Proc. 17th IEEE Int. Conf. on Data Engineering (ICDE), Heidelberg, Germany, 2001.

A Cost Model and Index Architecture for the Similarity Join

Christian Böhm and Hans-Peter Kriegel
University of Munich, Germany
{*boehm,kriegel*}*@dbs.informatik.uni-muenchen.de*

Abstract

The similarity join is an important database primitive which has been successfully applied to speed up data mining algorithms. In the similarity join, two point sets of a multidimensional vector space are combined such that the result contains all point pairs where the distance does not exceed a parameter ε. Due to its high practical relevance, many similarity join algorithms have been devised. In this paper, we propose an analytical cost model for the similarity join operation based on indexes. Our problem analysis reveals a serious optimization conflict between CPU time and I/O time: Fine-grained index structures are beneficial for the CPU efficiency, but deteriorate the I/O performance. As a consequence of this observation, we propose a new index architecture and join algorithm which allows a separate optimization of CPU time and I/O time. Our solution utilizes large pages which are optimized for I/O processing. The pages accommodate a search structure which minimizes the computational effort. In the experimental evaluation, a substantial improvement over competitive techniques is shown.

1. Introduction

Large sets of multidimensional vector data have become widespread to support modern applications such as CAD [17], multimedia [11], medical imaging [25], and time series analysis [2]. In such applications, complex objects are stored in databases. To facilitate the search by similarity, multidimensional feature vectors are extracted from the objects and organized in multidimensional indexes. The particular property of this *feature transformation* is that the Euclidean distance between two feature vectors corresponds to the (dis-) similarity of the original objects of the underlying application. Therefore, a similarity search can be translated into a neighborhood query in the feature space.

If a user is not only interested in the properties of single data objects but also in the properties of the data set as a whole he or she is supposed to run some data mining algorithms on the set of feature vectors. Data mining is the process of extracting implicit knowledge from the data set which is previously unknown and potentially useful. Standard tasks of data mining are clustering [12], i.e. finding groups of objects such that the intra-group similarity is maximized and the inter-group similarity is minimized, outlier detection [21], or the determination of association rules [19]. Considering these standard tasks, we can observe that many of the state-of-the-art algorithms require to consider all pairs of points which have a distance not exceeding a user-given parameter ε. This operation of generating all pairs is in essence a *similarity join*, and, as a consequence, many data mining algorithms can be directly performed on top of a similarity join [BBBK 00].

A typical example of such an algorithm is the clustering algorithm DBSCAN [9]. This algorithm defines a point *P* of the database to be a *core point* with respect to the user-given parameters ε and *min_pts* if at least a number *min_pts* of the points in the database have a distance of no more than ε from *P*. To compute the overall cluster structure, the algorithm transitively collects all core points which have a distance not exceeding ε from each other. The original definition of the algorithm performs a range query with the radius ε for each point stored in the database. We have shown in [3] that each of the two subtasks, core point determination and cluster collection, can be performed equivalently (i.e. yielding exactly the same result) by a single run of a *similarity join*. This transformation allows great performance improvements (up to 54 times faster) using standard join algorithms. But also other algorithms for knowledge discovery in databases are amenable to processing by similarity joins, such as the outlier detection algorithm RT [21], nearest neighbor clustering [16], single-link clustering [33], the hierarchical cluster analysis method OPTICS [1], proximity analysis [20], spatial association rules [19] and numerous other algorithms.

Due to the high impact of the similarity join operation, a considerable number of different algorithms to evaluate the similarity join have been proposed (the most relevant approaches will be reviewed in section 2). From a *theoretical* point of view, however, the similarity join has not been sufficiently analyzed. Our feeling is that the lack of insight into the properties of the similarity join is an obstacle in developing new methods with better performance.

In this paper, we analyze the performance of the similarity join using indexes. We will point out that the index selectivity is the central key to the performance analysis. In section 3, we will present the formula for the index selectivity with respect to the similarity join operation. We will further analyze how much selectivity is needed to justify the usage of an index for join processing. The surprising result is, that for the optimization of the CPU operations, a fine-grained index is indispensable. For the I/O operations, however, fine-grained indexes are disastrous. Our conclusion from these results is the necessity of decoupling the CPU optimization from the I/O optimization. Taking this into account, we will propose a suitable index architecture in section 6. Our solution uses an index with a coarse granularity in order to reduce the overhead of random I/O operations. To additionally reduce the computational overhead, these large

primary pages of the index are structured by a secondary search structure. This structure partitions the primary pages into a set of *accommodated buckets* which are, themselves, not subject to I/O, but only used to decide whether or not to exclude the corresponding pairs of secondary pages from CPU processing. Using this approach, the granularity of the index can be separately optimized for CPU and I/O operations. Our index architecture is additionally supposed to profit from ongoing trends in hardware development. E.g. [13] suggests that the transfer rates of disk drives continue to improve much faster than the rotational delay time. As a consequence the optimum page size with respect to I/O will even increase. As we will show in section 5, the optimum page size with respect to CPU time is determined by the proportion between two kinds of distance calculations: Distances between points and distances between rectangles. As it is not likely that one kind of distance calculation will improve more than the other, we assume that the optimum with respect to CPU time will not change much.

2. Related work

2.1. Multidimensional join processing

In the relational data model a join means to combine the tuples of two relations R and S into pairs if a *join predicate* is fulfilled. In multidimensional databases, R and S contain points (feature vectors) rather than ordinary tuples. In a *similarity join*, the join predicate is similarity, i.e. the pair $(p,q) \in R \times S$ appears in the result if the Euclidean distance between the two feature vectors p and q does not exceed a threshold value ε. If R and S are actually the same point set, the join is called a *self-join*.

Join algorithms using R-trees

Most related work on join processing using multidimensional index structures is based on the *spatial join*. The spatial join operation is defined for 2-dimensional polygon databases where the join predicate typically is the intersection between two objects. This kind of join predicate is prevalent in map overlay applications. We adapt the relevant algorithms to allow distance based predicates for point databases instead of the intersection of polygons.

The most common technique is the R-tree Spatial Join (RSJ) [7], which is based on R-tree like index structures built on R and S. RSJ is based on the lower bounding property which means that the distance between two points is never smaller than the distance between the regions of the two pages in which the points are stored. The RSJ algorithm traverses the indexes of R and S synchronously. When a pair of directory pages (P_R,P_S) is under consideration, the algorithm forms all pairs of the child pages of P_R and P_S having distances of at most ε. For these pairs of child pages, the algorithm is called recursively, i.e. the corresponding indexes are traversed in a depth-first order.

Various optimizations of RSJ have been proposed. Huan, Jing and Rundensteiner propose the BFRJ-algorithm [14] which traverses the indexes according to a breadth-first strategy. At each level, BFRJ creates an intermediate join index and deploys global optimization strategies to improve the join computation at the subsequent level. Improved cache management leads to 50% speed-up factors.

Join algorithms without index

If no multidimensional index is available, it is possible to construct the index on the fly before starting the join algorithm. Usually, the dynamic index construction by repeated insert operations performs poorly and cannot be amortized by performance gains during join processing. However, several techniques for bulk-loading multidimensional index structures have been proposed [18, 5]. Their runtime is substantially smaller compared to the runtime of the repeated insert operations, even if the data set does not fit in main memory. This effort is typically amortized by efficiency gains in join processing.

