Joining Massive High-Dimensional Datasets [*]

Tamer Kahveci Christian A. Lang Ambuj K. Singh
Department of Computer Science
University of California, Santa Barbara, CA 93106
{tamer,clang,ambuj}@cs.ucsb.edu

Abstract

We consider the problem of joining massive datasets. We propose two techniques for minimizing disk I/O cost of join operations for both spatial and sequence data. Our techniques optimize the available buffer space using a global view of the datasets. We build a boolean matrix on the pages of the given datasets using a lower bounding distance predictor. The marked entries of this matrix represent candidate page pairs to be joined. Our first technique joins the marked pages iteratively. Our second technique clusters the marked entries using rectangular dense regions that have minimal perimeter and fit into buffer. These clusters are then ordered so that the total number of common pages between consecutive clusters is maximal. The clusters are then read from disk and joined. Our experimental results on various real datasets show that our techniques are 2 to 86 times faster than the competing techniques for spatial datasets, and 13 to 133 times faster than the competing techniques for sequence datasets.

1 Introduction

Current databases contain a wide variety of information. This information can be stored as both spatial data and sequence data. Geographic Information Systems (GIS) are an important example of spatial data. GIS usually store data such as points, lines, or polygons. Stock market, video, text, genome data are examples of sequence data. The distance/similarity measure for these data varies based on the application and the data type. Some of the distance measures currently in use are Euclidean distance [1, 12, 16, 26], other vector norms like L_1 and L_∞ [2], shift/scale invariant distance measures [13, 27], edit distance [8, 25, 35], and score-matrix based similarity [21].

The sizes of spatial and sequence datasets are growing rapidly. Excessive amounts of data make similarity search challenging, incurring a large amount of disk I/O and CPU cost. An important primitive on such datasets is the *join*, also called *similarity join*, operator. A join query asks to find all similar data pairs from two datasets. Two example join queries are:

- *Spatial Join Query :* Find all hotels in California that are within three miles of a recreation area.
- *Sequence Join Query :* Find all pairs of companies from the New York Exchange and the Tokyo Exchange that have similar closing prices for one month.

The former query joins two spatial datasets, namely *hotels* and *recreation areas*. The latter query joins two sequence datasets for two stock markets. Current techniques for join queries are restrictive in that they usually consider only spatial data and Euclidean distance. A comparison of these techniques is presented in [37].

Join queries incur two major costs: 1) I/O cost, and 2) CPU cost. Disk I/Os are usually the bottleneck for dataset joins when the datasets are much larger than the available buffer. This is because most of the pages usually need to be accessed multiple times. If the whole dataset does not fit into buffer, a page that will be used later may be removed from the buffer incurring multiple random disk I/Os. Furthermore, the CPU performance and memory bandwidth have increased by 50% each year (a.k.a. Moore's law), while memory latency has stayed roughly equal. Therefore, the cost of memory access increases exponentially with respect to CPU cost [34].

In this paper, we consider the join problem for massive datasets when the similarity measure can be any metric. The dataset can be composed of spatial data as well as sequence data. We approximate each page of the dataset using a Minimum Bounding Rectangle (MBR). Given two datasets, we build a boolean matrix on the pages of these datasets using a lower bounding distance predictor. Each entry, $m_{i,j}$, of this matrix associates the i^{th} page of the first dataset to the j^{th} page of the second dataset. If an entry $m_{i,j}$ is marked, then a join of the data in the (i, j) page pair potentially contributes to the join of these datasets. Therefore, these page pairs need to be read from disk and joined.

We propose two techniques for joining such datasets. Our first technique restricts the join to the marked entries. It processes the page pairs by iteratively reading blocks of pages. Our second technique uses a cluster-based scheme to further

[*]Work supported partially by NSF under grants EIA-0080134, IIS-9877142, and DBI-0213903.

reduce the I/O cost. We propose two algorithms to cluster the marked entries of the prediction matrix such that the resulting clusters are dense, have minimal perimeter, and their contents fit into buffer. The first clustering technique simply allocates equal space in the buffer for both sequences. The second clustering technique works in a bottom-up manner. It chooses a single point from a dense region. This point defines the initial cluster. Later, this cluster is expanded to cover more points, minimizing the I/O cost, until the pages corresponding to that cluster fill the whole buffer. The points contained in this cluster are then removed from the prediction matrix and a new cluster is created iteratively on the rest of the prediction matrix. Once the clusters are determined, we define an order between these clusters such that the total amount of data overlap between consecutive clusters is maximal. Later, the clusters are read in this order and joined.

Experimental results show that our techniques are 2 to 86 times faster than the competing techniques for spatial datasets, and 13 to 133 times faster than the competing techniques for sequence datasets.

Besides the efficiency improvements for join operations, the main contribution of this paper is the introduction of a prediction matrix to speedup join operations under arbitrary distance measures. The speedup is achieved using prediction, clustering, and buffer reuse.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 discusses the subsequence join problem. Section 4 presents a brief overview of our techniques. Section 5 explains the construction of a global view of datasets. Section 6 presents our first technique that uses this global view. Section 7 discusses two clustering techniques to minimize I/O cost. Section 8 discusses cluster ordering. Section 9 presents the experimental results. We end with a brief discussion in Section 10.

2 Related work

Joining two datasets is a costly operation. Current techniques reduce this cost by pruning pairs of data points that do not appear in the final join. They can be classified into two groups based on the data structures they use: 1) no index is built on the datasets, or 2) an index is built on at least one of the datasets. Another way of classifying current techniques is based on the type of data they use: 1) spatial data, or 2) point data. Note that point data can also be considered as spatial data but the inverse is not true.

2.1 Joining without index structures

One of the simplest algorithms for joining datasets without index structure is *Nested Loop Join* (see [39]). Nested Loop Join iteratively reads an object from the first dataset. For each such object, it sequentially scans all the objects in the second dataset. The performance of Nested Loop Join can be improved by reading blocks of B-2 pages from the first dataset at each iteration, where B is the number of pages that buffer can hold. For each such block, the second dataset is sequentially read one page at a time. This technique is called *Block Nested Loop Join* (*NLJ*).

Sort Merge Join [5, 18, 33] sorts both of the datasets based on the join attribute and merges them. Orenstein [36] uses Z-curves to represent spatial objects. The bit-representation of the Z-value of objects is then used to find out whether the objects overlap.

Hash Join (see [39]) assigns objects in both datasets to buckets using the same hash functions. Later, the objects in the same bucket are joined. A variation of Hash Join, called *Grace Hash Join*, works in two phases. In the first phase, both datasets are mapped to buckets that reside on disk. In the second phase, one bucket is read at a time and a hash table is constructed from it (see [20] for a comparison).

Partition-Based Spatial Join (PBSM) [38] splits the data space into tiles using a grid. The tiles of each data set are then assigned to P partitions using a round robin or hashing scheme. The partitions corresponding to the same locations are then checked to find the candidate objects that may contribute to the final join. Next, objects are read from the first dataset into buffer, and the second dataset is sequentially scanned to find the actual join results.

Scalable Sweeping-based Spatial Join (SSSJ) [3] starts by sorting the objects according to their lower values in one of the dimensions in ascending order. Later, the objects are scanned sequentially in this order. For each object, the intersecting objects are found by scanning the other dataset. Gurret and Rigaux [19] show that the performance of join techniques based on plane sweep algorithms is better when one of the datasets is indexed using an R-tree.

Koudas and Sevcik [28, 29] propose to partition the space into finer grained cells by iteratively splitting it into two in each dimension. If the MBR of an object intersects a split line at the k^{th} iteration, it is assigned to level k. Once the objects are assigned to levels, the objects in each level are joined with objects in the same or lower levels. Dittrich and Seeger [14] improve the performance of this technique by introducing a modest data replication.

