Partitioning Strategies for Spatio-Textual Similarity Join

Jinfeng Rao[1,2], Jimmy Lin[1,2,3], and Hanan Samet[1,2]

[1] Department of Computer Science
[2] Institute for Advanced Computer Studies
[3] The iSchool – College of Information Studies
University of Maryland, College Park

jinfeng@cs.umd.edu, jimmylin@umd.edu, hjs@umiacs.umd.edu

ABSTRACT

Given a collection of geo-tagged objects with associated textual descriptors, the spatio-textual similarity join (STJoin) problem is to identify all pairs of similar objects that are close in distance. This task, which is useful in localized recommendations and other applications, is challenging since computing the join is super-linear with respect to the size of the collection. In this paper, we explore partitioning strategies for tackling STJoin. One approach is to start with a spatial data structure, traverse regions and apply a previous algorithm for identifying similar pairs of textual documents called All-Pairs. An alternative approach is to construct a global index but partition postings spatially and modify the All-Pairs algorithm to prune candidates based on distance. We evaluate these approaches on two real-world datasets and find that when running in a single thread, both approaches are comparable in terms of performance. However, a multi-threaded implementation of the global index approach is able to achieve far better speedup given its ability to parallelize at a finer granularity to avoid skewed distributions in task sizes. In addition to using All-Pairs as the underlying textual similarity join algorithm, we also explored an alternate algorithm known as PPJ: our findings are consistent, which suggests that load balancing is a fundamental issue affecting parallel implementations of STJoin algorithms.

Categories and Subject Descriptors: H.3 [Information Storage and Retrieval]: Content Analysis and Indexing

General Terms: Algorithms, Design, Performance

Keywords: Spatio-textual similarity join, geotagged data, indexing

1. INTRODUCTION

Our work is situated at the intersection of geo-spatial processing, data mining, and information retrieval: In spatio-textual similarity join (STJoin for short), we are given a collection of objects with both texts and geo coordinates and wish to efficiently identify all pairs of similar objects that are physically close.

There are many real-world applications of STJoin: One concrete example is localized recommendation. For example, Twitter may wish to suggest users who are both close by and share similar interests (as determined by tweet content). An online dating service might want to match people based on profile similarity and location constraints. Another sample application is near-duplicate detection, where we would like to group together multiple news stories that report on the same event based on location.

In this paper, we explore partitioning strategies for the STJoin problem. One approach is to start with a spatial data structure (in our case, either grids or quadtrees), traverse regions and apply a previous algorithm for identifying similar pairs of textual documents called All-Pairs [3]. We call this the *local* approach. An alternative approach is to construct a global index but partition postings spatially (either by grid or by quadtree, linearized by z-ordering) and modify the All-Pairs algorithm to prune candidates based on distance. We call this the *global* approach. Together, this yields four combinations: local grid, local quadtree, global grid, and global quadtree.

From experiments on two real-world datasets, we find that single-threaded implementations of these approaches are roughly comparable in performance. However, multi-threaded implementations of the global index approach achieve far better speedup due to their ability to parallelize work at a finer granularitity to avoid skewed distributions in task sizes. Finally, we varied the underlying similarity join algorithm, considering the PPJ algorithm [34] in addition to All-Pairs. Our findings about single- vs. multi-threaded performance hold, suggesting that load balance is a fundamental issue affecting the parallelization of STJoin algorithms.

Our work makes two contributions. First, we present a methodical analysis of the STJoin problem in multiple dimensions: local vs. global partitioning, grid vs. quadtree, All-Pairs vs. PPJ, and single- vs. multi-threaded. Although some combinations have been previously explored, our experiments evaluate the performance of STJoin under different variants and parameter settings. Second, we believe that our observation about the performance of multi-threaded STJoin algorithms is novel. Our results suggest that load balancing is a fundamental issue regardless of algorithm, and highlight the importance of evaluating parallel performance in today's multi-core computing environments.

2. RELATED WORK

Spatial Distance Join. As a component of the STJoin

http://dx.doi.org/10.1145/2676536.2676542

problem, spatial distance join and semi-join have been well studied in the literatures [33, 6, 15, 17, 19, 20, 25, 27]. In ϵ-distance join, given two geo dataset R and S, we wish to find all object pairs with $\text{dist}(r, s) < \epsilon$ where $r \in R$ and $s \in S$. The dominant approach is based on hierarchical decomposition using R-trees [6] or multi-level grid files [33]. These hierarchical methods are based on traversals of the spatial data structures, typically involving two stages: (1) the filter stage, which uses objects' minimum bounding rectangle approximations to find pairs of potentially intersected objects, and (2) the refinement stage, which checks whether the candidates satisfy the given distance threshold.

