Int. Conf. on Data Warehousing and Knowledge Discovery DaWaK ’99

Efficient Bulk Loading of Large High-Dimensional Indexes

Christian Böhm and Hans-Peter Kriegel

University of Munich, Oettingenstr. 67, D-80538 Munich, Germany
{boehm,kriegel}@informatik.uni-muenchen.de

Abstract. Efficient index construction in multidimensional data spaces is important for many knowledge discovery algorithms, because construction times typically must be amortized by performance gains in query processing. In this paper, we propose a generic bulk loading method which allows the application of user-defined split strategies in the index construction. This approach allows the adaptation of the index properties to the requirements of a specific knowledge discovery algorithm. As our algorithm takes into account that large data sets do not fit in main memory, our algorithm is based on external sorting. Decisions of the split strategy can be made according to a sample of the data set which is selected automatically. The sort algorithm is a variant of the well-known Quicksort algorithm, enhanced to work on secondary storage. The index construction has a runtime complexity of O(*n* log *n*). We show both analytically and experimentally that the algorithm outperforms traditional index construction methods by large factors.

1. Introduction

Efficient index construction in multidimensional data spaces is important for many knowledge discovery tasks. Many algorithms for knowledge discovery [JD 88, KR 90, NH 94, EKSX 96, BBBK 99], especially clustering algorithms, rely on efficient processing of similarity queries. In such a setting, multidimensional indexes are often created in a preprocessing step to knowledge discovery. If the index is not needed for general purpose query processing, it is not permanently maintained, but discarded after the KDD algorithm is completed. Therefore, the time spent in the index construction must be amortized by runtime improvements during knowledge discovery. Usually, indexes are constructed using repeated insert operations. This *‘dynamic index construction’*, however, causes a serious performance degeneration. We show later in this paper that in a typical setting, every insert operation leads to at least one access to a data page of the index. Therefore, there is an increasing interest in fast bulk-loading operations for multidimensional index structures which cause substantially fewer page accesses for the index construction.

A second problem is that indexes must be carefully optimized in order to achieve a satisfactory performance (cf. [Böh 98, BK 99, BBJ+ 99]). The optimization objectives [BBKK 97] depend on the properties of the data set (dimension, distribution, number of objects, etc.) and on the types of queries which are performed by the KDD algorithm (range queries [EKSX 96], nearest neighbor queries [KR 90, NH 94], similarity joins [BBBK 99], etc.). On the other hand, we may draw some advantage from the fact that we do not only know a single data item at each point of time (as in the dynamic index construction) but a large amount of data items. It is a common knowledge that a higher fanout and storage utilization of the index pages can be achieved by applying bulk-load operations. A higher fanout yields a better search performance. Knowing all data *a priori* allows us to choose an alternative data space partitioning. As we have shown in [BBK 98a], a strategy of splitting the data space into two equally-sized portions causes, under certain circumstances, a poor search performance in contrast to an unbalanced split. Therefore, it is an important property of a bulk-loading algorithm that it allows to exchange the splitting strategy according to the requirements specific to the application.

The currently proposed bulk-loading methods either suffer from poor performance in the index construction or in the query evaluation, or are not suitable for indexes which do not fit into main memory. In contrast to previous bulk-loading methods, we present in this paper an algorithm for fast index construction on secondary storage which provides efficient query processing and is generic in the sense that the split strategy can be easily exchanged. It is based on an extension of the Quicksort algorithm which facilitates sorting on secondary storage (cf. section 3.3 and 3.4). The split strategy (section 3.2) is a user-defined function. For the split decisions, a sample of the data set is exploited which is automatically generated by the bulk-loading algorithm.

2. Related Work

Several methods for bulk-loading multidimensional index structures have been proposed. Space-filling curves provide a means to order the points such that spatial neighborhoods are maintained. In the Hilbert R-tree construction method [KF 94], the points are sorted according to their Hilbert value. The obtained sequence of points is decomposed into contiguous subsequences which are stored in the data pages. The page region, however, is not described by the interval of Hilbert values but by the minimum bounding rectangle of the points. The directory is built bottom up. The disadvantage of Hilbert R-trees is the high overlap among page regions.