The seeded tree method [26] joins two point sets provided that only one is supported by an R-tree. The partitioning of this R-tree is used for a fast construction of the second index on the fly. The spatial hash-join [27, 29] decomposes the set R into a number of partitions which is determined according to system parameters. Sampling is applied to determine initial buckets. Each object of R is inserted into a bucket such that bucket enlargement and bucket overlap are minimized. Then, each object of S is inserted into every bucket having a distance not greater than epsilon from the object (object replication). If each bucket fits in main memory, a single scan of the buckets is sufficient to determine all join pairs.

A join algorithm particularly suited for similarity self joins is the ε-kdB-tree [34]. The basic idea is to partition the data set perpendicularly to one selected dimension into stripes of the width ε to restrict the join to pairs of subsequent stripes. The join algorithm is based on the assumption that the database cache is large enough to hold the data points of two subsequent stripes. In this case it is possible to join the set in a single pass. To speed up the CPU operations, for each stripe a main memory data structure, the ε-kdB-tree is constructed which also partitions the data set according to the other dimensions until a defined node capacity is reached. For each dimension, the data set is partitioned at most once into stripes of the width ε. Finally, a tree matching algorithm is applied which is restricted to neighboring stripes. Whether the ε-kdB join can be successfully applied depends on the choice of the similarity parameter ε. For instance, our experiments in section 7 using uniformly distributed points of an 8-dimensional feature space $[0..1]^8$ used $\varepsilon = 0.3$, which is at a first glance rather high, but in high-dimensional spaces it is necessary to choose such a large ε in order to find join mates at all (note that $0.3^8 = 6.5 \cdot 10^{-5}$). As the corresponding join algorithm needs to hold 2 stripes simultaneously in the main memory (i.e. 60% of the database size plus the overhead of the directory) the first partitioning step reduces the joining problem by less than 40%. The authors also extended their technique to cases where a single dimension is not sufficient to partition the data set into stripes fitting into the buffer. In this case, the authors propose to partition the data set according to 2 dimensions and to hold 4 neighboring partitions simultaneously in main memory for tree matching. In this case, however, it is not possible to perform a single-pass join. Additionally, for our running example, we still have to hold 36% ($=4 \cdot 0.3^2$) of the database in the main memory buffer. Even for all our real data experiments (where ε was much smaller) some of the ε-stripes contain too many data points (e.g. 35% for the meteorology data with $\varepsilon = 0.000$; due to the skew of the data, even very small

stripes such as $\varepsilon = 0.0001$ are filled with considerable amounts of data. The 35% have been determined by analyzing the data experimentally) and, therefore, the algorithm fails in the required configuration.

Koudas and Sevcik present the Size Separation Spatial Join [23] and the Multidimensional Spatial Join [24] which makes use of space filling curves to order the points in a multidimensional space. Each point is considered as a cube with side-length ε in the multidimensional space. Each cube is assigned a value l (level) which is essentially the size of the largest cell (according to the Hilbert decomposition of the data space) that contains the point. The points are distributed over several *level-files* each of which contains the points of a level in the order of their Hilbert values. For join processing, each subpartition of a level-file must be matched against the corresponding subpartitions at the same level and each higher level file of the other data set. For our running example of $\varepsilon = 0.3$ on uniform 8-dimensional data, the probability that an ε-cube intersects the first partitioning line is 0.3. That means 30% of all points are assigned to level $l = 0$ which must be compared with all other data points, and, therefore, be completely held in main memory during the scan of the database. Of the remaining 70% points, another 30% is assigned to level $l = 1$ which must be compared to half of all other points (these 21% = 0.7·0.3 must be held in main memory during half the scan time), and so on. For all levels, the expected value for the ratio of points to be held in main memory corresponds to $\sum_l \varepsilon \cdot (1-\varepsilon)^l / 2^l = 46\%$. Our real-data experiments yield analogous results (e.g. 26% of the CAD data needed simultaneously in main memory, as determined experimentally).

2.2. Cost models for similarity queries

Due to the high practical relevance of similarity queries, cost models for estimating the number of page accesses have already been proposed several years ago. The first approach is the well-known cost model proposed by Friedman, Bentley and Finkel [10] for nearest neighbor queries using the maximum metric. The original model estimates leaf accesses in a kd-tree, but can be easily extended to estimate data page accesses of R-trees and related index structures. The expected number of data page accesses in an R-tree is

$$A = \left(\sqrt[d]{\frac{1}{C_{\text{eff}}}} + 1 \right)^d ,$$

where C_{eff} is the effective capacity (i.e. average number of points) of a data page and d is the dimension of the data space.

Papadopoulos and Manolopoulos used these results for estimating data page accesses of R-trees when processing nearest neighbor queries in a Euclidean space [30]. As it is difficult to determine accesses of pages with rectangular regions for spherical queries, they approximate query spheres by minimum bounding and maximum enclosed cubes and thus determine upper and lower bounds of the number of page accesses.

Berchtold, Böhm, Keim and Kriegel [6] presented a cost model for query processing in high-dimensional data spaces. It provides accurate estimations for nearest neighbor queries and range queries using the Euclidean metric. In contrast to [30] it does not approximate the Euclidean query by a cube to cope with the Euclidean metric. The paper introduces the concept of

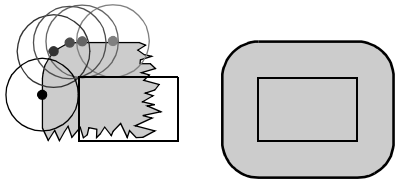

Figure 1. The Minkowski Sum

the Minkowski sum (cf. figure 1) to determine the access probability of a rectangular page for spherical queries (i.e. range queries and nearest neighbor queries). The Minkowski sum will be used in section 3 to determine the selectivity of an index with respect to the join operation. The cost model in [6] has found numerous applications. Weber, Schek and Blott [35] use it to show the superiority of the sequential scan in sufficiently high dimensions. They present the VA-file, an improvement of the sequential scan. Ciaccia, Patella and Zezula [8] adapt the cost model in [6] to estimate the page accesses of M-trees, an index structure for data spaces which are metric spaces but not vector spaces (i.e. only the distances between the objects are known, but no explicit positions). Papadopoulos and Manolopoulos [31] apply the cost model for declustering in disk arrays.

3. Problem analysis

In this section, we separately analyze the performance behavior of the similarity join with respect to CPU cost and I/O cost. For this purpose, we assume a simplified version of an R-tree like index structure which consists of a set of data pages on the average filled with a number C_{eff} of points and a flat directory. In our simplified index, similarity joins are processed by first reading the directory in a sequential scan, then determining the qualifying pairs of data pages, and, finally, accessing and processing the corresponding data pages.

When using an index, the only gain is the selectivity, i.e. not all pairs of pages must be accessed and not all pairs of points must be compared. For a join, the index selectivity σ is defined as the number of page pairs to be processed divided by the theoretically possible page pairs:

$$\sigma = \frac{\text{processed page pairs}}{B_R \cdot B_S} \approx \frac{\text{processed point pairs}}{|R| \cdot |S|} ,$$

where B_R and B_S are the numbers of blocks of the point sets and $|R|$ and $|S|$ are the sizes of the corresponding data sets (number of points). The index selectivity depends on the quality of the index and on the parameter ε of the similarity join. As a matter of fact, using an index for a join computation induces some overhead. We will first determine the possible overhead for the index usage. It is important to limit the overhead to a threshold, say 10%, to avoid that the join algorithm becomes arbitrarily bad in case of a large ε.