Set Similarity Join. In Set Similarity Join (SSJoin), the task is to find all pairs of objects from a potentially large collection whose similarity (e.g., Jaccard, cosine, etc.) is above specified threshold; of course, these objects can represent "bag of words". This problem has attracted the attention of researchers due to applications in data mining. Sarawagi and Kirpal [29] presented an early solution to this problem based on inverted indexing: for any object s, the corresponding inverted index lists of s are traversed to accumulate similarity with all other objects. The Prefix-Filter algorithm [1] is an additional optimization based on a subset of elements with the smallest frequencies, where the size of subset depends on the similarity function and threshold. Our work builds on the All-Pairs algorithm [3] for SSJoin, which is also an index-based approach. The algorithm takes advantage of the threshold property to prune the number of postings considered and is further optimized by sorting terms based on document frequencies. We provide the complete All-Pairs algorithm in Algorithm 1 for reference, but in this paper All-Pairs is used mostly as a "black box" in conjunction with spatial data structures.

The All-Pairs algorithm has two phases: building inverted indexes and finding similar objects. In the first phase, the algorithm builds postings lists for each term to store the list of objects containing that term. For an object x, it has a weight vector over terms that sum to one ($\sum_i x[i] = 1$), e.g, $x[i]$ denotes the weight of term t_i, $\sum_i \text{maxweight}_i(R) \cdot x[i]$ is the maximum similarity it could have with other objects, where $\text{maxweight}_i(R)$ is the maximum weight of term t_i for all objects in a collection R. If this similarity value is smaller than the given similarity threshold, then object x can not be considered as similar with any other object. For an object x, the All-Pairs algorithm utilizes this property to only index terms $\{t_j\}$ which satisfy following property:

$$\sum_{i<j} \text{maxweight}_i(R) \cdot x[i] \geq s$$

This idea corresponds to Lines 13-19 in the Algorithm 1. Line 12 calls the FindSimPairs function to find similar object pairs based on existing inverted indexes. In the second phase (FindSimPairs), the algorithm traverses the postings lists to accumulate similarity scores between objects (Line 25-30). The object pairs with non-zero similarities are further verified in Line 31-35 by computing the actual similarity score between objects (Line 32).

Spatial Keyword Queries. The spatial keyword query task is to find relevant POIs (points of interest) by considering distance and textual relevance to a given query q. An example would be "*find the nearest restaurant serving Chinese food*". Here "*Chinese food*" can be a textual descriptor of objects. Many researchers have studied this problem [9, 14, 8, 16, 21, 11]. A general framework to answer spatial keyword queries is to first build indexes on the textual and location attributes, then take into account both textual relevance and location proximity to prune the search space.

Algorithm 1 All-Pairs Similarity Join on cosine distance

```
1: Input: threshold s, collection R with n objects and m terms
2: Output: similar pair set S
3: function BuildIndex(ObjectSet R)
4:     Sort R in decreasing order of maxweight(x)
5:     Sort terms in decreasing order of num. of non-zero entries
6:     Denote the max. of x[i] for all x ∈ R as maxweight_i(R)
7:     Denote the max. of x[i] for i = 1...m as maxweight(x)
8:     S ← ∅                                          ▷ similar pair set
9:     I_1, I_2, ..., I_m ← ∅
10:    for each object x ∈ R do
11:        sim ← 0
12:        S ← S ∪ FindSimPairs(x, I_1, ..., I_m, s)
13:        for each i s.t. x[i] > 0 in increasing order of i do
14:            sim ← sim+maxweight_i(R) · x[i]
15:            if sim > s then
16:                I_i ← I_i ∪ {(x, x[i])}              ▷ indexed portion
17:                x[i] ← 0
18:            else
19:                I'_i ← I'_i ∪ {(x, x[i])}            ▷ unindexed portion
20:    return S
21:
22: function FindSimPairs(x, I_1, ..., I_m, s)
23:    Sim ← empty map from object to weight
24:    P ← ∅
25:    remscore = Σ_i x[i] · maxweight_i(R)
26:    for each i in reverse order s.t. x[i] > 0 do
27:        for each (y, y[i]) ∈ I_i do
28:            if Sim[y] ≠ 0 or remscore ≥ t then
29:                Sim[y] ← Sim[y] + x[i] · y[i]
30:        remscore ← remscore − x[i] · maxweight_i(R)
31:    for each y with non-zero weight in Sim do
32:        sim ← Sim[y] + dot(x, y')
33:              ▷ add similarity of index and unindexed portion
34:        if sim ≥ s then
35:            P ← P ∪ {(x, y, sim)}
36:    return P
```