The *Epsilon grid ordering technique* [6] considers the distance join problem for point data. This technique splits the data space into grid cells and assigns each data point to the cell that contains it. Later, these cells are ordered by assigning priorities to dimensions and constructing a lexicographic order. A pair of data points are considered a candidate if one of the points is within the ϵ interval of the other point based on the *epsilon grid ordering*. Although it is very efficient for point data, this technique fails to define an order for spatial data when an object spans more than one cell. It also incurs an additional cost for reordering the whole dataset according to the epsilon grid order. Since sequence datasets can not be reordered, the efficiency of this technique degrades for such datasets.

Two of the earlier techniques closer to ours in spirit are [11] and [42]. Chan and Ooi [11] model the page access sequence as a concatenation of segments. Each segment is a sequence of pages that does not contain pages that do not contribute to the join. The experimental results show that the number of page accesses of this technique is only 10% less than the naive greedy technique. Xiao, Zhang, and Jia [42] propose to construct a join graph. The vertices of this graph represent the data objects, and the edges of this graph are labelled with the size of the object pairs that need to be joined. This graph is then mapped to a join matrix. This matrix is diagonalized and partitioned into submatrices to fit into buffer. According to experimental results, the number of disk I/Os of this technique is 35% less than the simple technique without clustering. Both of these techniques have three major drawbacks: 1) they have a high time complexity, which is $O(N^2)$, where N is the number of pages in the dataset. 2) Either the data object or the pages need to be permuted. Therefore, most of the page accesses incur random disk seek cost unlike NLJ. 3) Their performance drops quickly as query selectivity increases.

Recently, Lang and Singh [30] proposed an acceleration technique for all-NN joins based on radius predictions. Two radius estimators are used to determine the join pages. These are then read with a modified NLJ algorithm. Even though the algorithm can degenerate to NLJ in the worst case, the experiments indicate that the accelerated algorithm is 3–4 times faster than NLJ on real data.

2.2 Joining using preconstructed index structures

Brinkhoff, Kriegel, and Seeger [10] propose to use R*-trees for spatial joins. Their approach traverses the trees in depth first order, expanding children of a node pair if these nodes intersect. Further optimizations are achieved by using either local plane sweep order, pinning, or local z-order. The performance of similarity joins can also be improved by decoupling CPU and I/O optimizations [7]. BFRJ [23] traverses the R-tree in Breadth First Search order. The authors show that this ordering reduces the number of accesses to the pages of the R-tree.

Lo and Ravishankar [31] consider the spatial join problem when the R-tree is precomputed on only one of the data sets. They propose to build a *seeded tree* on the other data set on the fly. The seeded trees can also be used to construct the buckets for spatial hash joins [32]. However, a spatial object may overlap with more than one hash bucket. In this case, either the spatial objects are assigned to multiple buckets or a bucket may join with more than one other bucket.

Hjaltason and Samet [22] give an index based incremental algorithm for *distance join* and *distance semijoin* of spatial datasets. The authors use R*-trees to index the objects. Later, the index structures are traversed keeping the partial results in a priority queue similar to the incremental NN-search algorithm.

Shim, Srikant, and Agrawal [41] propose an index structure called ϵ-kdB tree for joining high-dimensional point data. This index structure is a hierarchical tree where the root node corresponds to the entire search space. Each internal node splits the search space corresponding to that node into tiles of width ϵ in one dimension, where ϵ is a given similarity threshold. The partitions obtained by splitting an internal node determine the children of that internal node.

The performance of the join operation can be improved using conservative and progressive approximations to spatial objects [9]. A conservative approximation of an object is a simple shape (e.g. rectangle, circle, convex hull) that encloses that object. A progressive approximation of an object is a simple shape that is enclosed within that object.

3 Subsequence join

Current join techniques focus on spatial and point data (relational datasets can be considered point data as well.) We introduce a new type of join operation, called *subsequence join* which works for sequence data. A subsequence join query asks for all similar subsequence pairs of a prespecified length from two sequence datasets. A typical subsequence join query can be

"Find all pairs of companies from the New York Exchange and the Tokyo Exchange that have similar closing prices for one month." or

"Find all similar genome substring pairs of length 500, one from Human Genome and the other from Mouse Genome."

Formally, let $D_1 = \{A_1, A_2, \cdots, A_m\}$ and $D_2 = \{B_1, B_2, \cdots, B_n\}$ be two sequence datasets, where m and n are integers, and A_i and B_j are sequences for $1 \leq i \leq m$, $1 \leq j \leq n$, then

$\{(a, b)\} \in JOIN(D_1, D_2, w, \epsilon)$, iff

1. $\exists A_i \in D_1$: a is a subsequence of A_i, and
2. $\exists B_j \in D_2$: b is a subsequence of B_j, and
3. $|a| = |b| = w$, and
4. $distance(a, b) \leq \epsilon$.

Typical applications for such queries are in the area of stock market data, video data, biological data, and text data.

Current join techniques do not work well for subsequence join queries since sequence data contains many overlapping subsequences unlike point and spatial data. Most of the current join techniques for spatial data are based on the assumption that similar data are located closely on the disk. Similar subsequences can be brought closer on the disk in either of two ways: splitting or replicating. Splitting a sequence into nonoverlapping subsequences and reordering these sequences based on their proximity destroys the subsequences that overlap with the split location. Hence, the subsequences can not be reordered on disk. On the other hand, replicating all possible subsequences increases the memory usage dramatically, thus reduces the performance. For example, replicating all the subsequences of length w increases memory usage by a factor of w.

One simple way to overcome these problems might be to build an index structure (e.g. R-tree) on all possible subsequences. However, this solution poses two problems. First, the size of this index structure will be much larger than the dataset itself since many subsequences overlap. Second, the nodes of the index structure may contain subsequences that are far away from each other on disk. This would cause excessive amount of random seek cost for accessing a single node of the index structure.

4 An overview of our technique

Join queries incur two major costs: 1) I/O cost for reading pairs of pages from disk, and 2) CPU cost for joining the objects once the pages are read.

Throughout this paper, we will assume a finite buffer of B pages and a linear disk model. It is obvious to extend these techniques to other disk models. We will use LRU as the page replacement policy due to its simplicity and effectiveness.

Given two datasets R and S, our technique starts by constructing a boolean *prediction matrix* M for these datasets using a lower bounding distance predictor. Each entry, $m_{i,j}$, of this matrix associates the i^{th} page of R to the j^{th} page of S. If an entry $m_{i,j}$ is marked, then the join of the data in the (i, j) page pair potentially contributes to the join of these datasets.

We first propose a simple algorithm which is an improvement over NLJ. This technique uses the prediction matrix as a filter step to NLJ. Our second technique clusters the marked entries of the prediction matrix, such that each cluster fits into buffer and contains as many marked entries as possible. We propose two clustering strategies. The first clustering technique aims to minimize the total number of I/Os. The second clustering technique considers the random seek cost and sequential access cost. It aims to minimize the total disk I/O cost. After the clusters are determined we order the clusters such that consecutive clusters share as many common pages as possible. This ordering reduces the I/O cost by increasing the buffer reuse.

