Metric Space Similarity Joins

EDWIN H. JACOX and HANAN SAMET
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
jacox@cs.umd.edu and hjs@cs.umd.edu

Similarity join algorithms find pairs of objects that lie within a certain distance ϵ of each other. Algorithms that are adapted from spatial join techniques are designed primarily for data in a vector space and often employ some form of a multi-dimensional index. For these algorithms, when the data lies in a metric space, the usual solution is to embed the data in vector space and then make use of a multidimensional index. Such an approach has a number of drawbacks when the data is high dimensional as we must eventually find the most discriminating dimensions, which is not trivial. In addition, although the maximum distance between objects increases with dimension, the ability to discriminate between objects in each dimension does not. These drawbacks are overcome via the introduction of a new method called *Quickjoin* that does not require a multi-dimensional index and instead adapts techniques used in distance-based indexing for use in a method that is conceptually similar to the Quicksort algorithm. A formal analysis is provided of the Quickjoin method. Experiments show that the Quickjoin method significantly outperforms two existing techniques.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query Processing, Multimedia databases*; H.2.8 [**Database Management**]: Database Applications—*Image databases*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: similarity join, external memory algorithms distance-based indexing, nearest neighbor queries, range queries, ranking

1. INTRODUCTION

The similarity join is a typical data-mining operation that has applications in many domains, such as medical imaging [Korn et al. 1996], multimedia [Faloutsos et al. 1994], and GIS [Samet 1990a]. The operation is used to find association rules [Koperski and Han 1995], to analyze time series, such as stock histories [Agrawal et al. 1993; Agrawal et al. 1995], and can also be used to identify data clusters [Böhm et al. 2000]. The similarity join finds similar pairs of objects, such as similar images or similar sub-sequences. Similarity for a pair of objects means that they are within a certain distance, ϵ, of each other. The data can be of any type, as long as a distance function exists that returns a distance between any pair of objects. For example, in metric spaces, the Levenshtein edit distance [Levenshtein 1966] can be used in a similarity join on two sets of strings such as text or DNA sequences; the Hausdorff distance can be used in a similarity join on shapes [Huttenlocher et al. 1992] or images [Rucklidge 1995; 1996]; and the Jaccard coefficient [Zezula et al. 2006] can be used to find similarity between sets, such as sets of web documents. Additionally, any distance metric used for a similarity query can be used in a similarity join to answer the "all-pairs" version of the similarity query. For many domains, the data is often represented as points in a multi-dimensional space or as feature vectors, and techniques often rely on multi-dimensional indices [Samet 2006]. Other related operations are k-nearest neighbor joins [Xia et al. 2004], which find the k most similar objects for each object, approximate spatial joins [Papadias and Arkoumanis 2002], which find the most similar pairs within a given amount of time, and all-nearest-neighbor queries [Zhang et al. 2004], which find the nearest object for each object in the set.

We present a method that overcomes several deficiencies in existing similarity join techniques that are derived from spatial join techniques [Jacox and Samet 2007]. These similarity join techniques are only applicable to multi-dimensional point (vector) data. If the data is not in a vector format, then embedding methods [Faloutsos and Lin 1995; Hristescu and Farach-Colton 1999; Linial et al. 1995; Wang et al. 1999] are usually used to transform the data into a vector format. Furthermore, these similarity join algorithms, discussed in Section 2, must pick particular dimensions as either the primary sort dimensions or the dimensions to index. For example, the ϵ-kdb tree [Shim et al. 1997] splits the data along one dimension first, then another, and so forth. This presents several problems. First, a good dimension must be found. If a poor dimension is chosen, then performance could be poor. For instance, a bad dimension is one in which the range of values in the dimension is nearly ϵ. In this case, every object is similar in that dimension, thereby making it useless to the index and leading to wasted processing and storage.

Other methods try to index all of the dimensions at once using a grid approach [Kalashnikov and Prabhakar 2003; Koudas and Sevcik 1998]. However, with higher dimensions a problem arises in that it becomes very likely that most data points will have feature values that are near the grid boundaries, thereby confounding these index techniques. Additionally, for any type of multi-dimensional index, the ability to discriminate between objects in a particular dimension becomes more limited as the number of dimensions that make up the space increases. Therefore, in lower dimensions, the multi-dimensional index approaches will work fine, but for higher dimensions, especially very high dimensions, the performance of the multi-dimensional index methods will degrade. Another deficiency of multi-dimensional indices (and indices in general) is that they are built efficiently for only one distance function or for just one embedding of the data. In the past, this has not been an issue since the most commonly used distance function is one of the Minkowski metrics, such as the Euclidean distance (L_2-norm). However, as pointed out by Aggarwal [2003], the L_2-norm is not necessarily relevant to many emerging appli-

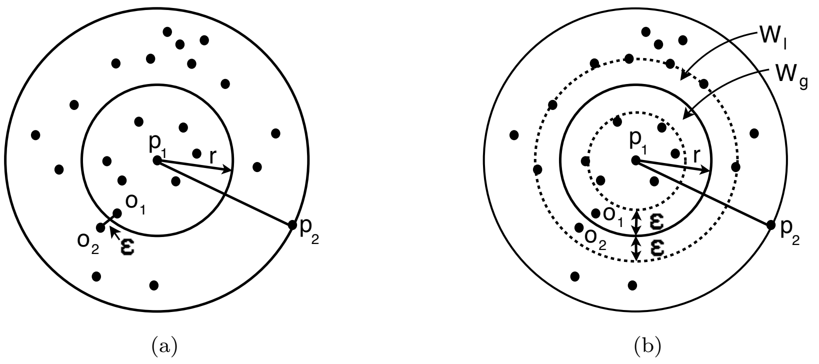


Fig. 1. (a) With ball partitioning, objects are partitioned by their distance from a randomly chosen object p_1, based on a distance r. In this example, r is determined using half the distance to the farthest object p_2. The problem with this partitioning is that it can separate result pairs that are within ϵ of each other, such as objects o_1 and o_2. (b) In order to correctly capture all result pairs, two windows of objects within ϵ of the partitioning distance, r, are created – one with objects whose distances from p_1 are between r and $r + \epsilon$, W_g, and one with objects whose distances from p_1 are between $r - \epsilon$ and r, W_l.

cations involving high-dimensional data. Furthermore, the distance function might change for a particular application based on user preferences. In these cases, a dimension that is primary for one distance function, might not be so for another distance function.

In this paper, we introduce a distance-based similarity join that avoids these problems and performs better than other non-index based similarity join techniques [Böhm et al. 2001; Dittrich and Seeger 2001]. Our algorithm uses concepts similar to those used in distance-based indexing methods [Chávez et al. 2001; Hjaltason and Samet 2003] in that only the distance function is used, rather than relying on a particular dimension. In other words, the data does not need to be in vector format (multi-dimensional points). Additionally, our method does not build an index, so it is able to adapt to changing distance functions. We only require that the triangle inequality hold, which is true in many applications. Finally, as shown experimentally in Section 5, our method performs competitively and often significantly outperforms competing methods.

Our method, termed *Quickjoin*, is based on the Quicksort algorithm [Hoare 1962] in that it recursively partitions the data until each partition contains a few objects, at which point a nested-loop join is used. The data can be partitioned using a variety of techniques, such as *ball partitioning*, which partitions the data based on their distance to a single random object (termed a *pivot*) [Tasan and Özsoyoglu 2004; Yianilos 1993], as shown in Fig. 1, or *generalized hyperplane partitioning*, which partitions the data on the basis of which of two randomly selected objects is closest [Uhlmann 1991]. Both techniques, as well as other variations, are described in Section 3.

By just partitioning the data, a significant portion of the desired results will be missed since objects in one partition might need to be joined with objects in another partition. For example, using ball-partitioning, in Fig. 1a, objects o_1 and o_2, which lie within ϵ of each other, would not be reported as similar since they are in different partitions. To overcome this problem, when the data is partitioned into two partitions, the data is replicated into two sets, which we call "windows": window W_l with distance values from p_1 less than or equal to the partitioning distance, r, but within ϵ of r, and set W_g with distance values from p_1 greater than r, but also within ϵ of r, as shown in Fig. 1b. These two windows are joined, and only pairs, (a, b), with an object from each window are reported, where $a \in W_l$ and $b \in W_g$ or $b \in W_l$ and $a \in W_g$. Again, a Quicksort-like partitioning is applied recursively, but this time to each of the windows.

The rest of the paper is organized as follows. After reviewing related work in Section 2, the details of the algorithm are presented in Section 3, including additional variations of the algorithm. In Section 4, the performance characteristics of the algorithm are analyzed in terms of CPU, internal memory, and I/O costs. Section 5 describes the result of several experiments showing that the method performs better than the EGO method [Böhm et al. 2001], described in Section 2.1 and the GESS method [Dittrich and Seeger 2001], described in Section 2.2. Section 6 contains concluding remarks.

2. RELATED WORK

A similarity join is similar to a spatial join [Orenstein 1989] or an interval join [Enderle et al. 2004], which find pairs of intersecting objects, such as intervals, polygons, or rectangles, in low dimensions. Algorithms have been devised for spatial joins using both indexed [Brinkhoff et al. 1993; Huang et al. 1997; Koudas and Sevcik 1997; Lo and Ravishankar 1995] and unindexed methods [Arge et al. 1998; Jacox and Samet 2003; Patel and DeWitt 1996]. For indexed methods, the join algorithm is frequently a synchronized traversal of the tree-like index structures [Brinkhoff et al. 1993]. If the data is not indexed, then an index can be built, or if the index will not be reused, then the expense of building an index can be avoided by using a partitioning method [Patel and DeWitt 1996] or a plane-sweep method [Arge et al. 1998; Jacox and Samet 2003]. As with the spatial join techniques, the first attempts at solving the similarity join used indices [Koudas and Sevcik 1998; Shim et al. 2002]. Unfortunately, traditional spatial indices, such as the R-tree [Guttman 1984], have a number of drawbacks in higher dimensions [Böhm and Kriegel 2001; Shim et al. 1997; 2002]. For example, the number of neighboring leaf nodes typically increases in higher dimensions, thereby degrading performance. To overcome such drawbacks, several indices have been proposed for indexing high-dimensional data in order to perform a similarity join:

(1) An MX-CIF quadtree [Kedem 1982] that uses a multi-dimensional linear order [Koudas and Sevcik 1998].

(2) A variant of the kdb-tree [Robinson 1981] that splits the data in each dimension into ϵ sized segments [Shim et al. 1997; 2002].

(3) A sophisticated R-tree variant with large pages and a secondary search structure [Böhm and Kriegel 2001].

(4) A simple grid-like index structure [Kalashnikov and Prabhakar 2003], similar to a grid file [Nievergelt et al. 1984].

Once the data is indexed, the similarity join can be performed using either a synchronized tree traversal [Brinkhoff et al. 1993] or other heuristic methods that globally order the access of the data pages [Harada et al. 1990; Kahveci et al. 2003; Kitsuregawa et al. 1989].

However, as mentioned in Section 1, similarity join techniques based on these indices might not perform well since they rely on indexing a subset of the dimensions and it might be difficult or impossible to choose a good subset of dimensions. Furthermore, these techniques assume a fixed, Minkowski distance function, which limits their applicability. For example, Shim et al. [1997; 2002] propose an index structure, termed an ϵ-kdB tree, in which each level of the tree splits on a different dimension into approximately ϵ sized pieces. After the indices are built, a synchronized tree traversal is used to perform the join. This method relies on choosing a good split dimension, which might not exist for larger ϵ values. Similarly, two good dimensions must be chosen for the *Grid-join* method of Kalashnikov and Prabhakar [2003], which uses a 2-dimensional grid to index multi-dimensional data. Even in their work, they noted that this approach is unsuitable for higher dimensions. In their experimental section they point out that "the Grid-join is not competitive for high-dimensional data, and its results are often omitted..." Instead of creating a new index, Böhm et al. [2001] propose an unindexed method, termed the *Epsilon Grid Order* or EGO, which is described in detail in Section 2.1.

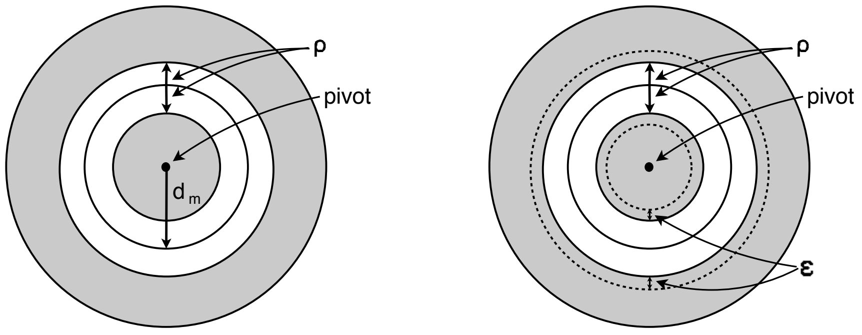


Fig. 2. (a) The D-index partitions the data into three sets using the median distance within the partition, d_m, and a parameter ρ. The outer set (shaded) consists of the objects farther than $d_m + \rho$ from the pivot object. The inner set (shaded) consists of the objects nearer than $d_m - \rho$ to the pivot object. The remaining objects form the third set, termed the *exclusion set* (not shaded). (b) The eD-index extends the D-index by replicating objects that are within ϵ of the inner and outer sets (shown by the dotted circles) into the exclusion set.