VAM-Split trees [JW 96], in contrast, use a concept of hierarchical space partitioning for bulk-loading R-trees or KDB-trees. Sort algorithms are used for this purpose. This approach does not exploit *a priori* knowledge of the data set and is not adaptable.

Buffer trees [BSW 97] are a generalized technique to improve the construction performance for dynamic insert algorithms. The general idea is to collect insert operations to certain branches of the tree in buffers. These operations are propagated to the next deeper level whenever such a buffer overflows. This technique preserves the properties of the underlying index structure.

3. Our New Technique

During the bulk-load operation, the complete data set is held on secondary storage. Although only a small cache in the main memory is required, cost intensive disk operations such as random seeks are minimized. In our algorithms, we strictly separate the split strategy from the core of the construction algorithm. Therefore, we can easily replace the split strategy and thus, create an arbitrary overlap-free partition for the given storage utilization. Various criteria for the choice of direction and position of split hyperplanes can be applied. The index construction is a recursive algorithm consisting of the following subtasks:

- determining the tree topology (height, fanout of the directory nodes, etc.)
- choice of the split strategy
- external bisection of the data set according to tree topology and split strategy
- construction of the index directory.

3.1 Determination of the Tree Topology

The first step of our algorithm is to determine the topology of the tree resulting from our bulk-load operation. The height of the tree can be determined as follows [Böh 98]:

$$h = \left\lceil \log_{C_{\text{eff,dir}}}\left(\frac{n}{C_{\text{eff,data}}}\right) \right\rceil + 1$$

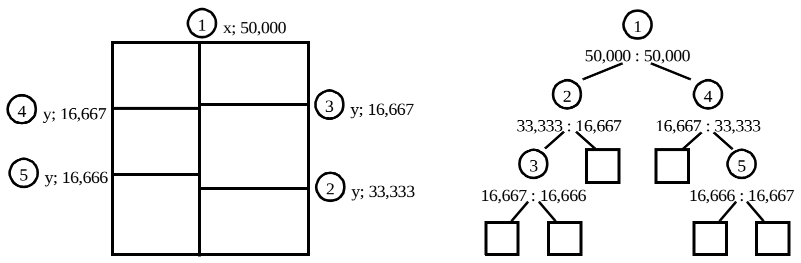

Figure 1: The Split Tree.

The fanout is given by the following formula:

$$\text{fanout}(h, n) = \min\left(\left\lceil \frac{n}{C_{\text{eff,data}} \cdot C_{\text{eff,dir}}^{h-2}} \right\rceil, C_{\text{max,dir}}\right)$$

3.2 The Split Strategy

In order to determine the split dimension, we have to consider two cases: If the data subset fits into main memory, the split dimension and the subset size can be obtained by computing selectivities or variances from the complete data subset. Otherwise, decisions are based on a sample of the subset which fits into main memory and can be loaded without causing too many random seek operations. We use a simple heuristic to sample the data subset which loads subsequent blocks from three different places in the data set.

3.3 Recursive Top-Down Partitioning

Now, we are able to define a recursive algorithm for partitioning the data set. The algorithm consists of two procedures which are nested recursively (both procedures call each other). The first procedure, *partition()*, that is called once for each directory page has the following duties:

- call the topology module to determine the fanout of the current directory page
- call the split-strategy module to determine a split tree for the current directory page
- call the second procedure, *partition_acc_to_split_tree()*

The second procedure partitions the data set according to the split dimensions and the proportions given in the split tree. However, the proportions are not regarded as fixed values. Instead, we will determine lower and upper bounds for the number of objects on each side of the split hyperplane. This will help us to improve the performance of the next step, the external bipartitioning. Let us assume that the ratio of the number of leaf nodes on each side of the current node in the split tree is $l : r$, and that we are currently dealing with N data objects. An exact split hyperplane would exploit the proportions:

$$N_{\text{left}} = N \cdot \frac{l}{l+r} \text{ and } N_{\text{right}} = N \cdot \frac{r}{l+r} = N - N_{\text{left}}.$$