5 Prediction matrix: a global view

5.1 Construction of the prediction matrix

We assume that the datasets are indexed prior to join operation. The index structure used depends on the type of the data. Table 1 shows the index structure, the distance measure between data, and the lower bounding distance measure between the index nodes used for each data type. For point and spatial data, the actual data is stored only at the leaf level nodes of the R*-tree. Once the R*-tree is constructed, the data objects are sorted so that the contents of each leaf level MBR appear contiguously on disk. For sequence data, these properties are already inherent in both MR-index and MRS-index (i.e. each MBR contains a contiguous disk block). The capacity of each MBR is set to one page size for simplicity in all of these index structures.

```
/* Let $M = (m_{i,j})$ be the prediction matrix (all entries initially 0). */
Algorithm PM($R, S, \epsilon$)
  1. $\varepsilon_R = \varepsilon_S = \emptyset$ /* Initialize active sets */
  2. FILTER(R,S);
  3. Extend all the MBRs in $R \cup S$ by $\epsilon/2$ in all directions.
  4. $E$ := ENDP(R) $\cup$ ENDP(S); /* Find the set of end points */
  5. While $E \neq \emptyset$
     (a) $p$ := EXTRACT_MIN(E);
     (b) If BOX(p) $\in R$
         /* The MBR of the current point is in set $R$ */
         i. If LEFT(p)
            A. Insert BOX(p) into $\varepsilon_R$;
            B. For all BOX(q) $\in \varepsilon_S$
                  If BOX(p) and BOX(q) intersect
                    If LEAF(p) and LEAF(q)
                      $m_{p,q}$ := 1;
                    else
                      PM(Children(p),Children(q),$\epsilon$);
         ii. If RIGHT(p)
            A. Remove $p$ from $\varepsilon_R$;
     (c) If BOX(p) $\in S$
         /* The MBR of the current point is in set $S$ */
           Symmetric to Step 5.b.
```

Figure 1. The algorithm for constructing the prediction matrix.

Given the MBRs of the index structures on the datasets and a threshold ϵ for the join condition, the prediction matrix is constructed in two phases: 1) The MBRs are extended by $\epsilon/2$ in all directions, and 2) a hierarchical plane sweep algorithm is used to find the intersecting MBRs. If two MBRs intersect, the lower bounding distance between these MBRs is less than ϵ and the corresponding matrix entry is marked.

Figure 1 presents the algorithm that constructs the prediction matrix. The algorithm takes two datasets, R and S, and a threshold ϵ as the input. It starts by initializing the active sets for R and S to empty sets (Step 1). Then, the MBR sets R and S are filtered using a fine-grained filtering technique in Step 2 (We will explain the filtering algorithm later in this section.). The end points of the remaining MBRs in R and S are extended by $\epsilon/2$ (Step 3), and stored in set E (Step 4). Now, all the end points are processed in ascending first-coordinate order. If the current MBR belongs to R, then Step 5.b is invoked. If the current end point is the leftmost point of an MBR, then its MBR is inserted into the active MBR list ε_R. Step 5.b.i checks whether the distance between its MBR and the active MBRs in ε_S is less than the threshold. If it is, then the levels of the boxes are checked. If both MBRs are at leaf level, then the corresponding entry in the prediction matrix is set to 1. Otherwise, their children are inspected similarly for intersection. If the current end point is the rightmost point of an MBR, then it is removed from the active MBR list ε_R (Step 5.b.ii). The algorithm proceeds similarly if the current MBR belongs to S (Step 5.c).

Figure 2 shows how the filtering at Step 2 of the prediction matrix construction algorithm works. Let R and S be two index node MBRs with children R_i and S_i, $1 \leq i \leq 6$.

Data Type	Index structure	Actual distance measure	Lower bounding distance measure
Point data	R*-tree [4]	Any vector norm (e.g. $\|\|.\|\|_2$)	Same
Spatial data	R*-tree [4]	Any vector norm (e.g. $\|\|.\|\|_2$)	Same
Time series data	MR-index structure [26]	Any vector norm (e.g. $\|\|.\|\|_2$)	Same
String data	MRS-index structure [25]	Edit distance	Frequency distance

Table 1. Index structures, distance measures, and the lower bounding distance measures used for different data types.

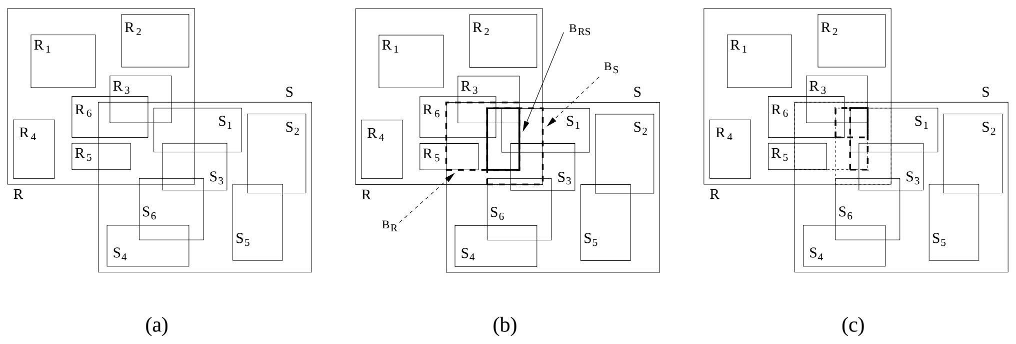


Figure 2. (a) Two index node MBRs R and S, and their children R_i and S_i, $1 \leq i \leq 6$. (b) First and (c) second iteration of the filtering technique on R and S. The dashed rectangles show the regions that are used for filtering.

If no filtering is done, then the intersection of 6 × 6 = 36 children MBR pairs must be checked. Brinkhoff et al. [10] propose to filter out the MBRs that do not intersect with the intersection of R and S. As a result of this filtering technique, only R_3, R_5, R_6, S_1, S_3, and S_6 remain (i.e. 3 × 3 = 9 MBR pairs). We propose a better filtering technique. Our technique is based on the following observation: Let B be the rectangle formed by the intersection of R and S. If R_i and S_j intersect, then R_i intersects with $B \cap S_j$ and S_j intersects with $B \cap R_i$. In order to use this fact in an efficient manner, we define B_R as the MBR that covers $B \cap R_i$, for all i. Similarly, B_S = the MBR that covers $B \cap S_i$, for all j. We define $B_{RS} = B_R \cap B_S$. Both B_R and B_S are shown using dashed lines, and B_{RS} is shown using a bold rectangle in Figure 2(b). We filter out all R_i and S_j that do not intersect with B_{RS}. As a result of this, only R_3, R_6, S_1, and S_3 remain for testing (i.e. 2 × 2 = 4 MBR pairs). This is one level of our filtering algorithm. We then update $R_i := R_i \cap B_{RS}$, for unfiltered R_i, and update R as the MBR that covers these R_i. A similar update is also applied to S and all S_j. Figure 2(c) shows the second iteration of our filtering algorithm on the same example. After the second iteration, only R_3 and S_1 remain for further inspection (i.e. 1 × 1 = 1 MBR pairs). We repeat this process until no further changes occur to B_R and B_S or τ iterations are performed, where τ is a reasonable upper bound to the number of iterations. We choose τ = 5 as default value. We assert such an upper bound on the filtering algorithm to guarantee that it runs in linear time. Since $B_{RS} \subseteq B$, it is guaranteed that our filtering technique does not perform any worse than that used in [10].

5.2 Analysis of prediction matrix construction

Since the end points of the MBRs for each data set are sorted once prior to prediction matrix construction, preprocessing incurs $O(n \log n)$ time. The filtering algorithm in Step 2 takes $O(n)$ time, where n is the number of MBRs in $R \cup S$, since it scans the MBRs at most $\tau = 5$ times. The plane sweep algorithm in Step 5 takes $O(n + A)$ time, where A is the number of marked entries, since each MBR has two end points and each end point is processed only once. Therefore, the total time complexity of this algorithm is $O(n \log n + A)$. This is equivalent to $O(\frac{|R|+|S|}{p} \cdot \log \frac{|R|+|S|}{p} + A)$, where $|R|$ and $|S|$ are the sizes of R and S, and p is the page size.