On the other hand, techniques based on metric space indexing have also been used for similarity joins (e.g., for text processing and data cleaning operations). For example, Dohnal et al. [003b] propose a metric space self-similarity join using a custom built metric space index, termed an *eD-index*, which repeatedly partitions the data using ball partitioning techniques. Once the data is partitioned, they use an in-memory technique that is similar to a plane-sweep technique [Preparata and Shamos 1985], which they call a "sliding window join", to find result pairs in each partition. To ensure that result pairs existing in separate partitions are reported, that is, results pair (o_1, o_2) where objects o_1 and o_2 are in different partitions, each partition is extended by ϵ, resulting in duplicate objects in the index. Duplicate results are avoided by the use of a tagging system (described in [Zezula et al. 2006], pages 134–136).

The eD-index is based on the D-index [Dohnal et al. 2003], whose partitioning scheme initially uses multiple ball-partitionings. Each partitioning around the separate pivots produces three sets based on the median distance in the set, d_m, and a parameter, ρ, as shown in Fig. 2a. Two sets (shaded in Fig. 2) are formed by those objects farther than ρ from the median distance. The inner set contains all objects, o, such that $d(o, p) \leq d_m - \rho$, and the outer set contains all objects such that $d(o, p) > d_m + \rho$. All other objects form a third set, called the *exclusion set*. The union of all objects in any exclusion set is recursively partitioned using further rounds of ball-partitionings until the resulting exclusion set is sufficiently small. The eD-index extends the D-index by replicating those objects farther than ρ from the median distance but within ϵ, which are the shaded regions within the dotted lines in Fig. 2b. In other words, those objects within ϵ of the inner and outer set boundary are replicated into the exclusion (middle) set.

As with the eD-index approach, the Quickjoin method also partitions the data to perform a similarity join, though it uses a different approach. Rather than extending the partitions by ϵ, the Quickjoin method creates subsets of the ϵ regions (see Section 3 for details), which are processed separately, avoiding duplicate results and the need for a duplicate removal technique. A potential drawback with the eD-index approach is that the index is built for small ϵ values and would need to be rebuilt for larger ϵ values. Furthermore, the technique is only applicable to self-similarity joins for pre-built indices. Indices built with different partitioning functions cannot be joined with this method. The Quickjoin method, since it does not build any structures, can process two sets by processing the sets as one and only reporting results containing one item from each set. Overall, the main difference between the Quickjoin and the eD-index approaches is the way in which the space is partitioned. Quickjoin always partitions the data while the eD-index method builds a structure which can be used several times. How this difference (and differences in duplicate removal and small partition processing) effects the overall performance is not addressed in this article and is left for future work.

2.1 Epsilon Grid Order (EGO)

The EGO algorithm of Böhm et al. [2001] is based on the EGO sorted order, which is described as a sort based on imposing an ϵ-sized multi-dimensional grid over the space, and ordering the cells, as shown in Fig. 3b. In effect, this is a loose version of a total multi-dimensional sort, where the points are primarily sorted in one dimension, secondarily on another dimension, and so forth for each dimension.

In the EGO method, the EGO-sorted data is read in blocks, and the key to the algorithm is to efficiently schedule reads of blocks of data so as to minimize I/O. Each block b of data needs to be joined with all other data blocks containing data within a distance of ϵ of b. In the best case, each data block can be read one at a time, and all corresponding data blocks (those that need to be joined) will fit in memory. In this way, each data block will be read only once. For larger ϵ values,

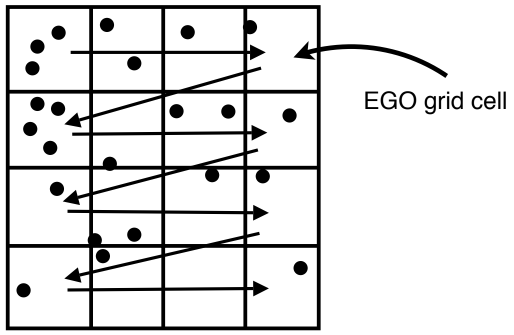


Fig. 3. The EGO sorted order traverses grid cells. Each grid cell is kept in memory until any points within the cell could not be within ϵ of grid cells yet to be seen. Points within a grid cell are not necessarily sorted, and are joined using a recursive divide-and-conquer approach.

this will not likely be the case, and the problem becomes similar to the classic join scheduling problem, which is known to be NP-Hard [Merrett et al. 1981; Neyer and Widmayer 1997]. To overcome this problem, Böhm et al. use heuristics to order the block reads and to determine which blocks in memory to evict. The goal of the heuristic, which they term *crab stepping*, is to try to minimize repeated reads of blocks.

To join two blocks of data, the data blocks are recursively divided until a minimum size has been reached, in which case a nested-loop join can be used. The recursion also stops when the data block subsets are at a distance greater than ϵ from each other. With the EGO* algorithm, Kalashnikov and Prabhakar [2003] use a better test for checking for this condition, which stops the recursion sooner and improves performance.

2.2 GESS

Dittrich and Seeger created the *Generic External Space Sweep* (GESS) method [Dittrich and Seeger 2001], which is a variant of the *Multidimensional Spatial Join* (MSJ) of Koudas and Sevcik [1998; 2000], which in turn is based on a technique for performing a spatial join on rectangles [Koudas and Sevcik 1997]. The technique is adapted to multi-dimensional points by creating hyper-squares centered around each point with ϵ length sides, and then using the original technique to join the hyper-squares. If two hyper-squares intersect, then the points must be within ϵ of each other along at least one of the dimensions of the underlying space. The MSJ method assigns the data to grid cells, sorts the data based on the assigned grid cells using a multi-dimensional linear order [Jagadish 1990; Samet 1990b], and then scans the data in sorted order to perform the join.

In the MSJ method, objects are assigned to levels of regular grids. Each level of grid cells is derived by dividing each dimension of the previous level in two, as shown in Fig. 4. The first level, level zero, encloses the entire data space. An object is assigned to the finest (highest) level in which it does not cross a grid boundary, thereby creating an *MX-CIF quadtree index* [Kedem 1982]. Each part of Fig. 4 shows an object assigned to a different level. Since each dimension is split, the fan-out will be large for higher dimensions, thereby limiting the depth of grid levels

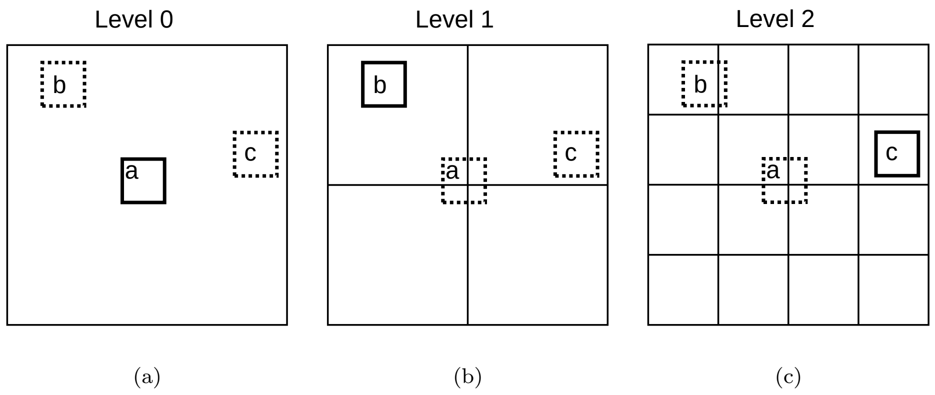


Fig. 4. In the MSJ method, objects are created out of multi-dimensional points by creating hyper-squares centered around each point with ϵ length sides. Each object is assigned to the finest regular grid in which it does not cross a grid boundary. In this example, only the square with the solid border in each figure is assigned to that level, with level 2 being the lowest level. (a) Object a is assigned to the root space, level 0, since it intersects the first (coarsest) grid level with four grid cells, that is, level 1. (b) Object b is assigned to the next level since it intersects the next finer grid level with sixteen cells, that is, level 2. (c) Object c is assigned to the lowest level.

since an enormous number of lower-level cells would create an unwieldy index.

By assigning objects to cells in this manner, objects whose hypersquares overlap grid boundaries will be assigned to shallower (coarser) levels, as shown in Fig. 4a. To overcome this inefficiency, Dittrich and Seeger introduce redundancy, thereby allowing an object to be replicated into multiple objects that can be assigned to a more appropriate level For example, object a in Fig. 4a would be replicated into four objects, each assigned to one of the four middle grid cells of level 2. Unfortunately, in higher dimensions, an object could cross boundaries in many dimensions and would need to be replicated into 2^k cells for k boundary crossings. For this reason, objects are not replicated if k exceeds a predefined limit. Dittrich and Seeger [2001] experimentally found that a small k factor, such as two or three, is best. By introducing redundancy in the dataset, duplicate results might appear, which can be removed in-line using a technique described later. Furthermore, the GESS method does not limit the type of partitioning. For instance, rather than a quadtree-like partitioning, a kd-trie partitioning can be used.

Once the data is assigned to grid cells, it is sorted based on the assigned grid cell using a linear order such as a Hilbert or Peano-Hilbert order [Jagadish 1990; Samet 1990b]. The data is secondarily sorted by level so that objects assigned to larger grid cells are seen first, immediately followed by objects within the larger grid cells that have been assigned to the next lower level of grid cells, and so forth. As noted by Dittrich and Seeger [2001], the data does not need to be physically written to level files, as is done with the MSJ method, but rather, the data can be assigned to grid cells as it is fed into a sorting operator, avoiding additional I/O cost.

After the data has been sorted, it is scanned in sorted order and joined using a technique attributable to Orenstein [1986] and similar to a plane-sweep technique [Preparata and Shamos 1985]. As each object is encountered, it is pushed onto a

```
procedure Quickjoin(eps, distFunc, objs)
begin
  if( objs.size < constSmallNumber )
   NestedLoop(eps, distFunc, objs);
   return;
  end if

  p1 ← randomObject(objs);
  p2 ← randomObject(objs-p1);
  (partL, partG, winL, winG) ←
        Partition(eps, distFunc, objs, p1, p2);

  QuickjoinWin(eps, distFunc, winL, winG);
  Quickjoin(eps, distFunc, partL);
  Quickjoin(eps, distFunc, partG);
end
```

Fig. 5. The Quickjoin algorithm recursively partitions the data into subsets that are small enough to be efficiently processed by a nested-loop join. The "windows" around the partition boundary, `winL` and `winG`, are handled by a separate recursive function, QuickjoinWin, shown in Fig. 9.

stack and joined with all other data on the stack. Once all of the data within a grid cell has been encountered, that data can be purged from the stack since it could not possibly be similar to any data yet to be seen because of the sorted order.

Because of the replication, duplicate results could be reported. Dittrich and Seeger use a *reference point method* [Aref and Samet 1994; Dittrich and Seeger 2000] to prevent duplicate results from being reported. In this method, a consistently chosen corner of the hyper-dimensional rectangle formed by the intersection of two ϵ expanded points is used, such as the lower left corner of a two-dimensional intersection. If this corner point is within the grid cell to which the objects are assigned, then the result (similarity) is reported. Since this point can only lie within one grid cell, duplicate results are avoided.

3. THE QUICKJOIN ALGORITHM

This section describes the details of the Quickjoin algorithm, as well as variations of the algorithms. Also, slightly different algorithms are used depending on whether the entire dataset fits into internal memory. The internal memory method using ball partitioning is presented in Section 3.1, and the generalized hyperplane variant of the Quickjoin algorithm is described in Section 3.2. A variant of the algorithm that assumes vector data is described in Section 3.3. Next, the external memory algorithm is described in Section 3.4, which uses the internal memory algorithm to join partitions of the data that are small enough to fit into internal memory. Finally, Section 3.5 describes a technique for improving the performance of the internal memory algorithm.

3.1 Quickjoin Using Ball Partitioning

The internal memory version of the Quickjoin algorithm, shown in Fig. 5, takes three arguments: the desired ϵ, `eps`; a distance function, `distFunc`; and the set of objects to be joined, `objs`, which is implemented as an array for the best performance for the internal memory algorithm. The procedure recursively partitions the data in

```
procedure NestedLoop(eps, distFunc,
                     objs)
begin
  for each a $\in$ objs do
    for each b $\in$ objs do
      if(a $\neq$ b AND distFunc(a,b)$\leq$eps)
          Report(a,b);
      end if
    end loop
  end loop
end
```

(a)

```
procedure NestedLoop2(eps, distFunc,
                      objsA, objsB)
begin
  for each a $\in$ objsA do
     for each b $\in$ objsB do
        if(distFunc(a,b) $\leq$ eps)
           Report(a,b);
        end if
     end loop
  end loop
end
```

(b)

Fig. 6. (a) The NestedLoop procedure performs a self join on the set of objects, `objs`, finding all similar pairs of objects. (b) The NestedLoop2 procedure performs a join on the two given sets objects, finding all pairs of similar objects between the two sets of objects, that is, pairs with an object from each set.

`objs` into smaller subsets until the subsets are small enough (defined by the constant `constSmallNumber`) to be efficiently handled by a nested-loop join, which is invoked on line 4 of the algorithm. Shown in Fig. 6a, the `NestedLoop` procedure simply performs a nested-loop join on the objects and reports the pairs of objects within `eps` of each other.