Felipe et al. [11] proposed a hybrid indexing approach to answer the *k-nearest-neighbor* query. It associates each R-tree node with a signature file to indicate keywords contained in the subtree rooted at that node. Cong et al. [9] and Li et al. [21] proposed another hybrid index structure, IR-Tree, which uses R-trees as the underlying spatial index and attaches an inverted file to each R-tree node. An inverted file contains two types of information: (1) the vocabulary of the whole collection, (2) a set of postings lists for objects located in the subtree. By extending the nearest neighbor algorithm of R-trees [18], the IR-tree can access the minimum number of R-tree nodes to find the best match objects with respect to both distance to the query location and textual similarity to the query keywords.

Another related spatial query problem is region-based keyword queries. Given a query consisting of a location and a set of keywords, find the relevant POIs satisfying the distance and textual similarity thresholds. Chen et al. [8] studied the impact of different indexing orders in hybrid index construction, e.g, first R-tree then inverted file or first inverted file then R-tree. Instead of pruning the search space by location and text separately, Hariharan et al. [16] proposed an index approach exploiting the power of pruning location and text simultaneously. As a variant of region-based keyword queries, Fan et al. [14] studied another prob-

lem: given a set of ROIs (Region of Interests) and a query ROI, find similar ROIs by considering spatial overlap and textual similarity. The problem setting is changed from points (POIs) to regions (ROIs). The authors proposed a signature-based filtering algorithm to generate signatures for both texts and regions. These signatures are used in a filter-and-verification framework to generate similar candidates and finally identify answers.

Most spatial queries examine the individual behavior of whether an individual object satisfies a query or not. Cao et al. [7] extended this work to investigate group behavior: finding groups of objects where objects in a group collectively satisfy a query. The authors defined two types of group behaviors: (1) the group's keywords cover the query's keywords and the sum of the objects' spatial distances to the query location is minimized; (2) the group's keywords cover the query's keywords and objects are nearest to the query location and also have the lowest inter-object distances. Cao et al. [7] devised exact algorithms utilizing dynamic programming and greedy algorithms to solve these two problems.

2.1 Spatio-Textual Similarity Join

STJoin is a combination of SSJoin and ϵ-distance join [2]. This problem has not been fully studied yet, although there is some previous work [2, 5, 22, 23]. To the best of our knowledge, Ballesteros et al. [2] is the first work to study the STJoin problem. They built a MapReduce platform for identifying objects similar in texts and close in spatial locations. The similarity function they used is different from ours: $\text{sim}(x,y) = \frac{\text{sim}_t(x.text, y.text)}{1+\text{dist}(x.loc, y.loc)}$, where $\text{sim}_t(x.text, y.text)$ is the textual similarity of objects x and y and $\text{dist}(x.loc, y.loc)$ is the distance between x and y. Another difference is that for an object x it identifies object y that maximizes $\text{sim}(x,y)$ as the best match object, while our work aims to find all objects y with $\text{sim}(x,y) \geq s$ and $\text{dist}(x,y) \leq t$.

Liu et al. [22, 23] presented a hybrid signature method (both textual signatures and spatial signatures) to prune dissimilar object pairs. The spatial objects they studied are regions (polygons) while we view each spatial object as a point in our work. They adopted the *probability constrained region* technique. [30] to generate spatial signatures for each object. The spatial signatures are combined with textual signatures to generate candidates and finally to identify the results. Our work is most similar to Bouros et al. [5], who presented an approach based on dynamic grid partitioning and the PPJ algorithm for SSJoin [34]. Our work compares All-Pairs and PPJ and further explore different partitioning strategies. Our work also considers multi-threaded performance whereas previous work does not.

3. SPATIO-TEXTUAL SIMILARITY JOIN

We begin with a formal problem definition: given a collection of objects R, textual similarity threshold s, and distance threshold t, spatio-textual similarity join aims to find all object pairs (u,v) with $\text{sim}(u,v) > s$ and $\text{dist}(u,v) < t$ where $(u,v) \in R \times R$. Each object $d \in R$ is associated with a textual descriptor and location information. In our case, the text is represented by a document vector using *tf-idf* weighting, and the location is represented by longitude and latitude. For $\text{sim}(u,v)$ we use cosine similarity and for $\text{dist}(u,v)$ we use Euclidean distance, converted into kilometers for easy comprehension.

