw_data* is reference to the structure containing informations created in *FW_PrepareInsert*.
- void *FW_FinishInsert*(void *fw_data);

– Cleans memory after finishing a scan. *fw_data* represents reference to global data (which should be the object of the cleaning).

- void *FW_InitSearch*(IndexScanDesc scan, ScanDirection dir);

  – If the external method is built in the way that it fetches all the records in one pass and then returns it one by one, it is accomplished here. The returned records are stored in the *scan* parameter.

- bool *FW_GetNextTID*(IndexScanDesc scan, ScanDirection dir, BlockNumber *block_number, OffsetNumber *offset);

  – Fetching the next record (see the previous function).

- void *FW_DeleteTuple*(BlockNumber block_number, OffsetNumber offset);

  – Deletes a record from the external storage.

If the external indexing framework is written in C, it can be compiled with the *FW_** functions (PostgreSQL's framework). Otherwise, the framework has to be compiled into a library and wrapped by C functions.

We used *ATOM* [1] as the external framework which is written in C++ and hence we needed to wrap it in order to communicate with *FW_** functions, as can be seen in Fig. 8.

3.4.1 Advantages of External Indexing

There are several important reasons why to use the developed framework:

- *Minimization of necessary PostgreSQL mastering* – it is sufficient to be familiar with the input structures that enter the framework's functions.

- *Implementation time* – implementing an existing indexing method into PostgreSQL through the connection bridge takes insignificant time. Most of the time is spent on understanding *index_** functions and working with memory in PostgreSQL.

- *Implementation of the indexing method is database independent* and it is possible to use it with other database platforms as well.

3.4.2 Implementation Challenges

In the previous section we described advantages of using an external index for implementing an access method. Although, such a solution has also disadvantages:

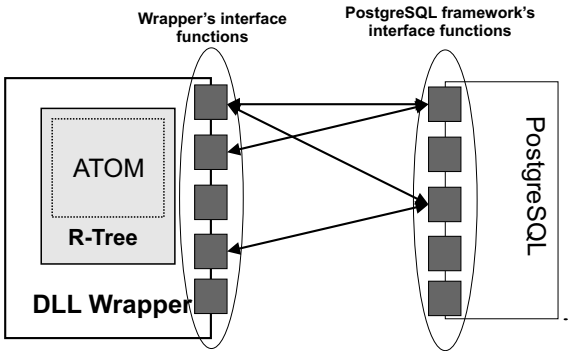

Figure 8: Communication between indexing and PostgreSQL framework.

- *Duplicated reading of pages.* This problem is caused by the need of storing IRs not only in the external indexing method only, but also in PostgreSQL (links to HRs). The problem is especially unpleasant when dealing with physical pages (which have to be read from an external physical storage). On the other hand, records in the framework are much smaller than those in PostgreSQL and hence more of them can be stored in a cache to avoid reading them from disk.

- *Global data.* The external method might need to store global data for a time spanning over a *scan's* lifetime. Unfortunately, PostgreSQL does not offer the possibility to store global data. Hence, the solution is to use temporary files or specific abilities of the operating system.

- *Impossibility of recognizing end of existence of an index.* In our point of view, this issue has arisen due to the inconsistence when implementing user defined indexes in PostgreSQL. When index is created, user defined function is called but there does not exist such a possibility when index is destroyed. Hence, an external indexing method that uses some kind of external storage can not recognize when to delete its repository for this storage. To the best of our knowledge, the only solution is to recycle the storage when the same name is reused in future (which does not solve the problem how to handle dropped and never recreated indexes).

3.5 Native R-tree Indexing

Using the framework described above we have implemented the R-tree as a native multidimensional indexing method into PostgreSQL. This way one could implement also other multidimensional indexing methods, like the UB-tree. One could even implement indexing methods suitable for different purposes, e.g., the M-tree [8] for similarity search, however, for such an extension the implementation using the framework

must additionally define new SQL operators (similarity predicates), which is out of scope of the framework and of our paper, but is supported natively by PostgreSQL.