If there are more than a few objects, the set of objects is partitioned in two using the `Partition` function on lines 10 and 11. The `Partition` function, shown in Fig. 7, requires two random objects, `p1` and `p2`, which are chosen on lines 8 and 9, of the Quickjoin algorithm. The `Partition` function divides the objects into two sets, `partL` and `partG`, and also creates two "window" subsets, `winL` and `winG`, that contain objects that are within `eps` of the opposing partition, that is, `winL` contains objects in `partL` that might be within `eps` of the objects in `partG`, and `winG` contains objects in `partG` that might be within `eps` of objects in `partL`. The window partitions are created, since, as was shown in Fig. 1a, a simple partitioning in two will miss reporting some pairs of objects that are similar because they lie in different partitions. For the internal memory algorithm, the window partitions can use references to the objects rather than duplicate the objects to save internal memory and replication costs.

Lines 14 and 15 recursively reinvoke the `Quickjoin` procedure to find similar objects within the `partL` and `partG` partitions. To find similar objects between the two partitions, that is, similar pairs of objects containing one object from `partL` and one object from `partG`, the two window sets, `winL` and `winG` are joined. This is done using a modified version of the `Quickjoin` procedure, termed `QuickjoinWin`, shown in Fig. 9, which is invoked on line 13.

The key to the Quickjoin algorithm is the `Partition` function, given in Fig. 7, which partitions the data using the distance of every object to a random object (the pivot), `p1`, as shown in Fig. 8. The variable `r` is the *pivot distance*, calculated on line 4, which, in this instance, is the distance from `p1` to a second random object, `p2`, that is used to partition the objects. Objects with a distance from `p1` less than or equal to `r` form the partition `partL`, and objects with a distance from `p1` greater than `r` form the partition `partG`.

The `Partition` function uses in-place partitioning, which assumes that the ob-

```
function Partition(eps, distFunc, objs[], p1, p2)
         : (partL, partG, winL, winG)
begin
   r ← distFunc(p1,p2);
   startIdx ← 0;
   endIdx ← objs.length-1;
   startDist ← distFunc(objs[startIdx], p1);
   endDist ← distFunc(objs[endIdx], p1);
   while( startIdx < endIdx )
      while( endDist > r AND startIdx < endIdx )
         if(endDist <= r+eps) winG.insert(objs[endIdx]);
         endDist ← distFunc(objs[--endIdx], p1);
      end loop
      while( startDist <= r AND startIdx < endIdx )
         if(startDist >= r-eps) winL.insert(objs[startIdx]);
         startDist ← distFunc(objs[++startIdx], p1);
      end loop

      if( startIdx < endIdx )  // exchange objects
         // check if either goes in a window
         if(endDist >= r-eps) winL.insert(objs[endIdx]);
         if(startDist <= r+eps) winG.insert(objs[startIdx]);
         // exchange items
         objs[startIdx] ↔ objs[endIdx];
         startDist ← distFunc(objs[++startIdx], p1);
         endDist ← distFunc(objs[--endIdx], p1);
      end if
   end loop

   // clean up last object
   if( startIdx == endIdx )
         // check if this needs to be added to a window
         if( endDist > r AND endDist <= r+eps )
            winG.insert(objs[endIdx]);
         end if
         if( startDist <= r AND startDist >= r-eps )
            winL.insert(objs[startIdx]);
         end if
         // check last object
         if( endDist > r ) endIdx--;
    end if

    return (objs[0:endIdx], objs[endIdx+1:objs.length-1], winL, winG );
end
```

Fig. 7. The `Partition` function partitions the data into two sets, those whose distances from object `p1` are greater than the partitioning distance, `r`, and those whose distances from `p1` are less than or equal to `r`. The value `r` is the distance from the randomly chosen object `p1` to another randomly chosen object, `p2`. Also, objects are inserted into two other "window" sets if the objects are within ϵ, `eps`, of `r`.

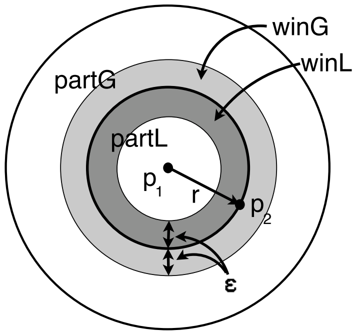


Fig. 8. Objects are partitioned into two partitions – `partG` contains objects whose distances from `p1` are greater than `r` and `partL` contains objects whose distances from `p1` are less than or equal to `r`. The objects are also replicated into two windows. The window, `winG` is a subset of `partG` containing objects within ϵ of `r` and `winL` is a subset of `partL` also containing objects within ϵ of `r`.

jects are stored in an internal memory array. The array is partitioned in place, which results in placing objects whose distance from `p1` is greater than `r` to the right of all objects whose distance from `p1` is less than or equal to `r`. The outer loop, starting at line 9, scans the objects from both ends of the array using two inner loops: one from the end (right side) on line 10, and one from the start (left side) on line 14. When out of place objects are found on each side, the two objects are exchanged on line 24. Additionally, as the objects are scanned, they are placed into window partitions if they are within `eps` of `r`. This is done on lines 11, 15, 21, 22, 34, and 37. Finally, the object dividing the two sets is checked starting on line 31, and serves as the dividing point for returning the two ends of the array as partitions, on line 43, along with the window partitions.

Different partitioning methods can also be used in the `Partition` function. The data can be partitioned using one random object and aggregate properties of the distances. For instance, the data can be partitioned on the basis of half the distance to the farthest object. The data could also be partitioned based on the average distance or the median distance in order to ensure a more equal-sized partitioned. See Section 5.4 for the results of experiments comparing these various techniques. Methods other than ball partitioning are described in Section 3.2 and Section 3.3.

The `QuickjoinWin` procedure, shown in Fig. 9, is only slightly different than the `Quickjoin` procedure. A nested-loop join is also used on line 5, which is termed `NestedLoop2`, shown in Fig. 6b, in which only similar pairs with an object from each set are reported. Additionally, the `Partition` function is called separately on each set of objects on lines 12 through 15, partitioning both sets using the same random objects, `p1` and `p2`. In our notation, the `Partition` function partitions `objs1` into objects whose distances from `p1` are less than or equal to `r`, `partL1`, and objects whose distances from `p1` are greater than `r`, `partG1`. Similarly, `objs2` is partitioned into `partL2` and `partG2`. The two sets whose distances from `p1` are less than or equal to `r`, `partL1` and `partL2`, are recursively joined using `QuickjoinWin` on line 19, since the purpose of the `QuickjoinWin` procedure is to find pairs between the two object sets. Similarly, `partG1` and `partG2` are recursively joined on line 20. In

```
 1 procedure QuickjoinWin(eps, distFunc, objs1, objs2)
 2 begin
 3   totalLen ← objs1.size+objs2.size
 4   if( totalLen < constSmallNumber2 )
 5      NestedLoop2(eps, distFunc, objs1, objs2);
 6      return;
 7   end if
 8
 9   allObjects ← objs1 ∪ objs2;
10   p1 ← randomObject(allObjects);
11   p2 ← randomObject(allObjects-p1);
12   (partL1, partG1, winL1, winG1) ←
13         Partition(eps, distFunc, objs1, p1, p2);
14   (partL2, partG2, winL2, winG2) ←
15         Partition(eps, distFunc, objs2, p1, p2);
16
17   QuickjoinWin(eps, distFunc, winL1, winG2);
18   QuickjoinWin(eps, distFunc, winG1, winL2);
19   QuickjoinWin(eps, distFunc, partL1, partL2);
20   QuickjoinWin(eps, distFunc, partG1, partG2);
21 end
```

Fig. 9. The `QuickjoinWin` procedure, which is similar to the `Quickjoin` procedure given in Fig. 5, joins two sets of objects by partitioning the two sets and then recursively joining the partitions. Only result pairs with one object from each set are reported.

addition, the partitioning creates windows of objects that are within ϵ of `r`. The set `winL1` is a subset of `partL1` that is within ϵ of `r` and `winG1` is a subset of `partG1` within ϵ of `r`. The sets `winL2` and `winG2` are defined similarly. To determine which windows to recursively join, the only ones to consider are those between subsets of `objs1` and `objs2`. The join between `winL1` and `winL2` is covered by the join between `partL1` and `partL2`. Similarly, the join between `winG1` and `winG2` is covered by the join between `partG1` and `partG2`. The remaining combinations are performed – between `winL1` and `winG2` on line 17, and between `winG1` and `winL2` on line 18. See Section 3.2 for a graphical depiction of the `QuickjoinWin` partitioning in which hyperplane partitioning is used instead of ball partitioning.

3.2 Quickjoin Using Generalized Hyperplane Partitioning

Instead of using ball partitioning, we could also use generalized hyperplane partitioning. In this case, the partitioning hyperplane is used to separate the objects on the basis of which of the two pivot objects is closer. Unfortunately, in a general metric space, the partitioning hyperplane does not exist explicitly. Instead, it is implicit and known as a *generalized hyperplane*. Therefore, we cannot, in general, compute the exact distance from an object a to the partitioning hyperplane. However, we do have a lower bound for it (see Lemma 4.4 in [Hjaltason and Samet 2003, p. 539]) which is given in Equation 1, where $dist(a, p1)$ is the distance from object a to random object *p1* and $dist(a, p2)$ is the distance from a to the second random object, *p2*.

$$dist = (dist(a, p1) - dist(a, p2))/2 \qquad (1)$$

To implement this variation, the `Partition` function in Fig. 7 is modified slightly.

```
(1) r ← 0;
(2) dist ← (distFunc(a,p1)-distFunc(a,p2))/2;
```

Fig. 10. (1) The pivot distance is zero for hyperplane partitioning and replaces the code on line 4 of Fig. 7. (2) The distance function used for generalized hyperplane partitioning. This code replaces the distance function on lines 7, 8, 12, 16, 25, and 26 in Fig. 7.

In particular, the only change is in how the distance to each object is calculated and the value of `r`, which becomes zero since objects closer to object `p1` will have a positive distance and objects closer to `p2` will have a negative distance. These changes are shown in Fig. 10.

Equation 1 works for any distance metric. If exact distances from the partitioning hyperplane are known, then the number of items in the windows can be reduced by using these exact distances instead of lower bounds on the distances, which improves the performance of the algorithm. For example, when the distance metric is the L_2-norm, the exact form of Equation 1 is given by Equation 2, where $dist(p1, p2)$ is the distance between the two random objects *p1* and *p2*.

$$dist = (dist(a, p1)^2 - dist(a, p2)^2)/(2 * dist(p1, p2)) \tag{2}$$

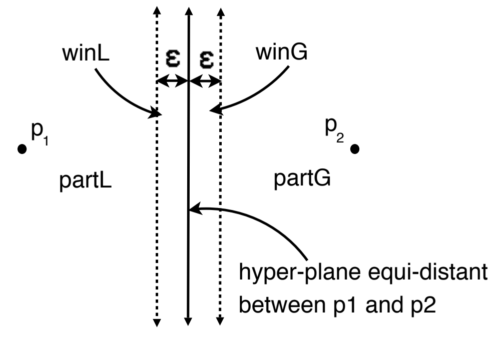


Fig. 11. Using hyperplane partitioning, objects are partitioned into two sets – `partG` containing objects whose distances from the partitioning hyperplane are greater than zero, and `partL` containing objects whose distances to the partitioning hyperplane are less than or equal to zero. The objects are also inserted into two windows. The window, `winG` is a subset of `partG` containing objects within ϵ of the partitioning hyperplane, and `winL` is a subset of `partL` also containing objects within ϵ of the partitioning hyperplane.

To use this version of the hyperplane, Equation 2 is used to calculate distance, replacing the distance function in Fig. 7, as was done with the generalized hyperplane function in Fig. 10. Fig. 11 shows the use of the hyperplane for partitioning, while Fig. 12 shows its use for partitioning within the `QuickjoinWin` procedure.

3.3 Dimensional Version – Vector Data

If the data is represented as vectors, then another partitioning approach is to partition the data using one of the dimensions of the vector data. This partitioning function requires only slight modifications to the `Partition` function given in Fig. 7,

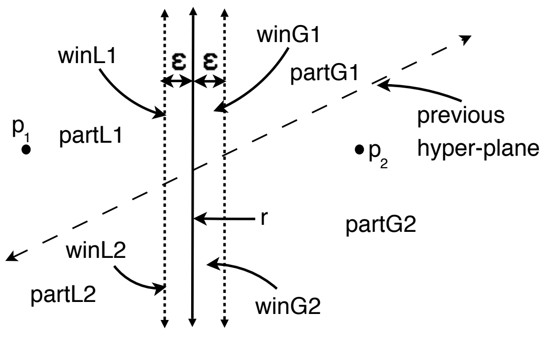


Fig. 12. Objects are partitioned into four sets in the procedure `QuickjoinWin`, given in Fig. 9. The procedure partitions sets `objs1` and `objs2`, which were divided by a previous hyperplane, as shown. Partitions `partG1` and `partG2` contain objects from object sets `objs1` and `objs2`, respectively, that are greater than the partitioning distance, `r`, and object sets `partL1` and `partL2` similarly contain objects less than or equal to the partitioning distance `r`, again from object sets `objs1` and `objs2`, respectively. As in the `Quickjoin` procedure, the objects are also replicated into windows containing objects within ϵ of the partitioning hyperplane.

```
(1) d ← pickPartitioningDimension();
(2) r ← p1[d];
(3) dist ← obj[d]
   (e.g., startDist ← objs[startIdx][d];)
```

Fig. 13. Modifications to the `Partition` function from Fig. 7 to implement the dimensional variant of the Quickjoin method. (1) Between lines 3 and 4, a single dimension, `d` is chosen, which can be done in many ways, such as cycling through the dimensions. (2) The pivot, `r`, on line 4 becomes one dimension of the randomly chosen object `p1`. (3) The value used for dimensional partitioning. This code replaces the distance function on lines 7, 8, 12, 16, 25, and 26. As an example, line 7 of the `Partition` function is rewritten to use the $\mathtt{d}^{th}$ dimension of object `objs[startIdx]`. The object `p2` is not needed.

which are shown in Fig. 13. In this case, each invocation of the `Partition` function must choose a dimension with which to partition the data. For instance, the first invocation could use the first dimension to partition the data, and subsequent calls could use the second dimension, then the third dimension, and so forth, cycling through the dimensions.