The distance calculations in the directory are the most important overhead for the CPU. The calculation of a Euclidean distance between two boxes (time t_{box}) can be assumed to be by a factor α more expensive than a distance calculation between two points (time t_{point}) with

$$\alpha = t_{\text{box}} / t_{\text{point}} , \text{ typically } \alpha \approx 5 ,$$

because it requires 2 additional case distinctions per dimension d (since both times t_{point} and t_{box} are linear in d, α does not de-

pend on d). Therefore, the relative CPU overhead when processing a page filled with C_{eff} points is

$$v_{\text{CPU}} = \frac{t_{\text{box}}}{C_{\text{eff}} \cdot t_{\text{point}}} = \frac{\alpha}{C_{\text{eff}}}.$$

Limiting the CPU overhead $v_{\text{CPU}} \le 10\%$ requires $C_{\text{eff}} \ge \alpha / v_{\text{CPU}} \ge 10\alpha$. A similar consideration is possible for the I/O. Here, the time for reading the directory is negligible (less than 5%). Important are, however, the seek operations which are necessary because index pages are loaded by random accesses rather than sequentially. The overhead is the time necessary for disk arm positioning (t_{seek}) and for the latency delay (t_{lat}), divided by the "productive" time for reading the actual data from disk (t_{tr} is the transfer time per Byte):

$$v_{\text{I/O}} = \frac{t_{\text{seek}} + t_{\text{lat}}}{C_{\text{eff}} \cdot 4d \cdot t_{\text{tr}}} = \frac{\beta}{C_{\text{eff}} \cdot 4d}$$

with the hardware constant $\beta = (t_{\text{seek}} + t_{\text{lat}}) / t_{\text{tr}}$ ($\beta \approx 40000$ for typical disk drives). We assume 4 bytes for a floating point value. Limiting the I/O overhead $v_{\text{I/O}} \le 10\%$ requires

$$C_{\text{eff}} \ge \beta / (4d \cdot v_{\text{I/O}}) \ge 100000 / d$$

which is even for a high data space dimension $d \ge 100$ orders of magnitude larger than the corresponding CPU limit.

Next we analyze how much selectivity is needed to brake-even with the overhead in index-based query processing. Again, we separately treat the CPU cost and the I/O cost. For the CPU cost, we know that we have to perform one distance calculation for every pair of pages in the directory. Additionally, for those page pairs which are mates (i.e. $\sigma \cdot |R| \cdot |S| / C_{\text{eff}}^2$ pairs) all pairs of the stored points must be distance compared (C_{eff}^2 distance computations). Altogether, we get $\sigma \cdot |R| \cdot |S|$ distance computations for the points. For join processing without index, $|R| \cdot |S|$ distance calculations must be performed. To justify the index, we postulate:

$$\sigma \cdot |R| \cdot |S| \cdot t_{\text{point}} + \frac{|R| \cdot |S|}{C_{\text{eff}}^2} \cdot t_{\text{box}} \le |R| \cdot |S| \cdot t_{\text{point}}$$

$$\text{and thus } \sigma \le 1 - \frac{t_{\text{box}}}{C_{\text{eff}}^2 \cdot t_{\text{point}}} = 1 - \frac{\alpha}{C_{\text{eff}}^2}$$

For each pair of pages which must be processed, we assume that a constant number λ of pages must be loaded from disk. If there is no cache and a random order of processing then $\lambda = 2$. If a cache is available λ is lower, but we assume that λ is not dependent on the page capacity, because the *ratio* of cached pages is constant (e.g. 10 %). We postulate that the cost for the page accesses using a page capacity C_{eff} and a selectivity σ must not exceed the cost for the page accesses for low-overhead pages without selectivity:

$$\sigma\lambda \frac{|R| \cdot |S|}{C_{\text{eff}}^2} \cdot (t_{\text{seek}} + t_{\text{lat}} + 4dC_{\text{eff}} \cdot t_{\text{tr}}) \le \frac{\lambda \cdot |R| \cdot |S|}{\beta^2 / (4dv_{\text{I/O}})^2} \cdot \frac{\beta t_{\text{tr}}}{v_{\text{I/O}}}$$

We obtain the following selectivity which is required to justify an index with respect to the I/O cost:

$$\sigma \le \frac{C_{\text{eff}}^2 \cdot 16d^2 \cdot v_{\text{I/O}}}{\beta \cdot (\beta + 4d \cdot C_{\text{eff}})}$$

The actual selectivity of an index with respect to the similarity join operation can be modeled as follows. As we assume no knowledge about the data set, we model a uniform and independent distribution of the points in a d-dimensional unit hypercube $[0..1]^d$. Furthermore, we assume that the data pages have the side length $\sqrt[d]{C_{\text{eff}}/|R|}$ and $\sqrt[d]{C_{\text{eff}}/|S|}$, respectively because $C_{\text{eff}}/|R|$ is the expected volume of a page region.

The index selectivity can be determined by the concept of the Minkowski sum [6]. A pair of pages is processed whenever the minimum distance between the two page regions does not exceed ε. To determine the probability of this event, we fix one page at some place and we (conceptually) move the other page over the data space. Whenever the distance is less than or equal to ε we mark the data space at the position of the center of the second page (cf. figure 2). As we mark the complete area where the variable page is a join mate of the fixed page, the probability of an arbitrary page to be a mate of the fixed page, corresponds to the marked area divided by the area of all possible positions of the page (which is the data space, $[0..1]^d$). The Minkowski sum is a concept often used in robot motion planning. Understanding two geometric objects A and B each as an infinite number of vectors (points) in the data space (e.g. $A = \{a_1, a_2, ...\}$) the Minkowski sum $A \oplus B$ is defined as the set of the vector sums of all combinations between vectors in A and B, i.e. $A \oplus B = \{a_1+b_1, a_1+b_2, a_2+b_1, ...\}$. For cost modeling we are only interested in the volume of the Minkowski sum, not in its shape. The example in figure 2 is now constructed, step by step: On the left hand side, simply the fixed page region with side length $\sqrt[d]{C_{\text{eff}}/|R|}$ is depicted. Next we show the complete area of the data space where the distance from the page region does not exceed ε. This corresponds to the Minkowski sum of the page region and a sphere of radius ε. Then, we show an example of a marginally mating page. The center point of the page is marked, as depicted. If we move this page around the shaded contour, we obtain the geometric object depicted on the right hand side. It corresponds to the Minkowski sum of three objects, the two page regions and the ε-sphere. The Minkowski sum of the two cubes is a cube with added side length. The Minkowski sum of the resulting cube and the ε-sphere can be determined by a binomial formula which was derived first in [6]:

$$\sigma_{\text{Mink}} = \sum_{0 \le i \le d} \binom{d}{i} \cdot \left(\sqrt[d]{\frac{C_{\text{eff}}}{|R|}} + \sqrt[d]{\frac{C_{\text{eff}}}{|S|}} \right)^{d-i} \cdot V_{i\text{-dim. Sphere}}(\varepsilon)$$

In figure 3 we compare the required and the estimated selectivities along with varying block sizes. The thin, dashed line shows σ as it is needed to justify the CPU overhead of the index. The curve is increasing very fast. Therefore, no good (i.e. low)