4 Experimental Results

We performed experiments to prove the benefits of native multidimensional indexing when compared with the conventional compound-key indexing. The testing platform was a Pentium4-3GHz, 1GB RAM, 80GB HD - 5400 rpm. We used PostgreSQL 8.1 (denoted PG in the figures), Oracle 9i Release2, Microsoft SQL Server 2000 and Transbase 6.4.1 as the database platforms in the competition. Each of the platforms used 8KB disk pages and so did the ATOM framework. To be able to compare all the platforms correctly, we did not use any optimization techniques such as clustered indexes in SQL Server or index-organized tables in Oracle.

As mentioned earlier, Microsoft SQL Server [13] uses B^{++}-tree[5] which is B^+-tree with linked lists connecting also inner nodes at each level, Oracle [15] uses B^*-tree which is B^+-Tree with 2/3 node utilization, and PostgreSQL uses the standard B^+-tree. We have also included the native multidimensional indexing in terms of R-tree-enhanced PostgreSQL (our approach described earlier) and Transbase [12] with its UB-tree.

4.1 The Testbed

We used two synthetic and one real dataset. The first synthetic dataset (denoted Uniform) was based on clusters of uniformly distributed (up to 15-dimensional) points (abstracting from table records). The number of generated clusters was related to the dataset size (the greater dataset, the more clusters). In the second synthetic dataset (denoted Gauss), the data (up to 3-dimensional points) followed the Gauss (normal) distribution without clusters. Finally, in order to test our method also with real-world data, we used a table from the DBLP database [9] which contained 435,373 records (denoted DBLP). From the DBLP table we chose attributes *author*, *type of publication*, *year of publication* and *number of pages* for indexing (i.e., four dimensions).

The Uniform dataset was chosen to see how window query selectivity, dimension and dataset size impacts the number of accesses to the index (logical nodes fetched from the disk). For each of the tests, 100 queries were performed and the results averaged.

The Gauss dataset was tested differently. One of the space's diagonal was divided into 11 parts and each of the parts formed a diagonal of an n-dimensional cube. Inside each of the cube 100 points were randomly picked up and selected as centers of windows queries (whose ranges corresponded to 1/11 of the space's diagonal). 100 window queries originated from each cube were used to average the results.

We studied the number of accesses to index and time in seconds to answer the query. Note that the number of accesses is a logical unit, since it is not dependent on the implementation details of a particular database platform.

[5]This is not the official name of the structure, just our label.

4.2 Index Size

We measured the index size for platforms where we were able to get this information. The Tab. 2 shows that space needed by B-tree is virtually the same as the space needed by R-tree. In both cases, the size grows linearly with the volume of data stored. The double size of the R-tree index incorporated into PostgreSQL is the consequence of the need to store the IRs twice (as described in section 3.4). To show the difference, the first column (ATOM(R)) refers to the pure R-tree size without the other PG overhead.

Dataset size	ATOM(R)	PG+ ATOM(R)	PG(B^+)	MSSQL (B^{++})
10^4	207	412	246	248
$5 * 10^4$	1 053	2 061	1 139	1 112
10^5	2 113	4 129	2 261	2 280
$2 * 10^5$	4 085	8 116	4 514	4 440
$5 * 10^5$	10 261	20 329	11 256	11 024
10^6	20 695	40 831	22 487	21 936
$2 * 10^6$	41 595	81 859	44 950	43 752
$5 * 10^6$	111 970	212 617	112 334	109 200

Table 2: Index size - dimension 2 (in KB)

4.3 Synthetic Data

The first set of experiments concerned the impact of the dataset size on the number of accesses to the index. As Fig. 9 shows, with growing size of the dataset the number of accesses grows linearly. In this one-dimensional case, the R-tree outperforms the other methods, while the relative difference stays approximately constant with growing dataset size. With higher query selectivity (in % of dataset size), the difference is even more obvious.