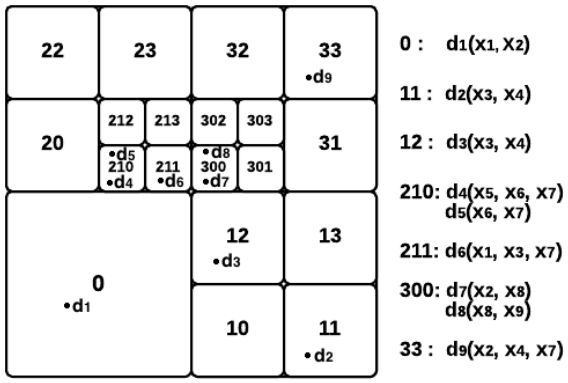


Figure 1: Simple quadtree example.

Consider the example shown in Figure 1. We have a collection of objects $R = \{d_1, d_2, ..., d_9\}$, whose locations are shown in the quadtree; the associated texts are shown on the right. We can think the object as a geo-tagged tweet or document. For example, the object d_4 is located in quadtree node 210 and it contains three terms(words) x_5, x_6 and x_7. Let the unit size of the quadtree be 1km, e.g., the size of node 210 is 1km. Given textual similarity threshold $s = 0.6$ (for simplicity, let us assume Jaccard similarity, i.e., Jaccard(A,B) = $\frac{A \cap B}{A \cup B}$, and distance threshold $t = 1$km, then objects (d_4, d_5) would be considered a similar pair with $\text{sim}(d_4, d_5) = \frac{|x_6, x_7|}{|x_5, x_6, x_7|} = 2/3$ and $\text{dist}(d_4, d_5) < 1$km. Although object pairs (d_7, d_8) satisfy the distance threshold, they fail to reach the similarity threshold and thus would not be identified by the algorithm.

This diagram nicely illustrates the intuition behind our first approach: the general strategy is to build a spatial index over the data and then run a SSJoin algorithm (in our case, All-Pairs) over regions with potential candidates—we refer to this as the *local* approach. The alternative approach is to build a global inverted index on the texts, but to partition each postings list spatially: this can be accomplished by linearizing each postings list via z-ordering or performing a grid partitioning of each postings list. This technique allows us to adapt the All-Pairs algorithm to efficiently prune pairs based on distance. We call this the *global* approach. Although these local and global approaches have been tried with grid partitions previously [5], we show how to extend to quadtrees as we believe that quadtrees represent a better partitioning strategy since geo data often have skewed distributions. Furthermore, we present a methodical analysis of the STJoin problem in multiple dimensions: local vs. global partitioning, grid vs. quadtree, All-Pairs vs. PPJ, and single- vs. multi-threaded. Below, we describe both approaches in more detail, using All-Pairs as the underlying similarity join algorithm; later, we replace All-Pairs with PPJ.

3.1 The Local Index Approach

Our algorithm for integrating quadtrees with All-Pairs is as follows: First, we build a PR-quadtree over the dataset. Based on common practice, we use the spatial range of (−180, −90, 180, 90). Given distance threshold t, we recursively decompose each node into four child nodes until the node contains less than b objects (by default, one) or the size of the node is about to less than the threshold t. Each node maintains a list of object ids. With this partitioning approach, when searching for similar objects for a

Algorithm 2 Quadtree STJoin

```
1: Input: similarity threshold s, distance threshold t, set R
2: Output: similar pair set S
3: function QuadtreeJoin
4:     T ← BuildQuadtree(R, t)
5:     for each nonempty node n ∈ T.leafnodes do
6:         Find neighbor nodes in {W, NW, NE, N} directions
7:         AllPairs-BuildIndex(node.objects ∪ neighbors.objects)
8:
9: function AllPairs-BuildIndex(allobjects)
10:    for each x ∈ allobjects do
11:        if x ∈ node.objects then
12:            S ← S ∪ AllPairs-FindSimPairs(x, I_1, ..., I_m, s, t)
13:        Index neighbor.objects first, then node.objects
```

target object x, only objects in the same or neighbor nodes of the target object need to be checked. Consider the example shown in Figure 1. Assume object x is contained in the node 210. To find all candidate pairs, only objects in the locating and neighbor nodes $\{20, 210, 212, 213, 211, 0\}$ need to be checked. As an additional optimization, since distance is symmetrical, once we check (x, y), it is unnecessary to check (y, x). Thus, we only need to consider neighbor nodes in $\{W, NW, NE, N\}$ directions: for node 210, these are nodes $\{20, 212, 213, 210\}$.