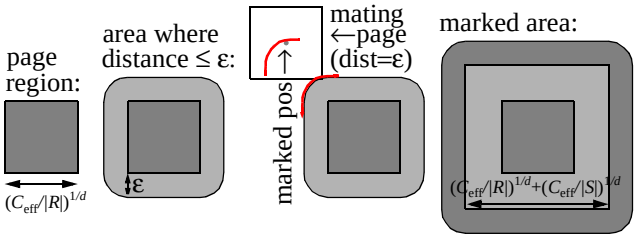

Figure 2. The Minkowski Sum for Page Pairs

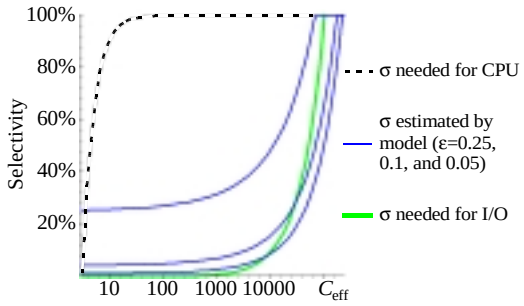

Figure 3. Selectivities Needed to Justify an Index

selectivity is needed unless the block size is *very* small (<10). Quite the opposite is true for the I/O cost (thick gray line). Until a block size of at least 10,000 points, an unrealistic good selectivity is needed. Only for block sizes starting at 10,000, index selectivities above 10% are allowed. Also depicted are 3 actual selectivities, estimated by our model. These curves are typical examples to demonstrate the range of possible curves.

The index usage is justified if the actual selectivity is below the needed selectivity. The higher the difference between "actual" and "needed" is, the more the index will outperform non-index joins. Over a wide range of block sizes, the actual selectivity curve is below the curve for the CPU cost. The highest difference is obtained between 10 and 100 points. In contrast the I/O curve needs always a better selectivity than the index has, if the distance parameter ε is high. For lower ε, the index is justified, but only for very large pages. The difference is never high.

It would be possible to determine the optimum block size for the CPU-cost and for the I/O cost. For this purpose we would have to choose a fixed distance parameter ε. As our objective is to create an index which is suitable in every join situation, it would be bad to optimize for a specific ε.

However, we can learn some important lessons from figure 3. Smaller pages are very good for minimizing the CPU cost. But for the I/O cost, small pages of 10..100 points are disastrous. Large pages, in contrast, minimize the I/O cost but are bad for the CPU cost. Gains at one of the sides are always paid by a higher price on the other side. Optimizing the overall-cost can merely bring the two cost components in balance.

To escape from this dilemma, it is necessary to decouple the I/O cost from the CPU cost by a new index architecture which will be proposed in section 6. This architecture consists of large blocks which are subject to I/O. These large blocks accommodate a secondary search structure with "subpages" which are used for reducing the computational effort.

4. Optimization of the I/O time

We have seen in the previous sections that limiting the overhead of I/O operations requires large pages with C_{eff} in the area of at least some 10,000s points. Additionally, only for such large pages, the actual selectivity is below the needed selectivity (cf. figure 3). When the block size is small, the selectivity which is needed to compensate for the index overhead is much smaller than the actually achievable selectivity of the multidimensional index.

We may ask ourselves whether or not the page size has an influence on the performance if the selectivity is close to 100%. For similarity queries with a bad index selectivity, the sequential scan is optimal, i.e. an infinitely large page size. For joins, however, the situation may be different and at the end of this section, we will know that a careful page size optimization is important.

In section 6 we will propose a join algorithm which loads several pages of R into the buffer and combines them with those pages of S which have a distance less than or equal to ε to at least one of the buffered S-pages. For such algorithms, the number of page accesses is

$$A = B_R + \sigma \frac{B_R \cdot B_S}{C-1} = \left\lceil \frac{f_R}{b} \right\rceil + \sigma \frac{\lceil f_R/b \rceil \cdot \lceil f_S/b \rceil}{\lfloor c/b \rfloor - 1},$$

where B_S and B_R are the numbers of blocks into which the point sets are decomposed, C is the number of blocks of the buffer, b is the block size in Bytes and f_R, f_S, and c are the sizes of the point sets and of the buffer in Bytes. The formula states the fact that the point set R is scanned once (B_R accesses) and the blocks of S are considered $\sigma B_R/(C-1)$ times. As S is scanned blockwise we face the following trade-off: If b is too large, i.e. close to $c/2$, then S must be scanned more often than necessary. In contrast, if b is chosen too small (e.g. 1 KByte), then the disk yields a latency delay after each block access.

The total cost of the join can be summarized as follows: For every block access of S, we have the corresponding transfer time $b \cdot t_{tr}$ and the latency delay t_{lat}. Additionally, for each of the $B_R/(C-1)$ traversals of R we have two disk arm positioning operations (t_{seek}), one more latency delay, and the transfer time:

$$t_{I/O} = f_R t_{tr} + \frac{\lceil f_R/b \rceil}{\lfloor c/b \rfloor - 1} \cdot (2t_{seek} + t_{lat}) + \sigma \cdot \frac{\lceil f_R/b \rceil \cdot \lceil f_S/b \rceil}{\lfloor c/b \rfloor - 1} \cdot (t_{lat} + b \cdot t_{tr})$$

As R is scanned only once and in larger blocks than the inner point set, we can neglect the cost for that. Further we can omit the ceiling-operator in f_R/b and f_S/b, because the point sets are much larger than the block size, and thus the relative error by this approximation is negligible, too:

$$t_{I/O} \approx \sigma \cdot \frac{f_R \cdot f_S}{b^2 \cdot (\lfloor c/b \rfloor - 1)} \cdot (t_{lat} + b \cdot t_{tr})$$

We are looking for the block size b which minimizes $t_{I/O}$. The only obstacle in optimization by setting the derivative to 0 is the floor-rounding in c/b which cannot be neglected because $c >> b$ is not guaranteed (we are basically out to determine whether the buffer should be assigned to R and S more balanced or more unbalanced). We solve this problem by first optimizing a hull function t_{hull} with $t_{hull} = t_{I/O}$ if b divides c and $t_{hull} < t_{I/O}$ otherwise:

$$t_{hull} = \sigma \cdot \frac{f_R \cdot f_S}{b \cdot (c-b)} \cdot (t_{lat} + b \cdot t_{tr})$$

Figure 4 depicts the actual cost function $t_{I/O}$ and the hull function t_{hull} for a file size of 10 MByte and a buffer of 500 KByte. It is easy to see that the optimum of $t_{I/O}$ cannot be at some position where $t_{I/O}$ is continuous, because the remaining term

$$\frac{\sigma \cdot f_R \cdot f_S}{b^2 \cdot \gamma} \cdot (t_{lat} + b \cdot t_{tr})$$