Instead of using the exact values, we compute an upper bound for N_{left} such that N_{left} is not too large to be placed in l subtrees with height $h-1$ and a lower bound for N_{left} such that N_{right} is not too large for r subtrees:

$$N_{\text{max,left}} = l \cdot C_{\text{max,tree}}(h-1) \qquad N_{\text{min,left}} = N - r \cdot C_{\text{max,tree}}(h-1)$$

An overview of the algorithm is depicted in C-like pseudocode in figure 2. For the presentation of the algorithm, we assume that the data vectors are stored in an array on secondary

```
index_construction (int n)
{
        int h = (int)(log (n/Ceffdata) / log (Ceffdir) + 1) ;
        partition (0, n, h) ;
}

partition (int start, int n, int height)
{
        if (height == 0) {
                ... // write data page, propagate info to parent
                return ;
        }
        int f = fanout (height, n) ;
        SplitTree st = split_strategy (start, n, f) ;
        partition_acc_to_splittree (start, n, height, st) ;
        ... // write directory page, propagate info to parent
}

partition_acc_to_splittree (int start, int n, int height, SplitTree st)
{
        if (is_leaf (st)) {
                partition (start, n, height - 1) ;
                return ;
        }
        int mtc = max_tree_capacity (height - 1) ;
        n_maxleft = st->l_leaves * mtc ;
        n_minleft = N - st->r_leaves * mtc ;
        n_real = external_bipartition (start, n, st->splitdim,
                           n_minleft, n_maxleft) ;
        partition_acc_to_splittree (start, n_real,
                         st->leftchild, height) ;
        partition_acc_to_splittree (start + n_real, n - n_real,
                         st->rightchild, height) ;
}
```

Figure 2: Recursive Top-Down Data Set Partitioning.

storage and that the current data subset is referred to by the parameters *start* and *n*, where *n* is the number of data objects and *start* represents the address of the first object.

The procedure *index_construction(n)* determines the height of the tree and calls *partition()* which is responsible for the generation of a complete data or directory page. The function *partition()* first determines the fanout of the current page and calls *split_strategy()* to construct an adequate split tree. Then *partition_acc_to_splittree*() is called to partition the data set according to the split tree. After partitioning the data, *partition_acc_to_splittree*() calls *partition()* in order to create the next deeper index level. The height of the current subtree is decremented in this indirect recursive call. Therefore, the data set is partitioned in a top-down manner, i.e. the data set is first partitioned with respect to the highest directory level below the root node.

3.4 External Bipartitioning of the Data Set

Our bipartitioning algorithm is comparable to the well-known Quicksort algorithm [Hoa 62, Sed 78]. Bipartitioning means to split the data set or a subset into two portions according to the value of one specific dimension, the split dimension. After the bipartitioning step, the “lower” part of the data set contains values in the split dimension which are lower than a threshold value, the *split value*. The values in the “higher” part will be higher than the split value. The split value is initially unknown and is determined during the run of the bipartitioning algorithm.

Bipartitioning is closely related to sorting the data set according to the split dimension. In fact, if the data is sorted, bipartitioning of any proportion can easily be achieved by cutting the sorted data set into two subsets. However, sorting has a complexity of o(n log n), and a complete sort-order is not required for our purpose. Instead, we will present a bipartitioning algorithm with an average-case complexity of O(n). The basic idea of our algorithm is to adapt Quicksort as follows: Quicksort makes a bisection of the data according to a heuristically chosen pivot value and then recursively calls Quicksort for both subsets. Our first modification is to make only one recursive call for the subset which contains the split interval. We are able to do that because the objects in the other subsets are on the correct side of the split interval anyway and need no further sorting. The second modification is to stop the recursion if the position of the pivot value is inside the split interval. The third modification is to choose the pivot values according to the proportion rather than to reach the middle.

Our bipartitioning algorithm works on secondary storage. It is well-known that the Mergesort algorithm is better suited for external sorting than Quicksort. However, Mergesort does not facilitate our modifications leading to an O(n) complexity and was not further investigated for this reason. In our implementation, we use a sophisticated scheme reducing disk I/O and especially reducing random seek operations much more than a normal caching algorithm would be able to.