The pseudo code of our STJoin algorithm is shown in Algorithm 2. We iterate over the quadtree leaf nodes (Line 5); for each non-empty node, we apply the All-Pairs algorithm over the node and its appropriate neighbors (which entails building the inverted index and traversing the postings). When verifying similar objects for target object x, both similarity and distance thresholds are used to prune objects. Note that with a minor modification, it is possible to recompute STJoin with different distance and similarity thresholds without rebuilding the entire quadtree. We can over-partition the quadtree (i.e., build it to a very fine granularity), and then back off to coarser nodes for larger distance thresholds.

3.2 The Global Index Approach

In the previous approach, we first spatially partition, and then build inverted indexes on individual regions. Alternatively, we can build a global inverted index, and then partition each postings list spatially. The All-Pairs algorithm can then take advantage of the spatial information when traversing postings. This is accomplished by sorting each postings list by its z-order, which is a standard way to linearize spatial data structures [26]; in fact this is equivalent to partitioning the postings lists since there is a direct relationship between z-ordering and quadtrees [26]. The postings lists of the objects in the quadtree example are shown in Figure 2. Objects are sorted by its z-order in each postings list. Thus, when we are applying the All-Pairs algorithm, instead of iterating over the entire postings list, we only need to consider the objects within certain range of z-orders. For example, in Figure 1, when processing object d_6 with z-order 211, only objects located in z-orders 210, 211, 212, 213 and 302 will be considered as similar candidates. To make finding these z-orders efficient, we store an auxiliary data structure to keep track of the beginning and end locations of each z-order ($id.startPos$, $id.endPos$). As an additional optimization, we can avoid checking the pair (y, x) if we've already checked (x, y). When processing object x with z-order i, instead of checking all objects with z-order in the range ($i.startPos$, $i.endPos$), only objects in the range ($x.loc + 1$, $i.endPos$) need to be considered. Note that the global index approach can be extended to grids as grid cells can also be linearized by z-order; the only difference is that we access a grid node by a 2-dimensional array, while for quadtrees, we need to visit a z-order map to access a quadtree node.

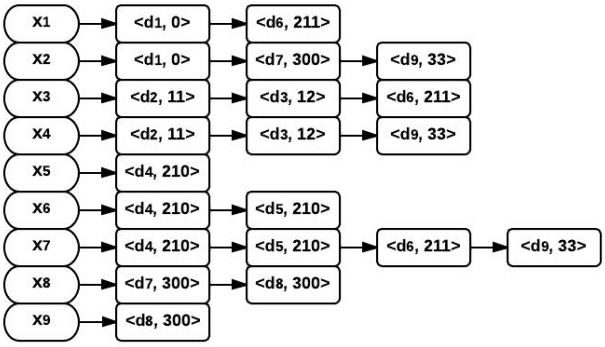


Figure 2: Postings lists of global index approach

It's also worth noting that some set join approaches require objects to be sorted in ascending order of size, conflicting with the global index requirement that objects to be sorted by z-order. In such approaches, when finding similar objects of x, only objects with smaller sizes than x will be checked. In this case, we would miss some similar pairs (x, y) if object y is located in x's neighbor nodes $\{W, NW, NE, N\}$ but has a larger size than x. Note that pair (y, x) will not be checked since object x is *not* located in y's neighbor nodes in the $\{W, NW, NE, N\}$ directions. Therefore, instead of only checking neighbors in the $\{W, NW, NE, N\}$ directions, we examine all the neighbor nodes if the technique requires sorting objects in order of size. However, this modification increases filtering overhead as more neighbor nodes need to be visited.

We provide the pseudo code in Algorithm 3. First, a quadtree or grid is constructed (Line 4) and objects are sorted by z-order (Line 5). Then we call the BuildGlobalIndex function (Line 6), iterating over the sorted objects and constructing postings list for each term (Lines 14-21). Note that when building postings lists for object x, only objects in the same or neighbor nodes are considered for computing $\text{maxweight}_i(r)$ (Lines 15-16). In Lines 22-24, for each postings list, we record the begin and end position of each node and the exact position of each object for subsequent access. In the FindSimPairs function, Line 28 accesses the corresponding postings list I_i for each term i in the target object x. Instead of visiting the entire postings lists, we only need to consider the parts belonging to target object x's location or neighbor nodes (Line 29). For the neighbor nodes, we consider the part of the postings list with position in the range of $[n.startPos, n.endPos]$ (Lines 31-32), while for the locating node, we consider the part with position in the range of $[x.pos + 1, n.endPos]$ (Lines 34-35), where $n.startPos, n.endPos$ denote the start and end position of node n in postings list I_i respectively, $x.pos$ denotes the exact position of object x in postings list I_i. In Lines 39-40, we generate and verify the candidates by incorporating the similarity and distance filters in Algorithm 1.