Theorem 1 states the correctness of the algorithm:

Theorem 1 (Correctness) *Let R and S be two datasets. Let $M = (m_{i,j})$ be the prediction matrix for the similarity join of R and S. If page i of R and page j of S contain a tuple pair in the result of the join operation, then $m_{i,j}$ = 1.*

If an entry of the prediction matrix is set to zero, then the corresponding disk page pair do no contain any result pair in the join operation. Therefore, the join operation does not need to compare the tuples in these page pairs.

6 A naive technique: pm-NLJ

Once the prediction matrix is determined, only the marked page pairs need to be read for the join operation. A simple improvement over the NLJ algorithm restricts the page pairs to these marked entries. We call this technique *prediction matrix-NLJ (pm-NLJ)*. It is given in Figure 4.

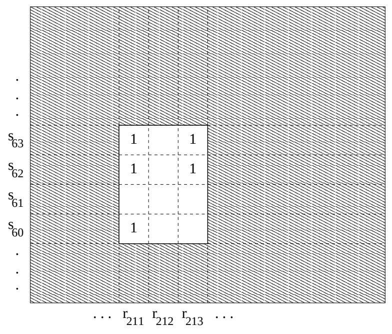


Figure 3. The illustration of a sample prediction matrix for datasets R and S. A small region within the prediction matrix is presented in more detail.

```
/* Let $S$ and $R$ be the datasets that will be joined, and $B$ be the buffer size
in pages. Assume that the size of $S$ is smaller than that of $R$. */
Algorithm pm-NLJ($R, S, B$)
    If all the marked pages of $S$ fit into buffer,
        1. Read all of them into buffer;
        2. Iteratively, read blocks of marked pages of $R$ into rest of buffer
           space and compute join;
    else
        Iteratively, read one marked page from $R$ and blocks of $B - 1$
        marked pages of $S$ and compute join;
```

Figure 4. Prediction matrix-NLJ algorithm.

Example 1 *Consider the prediction matrix in Figure 3. Assume that the buffer size is 5 pages. Simple NLJ algorithm iteratively reads all r_i. For each r_i, NLJ scans all s_j. This causes I/O thrashing since sequential scan sweeps the buffer as new pages are read. In order to join the pages in the unshaded region of Figure 3, NLJ performs 15 disk I/Os (read r_{211}, r_{212}, and r_{213} once, s_{60}-s_{63} three times.). On the other hand, for pm-NLJ, 7 disk I/Os are enough for the unshaded region (First read r_{211}, s_{60}, s_{62}, and s_{63}, then read r_{213}, s_{62} and s_{63}.).* □

In the following lemma, we calculate the minimum number of disk I/Os for pm-NLJ for each such subregion of the prediction matrix.

Lemma 1 *Assume that a cluster of the prediction matrix contains e marked entries, r marked rows and c marked columns, then pm-NLJ performs at least $e + min\{r, c\}$ disk I/Os for that cluster.*

Proof: omitted (cf. [24]) □

For the example in Figure 3, r = 3, c = 2, and e = 5. The total number of disk I/Os is 5 + $min\{2, 3\}$ = 7.

Note that NLJ is the same as pm-NLJ when all the entries of the prediction matrix are set to 1. Therefore, using Lemma 1, the number of pages read by NLJ can be calculated as $r' \cdot c' + min\{r', c'\}$, where r' and c' are the number of rows and columns of the prediction matrix ($r' \geq r$ and $c' \geq c$). Hence, we conclude that pm-NLJ saves $(r' \cdot c' - e) + (min\{r', c'\} - min\{r, c\})$ disk reads and $(r' \cdot c' - e)$ CPU comparisons over NLJ.

7 Minimizing the I/O cost

Although restricting the join operation to the marked entries reduces the number of disk reads, reading the pages in NLJ order still causes I/O thrashing. One can reduce the number of disk I/Os by clustering the marked entries of the prediction matrix, and reading the pages in each cluster separately. For example, assume that the unshaded region in Figure 3 is such a cluster. Assume that the buffer size is 5 pages. In order to join the page pairs within this cluster, pages r_{211}, r_{213}, s_{60}, s_{62}, and s_{63} are read into the buffer. The unshaded region can be joined without further disk I/Os since all marked page pairs in this region are in the buffer. Using such a cluster reduces the number of disk I/Os to 5. Lemma 2 establishes the number of disk reads needed to join a cluster.

Lemma 2 *Assume that a cluster of the prediction matrix contains r marked rows and c marked columns. Let B be the buffer size. If $r + c \leq B$ then $r + c$ disk I/Os are sufficient to join the pages within this cluster.*

Proof: The proof of Lemma 2 follows from the fact that reading the pages corresponding to marked rows and columns is sufficient to join all the marked page pairs in that cluster. □

7.1 Minimizing the number of disk I/Os

One of the primary factors that affect the cost of joins is the number of disk I/Os. Theorem 2 defines the amount of disk I/Os saved by reading one cluster at a time over pm-NLJ.

Theorem 2 (I/O Savings) *Assume that a cluster of the prediction matrix contains e marked entries, r marked rows and c marked columns. Let B be the buffer size. If $r + c \leq B$, then joining this cluster separately reduces the number of disk I/Os by at least $e - max\{r, c\}$ over pm-NLJ.*

Proof: follows from lemma 1 and 2. □

Two important observations follow from Theorem 2:

1. The amount of I/O savings increases as $max\{r, c\}$ decreases. If we assume that $r + c = B$ is fixed (i.e. we consume all the buffer for each cluster), then $max\{r, c\}$ is minimized when $r = c = B/2$. In other words, the performance for a cluster is improved if the cluster contains equal number of marked rows and columns.

2. The amount of I/O savings increases as the number of marked entries (e) within a cluster increases. Intuitively, this means that dense clusters result in better performance improvements.

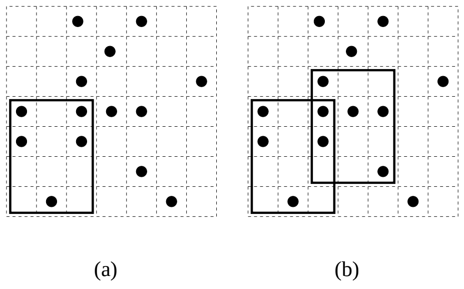


Figure 5. The first two iterations of square clustering algorithm on a prediction matrix. The buffer size is 6 pages in this example. The black dots correspond to marked entries of the prediction matrix.

Our first clustering technique follows from these observations: it iteratively forms clusters that have an equal number of marked rows r, and columns c, where $r + c = B$, except at the boundaries. Therefore, we call this technique the *Square Clustering (SC)* technique.

Figure 5 depicts the cluster boundaries formed by the first two iterations of SC. The black dots correspond to marked entries of the prediction matrix. We assume that the buffer size is 6 pages in this example. In Figure 5(a), SC starts by finding the first cluster which has all the following properties:

1. it has an equal number or marked rows and columns,
2. the sum of the number or marked rows and columns within that cluster is equal to the buffer size, and
3. its width is as small as possible.