The algorithm can run in two modes, *internal* or *external*, depending on the question whether the processed data set fits into main memory or not. The internal mode is quite similar to Quicksort: The middle of three split attribute values in the database is taken as pivot value. The first object on the left side having a split attribute value larger than the pivot value is exchanged with the last element on the right side smaller than the pivot value until left and right object pointers meet at the bisection point. The algorithm stops if the bisection point is inside the goal interval. Otherwise, the algorithm continues recursively with the data subset containing the goal interval.

The external mode is more sophisticated: First, the pivot value is determined from the sample which is taken in the same way as described in section 3.2 and can often be reused. A complete internal bipartition runs on the sample data set to determine a suitable pivot value. In the following external bisection (cf. figure 3), transfers from and to the cache are always processed with a blocksize half of the cache size. Figure 3a shows the initialization of the cache from the first and last block in the disk file. Then, the data in the cache is processed by internal bisection with respect to the pivot value. If the bisection point is in the lower part of the cache (figure 3c), the right side contains more objects than fit into one block. One block, starting from the bisection point, is written back to the file and the next block is read and internally bisected again. Usually, objects remain in the lower and higher ends of the cache. These objects are used later to fill up transfer blocks completely. All remaining data is written back in the very last step into the middle of the file where additionally a fraction of a block has to be processed. Finally, we test if the bisection point of the external bisection is in the split interval. If the point is outside, another recursion is required.

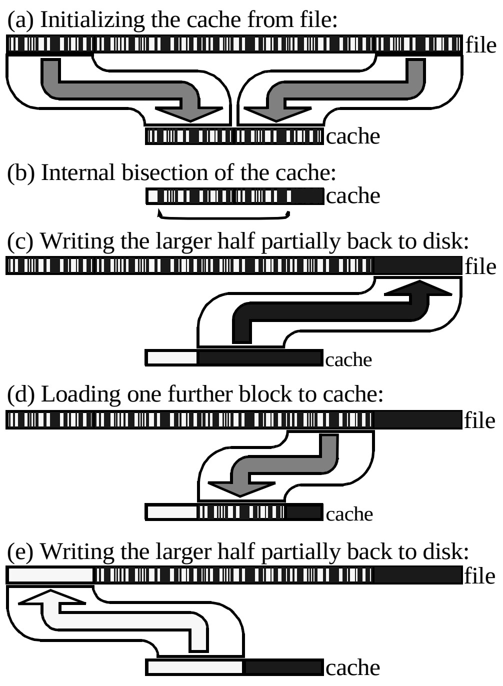
(a) Initializing the cache from file:
file
cache
(b) Internal bisection of the cache:
cache
(c) Writing the larger half partially back to disk:
file
cache
(d) Loading one further block to cache:
file
cache
(e) Writing the larger half partially back to disk:
file
cache

Figure 3: External Bisection.

3.5 Constructing the Index Directory

As data partitioning is done by a recursive algorithm, the structure of the index is represented by the recursion tree. Therefore, we are able to create a directory node after the completion of the recursive calls for the child nodes. These recursive calls return the bounding boxes and the corresponding secondary storage addresses to the caller, where the information is collected. There, the directory node is written, the bounding boxes are combined to a single bounding box comprising of all boxes of child nodes, and the result is again propagated to the next higher level. A depth-first post-order sequentialization of the index is written to the disk.

3.6 Analytical Evaluation of the Construction Algorithm

In this section, we will show that our bottom-up construction algorithm has an average case time complexity of O(n log n). Moreover, we will consider disk accesses in a more exact way, and thus provide an analytically derived improvement factor over the dynamic index construction. For the file I/O, we determine two parameters: The number of random seek operations and the amount of data read or written from or to the disk. Unless no further caching is performed (which is true for our application, but cannot be guaranteed for the operating system) and provided that seeks are uniformly distributed variables, the I/O processing time can be determined as

$$t_{\text{i/o}} = t_{\text{seek}} \cdot \text{seek\_ops} + t_{\text{transfer}} \cdot \text{amount}\,.$$

In the following, we denote by the cache capacity C_{cache} the number of objects fitting into the cache:

$$C_{\text{cache}} = \frac{\text{cachesize}}{\text{sizeof (object)}}$$

Lemma 1. Complexity of *bisection*

The bisection algorithm has the complexity O(n).