In the local index approach, inverted indexes are dynamically created with given similarity thresholds. This wastes computations in reconstructing inverted indexes when thresh-

Algorithm 3 Global Index STJoin

```
1: Input: similarity threshold s, distance threshold t, set R
2: Output: similar pair set S
3: function GlobalIndexJoin(R, s, t)
4:     T ← BuildQuadtree(R, t) or BuildGrid(R, t)
5:     Sort objects in ascending order of its z-order
6:     I_1, ..., I_m ← BuildGlobalIndex(R', s)
7:                                  ▷ R' denotes sorted object list
8:     for each x ∈ R' do
9:         S ← S∪ FindSimPairs(x, I_1, ..., I_m, s, t)
10:    return S
11:
12: function BuildGlobalIndex(R', s)
13:    Initializations; // Lines 5-9 in Algo. 1
14:    for each x ∈ R' do
15:        r ← set of objects in x.node or x.neighbors
16:        Denote the max. of x[i] for all x ∈ r as maxweight_i(r)
17:        for each term i s.t. x[i] > 0 in increasing order of i do
18:            sim ← sim+maxweight_i(r) · x[i]
19:            if sim > s then
20:                Construct postings list I_i
21:                                  ▷ // Lines 16-19 in Algo. 1
22:    for each postings list I_i do
23:        Record the [start, end] position of each node in the list
24:        Record the exact position of each object in the list
25:
26: function FindSimPairs(x, I_1, ..., I_m, s, t)
27:    Preparations; // Lines 23-25 in Algo. 1
28:    for each i in reverse order s.t. x[i] > 0 do
29:        for each node n ∈ x.node ∪ x.neighbors do
30:            if n ∈ x.neighbors then
31:                startPos = n.startPos
32:                endPos = n.endPos
33:            else                          ▷ x located in node n
34:                startPos = I_i.getPos(x) + 1
35:                endPos = n.endPos
36:            for each y ∈ I_i[startPos, endPos] do
37:                if x.id == y.id then
38:                    Continue;
39:                Generate Candidates; // Line 28-30 in Algo. 1
40:    Verify Candidates // Line 31-35 in Algo. 1
```

old parameters change. A simple modification to our global index algorithms allows us to compute STJoin using different thresholds without reindexing the collection. The basic idea is the same: we over-partition the inverted indexes at a very fine granularity and then choose the right level of granularity at which to apply the filtering: it is very straightforward to merge z-order ranges, which is the equivalent of ascending the quadtree or grid hierarchy.

The global index approach and the local index approach manifest different tradeoffs. With the global index approach, each object is indexed only once, but with the local index approach, each object might participate in All-Pairs search with several of its neighbors (and thus be indexed multiple times). On the other hand, the index structures in the local index approach are much more compact, which gives rise to better cache locality when traversing postings. How these two factors balance out is an empirical question we explore.

3.3 Parallelization

Most previous work on STJoin focuses on sequential algorithms. However, today's servers have multiple processors, each with multiple cores. To take advantage of modern hardware, we experimented with multi-threaded implementations of our approaches.

In the parallel global index approach, we take advantage of a thread pool. The parallelization is straightforward: we divide the work into subtasks, and each subtask is responsible for processing predefined number of objects β (by default 100). The subtasks are queued and executed by the thread pool. For the local quadtree or grid partition approaches, since building inverted indexes for each node is blocking, it is more natural to parallelize at the granularity of nodes. Thus, we divide the work into subtasks, and each subtask is responsible for processing a pre-defined number of nodes α (by default 10). The subtasks are queued and executed by the thread pool. We see that the global index approach lends itself to finer-grained parallelism, which is consequential for skewed distributions in geo data, as we see later.

4. EXPERIMENTS

4.1 Setup

Our experiments used two real-world geo-tagged datasets: The NewsStand dataset is a collection of news articles from the NewsStand system [28, 31], where each document is associated with its dominant geo-tagged location (longitude and latitude). This dataset contains 100k documents with 221k distinct words in total. The lengths of documents range from 2 to 7991, with an average length of 218. The World Cities dataset[1] [32] consists of 2.23m geo-tagged images from 40 cities, crawled from Flickr using geographic queries covering a window centered around each city center. For each image, we extracted values from the 'title', 'description', and 'tags' elements, which were concatenated as the textual descriptor; We used the latitude and longitude value split from the 'location' element to represent each image's real-world location. This dataset contains 420k distinct terms. The average size of the images' textual descriptors is 17. For both datasets stopwords are removed.