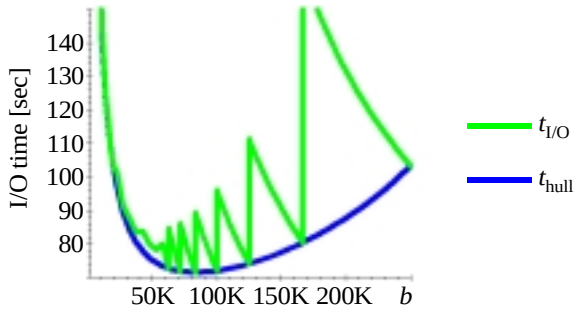

Figure 4. Optimizing the Join for I/O

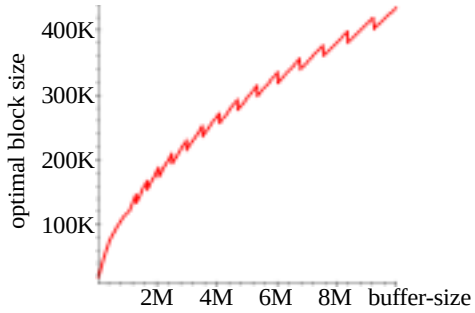

Figure 5. I/O-Optimal Block Size

(when the floor-expression is some constant γ) is strictly monotonically decreasing. This can be shown by the derivative. So, the minimum of $t_{I/O}$ must be at a position where b divides c without rest. As t_{hull} meets $t_{I/O}$ at all such positions, we know that the optimum of $t_{I/O}$ can only be at the first meeting point ($t_{I/O} = t_{hull}$) immediately left or right from the minimum of t_{hull}. The minimum of t_{hull} can be determined by setting the derivative to zero which yields two results. Only one is positive and it is a minimum, which can be shown according to the second derivative. The positive solution of $\frac{\partial}{\partial b} t_{hull} = 0$ is (because $\partial\sigma/\partial b \approx 0$ for large pages):

$$b_{opt,hull} = \frac{\sqrt{t_{lat}^2 + t_{tr} \cdot t_{lat} \cdot c} - t_{lat}}{t_{tr}}$$

The two possible positions of the actual optimum of $t_{I/O}$ are

$$b_1 = \frac{c}{\left\lfloor \frac{c}{b_{opt,hull}} \right\rfloor} \qquad b_2 = \frac{c}{\left\lceil \frac{c}{b_{opt,hull}} \right\rceil}.$$

These two values must be substituted in the cost function to determine the actual minimum.

$$b_{opt,I/O} = \begin{cases} b_1 \text{ if } t_{hull}(b_1) \le t_{hull}(b_2) \\ b_2 \text{ if } t_{hull}(b_1) \ge t_{hull}(b_2) \end{cases}$$

As the minimum of t_{hull} is very stable (cf. figure 4), it is also possible to use e.g. b_1 without considering b_2.

Figure 5 depicts b_1 with a buffer size varying from 0 to 10 MByte for a disk drive with a transfer rate of 4 MByte/s and a latency delay of 5 ms. The optimum for a 10 MByte buffer, for instance, is 455903 Bytes (i.e., 23 buffer pages).

5. Optimization of the CPU time

The CPU cost are composed of two components: cost of directory processing (i.e. distance computations among page regions) and cost of data level processing (i.e. point distance calculations). In our simplified index structure, the distance between every pair of pages must be calculated, i.e. $|R| \cdot |S| / C_{eff}^2$ calculations. The number of point distance calculations depends on the index selectivity and is $\sigma \cdot |R| \cdot |S|$. The total CPU cost is:

$$t_{CPU} = \sigma \cdot |R| \cdot |S| \cdot t_{point} + \frac{|R| \cdot |S|}{C_{eff}^2} \cdot t_{box}$$

As we have $t_{box} = \alpha \cdot t_{point}$, we can rewrite this and insert our estimate of the selectivity:

$$t_{CPU} = |R| \cdot |S| \cdot t_{point} \cdot \left(\left(\sqrt[d]{\frac{C_{eff}}{|R|}} + \sqrt[d]{\frac{C_{eff}}{|S|}} + 2\varepsilon \right)^d + \frac{\alpha}{C_{eff}^2} \right)$$

We do not want to optimize the index for a specific distance parameter ε, because we must create an index which is good for every similarity join. Therefore, we consider the two extreme situations of very low and very high distance parameters. For small ε, we can rewrite our CPU cost formula to

$$t_{low\varepsilon} = |R| \cdot |S| \cdot t_{point} \cdot \left(C_{eff} \cdot \left(\sqrt[d]{\frac{1}{|R|}} + \sqrt[d]{\frac{1}{|S|}} \right)^d + \frac{\alpha}{C_{eff}^2} \right)$$

which is optimized by

$$C_{low\varepsilon} = \sqrt[3]{2\alpha / \left(\sqrt[d]{\frac{1}{|R|}} + \sqrt[d]{\frac{1}{|S|}} \right)^d}.$$

If ε is very large, then the index cannot yield any selectivity. In this case, it is merely necessary to limit the overhead as in the beginning of section 3. For a 10% limit at least 10α points must be stored in a data page. Therefore, we have the following value for the effective capacity:

$$C_{opt} = \max\left\{ 10\alpha, \sqrt[3]{2\alpha / \left(\sqrt[d]{\frac{1}{|R|}} + \sqrt[d]{\frac{1}{|S|}} \right)^d} \right\}$$

6. The Multipage Index (MuX)

It has been shown in section 3 that it is necessary to decouple the I/O and CPU optimization to achieve a satisfactory performance in multidimensional join processing. In section 4 and section 5, it was shown how to optimize join processing with respect to I/O and CPU performance. We now introduce an index architecture and the corresponding algorithms which enable the separate optimization. In essence, our index consists of large I/O pages that are supported by an additional search structure to speed up the main-memory operations. A few index structures with supporting search structures have already been previously proposed. For instance, Lomet and Salzberg propose the hB tree [28] which uses a kd-tree like structure to organize directory pages. Their objective is improve the insert operations in order to achieve an overlap-free space decomposition in their index, not a separate optimization of CPU and I/O operations. Also, some quad tree based structures can be used in such a way. Kornacker [22] provides an interface

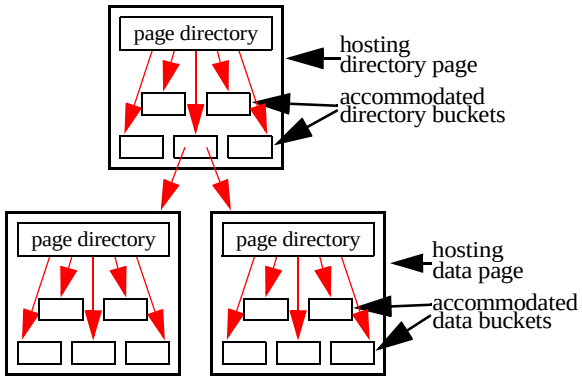

Figure 6. Index architecture of the multipage index

for GIST that allows the application of supporting search structures in index pages. Our solution uses a simple R-tree like secondary search structure. In the current paper, we have not yet evaluated which kind of search tree serves the best purpose. Our motivation for using minimum bounding rectangles for both, the primary and the secondary search structure, is to be able to apply the same cost model for both optimizations. Using different concepts for the primary and secondary search structure is viable, but requires different cost models and makes the analysis thus more complex. It remains as an issue for future work to evaluate different secondary search structures with respect to high-dimensional indexing and similarity join processing.