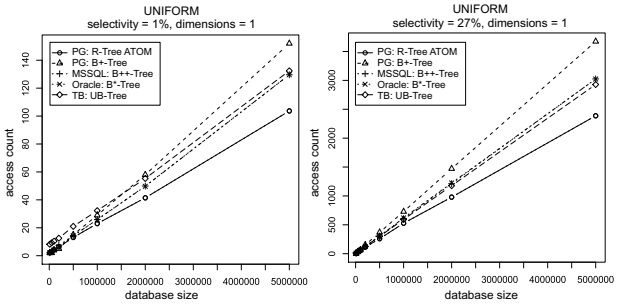

Figure 9: Impact of dataset size on access count (different query selectivities)

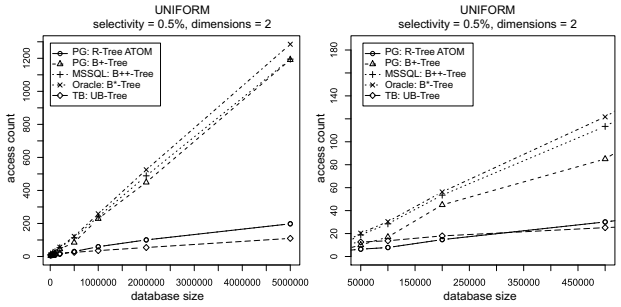

Figure 10: Impact of dataset size on access count (zoomed on the right)

If we increase the dimension to 2, it is apparent that the R-tree still outperforms compound keys methods (Fig. 10) but, on the other hand, R-tree is outperformed by UB-tree when it comes to larger databases.

If we further increase the dimension to 3, we can see that the difference between UB-tree and R-tree remains very similar but standard solutions begin to lag noticeably (Fig. 11).

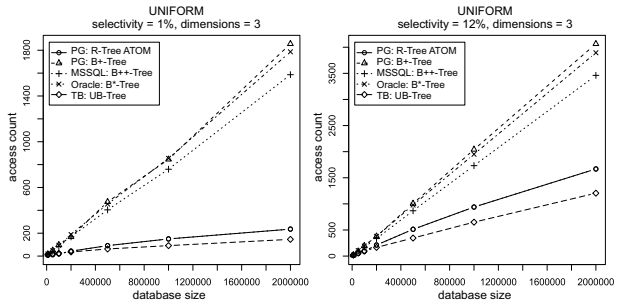

Figure 11: Impact of dataset size on access count with growing selectivity

It seems that with growing dimension the difference between standard solutions and R-tree/UB-tree also grows. In the next experiments we focused on the growing dimensionality, see Fig. 12. It turns up that R-tree performs efficiently just up to 8 dimensions. Then the overall volume of MBR overlaps grows rapidly, and the efficiency of the index deteriorates. Moreover, this trend is even more considerable in larger dataset.

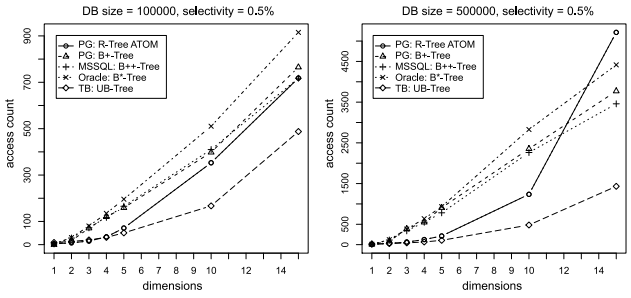

Figure 12: Impact of dimensionality on access count with growing dataset size

The next experiments considered the query selectivity. In Fig. 13, we can notice that R-tree's and UB-tree's number of accesses grows linearly with the increasing selectivity (independently on the dimension). On the other hand, standard solutions' growth shows logarithmic trend in higher dimensions. The reason for this trend is the fact that with growing number of fetched records, the disadvantage of the standard methods caused by the asymmetry of the first key decreases (many branches will be traversed, anyway).

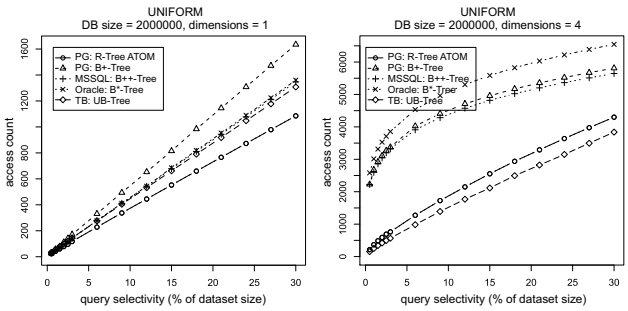

Figure 13: Impact of selectivity on the number of accesses (growing dimension)

When looking at Gauss, it is interesting how the methods behave when the query window "moves" to the center of the space (i.e., when the selectivity grows). The horizontal axis in Fig. 14 represents the diagonal of the space, hence in the center the selectivity is highest. The results confirm the previously observed measurements. In a single dimension the R-tree is the best method, while in higher dimensions and higher selectivities (on the sides of the axis) the winner is UB-tree.