With this partitioning method, the distance to each object does not need to be calculated, but a randomly chosen object still needs to be chosen to partition the data. Alternatively, as with ball partitioning, a median or average value could be used. Note that even if the dataset is not represented by vectors, as long as we have a distance function, we can use an embedding method [Faloutsos and Lin 1995; Hristescu and Farach-Colton 1999; Linial et al. 1995; Wang et al. 1999] to transform it into a multi-dimensional vector space.

3.4 External Memory Algorithm Using Multiple Pivots (Partitions)

In order to improve I/O performance, we present an external memory algorithm which differs from the internal memory algorithm in that it uses multiple pivots during partitioning. Instead of creating two sets at a time, n partitions are cre-

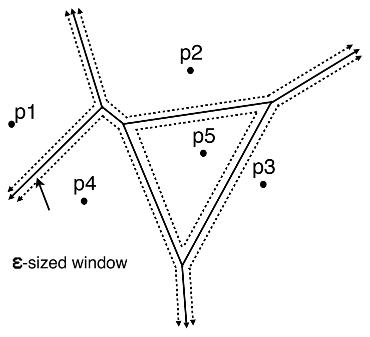


Fig. 14. Multiple partitions are created using multiple pivots, $p1$ through $p5$, along with corresponding window partitions (dashed lines).

```
procedure QuickjoinExt(eps, distFunc, objFile)
begin
  if( sufficientInternalMemory( objFile.size ) )
      objs ← readIntoInternalMemory( objFile );
      Quickjoin( eps, distFunc, objs );
      return;
  end if

  pivots ← selectRandomPivots( NUM_PIVOTS );
  (part[], win[][]) ← PartitionExt(eps, distFunc, objFile, pivots);

  for i=0:NUM_PIVOTS-1 do
   QuickjoinExt(eps, distFunc, part[i]);
  end loop
  for i=0:NUM_PIVOTS-2 do
    for j=i+1:NUM_PIVOTS-1 do
      QuickjoinWinExt(eps, distFunc, win[i][j], win[j][i] );
    end loop
  end loop
end
```

Fig. 15. The external memory version of the Quickjoin algorithm.

ated for each call to the `Partition` function, based on n randomly chosen objects (pivots). In this way, I/O is reduced since multiple partitionings occur at once and smaller partitions are created, thereby allowing for the objects to be read entirely into internal memory sooner. A 2-dimensional example is shown in Fig. 14 using five pivots, which results in partitions that form a Voronoi diagram [Voronoi 1909]. This technique is similar to the multiple-pivot similarity search technique used by GNAT (Geometric Near-neighbor Access Tree) [Brin 1995] for generalized hyperplane partitioning and an extension of the vp-tree [Yianilos 1993] for ball partitioning.

The partitions formed by the external memory algorithm are stored externally on disk and can be read and written sequentially using block I/O for efficiency. Once the partitions are small enough to fit in internal memory, the internal memory algorithm can be called for faster processing.

```
function PartitionExt(eps, distFunc, objFile, pivots)
          : (part[], win[][])
begin
   Initialize part and win files.

   while !EOF(objFile) do
      a ← objFile.ReadObject();
      minDist ← MAX_DOUBLE;
      for each i=0:NUM_PIVOTS-1  do
         dist ← distFunc(a,pivots[i]);
         if( dist < minDist )
            minDist ← dist;
            minPivot ← i;
         end if
      end loop
      part[minPivot].insert(a);
      p ← pivots[minPivot];
      for each i=0:NUM_PIVOTS-1  do
         if( i != minPivot )
            pX ← pivots[i];
            dist ← (distFunc(a,p)-distFunc(a,pX))/2;
            if( -dist < eps )
               win[i][j].insert(a);
            end if
         end if
      end loop
   return (part, win);
end
```

Fig. 16. The external partitioning function determines, for each object `a`, which pivot, `p`, is the closest, and places the object in the corresponding `part` partition. Also, for every other pivot, `pX`, if the distance from `p` to the generalized hyperplane between `p` and `pX` is less than or equal to `eps`, then the object `a` is also inserted into the corresponding window partition.

The external memory algorithm using multiple pivots is given in Fig. 15. First, if there is sufficient internal memory, the externally stored objects, `objFile`, are read into memory and the internal memory algorithm is used on line 5. Otherwise, `NUM_PIVOTS` random objects are chosen on line 9, and the objects are partitioned on line 10 using the `PartitionExt` function, given in Fig. 16. Note that since the objects are in external memory, the random objects (pivots) can be retrieved using non-sequential access, or, more efficiently, they can be selected and saved when the objects are written to external memory during the previous partitioning phase. The external partitioning function, `PartitionExt`, places each object into one `part[]` partition, corresponding to the closest pivot. The `win` set of partitions contains duplicates of the objects (stored externally) which correspond to objects that might be within `eps` of a partition. For instance, the `win[i][j]` partition correspond to sets of objects that are within the `part[i]`, but might be within `eps` of objects in `part[j]`. The `QuickjoinExt` function is recursively called on line 13 to join the `part` partitions while the `QuickjoinWinExt` procedure, shown in Fig. 17, is called on line 17 to join the `win` partitions.

The `PartitionExt` function, shown in Fig. 16, partitions the objects around a set pivots. The concept is similar to generalized hyperplane partitioning in Section 3.2,

```
 1 procedure QuickjoinWinExt(eps, distFunc, objFile1, objFile2)
 2 begin
 3   totalSize ← objFile1.size + objFile2.size;
 4   if( sufficientInternalMemory( totalSize ) )
 5       objs1 ← readIntoInternalMemory( objFile1 );
 6       objs2 ← readIntoInternalMemory( objFile2 );
 7       QuickjoinWin(eps, distFunc, objs1, objs2);
 8       return;
 9   end if
10
11   pivots ← selectRandomPivots( NUM_PIVOTS );
12   (part1[], win1[][]) ← PartitionExt(eps, distFunc, objFile1, pivots);
13   (part2[], win2[][]) ← PartitionExt(eps, distFunc, objFile2, pivots);
14
15   for i=0:NUM_PIVOTS-1 do
16    QuickjoinWinExt(eps, distFunc, part1[i], part2[i]);
17   end loop
18   for i=0:NUM_PIVOTS-2 do
19     for j=i+1:NUM_PIVOTS-1 do
20       QuickjoinWinExt(eps, distFunc, win1[i][j], win2[j][i] );
21       QuickjoinWinExt(eps, distFunc, win1[j][i], win2[j][i] );
22     end loop
23   end loop
24 end
```

Fig. 17. The `QuickjoinWin` function from Fig. 9 is also modified to create multiple partitions at a time.

where a generalized hyperplane exists between each pair of objects. For each object, the nearest pivot is found using the loop on line 9, and the object is inserted into the corresponding `part` partition on line 16. Then, in the loop starting on line 18, the distance to every other pivot is checked and if the distance to the generalized hyperplane, calculated on line 21, is less than or equal to `eps`, the object is also added to the corresponding window, `win`, partition. As in Section 3.2, exact distances, if applicable, can be used to calculate the hyperplane on line 21 to improve performance. Note that since the data resides in external memory, the objects should be read sequentially in blocks for efficiency.

The `QuickjoinWinExt`, shown in Fig. 17, joins two sets objects, `objFile1` and `objFile2`, using multiple pivots. Both sets of objects are partitioned using the same set of pivots on lines 12 and 13. Next, corresponding `part1` and `part2` partitions are joined on line 16 as well as corresponding window partitions on lines 20 and 21.

3.5 Split Windows

As a performance improvement, when the data is partitioned in two for internal partitioning (see Fig. 7), instead of two windows, four windows can be returned. This is illustrated in Fig. 18 with generalized hyperplane partitioning (Section 3.2), although any form of partitioning could be used as well. In the figure, the window partition `winL` from Fig. 11 is split into partitions `winInnerL` and `winOuterL`, and the window partition `winG` is split into `winInnerG` and `winOuterG` partitions. The inner windows (`winInnerL` and `winInnerG`) contain the window objects within $\frac{\epsilon}{2}$ of the pivot or hyperplane, and `winOuterL` and `winOuterG` contain the remaining objects. In this way, the join between the two outer windows, `winOuterL` and

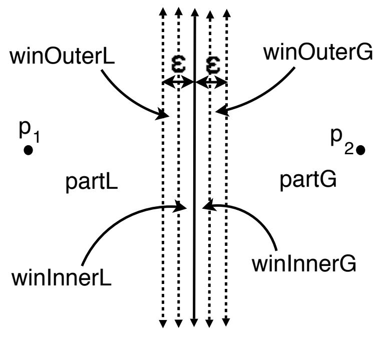


Fig. 18. For internal partitioning, the windows can be split further in order to improve performance. As shown with generalized hyperplane partitioning, each window is split so that the inner windows, (`winInnerL` and `winInnerG`), contain objects within $\frac{\epsilon}{2}$ of the hyperplane, while the outer windows (`winOuterL` and `winOuterG`) contain the remaining objects that lie within ϵ of the hyperplane. Since `winOuterL` and `winOuterG` are more than ϵ apart, they do not need to be joined.

```
procedure Quickjoin(eps, objs, distFunc)
begin
  if( objs.size < constSmallNumber )
   NestedLoop(eps, distFunc, objs);
   return;
  end if

  p1 ← randomObject(objs);
  p2 ← randomObject(objs-r);
  (partL, partG, winInnerL, winOuterL, winInnerG, winOuterG) ←
     PartitionSplitWindows(eps, distFunc, objs, p2, p2);

  QuickjoinWin(eps, distFunc, winInnerL, winInnerG);
  QuickjoinWin(eps, distFunc, winInnerL, winOuterG);
  QuickjoinWin(eps, distFunc, winOuterL, winInnerG);
  Quickjoin(eps, distFunc, partL);
  Quickjoin(eps, distFunc, partG);
end
```

Fig. 19. A variant of the Quickjoin function given in Fig. 5 modified to subdivide each window into two parts to improve performance, as illustrated in Fig. 18

`winOuterG`, can be avoided since they are further than ϵ apart. The remaining combinations of split windows are joined, as shown in Fig. 19.

4. ANALYSIS

The performance of the Quickjoin algorithm in terms of both CPU and I/O depends on how well the data is partitioned at each step. As with the Quicksort algorithm [Hoare 1962], because of the random nature of the algorithm, the Quickjoin algorithm will likely partition the data reasonably well on average, and each data object will be partitioned $O(\log(n))$ times. However, at each level of recursion, the size of the dataset grows due to the window partitions, leading, as will be shown in Section 4.1, to $O(n(1+w)^{\lceil \log(n) \rceil})$ performance on average, where w is the average fractional size of the window partitions. As shown experimentally in Section 5.1

with Fig. 22b, w is dependent upon ϵ, and as expected, the algorithm will perform well for smaller ϵ, and performance will degrade for larger ϵ values, as do all algorithms. Additionally, Section 4.2 discusses the I/O performance of the external memory algorithm and Section 4.3 discusses the internal memory requirements.