The first condition maximizes the I/O savings for that cluster by minimizing $max\{r, c\}$ (see Theorem 2.). The second condition ensures that the buffer space is not wasted. Finally, the last condition selects pages corresponding to one of the datasets from a small range since the width of the cluster corresponds to the span of the pages that will be read from disk. Once the first cluster is determined, SC removes the marked entries in this cluster from the prediction matrix and determines the next cluster in the same way. □

Figure 6 presents the SC algorithm. The run time complexity of the SC algorithm is $O(e)$, where e is the number of marked entries of the prediction matrix. This is because two passes over all marked entries suffice for the SC algorithm to determine all the clusters: one column-wise pass to determine the *CANDIDATE* points, and one row-wise pass to determine the *ASSIGNED* points. The space complexity of SC is $O(e)$, since the prediction matrix stores only the marked entries in sparse matrix format.

7.2 Minimizing the random seek cost

The cost of reading a set of disk pages highly depends on the location of these pages. This is because the random seek cost is much larger than a sequential page transfer cost. Therefore, reading a set of pages is cheaper if they are located close to each other.

```
/*Let $M$ be the prediction matrix.
Let $B$ be the buffer size.
All the marked entries in $M$ are initialized to UNASSIGNED.*/
Algorithm SQUARE-CLUSTER($M, B$)
  1. $id := 0$; $c := 0$;
  2. While there are UNASSIGNED entries
      (a) Set the UNASSIGNED entries in the first $B/2$-$c$ columns as
          CANDIDATE;
      (b) $r := min\{numberofCANDIDATErows, B/2\}$;
      (c) Set the CANDIDATE entries in the first $r$ CANDIDATE rows
          to ASSIGNED;
      (d) $c :=$ the number of ASSIGNED columns;
      (e) While $c + r < B$
            i. Set the marked entries in the next UNASSIGNED column to CANDIDATE
           ii. Update rows;
      (f) Insert the ASSIGNED entries to the cluster $id$ and remove
          them from prediction matrix;
      (g) $c :=$ the number of CANDIDATE columns;
      (h) $id$++;
```

Figure 6. Square clustering algorithm.

Our second clustering algorithm minimizes the I/O cost by considering the proximity of the pages as well as the shape and the density of the clusters. Therefore, we call this technique the *Cost-based Clustering (CC)* technique.

Figure 7 illustrates the construction of a cluster by the CC algorithm. The buffer size is 5 pages in this example. The densest region of the prediction matrix is selected as the *seed region*. This is achieved by computing a 2-dimensional density histogram on the matrix. CC randomly chooses a marked entry as *seed entry* in this region in Figure 7(a), and creates a (1×1) cluster that tightly covers this entry. Then, it iteratively chooses an entry that minimizes the increase in the I/O cost of reading the pages for that cluster and the new entry. The cluster is then extended to cover the new entry. This process continues until the contents of the cluster fill the buffer (Figure 7(d)).

Figure 8 presents the CC algorithm. Fagin's threshold algorithm (TA) [15] in Step 3.c is used because the two cluster expansion directions (horizontally and vertically) can be viewed as two lists sorted by increasing I/O cost. The next marked entry to add to the cluster is the one that incurs the lowest combined I/O cost when added. TA essentially gives a way of determining this minimum without inspecting all elements in the two lists.

Step 2 of the CC algorithm runs in $O(e)$ time, where e is the number of marked entries in the prediction matrix. Steps 3.a and 3.b are $O(1)$ operations. Each iteration of 3.c takes $O(\sqrt{e})$ time (see [15] for a detailed analysis of TA). Since the threshold algorithm is ran once for each marked entry in

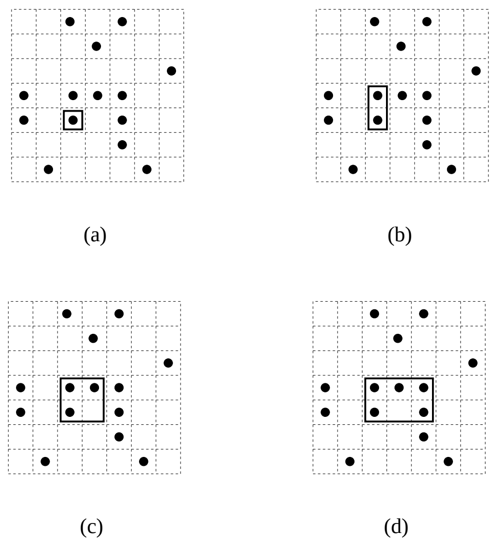


Figure 7. A trace of the CC algorithm.

the worst case, the worst case total time complexity of 3.c is $O(e^{3/2})$. Since each marked entry is assigned to a cluster once, the total cost of 3.d and 3.e are $O(e)$. Hence, the total time complexity of the CC algorithm is $O(e^{3/2})$ since Step 3.c dominates the total cost. The space complexity of the CC algorithm is only $O(e)$ since storing the marked entries in sparse matrix format suffices. Since CC is CPU-intensive, we will use it only as a lower bound of the total I/O cost.

8 Cache reuse

Once the clusters in the prediction matrix are determined (either with SC or CC), they are read and processed one at a time. For each such cluster

1. the marked pages of both datasets are read using optimal disk scheduling [40] and
2. the marked page pairs are joined. Since the content of each cluster fits into buffer, the join operation is always performed in memory.

In Section 7, we showed that each cluster is optimized in order to minimize the I/O cost per cluster. Further optimization can be obtained by maximizing the number of pages reused in the buffer. This improvement is achieved by scheduling the order of clusters in a clever way.

Example 2 *Assume that Figure 9 shows a prediction matrix and a set of clusters $\{C_1, C_2, C_3, C_4, C_5\}$ defined on it. Let the buffer size be 5 pages. The clusters contain the following pages: $C_1 = \{r_2, r_3, s_5, s_6, s_7\}$, $C_2 = \{r_2, r_3, r_4, s_3, s_4\}$, $C_3 = \{r_6, r_7, s_4, s_5\}$, $C_4 = \{r_3, r_4, r_5, s_2, s_3\}$, and $C_5 = \{r_1, s_1\}$. The total number of pages that need to be read for all the clusters is $\sum_{i=1}^{5} |C_i| = 21$. Now, consider the following scenarios:*

```
/*Let $M$ be the prediction matrix.
Let $B$ be the buffer size.
All the marked entries in $M$ are initialized to UNASSIGNED.*/
Algorithm COST-CLUSTER($M$, $B$)
  1. $id := 0$;
  2. Build a $10 \times 10$ histogram by finding the number of marked entries
     in each histogram bucket;
  3. While there are UNASSIGNED entries
     (a) Randomly choose a seed $s$ in the bucket that has the most
         marked entries;
     (b) Construct a $1 \times 1$ cluster $C$ on the prediction matrix that tightly
         covers $s$;
     (c) While there are UNASSIGNED entries and the number of
         pages in $C$ is less than $B$
           i. Find the marked entry $s'$ in the prediction matrix that
              minimizes the I/O cost of reading the contents of $C$ and
              $s'$ using Fagin's threshold algorithm;
          ii. Expand $C$ to cover $s'$;
     (d) Assign all the marked entries in $C$ to cluster $id$, and remove
         them from the prediction matrix;
     (e) Update the histogram;
     (f) $id$++;
```

Figure 8. Cost-based clustering algorithm.

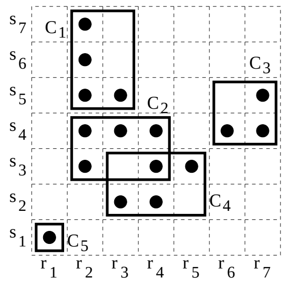


Figure 9. A set of clusters on a prediction matrix.