Our algorithms were implemented in Java, running on JDK7. Each experiment was repeated three times and we report average performance. Experiments were conducted on a server with dual Intel Xeon 8-core processors (E5620 2.4 GHz) and 128 GB RAM. With hyper-threading, the server supports up to 32 concurrent threads.

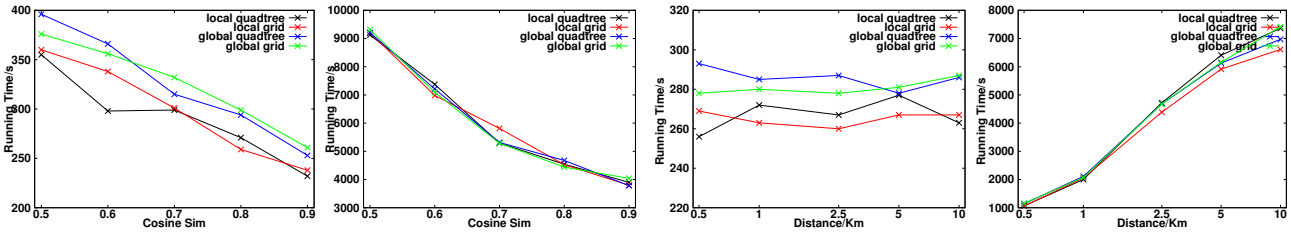
4.2 Local vs. Global Approaches

We begin with single-threaded experiments comparing the local and global index approaches described in the previous section. For the global index approaches, we first build spatial indexes (quadtree or grid), which is equivalent to constructing the z-order, and then use the z-order to sort the postings. To ensure a fair comparison, grid partitions include all the same optimizations as our quadtree partition approach. We make the size of a grid node the same as that of quadtree leaf node.

In total, we have four partitioning strategies: local quadtree, local grid, global quadtree, and global grid. In this experiment, we used All-Pairs [3] for joining objects based on text similarity, using cosine distance. In all cases we built the spatial data structures at a distance granularity of 0.5km; for larger distance thresholds we take advantage of the coarsening techniques discussed in the previous section (e.g., for quadtrees, ascending quadtree nodes to the correct spatial granularity).

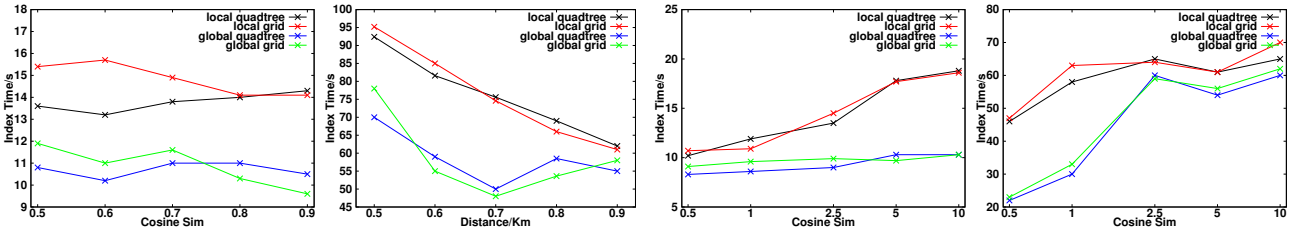
Results are shown in Figure 3 for both collections. In Figure 3a and 3b, we fixed the distance threshold to 2.5km,

[1] `http://image.ntua.gr/iva/datasets/wc/`



(a) NewsStand, fix t=2.5km (b) World Cities, fix t=2.5km (c) NewsStand, fix s=0.8 (d) World Cities, fix s=0.8

Figure 3: Performance of different partitioning strategies running in a single thread.