Proof (Lemma 1)

We assume that the pivot element is randomly chosen from the data set. After the first run of the algorithm, the pivot element is located with uniform probability at one of the n positions in the file. Therefore, the next run of the algorithm will have the length k with a probability $1/n$ for each $1 < k < n$. Thus, the cost function $C(n)$ encompasses the cost for the algorithm, $n + 1$ comparison operations plus a probability weighted sum of the cost for processing the algorithm with length $k - 1$, $C(k)$. We obtain the following recursive equation:

$$C(n) = n + 1 + \sum_{k=1}^{n} \frac{C(k-1)}{n}$$

which can be solved by multiplying with n and subtracting the same equation for $n - 1$. This can be simplified to $C(n) = 2 + C(n-1)$, and, $C(n) = 2 \cdot n = O(n)$.

❑

Lemma 2. Cost Bounds of Recursion

(1) The amount of data read or written during one recursion of our technique does not exceed four times the file-size.

(2) The number of seek operations required is bounded by

$$\text{seek\_ops}(n) \le \frac{8 \cdot n}{C_{\text{cache}}} + 2 \cdot \log_2(n)$$

Proof (Lemma 2)

(1) follows directly from Lemma 1 because every compared element has to be transferred at most once from disk to main memory and at most once back to disk.

(2) In each run of the external bisection algorithm, file I/O is processed with a blocksize of cachesize/2. The number of blocks read in each run is therefore

$$\text{blocks\_read}_{\text{bisection}}(n) = \frac{n}{C_{\text{cache}}/2} + 1$$

because one extra read is required in the final step. The number of write operations is the same and thus

$$\text{seek\_ops}(n) = 2 \cdot \sum_{i=0}^{r_{\text{interval}}} \text{blocks\_read}_{\text{run}}(i) \le \frac{8 \cdot n}{C_{\text{cache}}} + 2 \cdot \log_2(n).$$

❑

Lemma 3. Average Case Complexity of Our Technique

Our technique has an average case complexity of O(n log n) unless the split strategy has a complexity worse than O(n).

Proof (Lemma 3)

For each level of the tree, the complete data set has to be bisectioned as often as the height of the split tree indicates. As the height of the split tree is determined by the directory page capacity, there are at most

$$h(n) \cdot C_{\text{max,dir}} = O(\log n)$$

bisection runs necessary.Therefore, our technique has the complexity O(n log n).

❑

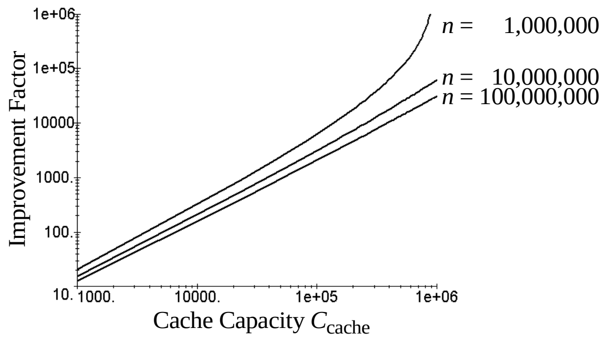

Figure 4: Improvement Factor for the Index Construction According to Lemmata 1-5.

Lemma 4. Cost of Symmetric Partitioning

For symmetric splitting, the procedure *partition()* handles an amount of file I/O data of

$$\left(\log_2\left(\frac{n}{C_{\text{cache}}}\right) + \log_{C_{\text{max,dir}}}\left(\frac{n}{C_{\text{cache}}}\right)\right) \cdot 4 \cdot \text{filesize}$$

and requires

$$\left(\log_2\left(\frac{n}{C_{\text{cache}}}\right) + \log_{C_{\text{max,dir}}}\left(\frac{n}{C_{\text{cache}}}\right)\right) \cdot \left(\frac{8 \cdot n}{C_{\text{cache}}} + 2 \cdot \log_2(n)\right)$$

random seek operations.

Proof (Lemma 4)

Left out due to space limitations, cf. [Böh 98].