6.1. Index architecture

The *Multipage Index* (MuX) is a height-balanced tree with directory pages and data pages (cf. figure 6). Both kinds of pages are assigned to a rectilinear region of the data space and to a block on secondary storage. The block size is optimized for I/O according to the model proposed in section 4. The I/O optimized pages are called the *hosting pages*. As in usual R-trees, both kinds of pages store a number of entries (directory entries and data points). In contrast to usual R-trees, where the entries of pages are stored in random order in a simple array, MuX uses a secondary search structure to organize the entries. The complete search structure is accommodated in the hosting pages. Therefore, search operations in the secondary search structure do not raise any further I/O operations once the hosting page has been loaded.

For the secondary search structure, we use a degenerated R-tree consisting of a flat directory (called *page directory*) and a constant number of leaves (called *accommodated buckets*). If the hosting page is a data page, the accommodated buckets are data buckets and contain feature vectors. If the hosting page is a directory page, the accommodated buckets are directory buckets which store pairs of a MBR and a pointer to another hosting page. The page directory is flat and consists of an array of MBRs and pointers to the corresponding accommodated buckets. Generally, it would be straightforward to use a hierarchical page directory. The actual number of buckets accommodated on a hosting page, however, is not high enough to justify a deep hierarchy. In our current implementation, the primary directory of MuX also consists of a single level (flat hierarchy), because hierarchical directories often do not pay off in high-dimensional query processing, as it was pointed out e.g. in [4].

6.2. Construction and maintenance

For a fast index construction, the bottom-up algorithm for X-tree construction [5] was adopted. The various R-tree algorithms for insertions and deletions can also be adapted to the MuX architecture. Due to space limitations we cannot go into further details at this point.

6.3. Similarity queries

Similarity range queries can be efficiently processed by a depth-first traversal of the multipage index. For nearest neighbor queries, *k*-nearest neighbor queries and ranking queries, we propose to adapt the HS algorithm [15] which uses a priority queue for page scheduling. In our implementation, only the hosting pages are scheduled by the priority queue. Once a hosting page is accessed, the corresponding accommodated buckets are processed in order of decreasing priority. Accommodated buckets can additionally be pruned whenever their query distance exceeds the current pruning distance.

6.4. Join processing

We use the following strategy for join processing: One block of the buffer memory with the size of one hosting page is reserved for *S* (the *S-buffer*). The rest of the buffer (*R-buffer*) is used for caching one or more hosting pages of *R*. In the outermost loop of the algorithm presented in figure 7, the *R*-buffer is filled with a chunk of pages of *R*. In line (*), each hosting page of *S* which is a join mate of (at least) one of the accommodated buckets in the *R*-buffer is accessed. Then each pair of accommodated buckets having a distance of at least ε is processed, i.e. the point pairs fulfilling the join criterion are determined.

In line (*) our algorithm considers the *accommodated* buckets of the chunk in the *R*-buffer to exclude *hosting* pages of *S* from consideration. Note that our algorithm could also use the hosting pages of *R* instead of the accommodated buckets. The buckets, however, exclude more *S*-pages from processing (i.e. the index selectivity is improved). It would also be desirable to use the accommodated buckets of *S* for this exclusion test, but

```
algorithm MuX_join
    for i := 1 to B_R step C − 1 do
        load hosting pages B_R(i) .. B_R(i + C − 1) ;
        for j := 1 to B_S do
(*)         if B_S(j) has some join mate in an accomm.
                        bucket of B_R(i) .. B_R(i+C−1) then
                load hosting page B_S(j) ;
                for each accomm. bucket of
                                B_R(i) .. B_R(i + C − 1) do
                    for each accomm. bucket of B_S(j)
                        if distance (buckets) ≤ ε then
                            process pair of buckets;
```

Figure 7. Join Processing for the Multipage Index

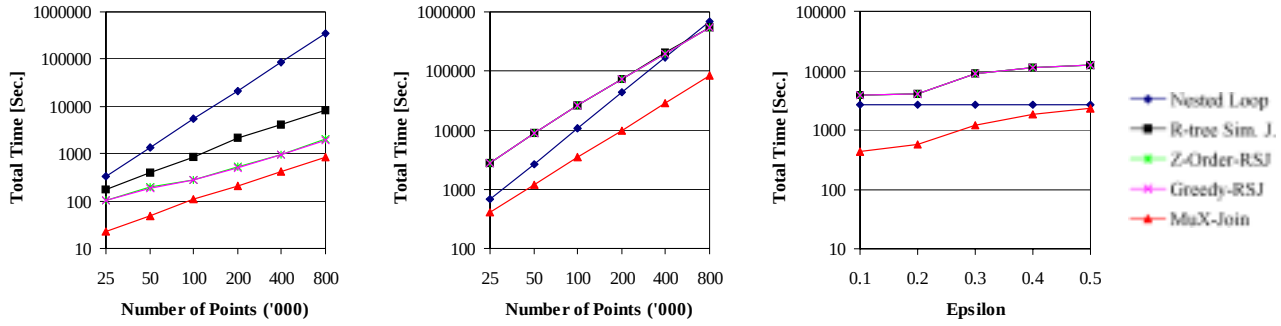

Figure 8. Uniform Data: 4D (left), 8D (middle and right)

the corresponding MBRs of these buckets are not known until the hosting page is loaded.

In the following two claims, we will point out why our MuX structure achieves a separate optimization of CPU and I/O performance and why this leads to a superior performance compared to the conventional R-tree join. For these claims we assume that the capacity of an accommodated bucket is at least 20 data points and that a hosting page stores at least 10 accommodated buckets.

Claim 1: The I/O cost of an R-tree and MuX are very similar if the page capacity of the R-tree corresponds to the capacity of a *hosting page* of MuX.

Claim 2: With respect to CPU cost, the MuX join performs similarly to an R-tree if the page capacity of the R-tree is chosen like the *accommodated buckets* of MuX.

Rationale for claim 1: Provided that the R-tree and the MuX structure apply the same insertion and splitting rules and provided that the page capacities are equal, both techniques lead to identical paginations. Therefore, the same page pairs have to be considered which leads to the same number of page accesses. The main difference is that MuX pages have to store additionally the page directory which increases the cost of a page access. The page directory stores pairs of lower bounds and upper bounds for each accommodated bucket. For each bucket we have to store as much information as for two data points. As the capacity of a bucket is at least 20 data points, the storage size of a MuX hosting page is at most 10% larger than the storage size of the R-tree. Therefore, the I/O cost of MuX is at most 10% higher than that of the R-tree.

Rationale for claim 2: Provided that the page capacity of the R-tree corresponds to the page capacity of the accommodated buckets, and provided that the same insertion and split strategy has been applied, the two structures exactly compare the same point pairs. The number of point distance computations is identical. The MuX structure determines at most as many distances between accommodated buckets as the R-tree determines distances between R-tree pages (in practice even much fewer because not all pairs of accommodated buckets have to be considered; only those located in mating hosting pages). The additional CPU cost in the MuX structure are the distance computations between the hosting pages. Because each hosting page stores more than 10 accommodated buckets there can be only one successful distance calculation per 10^2=100 distance calculations between accommodated buckets. MuX can in the worst case be 1% worse than the corresponding R-tree.

We optimize the capacity of the hosting pages of MuX such that they are I/O optimal. The capacity of the accommodated buckets is optimized such that they are CPU-optimal. Taken claim 1 and claim 2 together, we obtain a CPU performance which resembles a CPU-optimized R-tree and an I/O performance that resembles an I/O optimal R-tree (for both cases plus the overhead mentioned in the rationales of the claims).