Scenario 1: *The clusters are processed in the order C_4, C_1, C_3, C_5, C_2. In the first iteration, the pages for C_4 are read into the buffer. These pages are r_3, r_4, r_5, s_2, and s_3 (i.e. 5 pages are read). In the second iteration, the pages for C_1 (r_2, r_3, s_5, s_6, and s_7) need to be read. Since r_3 is already in the buffer from the first iteration, we do not need to read it from disk (i.e. 4 pages are read and 1 page is reused). The same idea is applied in the subsequent iterations, yielding overall 19 pages to be read.*
Scenario 2: *The clusters are processed in the order C_4, C_2, C_1, C_3, C_5. Here, as the reader may verify, only 15 pages need to be fetched from disk.* $\square$

The above example shows that different schedules of clusters may result in different numbers of page reads. This is because if consecutive clusters have overlapping marked pages, then these pages can be reused, thus reducing the number of pages that need to be read. Therefore, a good cluster scheduling algorithm should maximize the total overlap between consecutive clusters.

In order to model the amount of reuse for a schedule of clusters, we define *sharing graphs* as follows.

Definition 1 *Let $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ be the set of clusters on a prediction matrix. The* sharing graph *of $\mathcal{C}$ is defined as the undirected graph $G = (V, E)$ with weighted edges, such that $V = \mathcal{C}$ and $E = \{e_{i,j}\}$, where $1 \leq i, j \leq k$, $i \neq j$, $e_{i,j} > 0$, and $e_{i,j}$ = the number of pages shared by clusters C_i and C_j.*

Each scheduling of the clusters defines a unique path on the sharing graph. For example, scenario 1 in Example 2 corresponds to the path $P = C_4 \rightarrow C_1 \rightarrow C_3 \rightarrow C_5 \rightarrow C_2$. The following two lemmas discuss two important properties of the a cluster schedules.

Lemma 3 *A schedule of the clusters defines a path on the sharing graph that visits each vertex exactly once.*

Lemma 4 *Let P be the path on a sharing graph defined by a schedule of clusters, then the amount of savings in page reads is equal to the sum of the weights of the edges of P.*

We conclude from Lemma 3 and 4 that a good path visits all the vertices once and maximizes the sum of the weights of the edges on this path. This is equivalent to the well known *Traveling Salesman Problem (TSP)* [17] which is NP-complete. We therefore use a greedy-based heuristics to construct a good path: the algorithm iteratively chooses the next edge that has the largest weight, does not cause a cycle, and does not increase the degree of a vertex to three. The total time complexity of this scheduling algorithm is $O(|E| \log |E|)$ where E is the set of edges in the sharing graph.

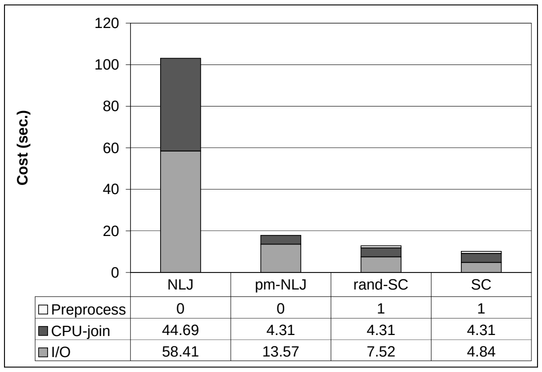
9 Experimental evaluation

We use three classes of datasets for the experimental evaluation of our techniques. These are

1. **Low dimensional datasets** *LBeach* and *MCounty*. They are road intersections of Long Beach and Montgomery County. Both of these datasets contain 2-dimensional vectors. *LBeach* contains 53,145 vectors and *MCounty* contains 39,231 vectors.

2. **High dimensional dataset** *Landsat* contains 275,465 60-dimensional feature vectors of satellite images. We split this dataset randomly into 8 equal sized and non-overlapping datasets. These datasets are denoted by *Landsat1*, *Landsat2*, $\cdots$, *Landsat8*.

3. **Sequence datasets** *HChr18* and *MChr18*. These datasets contain the nucleotides of human chromosome 18 and mouse chromosome 18. *HChr18* contains 4,225,477 nucleotides while *MChr18* contains 2,313,942 nucleotides. Both of these datasets are publicly available at *ftp://ncbi.nlm.nih.gov/genbank/genomes/*.



	NLJ	pm-NLJ	rand-SC	SC
□ Preprocess	0	0	1	1
■ CPU-join	44.69	4.31	4.31	4.31
▨ I/O	58.41	13.57	7.52	4.84

Figure 10. Costs of joining the LBeach and MCounty datasets when $\epsilon = 0.1$ for NLJ, pm-NLJ, random-SC, and SC.

We implemented pm-NLJ, CC, and SC along with three competing techniques: block nested loop join (NLJ), epsilon grid ordering (EGO) [6], and BFRJ [23]. NLJ is one of the earliest and simplest yet effective join techniques, while EGO is shown to be superior than other techniques proposed earlier. BFRJ is one of the most efficient index-based join techniques. We also implemented a simpler version of SC, called *random-SC*. This program builds the clusters exactly the same as SC, but it processes clusters in random order.

All of these techniques were implemented in C on a Unix platform. In order to run EGO and BFRJ on the sequence data we used the original implementation of the MRS index structure [25] to extract the frequency vectors of *HChr18* and *MChr18*. We ran our experiments on a 400 MHz Pentium II computer with 1 GB memory.

9.1 Evaluation of clustering components

SC and CC optimize the cost of join queries in three ways:

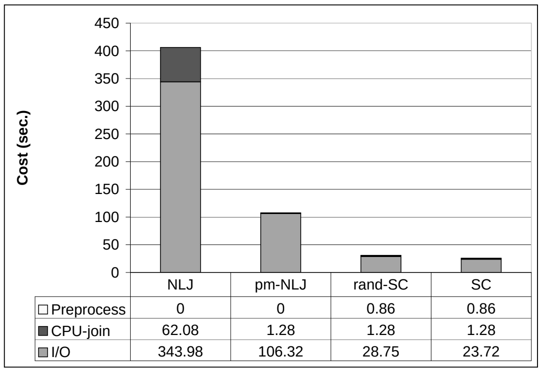
- *Optimization 1:* Restricting the join to the marked page pairs reduces the number of page pairs that need to be joined. This reduces both I/O cost and CPU cost (Sections 5 and 6).

- *Optimization 2:* Clustering the marked entries and reading one cluster at a time reduces the I/O cost (Section 7).

- *Optimization 3:* Processing clusters in a clever order increases memory reuse. That is, it reduces the I/O cost (Section 8).

Our first set of experiments evaluates the performance gain obtained by each of the abovementioned optimizations. We use NLJ as the simplest technique that does not use any information on the datasets. The difference between pm-NLJ and NLJ captures *Optimization 1*. Similarly, the difference between pm-NLJ and random-SC gives *Optimization 2*, and the difference between random-SC and SC shows the performance gained by *Optimization 3*.

Figure 10 shows the I/O cost, CPU cost and preprocessing cost of joining the LBeach and MCounty datasets when



	NLJ	pm-NLJ	rand-SC	SC
□ Preprocess	0	0	0.86	0.86
■ CPU-join	62.08	1.28	1.28	1.28
▨ I/O	343.98	106.32	28.75	23.72

Figure 11. Costs of self join on the HChr18 dataset when $\epsilon = 0.01$ for NLJ, pm-NLJ, random-SC, and SC.