4.1 CPU Performance Analysis

The basic performance analysis for the `Quickjoin` procedure from Fig. 5 is given in Equation 3, where QJ denotes a call to the `Quickjoin` procedure, P denotes a call to the `Partition` function from Fig. 7, and QJW denotes a call to the `QuickjoinWin` procedure from Fig. 9. Variable x represents the size of the object set. The `Partition` function divides the x objects into two partitions, represented by ax and $(1-a)x$, where a is the fractional size of the first partition and $(1-a)$ is the fractional size of the second partition. These two partitions are passed to the `Quickjoin` procedure, as represented by $QJ(ax)$ and $QJ((1-a)x)$. The `Partition` function also creates two window partitions, which are passed to the `QuickjoinWin` procedure, as represented by $QJW(wax, w(1-a)x)$, where w represents the fraction of objects from each partition that lie within ϵ of the partition boundary (the generalized hyperplane, for example) and are inserted into the window partitions. This analysis uses single values for a and w, even though they will vary at each level of the recursion. However, for bounding purposes, these values can be assumed to be average or maximum values. The analysis will be unchanged as long as a and w are between 0 and 1, inclusively, which is true.

$$QJ(x) = P(x) + QJ(ax) + QJ((1-a)x) + QJW(wax, w(1-a)x) \tag{3}$$

The performance of the `QuickjoinWin` procedure is similar to the `Quickjoin` procedure, and is given in Equation 4, where x_1 and x_2 are the sizes of the two input partitions.

$$\begin{aligned} QJW(x_1, x_2) &= P(x_1) + P(x_2) + QJW(ax_1, ax_2) + QJW((1-a)x_1, (1-a)x_2) \\ &+ QJW(wax_1, w(1-a)ax_2) + QJW(w(1-a)x_1, wax_2) \end{aligned} \tag{4}$$

The partitioning function, P, scans its data, and applies the distance function once or twice to each object, depending on whether ball partitioning or hyperplane partitioning is used, respectively. Letting d be the cost of the distance function, the cost of partitioning the data is $P(x) = cdx$, where c is 1 or 2, plus any constant overhead. Using this equation to replace $P(x)$ and expanding Equation 3 using both Equations 3 and 4 yields Equation 5.

$$\begin{aligned} QJ(x) &= cdx + acdx + QJ(a^2x) + QJ(a(1-a)x) + QJW(wa^2x, w(a(1-a)x) \\ &+ (1-a)cdx + QJ((1-a)ax) + QJ((1-a)^2x) \\ &+ QJW(w(1-a)ax, w(1-a)^2x) + wacdx + w(1-a)cdx \\ &+ QJW(wa^2x, wa(1-a)x) + QJW(wa(1-a)x, w(1-a)^2x) \\ &+ QJW(w^2a^2x, w^2(1-a)^2x) + QJW(w^2a(1-a)x, w^2a(1-a)x) \end{aligned} \tag{5}$$

Intuitively, in the algorithm, at each level of the recursion, all of the objects are partitioned once. This can be seen by collecting just the cdx terms and ignoring the QJ and QJW terms, yielding $cdx + cdx(1 + w)$, since the a terms drop out. Objects in each window are also partitioned, which is reflected in the $(1+w)$ term.

Continuing the expansion further yields the sequence in Equation 6, where each level is shown in brackets.

$$QJ(x) = cdx + [cdx(1+w)] + [cdx(1+2w+w^2)] + [cdx(1+3w+3w^2+w^3)] + ... \quad (6)$$

Equation 6 is the expansion of the recurrence shown in Equation 7. Intuitively, at each level, the number of objects partitioned is $(1 + w)$ times the previous level.

$$QJ(x) = cdx \sum_{i=0}^{\infty} (1 + w)^i \quad (7)$$

This equation is unbounded and growing. However, the depth of recursion is limited. The Quickjoin algorithm partitions the data in the same manner as the Quicksort algorithm, which is known to have an average case depth of $O(\log(n))$ [Ja'Ja' 2000] (see Appendix A.1 for an analysis). Using this fact yields Equation 8.

$$QJ(x) = cdx \sum_{i=0}^{\lceil \log(n) \rceil} (1 + w)^i \quad (8)$$

Using induction, it can be show that the summation in Equation 8 has an upper bound of $O((1 + w)^{\lceil \log(n) \rceil})$ (see Appendix A.2 for proof), which yields a final bound on the Quickjoin algorithm of $O(n(1 + w)^{\lceil \log(n) \rceil})$. This bound shows that the Quickjoin algorithm will perform well $(O(n))$ for smaller average windows since the $(1 + w)^{\lceil \log(n) \rceil}$ term will be small. This term will be maximum when w is one, meaning that all objects are put into the windows. In this case, $2^{\log_b(n)} \leq n$ for $b \geq 2$, where b is the constant base of the logarithm. In other words, the performance for the largest window size is $O(n^2)$. The theoretical worst case, as with the Quicksort algorithm, occurs if the depth of recursion approaches n. In this case, the Quickjoin algorithm would have performance $O(n2^n)$. However, this is unlikely to occur, and the algorithm can be written to avoid such worst cases by, for example, using median values instead of random objects for the ball partitioning method or immediately repartitioning the data when the data is partitioned poorly.

Furthermore, the w factor is dependent on ϵ, and as shown experimentally in Section 5.1, this dependence is nearly linear. For smaller ϵ values, w is also small and will not have a significant impact on performance. In particular, only for larger ϵ values will w have a significant impact on performance.

4.2 I/O Analysis

For the external memory algorithm (Section 3.4), given a fixed amount of internal memory, the external data must be partitioned until each external partition will fit within internal memory, at which time the internal memory algorithm can be used (Section 3.1). The analysis to determine the amount of I/O is nearly identical to that of the performance analysis given in Section 4.1 since each object is read from memory once each time it is partitioned, but the I/O analysis differs in two aspects. First, since multiple pivots are used, the partition function calculates the distance function for each pivot. However, the previous analysis used $P(x) = cdx$ for the performance of the partitioning function and the additional pivots can be incorporated into the c factor. Furthermore, the multiple pivots create a larger fan-out which shortens the depth of the recursion. This would be incorporated into the base of the logarithm of Equation 8, leaving the analysis unchanged.

The second difference is that the external algorithm stops, switching to the internal algorithm, when there is sufficient internal memory to hold the partition. Letting M be the number of objects that will fit in internal memory, the depth of the recursion is $\log(n/M)$. Additionally, since the data is read in blocks, the n variable should be represented by N, where $N = \frac{n}{m}$, and m is the number of objects per block. Applying these two modifications to the final average case performance yields an I/O average performance of $O(N(1+w)^{\lceil \log(N/M) \rceil})$.

4.3 Internal Memory Analysis

For the internal memory memory algorithm, the size of the dataset will mostly determine how much internal memory is required, that is, there must be a sufficient amount of internal memory to store the entire dataset in memory. To keep the internal memory usage from growing, the window partitions should use pointers, rather than copies of the data. However, there is a concern that the number of pointers could grow beyond the internal memory limits when the depth of the recursion is high. As we show, the number of pointers is bounded, though, and this bound can be used to determine if there will definitely be sufficient internal memory.

At each level of recursion (calls to `QuickjoinWin` in Fig. 9), the partitioning function expands the current pointer set by creating window partitions. Note that calls to the `Quickjoin` procedure do not increase the number of pointers since it partitions the original dataset. The window partitions created are given to the `QuickjoinWin` procedure to be processed and are released when the call completes, meaning that the `Quickjoin` procedure has no pointers when it calls itself. To determine the maximum number of pointers, $max\_pointers$, let w be the maximum window size (as a fraction of the total dataset size) created by the partitioning function and given to the `QuickjoinWin` procedure. In the `QuickjoinWin` procedure, two window partitions are created from the dataset, and hence, the number of new pointers created at each level of recursion is at most $2 * w * n$, where n is the size of the dataset given to `Quickjoin` or the combined size of the datasets given to `QuickjoinWin`. The next level of recursion repeats this, producing $2 * w * (2 * w * n)$ pointers. Since the algorithm employs depth-first recursion, to derive the maximum number of pointers we simply sum these values to the maximum depth of recursion, MAX_DEPTH, as shown in Equation 9. As each call returns, the pointers should be released to conserve internal memory.

$$max\_pointers = \sum_{i=1}^{MAX\_DEPTH} (2 * w)^i * n \tag{9}$$

Equation 9 converges if MAX_DEPTH is infinite and w is less than 0.5. If w is 0.25, then it converges to n. If w is 0.5 or greater, than potentially, the entire dataset is passed as windows to the `QuickjoinWin` and the algorithm is behaving more like a nested-loop join. In this case, the best performance will be $O(n^2)$ since that is how many results will be expected (everything is near everything else), and the algorithm should just perform a nested-loop join. If w is consistently less than 0.5 and greater than 0.25, then the algorithm will be behaving more like the nested-loop join. In this case, the algorithm should also just use a nested-loop join to avoid unnecessary partitioning. In practice, this condition where the algorithm is behaving like a nested-loop join can be detected by the algorithm if the data is continually partitioning the same data with large window sizes. Alternatively, the algorithm can track the number of pointers used. If a predefined limit, such as n, is reached, then the algorithm can end the recursion by using a nested-loop join.

In practice, the performance will be much better than the worst case. When the data is initially partitioned, the window sizes are larger, both in real size and as a fraction of the whole dataset. However, the separation will tend to be better, thereby creating a very short recursive depth. As the depth of recursion increases, the data will be partitioned into smaller chunks with less separation, resulting in deeper recursion, but starting with smaller windows (number of pointers).

As for the external memory algorithm, the internal memory requirements algorithm are quite low. In the algorithm, shown in Fig. 15, a block of data, or even a single datum, can be read, and then immediately written to a partition. Beyond these minimal requirements, the amount of internal memory used is determined by the stopping condition `sufficientInternalMemory` on line 3, which indicates that all of the data can be read into internal memory and processed using the internal memory algorithm. `sufficientInternalMemory`. More internal memory will improve the performance by allowing larger partitions to be read into memory and to be fully processed, rather than having to incur the I/O cost of writing the data back out to external memory.

5. EXPERIMENTS

This section presents the results of similarity join experiments on real datasets. The algorithms were implemented in C++ and the experiments were performed on a computer with a Pentium 3 processor and 512MB of RAM, running Windows XP. The similarity join algorithms were tested using color data, forest cover type data, and NSF research awards abstracts from the UC Irvine Knowledge Discovery in Databases (UCI-KDD) repository [Hettich and Bay 1999]. The color data consists of multi-dimensional features extracted from a Corel image data collection with 68,040 objects. The experiments presented here used the 9-dimensional color moment data, with data values ranging from −4.8 to 4.4, and 32-dimensional color histograms, with data values ranging from 0 to 1. On disk, the color moments data is 6.1 MB large and the histogram data is 19.2 MB. The forest cover type data from the repository consists of 581,012 objects with 54 multi-dimensional features describing forest regions, using such features as elevation, slope, shade, and binary values for soil type. We only used the 10 non-binary features in the dataset. These features have different ranges of values, such as 0 to 3,858 for the elevation and

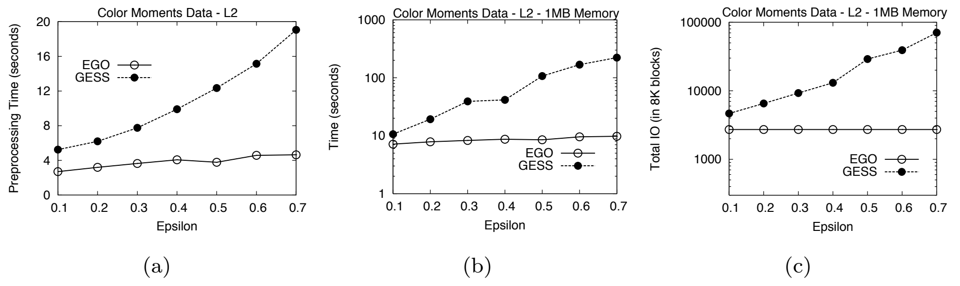


Fig. 20. (a) Internal memory sort times for the color moments data. (b) External memory sort times for the color moments data. (c) Total I/O counts for the external memory sort.

0 to 255 for the shade values. On disk, this dataset takes up 76.1 MB. We used 10,000 sentences from the NSF research awards abstract data to perform metric space experiments. On disk, the dataset takes up 1.7 MB. The sentences ranged in length from 30 to 649 words, with an average length of 169 words.

In the experiments, the variations of the Quickjoin algorithms are compared to the EGO algorithm, described in Section 2.1, and the GESS algorithm, described in Section 2.2. The EGO algorithm was implemented with the suggested EGO* improvements [Kalashnikov and Prabhakar 2003], and the block size of the EGO algorithm was tuned to optimize its performance. The GESS algorithm was implemented using both a Hilbert order and a Morton (Peano-Hilbert) order. Since both orders performed similarly, only results with the Morton order are shown. We used a kd-trie partitioning and tuned the GESS algorithm to optimize its performance by adjusting the *msl* (maximum split level) and k (maximum number of split-lines) factors, described in [Dittrich and Seeger 2001]. We implemented the linear codes as bit vectors, providing unlimited depth of levels.

The preprocessing times for the EGO and GESS methods are shown in Fig. 20 for both (a) internal and (b) external memory, and (c) total I/O. The preprocessing time consists of sorting the data, and for the GESS method, determining the appropriate grid cell (note that this can be done as the data is fed into the sorting operator). We used a Quicksort algorithm [Hoare 1962] for the internal sorting operation and an external merge sort for external sorting [Cormen et al. 2001]. For both methods, the sort time increases with increasing ϵ. The EGO method uses ϵ to calculate the EGO grid cell, as described in Section 2.1. Since the grid cells are of size ϵ, as ϵ increases, there are fewer cells in which to place objects, thereby making it more likely that two objects will be in the same grid cell in a single dimension. Therefore, on the average, more dimensions will be needed to determine which object takes precedence in the ordering. For the GESS method, as ϵ increases, the size of each hyper-square around each point increases. This increases processing time by requiring more replication since the larger hyper-squares are more likely to intersect grid boundaries. Similarly, as shown in Fig. 20c, the total I/O increases for the GESS method because of increased replication.