Compared to conventional index join algorithms which traverse the indexes depth-first [7] or breadth-first [14], our new algorithm improves the performance with respect to CPU and I/O. The I/O effort is reduced by two ideas: The first idea is to use more cache for the point set *R* which is scanned in the outermost loop. The advantage is that in the case of a bad index selectivity the number of scans of the other point set *S* is minimized. Therefore, the I/O cost cannot become substantially worse than the I/O cost of a nested loop join. In the case of a good index selectivity, in the inner loop only those *S*-pages are loaded which are actually needed. Therefore, the performance cannot become substantially worse than a breadth-first or depth-first index traversal. For these extreme cases, we have always the performance of the best of the two worlds: nested loops or tree traversal. In the cases between these extremes, we combine the advantages of both paradigms and outperform them both clearly. The second idea leading to reduced I/O cost is that we use the page regions of the accommodated *R*-buckets to exclude hosting *S*-pages. While only I/O optimized pages are subject to I/O operations, the more selective bucket regions are used for excluding, leading to a clear advantage in the index selectivity. The CPU effort is minimized due to the optimization of the bucket size for minimum computational cost. Additionally, many distance computations between bucket regions are avoided, because buckets can only mate if their hosting pages mate, too.

7. Experimental evaluation

To show the superiority of our proposal over competitive techniques, we have performed an extensive experimental evaluation. For this purpose, we implemented our multipage index join algorithm. For comparison, we also implemented a similarity join algorithm using nested loops and a similarity join algorithm based on the R-tree spatial join (RSJ) algorithm [7] with three different scheduling and caching schemes.

The cache for the nested loop-join was assigned according to our optimization presented in section 4. All RSJ variants used a caching strategy discarding the page which will not be used for the longest time in the future. Note that, in contrast to usual paging algorithms applied in general-purpose operating systems, the join algorithm allows to exploit the knowledge of the

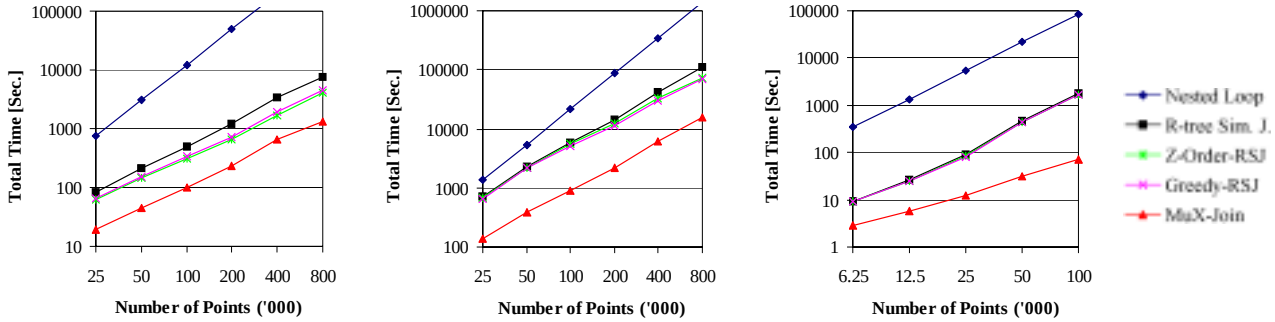

Figure 9. Application Data: Meteorology Data 9D (left), CAD Data 16D (middle), Color Images 64D (right)

page schedule in the future. The basic RSJ algorithm accesses the data pages of the index in a random order. The cache hit rate can be improved by accessing the pages of the index in an order preserving the spatial proximity. In [14], 4 different kinds of page ordering were proposed, including the Hilbert curve, and an improvement of the cache hit ratio of up to 50% was reported. We implemented a page scheduling strategy based on Z-ordering and a greedy optimization strategy which starts with an arbitrary page and accesses in each step the unprocessed page with the smallest distance to the last previously accessed pages. We will refer to the three variants as "*R-tree Similarity Join* (*RSJ*)", "*RSJ with Z-ordering optimization*", and "*RSJ with greedy optimization*." All algorithms were allowed to use the same amount of buffer memory (5% of the database size).

All our experiments were carried out on HP 9000/780 workstations under HPUX-10.20. We used a disk device with a transfer rate of 4 MByte/sec, a seek time of 5 msec, and latency time of 5 msec. Our algorithms do not exploit parallelism between CPU and I/O, which would be possible in all approaches. Therefore, our reported total query time corresponds to the sum of the CPU time and the I/O time. The index construction was not taken into account.

For our experiments, we used synthetic as well as real data. Our synthetic data sets consist of up to 800,000 uniformly distributed points in the unit hypercube with the dimensions 4 and 8. Our real-world data stem from three application domains: A CAD database with 16-dimensional feature vectors extracted from geometrical parts, a color image database with 64-dimensional feature vectors representing color histograms, and a meteorology database with 9-dimensional feature vectors generated by weather observation. In the similarity join, we used the Euclidean distance. Appropriate distance parameters ε for each data set were determined such that they are useful in clustering [9] and that each point of the data set is combined with a few other points on the average. That means in particular that we avoided in our experiments the extreme cases of no resulting pair (or in the case of self joins: each point is only a join mate of itself), or each point is combined with every other point.

Figure 8 shows our experiments on uniformly distributed point data. In the left diagram, the data space is 4-dimensional and an appropriate ε = 0.05 (i.e. in the result, each point has an average of 8.5 join mates). The nested loop join has the worst performance over all scales. With increasing database size, this technique is outperformed by all other techniques by increasing factors. For low-dimensional data spaces, the scheduling strategy in the R-tree similarity join plays a relatively important role. Therefore, the more sophisticated strategies which order the page accesses by Z-ordering or a greedy strategy improve the performance of the R-tree similarity join by factors up to 4.2. The clear winner over all database sizes is our new technique, the MuX-join. It outperforms the nested loop join up to 400 times and is up to 10 times faster than the R-tree similarity join. Even the improved R-tree join versions are outperformed with factors between 2.3 and 4.6. The diagram in the middle shows our experiments with an 8-dimensional data space (ε = 0.3; each point has an average of 22.3 join mates). In this dimension, the various R-tree join variants do not differ much. As the index selectivity begins to deteriorate in medium-dimensional data spaces, the nested loop join is much more competitive and is only for the largest database (800,000 points) outperformed by the three R-tree join variants. Our new technique, in contrast, outperforms the other techniques by a factor of 6.3 (over R-trees) and 8.1 (over nested loop) for the largest database size. For 100,000 points, the corresponding factors are 7.4 (over R-trees) and 3.1 (over nested loop). The diagram on the right side depicts the performance of the join algorithms with varying distance parameter ε (d = 8; n = 50,000). It is obvious that for very large ε the nested loop join must be the winner, because the join result combines each point with every other point, and the nested loop join has no additional index overhead. Therefore, the R-tree variants are clearly outperformed. As our new technique strictly limits the index overhead by an appropriate optimization of I/O as well as CPU, it is never clearly outperformed. Instead, the performance slowly approaches the performance of the nested loop join with increasing ε.