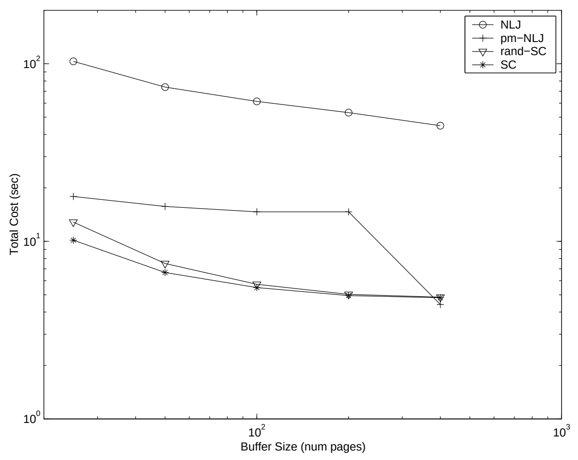


Figure 12. The total cost of self joining the HChr18 dataset for NLJ, pm-NLJ, random-SC, and SC for different buffer sizes.

$\epsilon = 0.1$ for NLJ, pm-NLJ, random-SC, and SC. The query selectivity here is approximately 10% and the buffer size is 25 pages, where the page size is 1K. Preprocessing cost corresponds to the cost of determining the clusters. Although this is a CPU cost, we report it separately since NLJ and pm-NLJ do not incur these costs. The reduction in CPU cost is on account of filtering the search space using the prediction matrix. The CPU cost of pm-NLJ (i.e. Optimization 1) is 10 times less than NLJ. In addition to this, embodying the prediction matrix on NLJ reduces the I/O cost 4.3 times over NLJ. Algorithm random-SC (i.e. Optimizations 1 and 2) decreases the I/O cost of pm-NLJ by one-half. Although clustering has an additional preprocessing cost, it is very small compared to the total cost. Finally, SC (i.e. Optimizations 1–3) reduces the I/O cost by 35% over random-SC. The total cost of SC is 10 times less than that of NLJ.

Figure 11 gives the results for the self join of the HChr18 dataset when $epsilon = 0.01$. The query selectivity here is approximately 2% and the buffer size is 100 pages, where the page size is 4K. The total cost of SC is approximately 16 times less than that of NLJ.

Figure 12 shows the total cost of self joining the HChr18 dataset for NLJ, pm-NLJ, random-SC, and SC for increasing buffer sizes in log-log scale. The total cost of pm-NLJ is always much less than that of NLJ. This difference reflects the performance gain obtained by Optimization 1. Both NLJ and pm-NLJ have a knee at B = 800. This is because one of the datasets fits into buffer for the next value of B. When one of the datasets fits into buffer, the I/O cost of both NLJ and pm-NLJ decrease dramatically and converge to that of SC. Since the CPU cost of pm-NLJ is equivalent to that of SC and random-SC, and SC (and random-SC) incurs an additional preprocessing cost for detecting clusters, the total cost of pm-NLJ becomes less than that of SC and random-SC for very large buffer sizes. For smaller buffer sizes, SC is up to two orders of magnitude faster than NLJ, up to 30 times faster than pm-NLJ, and up to 26% faster than random-SC.

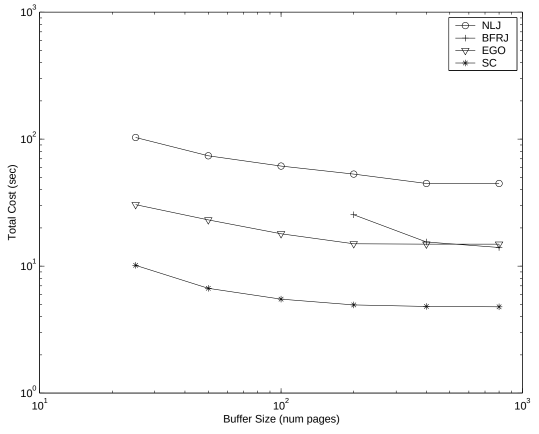
Buffer Size	50	100	200	400	800
LBeach/MCounty	2.06 (1.68)	1.02 (0.98)	0.51 (0.59)	0.37 (0.45)	0.34 (0.38)
Buffer Size	125	250	500	1000	2000
Landsat1/Landsat2	7.40 (6.46)	3.53 (2.93)	1.62 (1.44)	1.14 (1.27)	0.88 (0.88)
Buffer Size	100	200	400	800	1600
HChr18/HChr18	23.72 (12.02)	14.35 (6.56)	7.31 (3.56)	2.63 (2.01)	1.47 (1.07)
Buffer Size	50	100	200	400	800
HChr18/MChr18	46.08 (29.71)	26.46 (15.45)	13.27 (7.70)	6.72 (4.23)	3.11 (1.96)

Table 2. The I/O costs (in s) of SC and CC for joining various datasets for different buffer sizes. The results for CC are given in parentheses.

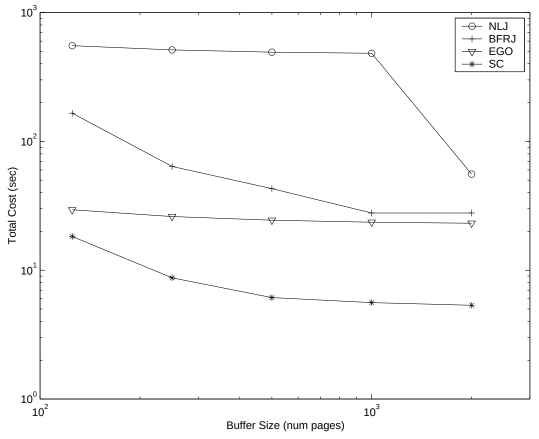
9.2 Effect of the buffer size

In this set of experiments, we evaluate the performance of SC and CC for different buffer sizes. Table 2 lists the I/O costs of SC and CC for joining various datasets for increasing buffer sizes (number of pages). CC almost always has lower I/O cost than SC. Although the I/O cost of CC is low, the total cost of CC is high. This is because finding clusters using CC is expensive. Therefore, we use CC to determine an approximate lower bound of the I/O cost of join operations. In all of the experiments, the I/O cost of SC is very close to that of CC. Therefore, we conclude that SC is a very competitive clustering technique despite its simplicity.

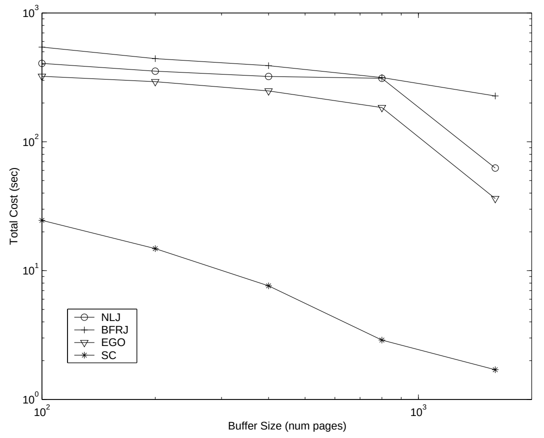
Figures 13(a) to 13(c) show the total cost of the join operation for NLJ, BFRJ, EGO, and SC for increasing buffer sizes in log-log scale. In all of these experiments, SC has the lowest cost and EGO has the next lowest cost. In Figure 13(a), we do not show the result for BFRJ for buffer sizes less than 200 pages. This is because even the intermediate structures do not fit into buffer for these buffer sizes. SC is 2 to 86 times faster than the competing techniques for spatial datasets, and



(a)



(b)



(c)

Figure 13. The total cost of (a) joining LBeach and MCounty datasets, (b) joining Landsat1 and Landsat2 datasets, and (c) self joining the HChr18 dataset using NLJ, BFRJ, EGO, and SC for different buffer sizes.