Results of experiments on datasets that only require internal memory (no disk I/O) are presented in Section 5.1, results of experiments that require external memory (disk I/O) are presented in Section 5.2, and results of metric space experiments are presented in Section 5.3. While external memory is required to handle datasets of any size, the internal memory experiments not only demonstrate the various internal memory Quickjoin algorithms, but are also of practical significance since large datasets can be processed entirely in memory with today's internal memory capacities. Additionally, experiments with pivot selection are presented in Section 5.4.

5.1 Internal Memory Similarity Joins

This section presents the results of experiments using only internal memory and datasets that fit entirely into internal memory. The four internal memory variations of the Quickjoin algorithm were used in the experiments in this section:

(1) The ball-partitioning method described in Section 3.1, using the average distance to a pivot to partition the data, referred to as QuickJoinBall in the plots.

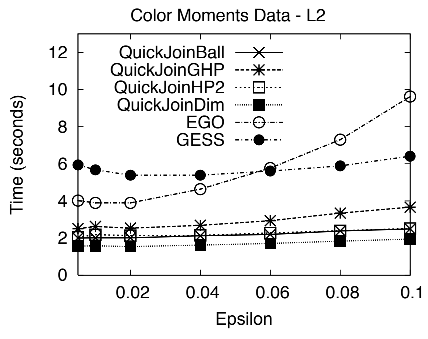
(2) The generalized hyperplane version described in Section 3.2, using two randomly chosen pivots, referred to as QuickJoinGHP.

(3) The L_2-norm hyperplane described in Section 3.2, using two randomly chosen pivots referred to as QuickJoinHP2.

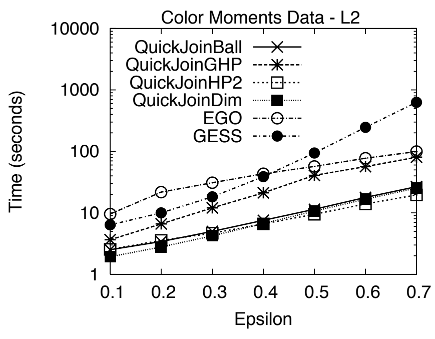
(4) The dimensional version for vector data described in Section 3.3, using the method of cycling through the dimensions to chose the pivot for partitioning, referred to as QuickJoinDim.

Each method was also implemented with the split windows enhancement described in Section 3.5.

The plots in this section show the time needed to perform the similarity join, assuming an unsorted dataset. The Quickjoin algorithms immediately execute the join, while the EGO and GESS methods preprocess the data by sorting the data, and in the case of the GESS method, assign the data linear order codes. Except for the smallest of experiments, this preprocessing tends to take up only a small fraction of the total processing time.



(a)

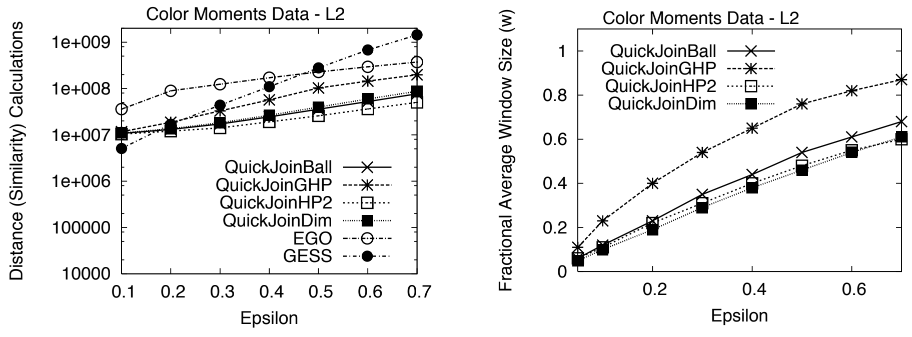


(b)

Fig. 21. Experiments with color moment data for (a) smaller ϵ values using an L_2-norm, and (b) larger ϵ values using an L_2-norm.

The results of similarity join experiments with the 9-dimensional color moment data, with data values ranging from −4.8 to 4.4, are shown in Figs. 21a and 21b, both using an L_2-norm. Fig. 21a shows experiments with smaller ϵ values (less than 1%), which represent queries where a few hundred results are reported, while Fig. 21b shows experiments with larger ϵ values, which represent queries where more results are reported, ranging from a few hundred up to several hundred thousand results. For smaller ϵ values, shown in Fig. 21a, the Quickjoin algorithms outperform the EGO and GESS algorithms. The GESS algorithm exhibits nearly a constant performance for these smaller ϵ values, while the EGO method is more sensitive to the ϵ value. The Quickjoin algorithms, except for the generalized hyperplane version (QuickJoinGHP), also exhibit an almost constant performance with increasing ϵ values. In these cases, the ϵ sized window partitions represent only a small fraction of the processing.

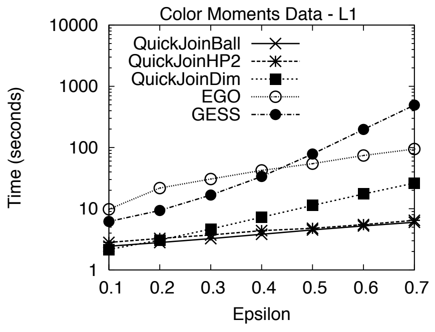
For larger ϵ values, shown in Fig. 21b, the Quickjoin algorithms begin to perform slightly worse. The ball partitioning, hyperplane partitioning and dimensional versions of the Quickjoin algorithm exhibit nearly identical performance on this dataset as well as most other datasets. The generalized hyperplane version (QuickJoinGHP) which makes use of a lower bound on the distance (hence resulting in a larger window) performs worse than these variations. Nevertheless, the Quickjoin algorithms still outperform both the EGO and the GESS algorithms. The GESS algorithm begins to degrade in performance due to increased replication. Tuning the algorithm to decrease replication causes more objects to reside in shallower levels, which results in worse performance.



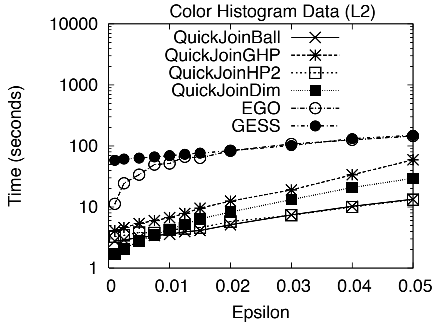
(a) (b)

Fig. 22. For the color moment data using an L_2-norm, (a) the number of distance (similarity) calculations, and (b) the average fractional window size created by the `Partition` function.

Fig. 22a shows the number of distance calculations for the experiments shown in Fig. 21b, which demonstrates that the Quickjoin algorithms improved performance over the EGO algorithm is due to the fewer distance (similarity) tests that it performs. At smaller ϵ values, the GESS algorithm performs fewer similarity calculations, though its performance is still worse due to the preprocessing time. For larger ϵ values, the GESS algorithm performs more similarity calculations because



(a)



(b)

Fig. 23. (a) Experiments with color moment data for larger ϵ values using an L_1-norm. (b) Experiments with 32-dimensional color histogram data, using the L_2-norm.

of the increased size of the dataset due to a greater rate of replication. Further insight into the performance of the Quickjoin algorithms can be gained by examining the fractional size of the windows created by the `Partition` function from Fig. 7, which is called by the `Quickjoin` and `QuickjoinWin` procedures given in Figs. 5 and 9. As shown in Fig. 22b, the average window size increases nearly linearly with ϵ. Note that w is a function of ϵ. In this case, with the color moment data, the data can range in value from -4.4 to 4.8. In 9 dimensions, the maximum ϵ would be $\sqrt{9 \cdot (4.8 - (-4.4))^2} = 27.6$. However, w would be close to its maximum value of 1 at a smaller value, as can be seen with the plot of QuickJoinGHP, where w approaches 1. Intuitively, the size of the result set increases dramatically for the ϵ values shown in the plot, and for even larger values, nearly all pairs of objects will be returned, resulting in $O(n^2)$ performance of the evaluated algorithms. Fig. 22b also illustrates the fact that the generalized hyperplane version (QuickJoinGHP) uses window sizes that are twice as large as the L_2-norm hyperplane version (QuickJoinHP2).

Fig. 23a shows the results of repeating the experiments in Fig. 21b using an L_1-norm. In this case, the EGO and GESS algorithms are shown to perform nearly the same as before, while the performance of the Quickjoin method improves. Note that the QuickjoinHP2 method is not shown as it assumes the use of the L_2-norm. Because the distances between objects are larger when using the L_1-norm as compared to when using the L_2-norm, the windows created by the Quickjoin method are relatively smaller, thereby leading to improved performance. Also, since the distances between objects are increased, the number of objects in the join result tends to be in the hundreds, rather than in the thousands, as with the join result from Fig. 21b.

The results of experiments with the 32-dimensional color histogram data are shown in Fig. 23b. The data ranges in value from 0 to 1, and the ϵ values produce thousands of results. In this case, the Quickjoin algorithms significantly outperforms the EGO and GESS methods, which interestingly, perform identically for the larger ϵ values.

5.2 External Memory Similarity Joins

This section presents the results of experiments that require the algorithms to use external memory to perform the similarity join, which demonstrates the I/O performance of the algorithms. All of the algorithms perform sequential I/O. The algorithms were again tested using both sets of color data, and additionally the forest cover type data. The amount of internal memory available to the algorithms is controlled, thereby forcing them to use external memory. The overall performance of the Quickjoin algorithms is still better than that of the EGO and GESS methods, even though the Quickjoin methods perform slightly more I/O than the EGO method. Since the CPU costs are a significant factor in the performance of similarity joins due to the expensive distance (similarity) calculations, the Quickjoin methods perform better overall since their internal memory performance is so much better, as was shown in Section 5.1.

The external memory version of the Quickjoin algorithm was implemented as described in Section 3.4. The window partitions are calculated using both the generalized hyperplane applicable to any distance metric (again termed QuickjoinGHP in the plots) and the L_2-norm hyperplane (again termed QuickjoinHP2 in the plots). As appropriate, the internal memory Quickjoin algorithm used is the corresponding generalized hyperplane or hyperplane version of the internal memory Quickjoin algorithm.

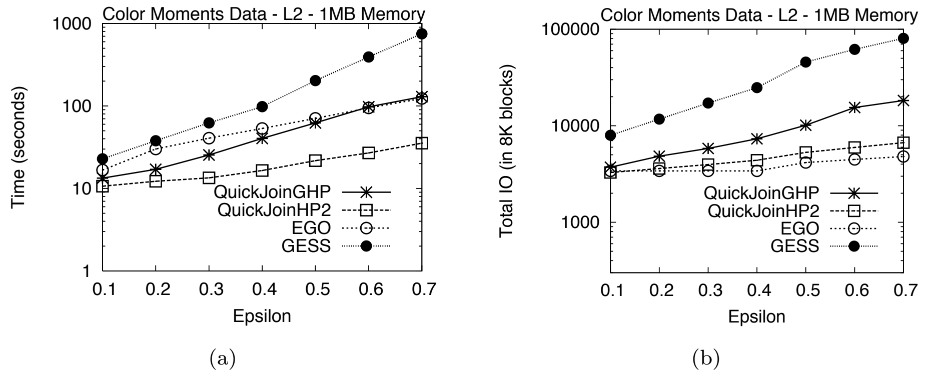


Fig. 24. Experiments with color moment data for larger ϵ values using an L_2-norm and with memory restricted to 1MB. (a) Execution time, and (b) I/O.

Fig. 24a shows the results of experiments with external memory, which duplicate the experiment in Fig. 21b, but with internal memory limited to 1MB. The hyperplane version of Quickjoin still performs better than the EGO method, while the performance of the generalized hyperplane version is similar to the EGO method. Fig. 24b shows the I/O performance for the experiments in Fig. 24a. Here we find that the hyperplane Quickjoin method performs slightly more input and output than the EGO method, but the generalized hyperplane version performs more I/O, which explains the performance difference shown in Fig. 24a. While the performance of the GESS method compared well to that of the EGO method for the internal memory experiments (Section 5.1), the GESS method did not do as well

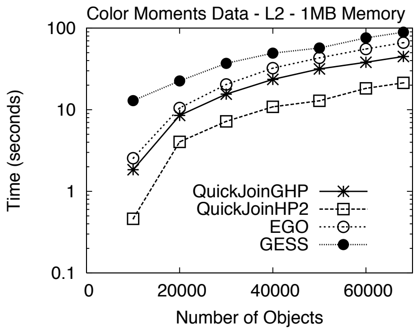


Fig. 25. A scalability experiment with color moment data for an ϵ value of 0.4 using an L_2-norm and with memory restricted to 1MB.

for the external memory experiments, as shown in Fig. 24a, due to the increased replication, which impacts I/O, as shown in Fig. 24b.

Additionally, Fig. 25 shows the results of varying the dataset sizes. As shown, the different algorithms maintain their relative performance at various dataset sizes, with the Quickjoin methods performing the best.