Lemma 5. Cost of Dynamic Index Construction

Dynamic X-tree construction requires $2\,n$ seek operations. The transferred amount of data is $2 \cdot n \cdot \text{pagesize}$.

Proof (Lemma 5)

For the X-tree, it is generally assumed that the directory is completely held in main memory. Data pages are not cached at all. For each insert, the corresponding data page has to be loaded and written back after completing the operation.

❏

Moreover, no better caching strategy for data pages can be applied, since without preprocessing of the input data set, no locality can be exploited to establish a working set of pages. From the results of lemmata 4 and 5 we can derive an estimate for the improvement factor of the bottom-up construction over dynamic index construction. The improvement factor for the number of seek operations is approximately:

$$\text{Improvement} \approx \frac{C_{\text{cache}}}{4 \cdot \left(\log_2\left(\frac{n}{C_{\text{cache}}}\right) + \log_{C_{\text{max,dir}}}\left(\frac{n}{C_{\text{cache}}}\right)\right)}$$

It is almost (up to the logarithmic factor in the denominator) linear in the cache capacity. Figure 4 depicts the improvement factor (number of random seek operations) for varying cache sizes and varying database sizes.

4. Experimental Evaluation

To show the practical relevance of our bottom-up construction algorithm, we have performed an extensive experimental evaluation by comparing the following index construction techniques: Dynamic index construction (repeated insert operations), Hilbert R-tree construction and our new method. All experiments have been computed on HP9000/780 workstations with several GBytes of secondary storage. Although our technique is applicable to most R-tree-like index structures, we decided to use the X-tree as an underlying index structure because according to [BKK 96], the X-tree outperforms other high-dimensional index structures. All programs have been implemented in C++.

In our experiments, we compare the construction times for various indexes. The external sorting procedure of our construction method was allowed to use only a relatively small cache (32 kBytes). Note that, although our implementation does not provide any further disk I/O caching, this cannot be guaranteed for the operating system. In contrast, the Hilbert construction method was implemented with internal sorting for simplicity. The construction time of the Hilbert method is therefore underestimated by far and would worsen in combination with external sorting when the cache size is strictly limited. All Hilbert-constructed indexes have a storage utilization near 100%.

Figure 5 shows the construction time of dynamic index construction and of the bottom-up methods. In the left diagram, we fix the dimension to 16, and vary the database size from 100,000 to 2,000,000 objects of synthetic data. The resulting speed-up of the bulk-loading techniques over the dynamic construction was so enormous that a logarithmic scale must be used in figure 5. In contrast, the bottom-up methods differ only slightly in their performance. The Hilbert technique was the best method, having a construction time between 17 and 429 sec. The construction time of symmetric splitting ranges from 26 to 668 sec., whereas unbalanced splitting required between 21 and 744 sec. in the moderate case and between 23 and 858 sec. for the 9:1 split. In contrast, the dynamic construction time ranged from 965 to 393,310 sec. (4 days, 13 hours). The improvement factor of our methods constantly increases with growing index size, starting from 37 to 45 for 100,000 objects and reaching 458 to 588 for 2,000,000 objects. The Hilbert construction is up to 915 times faster than the dynamic index construction. This enormous factor is not only due to internal sorting but also due to reduced overhead in changing the ordering attribute. In contrast to Hilbert construction, our technique changes the sorting criterion during the sort process according to the split tree. The more often the sorting criterion is changed, the more unbalanced the split becomes because the height of the

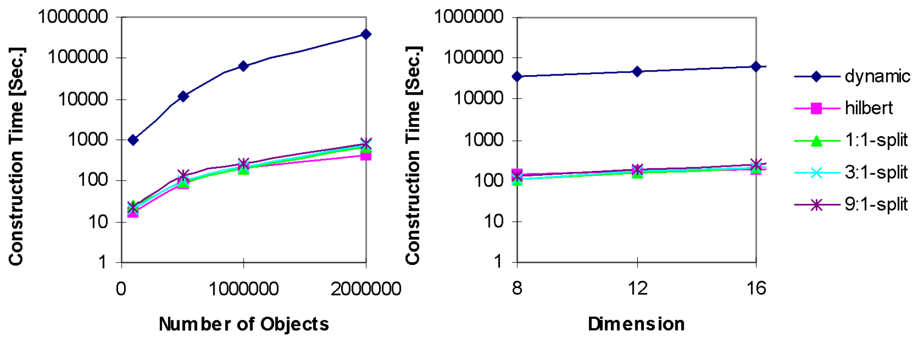

Figure 5: Performance of Index Construction Against Database Size and Dimension.