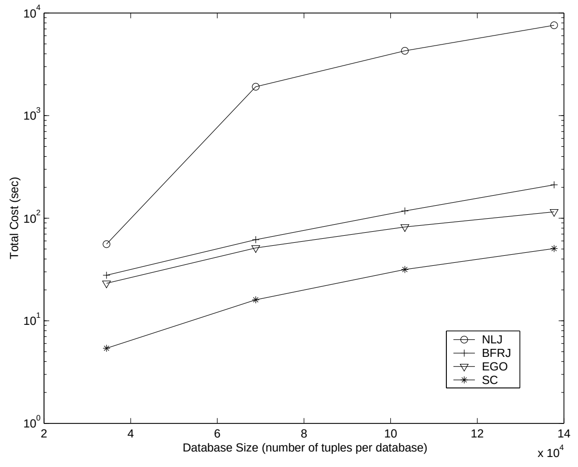


Figure 14. The total cost of joining Landsat datasets for NLJ, BFRJ, EGO, and SC for increasing dataset size when buffer size is 2000 pages.

13 to 133 times faster for sequence datasets. The performance of BFRJ and EGO deteriorates for sequence datasets due to many random disk seeks since the data cannot be reordered. On the other hand, the performance of SC does not drop since it avoids random disk seeks. A more detailed discussion of the cost components can be found in the full version of the paper [24].

9.3 Scalability comparison

In our last set of experiments, we inspect the total cost of SC and competing techniques for increasing dataset sizes. In order to carry out this experiment, we constructed two non-overlapping datasets of size 12.5%, 25%, 37.5%, and 50% of the original Landsat dataset by merging the Landsat1–8 datasets (i.e. we have two datasets with 34,433 vectors, two datasets with 68,866 vectors, two datasets with 103,299 vectors, and two datasets with 137,732 vectors.).

Figure 14 gives the total cost of NLJ, BFRJ, EGO, and SC for increasing dataset sizes for a buffer size of 2000 pages in log scale. The total cost of all techniques increases quadratically with the dataset size since the size of both join datasets are increased. SC is the fastest for all dataset sizes and the difference between SC and the competing techniques increases as datasets grow. SC is 2 to 4.3 times faster than EGO, 4 to 6.5 times faster than BFRJ, and 10 to 150 times faster than NLJ.

10 Discussion

In this paper, we considered the problem of joining massive datasets. We introduced a new type of join, called subsequence join, for sequence datasets. We proposed to minimize the I/O cost of the join operation using a global view of the datasets (namely, the prediction matrix). Our technique incorporates three optimizations: prediction, clustering, and cache reuse.

Our algorithm clusters the marked entries of the prediction matrix. We proposed two clustering techniques. The first technique, called square clustering (SC), assigns an equal number of pages from each dataset to each cluster. The second technique, called cost clustering (CC), minimizes the I/O cost for reading the contents of each cluster. We presented an algorithm for SC and showed its efficiency using a lower bound on the I/O cost obtained from CC. We then showed that the order of processing the clusters has an impact on the performance and proposed an efficient technique to define a good cluster order using a greedy technique.

According to our experiments, SC is 2 to 86 times faster than the competing techniques for spatial datasets, and 13 to 133 times faster than the competing techniques for sequence datasets. The I/O cost of SC is very close to that of our approximate lower bound determined by CC.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *FODO*, Evanston, Illinois, October 1993.

[2] R. Agrawal, K. Lin, H.S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *VLDB*, Zürich, Switzerland, September 1995.

[3] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *VLDB*, New York, 1998.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, Atlantic City, NJ, 1990.

[5] M.W. Blasgen and K.P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):362–377, 1977.

[6] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *SIGMOD*, Santa Barbara, CA, 2001.

[7] C. Böhm and H.P. Kriegel. A cost model and index architecture for the similarity join. In *ICDE*, Heidelberg, Germany, 2001.

[8] T. Bozkaya, N. Yazdani, and Z.M. Özsoyoglu. Matching and indexing sequences of different lengths. In *CIKM*, pages 128–135, 1997.

[9] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *SIGMOD*, pages 197–208, Minneapolis, MN, June 1994.

[10] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD*, pages 237–246, 1993.

[11] C.C. Chan and B.B. Ooi. Efficient scheduling of page access in index-based join processing. *TKDE*, 9(6):1005–1011, 1997.

[12] K.-P. Chan and A.W.-C. Fu. Efficient time series matching by wavelets. In *ICDE*, Sydney, Australia, March 1999.

[13] K.K.W. Chu and M.H. Wong. Fast time-series searching with scaling and shifting. In *PODS*, Philadelphia, PA, 1999.

[14] J.P. Dittrich and B. Seeger. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *KDD*, San Francisco, CA, 2001.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, Santa Barbara, CA, May 2001.

[16] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, Minneapolis MN, May 1994.

[17] M.R. Garey and D.S. Johnson. *Computers and intractability A guide to the theory of NP-Completeness*. Freeman, June 1979.

[18] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *ICDE*, Houston, Texas, February 1994.

[19] C. Gurret and P. Rigaux. The sort/sweep algorithm: A new method for R-tree based spatial joins. In *SSDBM*, pages 153–165, 2000.

[20] E.P. Harris and K. Ramamohanarao. Join algorithm costs revisited. *VLDB Journal*, 5(1):64–84, 1996.

[21] J.G. Henikoff and S. Henikoff. BLOCKS database and its applications. *Methods in Enzymology*, 266:88–105, 1996.

[22] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, pages 237–248, Seattle, Washington, 1998.

[23] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *VLDB*, Athens, Greece, 1997.

[24] T. Kahveci, C. A. Lang, and A. K. Singh. Joining massive high-dimensional datasets. Technical Report 30, UCSB, 2002.

[25] T. Kahveci and A. Singh. An efficient index structure for string databases. In *VLDB*, pages 351–360, Roma, Italy, September 2001.

[26] T. Kahveci and A. Singh. Variable length queries for time series data. In *ICDE*, Heidelberg, Germany, 2001.

[27] T. Kahveci, A. K. Singh, and A. Gürel. Similarity searching for multi-attribute sequences. In *SSDBM*, 2002.

[28] N. Koudas and K.C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, Tucson, AZ, June 1997.

[29] N. Koudas and K.C. Sevcik. High dimensional similarity joins: algorithms and performance evaluation. *TKDE*, 12(1), January/February 2000.

[30] C. A. Lang and A. K. Singh. Accelerating high-dimensional nearest neighbor queries. In *SSDBM*, pages 109–118, Edinburgh, Scotland, July 2002.

[31] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *SIGMOD*, Minnesota, USA, 1994.

[32] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *SIGMOD*, Montreal, Canada, 1996.

[33] H. Lu and K.-L. Tan. On sort-merge algorithm for band joins. *TKDE*, 7(3):508–510, June 1995.

[34] S. Manegold, P.A. Boncz, and M.L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, Cairo, Egypt, September 2000.

[35] S. Muthukrishnan and S. C. Sahinalp. Approximate nearest neighbors and sequence comparison with block operations. In *STOC*, Portland, Or, 2000.

[36] J.A. Orenstein. Spatial query processing in an object-oriented database system. In *SIGMOD*, pages 326–226, Washington, D.C., May 1986.

[37] A. Papadopoulos, P. Rigaux, and M. Scholl. A performance evaluation of spatial join processing strategies. In *SSD*, pages 286–307, 1999.

[38] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[39] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. MC Graw Hill, 2000.

[40] B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, 21(5):387–407, 1996.

[41] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. *TKDE*, 14(1):156–171, January/February 2002.

[42] J. Xiao, Y. Zhang, and X. Jia. Clustering non-uniform-sized spatial objects to reduce I/O cost for spatial-join processing. *The Computer Journal*, 44(5):384–397, 2001.