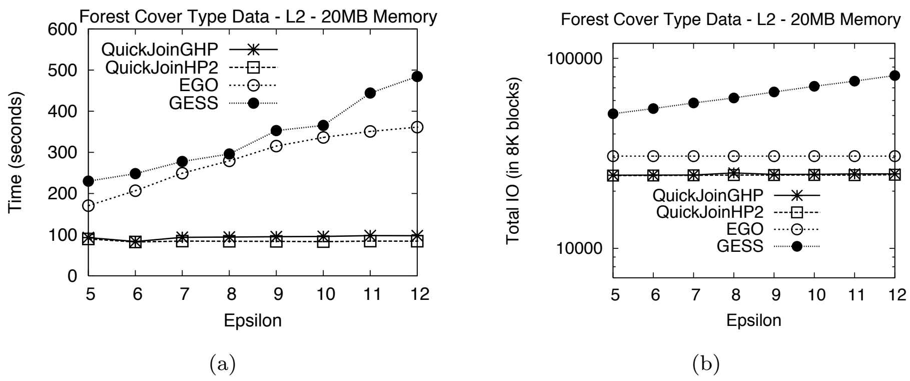
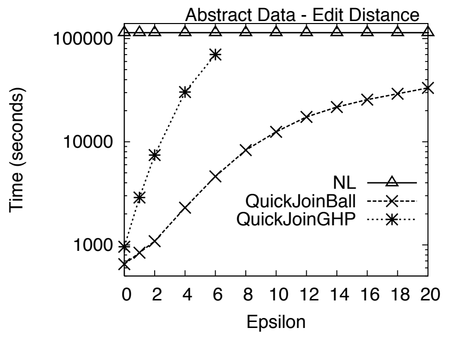


Fig. 26. Experiments with forest cover type data for using an L_2-norm and with memory restricted to 20MB. (a) Execution time, and (b) I/O.

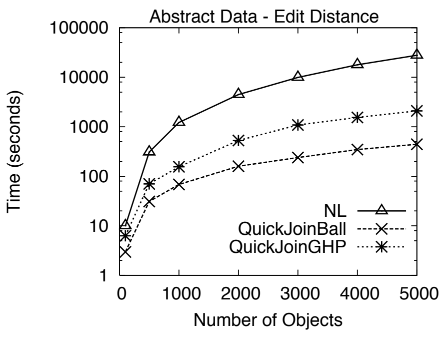
To test the algorithms on a larger dataset, we used 10 dimensions of the UCI-KDD forest cover type data [Hettich and Bay 1999], which consists of 581,012 objects. The external memory algorithms were allowed 20 MB of memory. The number of results returned ranged from 31 for an ϵ of 5 to 2,046 for an ϵ of 12. The results are shown in Fig. 26. The Quickjoin algorithms outperformed the EGO and GESS methods with near constant performance. The EGO and GESS methods were more effected by increasing ϵ, though they performed similarly on this dataset.

5.3 Metric Space Similarity Joins

In order to demonstrate the metric space capabilities of the Quickjoin algorithm, we ran experiments on 10,000 sentences extracted from the NSF research awards



(a)



(b)

Fig. 27. Internal memory experiments with 10,000 sentences extracted from scientific abstracts using the edit distance with varying (a) ϵ and (b) number of objects with an ϵ of 2.

abstracts in the UCI-KDD repository [Hettich and Bay 1999]. The sentences ranged in length from 30 to 649 words, with an average length of 169 words. For the metric, we used the Levenshtein edit distance [Levenshtein 1966], which is a measure of string (in this case, sentences) similarity. A score of zero indicates identical strings, while a higher score indicates the number of letter differences between the strings. We compared the metric version of the Quickjoin algorithm (the ball partitioning method, QuickJoinBall and the generic hyperplane version, QuickJoinGHP) to the naïve nested-loop join implementation (NL) [Elmasri and Navathe 2004] by varying ϵ, as shown in Fig. 27a. With a small edit distance (an ϵ of two), the ball-partitioning version of the Quickjoin algorithm performed two orders of magnitude better than the nested-loop implementation. As ϵ increases, the performance difference drops to just under an order of magnitude better. The generalized-hyperplane version of the Quickjoin algorithm performed well only for the lowest ϵ values, and quickly degraded to the performance of the nested-loop join. The number of results (pairs of sentences within the specified ϵ distance) did not vary much with ϵ, ranging from 1907 to 2346 pairs returned for ϵ values of 2 to 20. Additionally, as shown in Fig. 27b, the Quickjoin methods scale better than the nested-loop implementation.

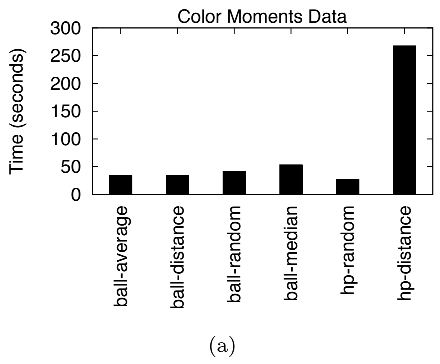
5.4 Pivot Selection

As described in Section 3.1, various methods can be used to choose a pivot when partitioning the data. We examined four methods for ball partitioning:

(1) Using the average distance, referred to as ball-average in the figure.
(2) Using half of the maximum distance, referred to as ball-distance.
(3) Using a randomly chosen object's distance, referred to as ball-random.
(4) Using the median distance, referred to as ball-median.

We examined two methods for hyperplane partitioning:

(1) Using a randomly chosen second pivot, referred to as hp-random.
(2) Using the object with the maximum distance from the first pivot as the second pivot, referred to as hp-distance.



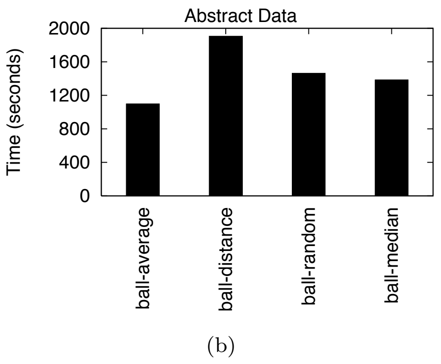


Fig. 28. (a) Experiments with alternative choices for pivot selections using the Color Moments data with an ϵ of 0.7. (b) Similar experiments using the abstract data with an ϵ of 2.

The results are shown in Fig. 28. For the ball-partitioning method, using the average distance (i.e., ball-average) for the pivot worked best on both datasets. Using the median distance (i.e., ball-median) was slower since finding the median can be costly. With the metric space abstract data in Fig. 28b, the median distance did prove more effective than with the color moment data since it creates even partitions, though the average distance was still faster. For the hyperplane-partitioning method, the random pivot (i.e., hp-random) proved much faster than choosing the most distant pivot (i.e., hp-distance), as shown in Fig. 28a. The experiments with the generalized-hyperplane partitioning on the metric space data were no more effective and are not shown in Fig. 28b.

6. CONCLUDING REMARKS

We have described the Quickjoin algorithm, a novel method for performing a similarity join, and shown it to be highly competitive. The algorithm works by partitioning the data using only the metric-space distance information, rather than using any vector-format information. In effect, the algorithm creates partitions using the data rather than partitioning based on space.

The method overcomes deficiencies that occur with many existing methods based on spatial join techniques that rely on multi-dimensional indices. These problems mostly arise because the data in higher dimensions will tend to reside near the boundaries of the data space and the ability to use a particular dimension to discriminate between objects is reduced. Additionally, these indexed methods are limited to multi-dimensional data. For other methods derived from spatial join techniques [Böhm et al. 2001; Dittrich and Seeger 2001], we have shown experimentally that the Quickjoin method performs better.

Future work involves comparing the Quickjoin technique with similarity join methods that make use of metric space indices [Dohnal et al. 003b] and other techniques used in text processing and data cleaning [Dohnal et al. 003a]. In addition, exploring the possibility of adapting indexed spatial join techniques to use metric-based indices is also worthwhile.

APPENDIX

A.1 Depth of Recursion

The Quickjoin algorithm partitions the data in the same manner as the Quicksort algorithm, which is known to have an average case depth of $O(\log(n))$ [Ja'Ja' 2000]. Here, we adapt the average case analysis of Ja'Ja' [2000] to the Quickjoin algorithm using Equation 3 from Section 4.1. Since we are interested in the depth of the recursion of the `Quickjoin` procedure from Fig. 5, we can drop the call to the `QuickjoinWin` procedure, $QJW()$, yielding Equation 10, with $P(x)$ replaced by cx, as was done with Equation 5, but dropping the unnecessary d constant and letting n denote the total number of objects.

$$QJ(n) = cn + QJ(an) + QJ((1-a)n) \tag{10}$$

In the average case, Equation 10 becomes Equation 11, which takes the average of all possible partitionings.

$$QJ(n) = \left(\frac{1}{n}\left(\sum_{i=0}^{n-1}(QJ(i) + QJ(n-i)\right)\right) + cn \tag{11}$$

With a base case shown in Equation 12.

$$QJ(1) = \Theta(1) \tag{12}$$

Combining $QJ(i)$ and $QJ(n-i)$ and multiplying Equation 11 by n starts the following series of algebraic manipulations.

$$\begin{aligned} nQJ(n) &= 2\left(\sum_{i=0}^{n-1} QJ(i)\right) + cn^2 \\ nQJ(n) - (n-1)QJ(n-1) &= 2QJ(n-1) + cn^2 - c(n^2 - 2n + 1) \\ nQJ(n) &= (n+1)QJ(n-1) + 2cn - c \\ \frac{QJ(n)}{n+1} &= \frac{QJ(n-1)}{n} + \frac{2c}{n+1} - \frac{c}{n(n+1)} \end{aligned} \tag{13}$$

Expanding Equation 13 and dropping the $\frac{c}{n(n+1)}$ term since it is immaterial to an upper bound calculation, leads to the following sequence of equations.

$$\begin{aligned} \frac{QJ(n)}{n+1} &= \frac{QJ(n-1)}{n} + \frac{2c}{n+1} \\ \frac{QJ(n-1)}{n} &= \frac{QJ(n-2)}{n-1} + \frac{2c}{n} \\ \frac{QJ(n-2)}{n-1} &= \frac{QJ(n-3)}{n-2} + \frac{2c}{n-1} \\ &\dots \\ \frac{QJ(2)}{3} &= \frac{QJ(1)}{2} + \frac{2c}{3} \end{aligned} \tag{14}$$

Summing these equations gives

$$\frac{QJ(n)}{n+1} = \sum_{i=3}^{n+1} \frac{2c}{i} = 2c\left(\log_e(n+1) + \gamma - \frac{3}{2}\right) \tag{15}$$

The final equation results from the fact that $\sum_{i=1}^{n+1} \frac{1}{i} = \log_e(n+1) + \gamma$, where the constant is $\gamma = 0.577$ (Euler's Constant). In other words, Equation 15 shows that the `Quickjoin` procedure will perform, on average, $O(n \log(n))$ calculations partitioning the data. Since, at each level of recursion, the `Quickjoin` function partitions all of the n data items, meaning that the `Quickjoin` function will terminate at $O(\log(n))$ depth, on average. Also, since the `QuickjoinWin` uses the same partitioning method, it will also terminate at $O(\log(n))$ depth.

A.2 Upper Bound On $\sum_{i=0}^{\lceil \log(n) \rceil} (1+w)^i$

$$QJ(x) = \sum_{i=0}^{\lceil \log(n) \rceil} (1+w)^i \tag{16}$$

We assert that Equation 16 has an upper bound of $O\left((1+w)^{\lceil \log(n) \rceil}\right)$. This is assertion is true if Equation 17 holds and $c \geq 1$, which we prove by induction.

$$\sum_{i=0}^{\lceil \log(n) \rceil} (1+w)^i \leq c(1+w)^{\lceil \log(n) \rceil} \tag{17}$$

First, the base case of $n = 1$ is as follows, where any $c \geq 1$ satisfies the relation:

$$\sum_{i=0}^{\lceil \log(1) \rceil} (1+w)^i = \sum_{i=0}^{0} (1+w)^i = (1+w)^0 = 1 \leq c(1+w)^{\lceil \log(1) \rceil} = c \tag{18}$$

Assuming Equation 17 is true for n, we prove the induction in two cases. First, if $\lceil \log(n) \rceil = \lceil \log(n+1) \rceil$, then the equation for $n+1$ becomes

$$\begin{aligned} \sum_{i=0}^{\lceil \log(n+1) \rceil} (1+w)^i &= \sum_{i=0}^{\lceil \log(n) \rceil} (1+w)^i \\ &\leq c(1+w)^{\lceil \log(n) \rceil} \\ &= c(1+w)^{\lceil \log(n+1) \rceil} \end{aligned} \tag{19}$$

Equation 19 holds for any $c \geq 1$. Next, if $\lceil \log(n) \rceil = \lceil \log(n+1) \rceil - 1$, then the equation for $n+1$ becomes

$$\begin{aligned} \sum_{i=0}^{\lceil \log(n+1) \rceil} (1+w)^i &= (1+w)^{\lceil \log(n+1) \rceil} + \sum_{i=0}^{\lceil \log(n) \rceil} (1+w)^i \\ &\leq (1+w)^{\lceil \log(n+1) \rceil} + c(1+w)^{\lceil \log(n) \rceil} \end{aligned}$$

$$
\begin{aligned}
&= (1+w)^{\lceil \log(n+1) \rceil} + \frac{c(1+w)^{\lceil \log(n) \rceil + 1}}{1+w} \\
&= (1+w)^{\lceil \log(n+1) \rceil} + \frac{c(1+w)^{\lceil \log(n+1) \rceil}}{1+w} \\
&= (1+w)^{\lceil \log(n+1) \rceil} \cdot \left(1 + \frac{c}{1+w}\right) \\
&= c(1+w)^{\lceil \log(n+1) \rceil} \cdot \left(\frac{1}{c} + \frac{1}{1+w}\right) \\
&\le c(1+w)^{\lceil \log(n+1) \rceil} \qquad (20)
\end{aligned}
$$

The final equation, Equation 20, is true provided that

$$\frac{1}{c} + \frac{1}{1+w} \le 1 \qquad (21)$$

Rewriting Equation 21 gives

$$c \ge \frac{1+w}{w} \qquad (22)$$

Since w is a positive constant, $\frac{1+w}{w}$ can be bounded by c, proving the bound of $O\left((1+w)^{\lceil \log(n) \rceil}\right)$.