split tree increases. Therefore, the 9:1-split has the worst improvement factor. The right diagram in figure 5 shows the construction time for varying index dimensions. Here, the database size was fixed to 1,000,000 objects. It can be seen that the improvement factors of the construction methods (between 240 and 320) are rather independent from the dimension of the data space.

Our further experiments, which are not presented due to space limitations [Böh 98], show that the Hilbert construction method yields a bad performance in query processing. The reason is the high overlap among the page regions. Due to improved space partitioning resulting from knowing the data set *a priori*, the indexes constructed by our new method outperform even the dynamically constructed indexes by factors up to 16.8.

5. Conclusion

In this paper, we have proposed a fast algorithm for constructing indexes for high-dimensional data spaces on secondary storage. A user-defined split-strategy allows the adaptation of the index properties to the requirements of a specific knowledge discovery algorithm. We have shown both analytically and experimentally that our construction method outperforms the dynamic index construction by large factors. Our experiments further show that these indexes are also superior with respect to the search performance. Future work includes the investigation of various split strategies and their impact on different query types and access patterns.

6. References

[BBBK 99] Böhm, Braunmüller, Breunig, Kriegel: ‘*Fast Clustering Using High-Dimensional Similarity Joins*’, submitted for publication, 1999.

[BBJ+ 99] Berchtold, Böhm, Jagadish, Kriegel, Sander: *‘Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces’*, submitted for publication, 1999.

[BBK 98a] Berchtold, Böhm, Kriegel: *‘Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations’,* Int. Conf. on Extending Database Techn., EDBT, 1998.

[BBKK 97] Berchtold, Böhm, Keim, Kriegel: ‘*A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*’, ACM PODS Symp. Principles of Database Systems, 1997.

[BK 99] Böhm, Kriegel: ‘*Dynamically Optimizing High-Dimensional Index Structures*’, subm., 1999.

[BKK 96] Berchtold, Keim, Kriegel: *‘The X-Tree: An Index Structure for High-Dimensional Data*’, Int. Conf. on Very Large Data Bases, VLDB, 1996.

[BSW 97] van den Bercken, Seeger, Widmayer:*‘A General Approach to Bulk Loading Multidimensional Index Structures’*, Int. Conf. on Very Large Databases, VLDB, 1997.

[Böh 98] Böhm: *‘Efficiently Indexing High-Dimensional Data Spaces’,* PhD Thesis, University of Munich, Herbert Utz Verlag, 1998.

[EKSX 96] Ester, Kriegel, Sander, Xu: *‘A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise’*, Int. Conf. Knowl. Disc. and Data Mining, KDD, 1996.

[Hoa 62] Hoare: *‘Quicksort’*, Computer Journal, Vol. 5, No. 1, 1962.

[JD 88] Jain, Dubes: ‘*Algorithms for Clustering Data’,* Prentice-Hall, Inc., 1988.

[JW 96] Jain, White: *‘Similarity Indexing: Algorithms and Performance’*, SPIE Storage and Retrieval for Image and Video Databases IV, Vol. 2670, 1996.

[KF 94] Kamel, Faloutsos: *‘Hilbert R-tree: An Improved R-tree using Fractals’*. Int. Conf. on Very Large Data Bases, VLDB, 1994.

[KR 90] Kaufman, Rousseeuw: *‘Finding Groups in Data: An Introduction to Cluster Analysis’*, John Wiley & Sons, 1990.

[NH 94] Ng, Han: *‘Efficient and Effective Clustering Methods for Spatial Data Mining’*, Int. Conf. on Very Large Data Bases, VLDB, 1994.

[Sed 78] Sedgewick: ‘*Quicksort*’, Garland, New York, 1978.

[WSB 98] Weber, Schek, Blott: ‘*A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*’, Int. Conf. on Very Large Databases, VLDB, 1998.