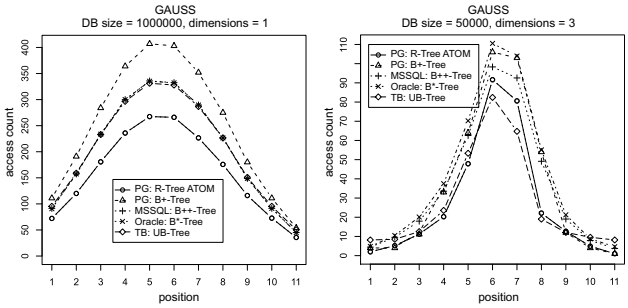

Figure 14: Moving window query

4.3.1 Realtimes

In further experiments we measured the realtimes. Here we should point out the fact that index accesses comprise disk and memory access.

Hence, the time spent while searching is noticeably influenced by caching of nodes which is dependent on the individual database platform, the size of cache (which often cannot be adjusted to narrow the exper-

Dim	*Query selectivity*	*PG(R)*	*PG(B^+)*	*MSSQL(B^{++})*	*Oracle(B^*)*	*TB(UB)*
2	2 %	3.62	71.88	51.59	52.87	8
	3 %	3.62	63.13	43.8	43.99	7.97
3	2 %	4.31	94.82	71.42	71.04	10.25
	3 %	3.95	109.45	78.71	78.88	10.05
4	2 %	9.87	86.31	70.83	71.44	12.43
	3 %	10.73	92.08	76.02	76.1	13.15

Table 3.: Access count – DBLP

Dim	*Query selectivity*	*ATOM(R)*	*PG(R)*	*PG(B^+)*	*MSSQL(B^{++})*	*Oracle(B^*)*	*TB(UB)*
2	2 %	3.13	5.31	5.78	0.46	0.93	3.58
	3 %	3.28	5.94	4.84	0.34	0.73	3.42
3	2 %	2.97	4.69	6.25	0.87	1.73	3.88
	3 %	2.66	5	7.19	1.06	1.76	2.82
4	2 %	3.91	6.88	6.72	1.6	1.69	3.58
	3 %	5	7.2	7.66	1.3	1.82	2.19

Table 4.: Realtime (s) – DBLP

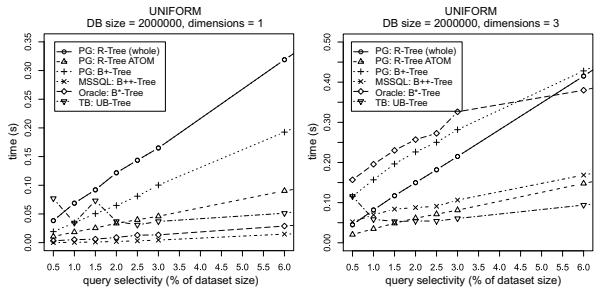

Figure 15: Impact of query selectivity on realtime

imental conditions), etc. The results can be seen in Fig. 15. In these experiments R-tree results are divided into two parts – the time needed by ATOM to find records and the overall time (with PostgreSQL overhead). We can see that in one dimension, commercial solutions are faster, however they have to access higher number of nodes (we can see the impact of a good implementation and optimizing). On the other hand, if the dimension is high enough the benefits of multidimensional indexing dominate (here only three dimensions sufficed) – both UB-tree and R-tree outperformed the other methods. Unfortunately, the R-tree-enhanced PostgreSQL is fairly slow due to the inherent overhead of PostgreSQL (access to HRs and IRs, see section 3.1).

The B^+-tree of PostgreSQL shows poor results, even though it is comparable to other solutions in the number of index accesses (Fig. 12). Anyway, under the scope of PostgreSQL the native multidimensional indexing extension brings a clear improvement for higher dimensions.