Our experiments on real application data depicted in figure 9 clearly confirm our experiments on uniform data. Partially, the improvement factors are even higher. The left diagram depicts the results on the 9-dimensional meteorology feature vectors (ε = 0.0001; 3.9 join mates per point). For the largest database size, our technique was 590 times faster than the nested loop join, 5.9 times faster than the R-tree similarity join, and 3.5 times faster than RSJ with the improved scheduling strategies. For the 16-dimensional CAD feature vectors (diagram in the middle; ε = 0.01; 7.5 join mates per point) our technique is up to 87 times faster than the nested loop join and between 6 and 7 times faster than the 3 R-tree similarity join variants. The right diagram shows the results on our color image database (ε = 0.0001; 1.1 join mates per point). For the largest database, our technique yields an improvement factor of 1203 over the nested loop join of 25 over all R-tree similarity join algorithms.

8. Conclusions

In this paper, we have proposed an analytical model and a performance study of the similarity join operation. In this context, a severe optimization conflict between CPU and I/O optimization has been discovered. To solve this conflict, we propose an

index architecture which allows a separate optimization of the CPU time and the I/O time. Our architecture utilizes large primary pages which are subject to I/O processing and optimized for this purpose. The primary pages accommodate a secondary search structure to reduce the computational effort. Our experimental evaluation has shown consistently good performance. Competitive approaches are outperformed by large factors. An open question for future work is the suitability of our secondary search structure. For simplicity, and in order to uniformly apply the same cost model for CPU and I/O optimization, we used minimum bounding rectangles for both, the primary and the secondary search structure. More sophisticated techniques, however, should have the potential to even improve our high speedup factors.

References

[1] Ankerst M., Breunig M. M., Kriegel H.-P., Sander J.: *OPTICS: Ordering Points To Identify the Clustering Structure*, ACM SIGMOD Int. Conf. on Management of Data, 1999.

[2] Agrawal R., Faloutsos C., Swami A. *Efficient similarity search in sequence databases.* Int. Conf. on Foundations of Data Organization and Algorithms, 1993.

[3] Böhm C., Braunmüller B., Breunig M. M., Kriegel H.-P.: *High Performance Clustering Based on the Similarity Join*, Int. Conf. on Inform. Knowledge Managem. (CIKM), 2000.

[4] Berchtold S., Böhm C., Jagadish H.V., Kriegel H.-P., Sander J.: *Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces*, IEEE Int. Conf. on Data Engineering (ICDE), 2000.

[5] Berchtold S., Böhm C., Kriegel H.-P.: *Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations*, Int. Conf. on Extending Database Technology (EDBT), 1998.

[6] Berchtold S., Böhm C., Keim D., Kriegel H.-P.: *A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*, ACM Symp. Principles of Datab. Syst. (PODS), 1997.

[7] Brinkhoff T., Kriegel H.-P., Seeger B.: *Efficient Processing of Spatial Joins Using R-trees*, ACM SIGMOD Int. Conf. on Management of Data, 1993.

[8] Ciaccia P., Patella M., Zezula P. *A cost model for similarity queries in metric spaces*. ACM Symposium on Principles of Database Systems (PODS), 1998.

[9] Ester M., Kriegel H.-P., Sander J., Xu X.: *'A Density Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise´*, 2nd Int. Conf. on Knowledge Discovery and Data Mining, 1996.

[10] Friedman J. H., Bentley J. L., Finkel R. A. *An algorithm for finding best matches in logarithmic expected time.* ACM Trans. on Mathematical Software, Vol. 3, No. 3, 1977.

[11] Faloutsos C., Barber R., Flickner M., Hafner J., et al.: *Efficient and Effective Querying by Image Content*, Journal of Intelligent Information Systems, Vol. 3, 1994.

[12] Guha S., Rastogi R., Shim K.: *CURE: An Efficient Clustering Algorithms for Large Databases*, ACM SIGMOD Int. Conf. on Management of Data, 1998.

[13] Gray J., Shenoy P. J.: *Rules of Thumb in Data Engineering*, IEEE Int. Conf. on Data Engineering (ICDE), 2000.

[14] Huang Y.-W., Jing N., Rundensteiner E. A.:*Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations*, Int. Conf. on Very Large Databases (VLDB), 1997.

[15] Hjaltasson G., Samet H.: *Ranking in Spatial Databases*, Symposium on Large Spatial Databases (SSD), 1995.

[16] Hattori K., Torii Y.: *Effective algorithms for the nearest neighbor method in the clustering problem*. Pattern Recognition, Vol. 26, No. 5, 1993.

[17] Jagadish H. V.: *A Retrieval Technique for Similar Shapes*, ACM SIGMOD Int. Conf. on Management of Data, 1991.

[18] Kamel I., Faloutsos C.: *Hilbert R-tree: An Improved R-tree using Fractals*. Int. Conf. on Very Large Databases, 1994.

[19] Koperski K. and Han J.: *Discovery of Spatial Association Rules in Geographic Information Databases*, Int. Symp. on Large Spatial Databases (SSD), 1995.

[20] Knorr E.M. and Ng R.T.: *Finding Aggregate Proximity Relationships and Commonalities in Spatial Data Mining*, IEEE Trans. on Knowledge and Data Engineering, 8(6), 1996.

[21] Knorr E.M. and Ng R.T.: *Algorithms for Mining Distance-Based Outliers in Large Datasets*, Int. Conf. on Very Large Databases (VLDB), 1998.

[22] Kornacker M.: *High-Performance Extensible Indexing*, Int. Conf. on Very Large Databases (VLDB), 1999.

[23] Koudas N., Sevcik C.: *Size Separation Spatial Join*, ACM SIGMOD Int. Conf. on Management of Data, 1997.

[24] Koudas N., Sevcik C.: *High Dimensional Similarity Joins: Algorithms and Performance Evaluation*, IEEE Int. Conf. on Data Engineering (ICDE), Best Paper Award, 1998.

[25] Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: *Fast Nearest Neighbor Search in Medical Image Databases*, Int. Conf. on Very Large Data Bases (VLDB), 1996.

[26] Lo M.-L., Ravishankar C. V.: *Spatial Joins Using Seeded Trees*, ACM SIGMOD Int. Conf. Management of Data, 1994.

[27] Lo M.-L., Ravishankar C. V.: *Spatial Hash Joins*, ACM SIGMOD Int. Conf. on Management of Data, 1996.

[28] Lomet D., Salzberg B.: The hB-Tree: *A Multiattribute Indexing Method with Good Guaranteed Performance*, ACM Transactions on Database Systems (TODS), Vol. 15, No. 4, 1990.

[29] Patel J.M., DeWitt D.J., *Partition Based Spatial-Merge Join*, ACM SIGMOD Int. Conf. on Management of Data, 1996.

[30] Papadopoulos A., Manolopoulos Y. *Performance of nearest neighbor queries in R-trees.* Int. Conf. Datab. Theory, 1997.

[31] Papadopoulos A., Manolopoulos Y. *Similarity query processing using disk arrays.* ACM SIGMOD Int. Conf. on Management of Data, 1998.

[32] Robinson J. T.: *The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes*, ACM SIGMOD Int. Conf. on Management of Data, 1981.

[33] Sibson R.: *SLINK: an optimally efficient algorithm for the single-link cluster method*, The Computer Journal 16(1), 1973.

[34] Shim K., Srikant R., Agrawal R.: *High-Dimensional Similarity Joins*, Int. Conf. on Data Engineering (ICDE), 1997.

[35] Weber R., Schek H.-J., Blott S.: *A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*, Int. Conf. Very Large Databases, 1998.