ACKNOWLEDGMENTS

We appreciate Professor David Mount's assistance with the analysis of the Quickjoin algorithm.

The support of the National Science Foundation under Grants EIA-0091474 and CCF-0515241, Microsoft Research, and the University of Maryland General Research Board is gratefully acknowledged.

REFERENCES

Aggarwal, C. C. 2003. Towards systematic design of distance functions for data mining applications. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Washington, D.C., 9–18.

Agrawal, R., Faloutsos, C., and Swami, A. N. 1993. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, D. B. Lomet, Ed. vol. 730 of Springer-Verlag Lecture Notes in Computer Science. Chicago, 69–84.

Agrawal, R., Psaila, G., Wimmers, E. L., and Zaït, M. 1995. Querying shapes of histories. In *Proceedings of 21st VLDB International Conference on Very Large Data Bases*, U. Dayal, P. M. D. Gray, and S. Nishio, Eds. Zurich, Switzerland, 502–514.

Aref, W. G. and Samet, H. 1994. Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*. Gaithersburg, MD, 347–354.

Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., and Vitter, J. S. 1998. Scalable sweeping-based spatial join. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, A. Gupta, O. Shmueli, and J. Widom, Eds. New York, 570–581.

Böhm, C., Braunmüller, B., Breunig, M., and Kriegel, H.-P. 2000. High performance clustering based on the similarity join. In *Proceedings of the 9th CIKM International Conference on Information and Knowledge Management*. McLean, VA, 298–305.

Böhm, C., Braunmüller, B., Krebs, F., and Kriegel, H.-P. 2001. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In *Proceedings of the ACM SIGMOD Conference*. Santa Barbara, CA, 379–390.

Böhm, C. and Kriegel, H.-P. 2001. A cost model and index architecture for the similarity join. In *Proceedings of the 17th IEEE International Conference on Data Engineering*. Heidelberg, Germany, 411–420.

Brin, S. 1995. Near neighbor search in large metric spaces. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, U. Dayal, P. M. D. Gray, and S. Nishio, Eds. Zurich, Switzerland, 574–584.

Brinkhoff, T., Kriegel, H.-P., and Seeger, B. 1993. Efficient processing of spatial joins using R-trees. In *Proceedings of the ACM SIGMOD Conference*. Washington, DC, 237–246.

Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. 2001. Searching in metric spaces. *ACM Computing Surveys 33,* 3 (Sept.), 273–322. Also University of Chile DCC Technical Report TR/DCC-99-3, June 1999.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms*, Second ed. MIT Press/McGraw-Hill, Cambridge, MA.

Dittrich, J.-P. and Seeger, B. 2000. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th IEEE International Conference on Data Engineering.* San Diego, CA, 535–546.

Dittrich, J.-P. and Seeger, B. 2001. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* San Francisco, California, 47–56.

Dohnal, V., Gennaro, C., Savino, P., and Zezula, P. 2003. D-index: Distance searching index for metric data sets. *Multimedia Tools Appl. 21,* 1 (sep), 9–33.

Dohnal, V., Gennaro, C., Savino, P., and Zezula, P. 2003a. Similarity join in metric spaces. In *Advances in Information Retrieval, 25th European Conference on IR Research, ECIR 2003.* Pisa, Italy, 452–467.

Dohnal, V., Gennaro, C., and Zezula, P. 2003b. Similarity join in metric spaces using ed-index. In *Database and Expert Systems Applications, 14th International Conference, DEXA 2003*. Prague, Czech Republic, 484–493.

Elmasri, R. and Navathe, S. B. 2004. *Fundamentals of Database Systems*, Fourth ed. Addison-Wesley, Upper Saddle River, NJ.

Enderle, J., Hampel, M., and Seidl, T. 2004. Joining interval data in relational databases. In *Proceedings of the ACM SIGMOD Conference.* Paris, France, 683–694.

Faloutsos, C., Barber, R., Flickner, M., Hafner, J., Niblack, W., Petkovic, D., and Equitz, W. 1994. Efficient and effective querying by image content. *Journal of Intelligent Information Systems 3,* 3/4, 231–262.

Faloutsos, C. and Lin, K.-I. 1995. FastMap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the ACM SIGMOD Conference.* San Jose, CA, 163–174.

Guttman, A. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference.* Boston, 47–57.

Harada, L., Nakano, M., Kitsuregawa, M., and Takagi, M. 1990. Query processing for multi-attribute clustered records. In *16th International Conference on Very Large Data Bases*, D. McLeod, R. Sacks-Davis, and H.-J. Schek, Eds. Brisbane, Queensland, Australia, 59–70.

Hettich, S. and Bay, S. D. 1999. The UCI KDD archive [`http://kdd.ics.uci.edu`]. Irvine, CA: University of California, Department of Information and Computer Science.

Hjaltason, G. R. and Samet, H. 2003. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems 28,* 4 (Dec.), 517–580.

Hoare, C. A. R. 1962. Quicksort. *Computer Journal 5,* 1 (Feb.), 10–15.

Hristescu, G. and Farach-Colton, M. 1999. Cluster-preserving embedding of proteins. Technical report, Department of Computer Science, Rutgers University, Piscataway, NJ.

Huang, Y.-W., Jing, N., and Rundensteiner, E. A. 1997. Integrated query processing strategies for spatial path queries. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, A. Gray and P.-A. Larson, Eds. Birmingham, United Kingdom, 477–486.

Huttenlocher, D. P., Kedem, K., and Kleinberg, J. M. 1992. On dynamic voronoi diagrams and the minimum Hausdorff distance for point sets under euclidean motion in the plane. In *SCG '92: Proceedings of the Eighth Annual Symposium on Computational Geometry*. Berlin, Germany, 110–119.

Jacox, E. and Samet, H. 2003. Iterative spatial join. *ACM Transactions on Database Systems 28,* 3 (Sept.), 268–294.

Jacox, E. and Samet, H. 2007. Spatial join techniques. *ACM Transactions on Database Systems 32,* 1 (Mar.). To appear and also an expanded version in University of Maryland Computer Science Technical Report TR–4730, June 2005.

Jagadish, H. V. 1990. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM SIGMOD Conference*. Atlantic City, NJ, 332–342.

Ja'Ja', J. 2000. A perspective on quicksort. *Computing in Science and Engineering 2,* 1 (Jan.), 43–49.

Kahveci, T., Lang, C., and Singh, A. K. 2003. Joining massive high-dimensional datasets. In *Proceedings of the 19th IEEE International Conference on Data Engineering*. Bangalore, India, 264–276.

Kalashnikov, D. and Prabhakar, S. 2003. Similarity joins for low- and high- dimensional data. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA'03)*. Kyoto, Japan, 7–16.

Kedem, G. 1982. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*. Las Vegas, NV, 352–357. Also University of Rochester Computer Science Technical Report TR–91, September 1981.

Kitsuregawa, M., Harada, L., and Takagi, M. 1989. Join strategies on KD-tree indexed relations. In *Proceedings of the 5th IEEE International Conference on Data Engineering*. Los Angeles, 85–93.

Koperski, K. and Han, J. 1995. Discovery of spatial association rules in geographic information systems. In *Advances in Spatial Databases—4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, Eds. vol. 951 of Springer-Verlag Lecture Notes in Computer Science. Portland, ME, 47–66.

Korn, F., Sidiropoulos, N., Faloutsos, C., Siegel, E., and Protopapas, Z. 1996. Fast nearest neighbor search in medical image databases. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds. Mumbai (Bombay), India, 215–226.

Koudas, N. and Sevcik, K. C. 1997. Size separation spatial join. In *Proceedings of the ACM SIGMOD Conference*, J. Peckham, Ed. Tucson, AZ, 324–335.

Koudas, N. and Sevcik, K. C. 1998. High dimensional similarity joins: algorithms and performance evaluation. In *Proceedings of the 14th IEEE International Conference on Data Engineering*. Orlando, FL, 466–475.

Koudas, N. and Sevcik, K. C. 2000. High dimensional similarity joins: algorithms and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering 12,* 1 (January/February), 3–18.

Levenshtein, V. A. 1966. Binary codes capable of correcting deletions, insertion, and reversals. *Cybernetics and Control Theory 10,* 8, 707–710.

Linial, N., London, E., and Rabinovich, Y. 1995. The geometry of graphs and some of its algorithmic applications. *Combinatorica 15*, 215–245. Also see *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science*, pages 577–591, Santa Fe, NM, November 1994.

Lo, M.-L. and Ravishankar, C. V. 1995. Generating seeded trees from data sets. In *Advances in Spatial Databases—4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, Eds. vol. 951 of Springer-Verlag Lecture Notes in Computer Science. Portland, ME, 328–347.

Merrett, T. H., Kambayashi, Y., and Yasuura, H. 1981. Scheduling of page-fetches in join operations. In *Very Large Data Bases, 7th International Conference*. Cannes, France, 488–498.

Neyer, G. and Widmayer, P. 1997. Singularities make spatial join scheduling hard. In *Algorithms and Computation, 8th International Symposium, ISAAC*. Singapore, 293–302.

Nievergelt, J., Hinterberger, H., and Sevcik, K. C. 1984. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems 9,* 1 (Mar.), 38–71.

Orenstein, J. A. 1986. Spatial query processing in an object-oriented database system. In *Proceedings of the ACM SIGMOD Conference*. Washington, DC, 326–336.

Orenstein, J. A. 1989. Strategies for optimizing the use of redundancy in spatial databases. In *Design and Implementation of Large Spatial Databases—1st Symposium, SSD'89*, A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, Eds. vol. 409 of Springer-Verlag Lecture Notes in Computer Science. Santa Barbara, CA, 115–134.

Papadias, D. and Arkoumanis, D. 2002. Approximate processing of multiway spatial joins in very large databases. In *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology*, C. S. Jensen, K. G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, and M. Jarke, Eds. Lecture Notes in Computer Science, vol. 2287. Prague, Czech Republic, 179–196.

Patel, J. M. and DeWitt, D. J. 1996. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD Conference*. Montréal, Canada, 259–270.

Preparata, F. P. and Shamos, M. I. 1985. *Computational Geometry: An Introduction*. Springer-Verlag, New York.

Robinson, J. T. 1981. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*. Ann Arbor, MI, 10–18.

Rucklidge, W. J. 1995. Locating objects using the Hausdorff distance. In *Proceedings of the Fifth International Conference on Computer Vision (ICCV'95)*. Boston, MA, 457–464.

Rucklidge, W. J. 1996. *Efficient Visual Recognition Using the Hausdorff Distance*. Springer-Verlag, Secaucus, NJ.

Samet, H. 1990a. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA.

Samet, H. 1990b. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA.

Samet, H. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco.

Shim, K., Srikant, R., and Agrawal, R. 1997. High-dimensional similarity joins. In *Proceedings of the 13th IEEE International Conference on Data Engineering*. Birmingham U.K., 301–311.

Shim, K., Srikant, R., and Agrawal, R. 2002. High-dimensional similarity joins. *IEEE Transactions on Knowledge and Data Engineering 14,* 1 (January/February), 156–171.

Tasan, M. and Özsoyoglu, Z. M. 2004. Improvements in distance-based indexing. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, M. Hatzopoulos and Y. Manolopoulos, Eds. Santorini, Greece, 161–170.

Uhlmann, J. K. 1991. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett. 40,* 4 (Nov.), 175–179.

Voronoi, G. 1909. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxiême mémoire: Recherches sur les parallèlloèdres primitifs. Seconde partie. *Journal für die Reine und Angewandte Mathematik 136,* 2, 67–181.

Wang, J. T., Wang, X., Lin, K.-I., Shasha, D., Shapiro, B. A., and Zhang, K. 1999. Evaluating a class of distance-mapping algorithms for data mining and clustering. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. San Diego, CA, 307–311.

Xia, C., Lu, J., Ooi, B. C., and Hu, J. 2004. Gorder: an efficient method for KNN join processing. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakely, and K. B. Schiefer, Eds. Toronto, Canada, 756–767.

Yianilos, P. N. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*. Austin, TX, 311–321.

Zezula, P., Amato, G., Dohnal, V., and Batko, M. 2006. *Similarity Search: The Metric Space Approach*, Advances in Database Systems, Vol. 32 ed. Springer, New York, NY.

Zhang, J., Mamoulis, N., Papadias, D., and Tao, Y. 2004. All-nearest-neighbors queries in spatial databases. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, M. Hatzopoulos and Y. Manolopoulos, Eds. Santorini, Greece, 297–306.