4.4 Real Data

In the last experiments we indexed the DBLP dataset where we measured both number of accesses and realtime. When looking at the results representing the number of accesses (Tab. 3), we can see that with the DBLP dataset the benefits of the multidimensional methods are even more noticeable than in the case of synthetic datasets. In some cases, the R-tree is even 20× more efficient than Oracle's or SQL Server's methods.

However, when it comes to realtimes, the same problem as in the case of synthetic datasets arises. The commercial solutions are able to beat the competitors because of their good implementation (Tab. 4).

5 Utilization of the Idea

The presented idea is quite simple, but still very powerful. It allows to store various multidimensional data types and query them effectively with standard SQL syntax. Let's point out few areas where such a type of index can find use:

- *Symmetric multidimensional queries.* Example is the vendor system where users need to query according to multiple attributes with similar selectivity. As stated at the beginning of the paper, in such a case only multiple B-trees would do the work in current commercial database systems. Hence the data can be accessed more easily.
- *Object features - multimedia databases.* Features of objects such as pictures usually contain many dimensions (color distribution, shapes, ...). For these features we can use multidimensional indexes only, but then the objects have to be bundled in some kind of BLOB object. That is not necessary when using native multidimensional indexing presented in this paper.
- *Mapped non-relational databases.* Various complex database models exploit the relational model

in order to efficiently implement access to their data. For example, besides native engines, XML databases are often transformed into plain tables [14]. XPath or XQuery queries are then expressed using the standard SQL, which leads to intensive multi-attribute queries.

6 Conclusion

In this paper we applied the R-tree into PostgreSQL for native multidimensional indexing of relational databases. We extensively tested our approach against today's most widely used database platforms and showed that our solution outperforms these solutions in terms of platform-independent efficiency measures. However, to beat the competitor also in realtimes, the native multidimensional indexing must be optimized and incorporated directly into the DBMS core in the future. Under the scope of PostgreSQL, however, the R-tree-based multidimensional indexing outperforms the standard B^+-tree indexing in both, the access costs as well as realtimes. As a by-product, we developed a framework that allows other interested researchers to implement their own indexing methods and test them within a real database environment (PostgreSQL) with just a minimal knowledge of the platform.

Acknowledgments. This research has been supported by grant GAUK 57907 provided by the Grant Agency of Charles University and by grant GAČR 201/06/0756 by the Czech Science Foundation.

References

[1] ARG. Amphora Research Group. Amphora Tree Object Model (ATOM) Book, `http://arg.vsb.cz`.

[2] R. Bayer. The universal b-tree for multidimensional indexing: general concepts. In *WWCA '97: Proceedings of the International Conference on Worldwide Computing and Its Applications*, pages 198–209, London, UK, 1997. Springer-Verlag.

[3] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141. ACM, 1970.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331. ACM Press, 1990.

[6] C. Böhm, S. Berchtold, and D. Keim. Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[7] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB'97*, pages 426–435, 1997.

[9] DBLP. Digital Bibliography & Library Project, `http://dblp.uni-trier.de/`.

[10] K. Douglas and S. S. P. Douglas. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases*. Developer's library. pub-SAMS, 2003.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[12] V. Markl, F. Ramsak, R. Pieringer, R. Fenk, K. Elhardt, and R. Bayer. The transbase hypercube rdbms: Multidimensional indexing of relational tables. In *ICDE Demo Sessions*, pages 4–6, 2001.

[13] Microsoft. SQL Server 2000 Books Online, `http://www.microsoft.com/sql/prodinfo/previousversions/books.mspx`.

[14] I. Mlynkova and J. Pokorny. Usermap : an adaptive enhancing of user-driven xml-to-relational mapping strategies. In *Nineteenth Australasian Database Conference (ADC 2008)*, volume 75 of *CRPIT*, pages 165–174, Wollongong, NSW, Australia, 2008. ACS.

[15] Oracle. 9i Release 2 Documentation, `http://www.oracle.com/technology/documentation/oracle9i.html`.

[16] PostgreSQL. 8.1 documentation, `http://www.postgresql.org/docs/8.1/interactive/`.

[17] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multidimensional objects. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 507–518. Morgan Kaufmann, 1987.