(a) NewsStand, fix t=2.5km (b) World Cities, fix t=2.5km (c) NewsStand, fix s=0.8 (d) World Cities, fix s=0.8

Figure 4: Indexing Time

and varied the similarity threshold in the range of [0.5, 0.9], while in Figure 3c and 3d, we fixed the similarity to 0.8 and varied the distance threshold from 0.5km to 10km. The y axis shows the total running time of each partitioning strategy. The running time consists of three parts: spatial partitioning, inverted indexing, and joining. Since we found that the spatial partitioning time is small compared to the total running time (usually ∼1 second for the NewsStand data and 10-20 seconds for the World Cities data), we don't show this part in our results. The corresponding inverted indexing time for each experiment in Figure 3 is shown in Figure 4. We can see that in all cases, most of the time is taken up by accumulating scores during postings traversal (joining). As expected, inverted indexing time for local quadtree and grid partitions is longer, since objects may be indexed multiple times. Since the World Cities dataset is much larger, we would expect the running time to be longer. For $s = 0.8, t = 2.5$km, we identified 79m similar pairs; the same setting on the NewsStand dataset produced 982k similar pairs.

Comparing the performance of the four partitioning strategies, we see that local partitions are generally faster than global partitions, although all these are still roughly comparable for different threshold settings. A somewhat surprising finding is that local grid partition doesn't suffer from poor performance for smaller distance thresholds. As we previously expected, a smaller distance threshold leads to an exponentially increasing number of grid cells, thus resulting in the inefficiency of local grid partition. This problem is solved by only generating non-empty grid cells instead of storing all cells. In this case, grid has almost the same shape and configuration as quadtree, thus is expected to obtain similar performance. Overall, single-threaded performance between all four partitioning strategies is comparable.

Finally, we note that in the global partitions, index construction time is shorter since we avoid (potentially) indexing an object multiple times. However, during the All-Pairs computation, we incur additional overhead since we need to consult the auxiliary data structures to look up the z-order ranges. These two factors roughly balance out, making the global partitions roughly as fast as local partitions. This is a somewhat surprising finding, given how different the partition-based and global index approaches are.

Note that in Figure 3, we fixed one threshold and varied the other. From these experiments, we noticed two trends of how parameter values influence partition efficiency: (1) higher similarity thresholds leads to less processing time and (2) partitioning strategies are faster for smaller distance thresholds. The first trend matches our intuition that increasing similarity will reduce the number of candidates considered as similar pairs and also improve the filtering process, thereby decreasing the total running time. The second trend reflects the observation that as nodes become smaller, the average number of objects in the nodes decreases. As we found that the inverted indexing and joining time is superlinear to the number of objects in a node and its neighbors, decreasing the average number of objects in each node will result in reducing the total running time.

4.3 Multi-Threaded Experiments

Next, we examined the multi-threaded performance of our techniques. Here, we set the thread pool size to 32, reflecting the number of hyper-threads supported by our machine. By default, for local partitions (quadtree/grid), we set the task size to the granularity of 10 leaf nodes, and for global partitions, each task consists of 100 objects. Table 1 and 2 show multi-threaded experimental results corresponding to the same conditions in Figure 3; each column represents a partitioning strategy, e.g, LQ stands for local quadtree partition and GG means global grid partition. We report both running times and speedups compared to the single-threaded implementations. Consistent with previous experiment settings, the running time is composed of three parts: spatial partitioning, inverted indexing, and joining. When computing speedup, we only take into account the parts which are processed by multiple threads. For local partitions, both the inverted indexing and joining parts are considered when computing multi-threaded speedups, while for global partitions, we only consider the joining time (the global inverted indexes are precomputed in a single-thread). For reference,

	LQ		LG		GQ		GG	
sim	time	speedup	time	speedup	time	speedup	time	speedup
0.5	221	**1.6**×	234	**1.5**×	35	**16.3**×	35	**16.3**×
0.6	211	**1.6**×	240	**1.4**×	31	**17.5**×	32	**18**×
0.7	190	**1.6**×	193	**1.6**×	30	**16.3**×	31	**16.7**×
0.8	173	**1.6**×	178	**1.5**×	28	**17.4**×	27	**17.9**×
0.9	152	**1.5**×	153	**1.6**×	27	**14.9**×	23	**16.5**×

(a) NewsStand

	LQ		LG		GQ		GG	
sim	time	speedup	time	speedup	time	speedup	time	speedup
0.5	3012	**3.1**×	2505	**3.7**×	1325	**7.4**×	1310	**7.3**×
0.6	2493	**3**×	1559	**4.5**×	882	**8.9**×	890	**8.9**×
0.7	1770	**3**×	1293	**4.5**×	750	**8.3**×	744	**8.4**×
0.8	1401	**3.2**×	1010	**4.5**×	579	**9.3**×	560	**9.2**×
0.9	1122	**3.5**×	939	**4**×	452	**9.7**×	456	**9.9**×

(b) World Cities

Table 1: Multi-threaded Experiments: fix t=2.5km, vary s

	LQ		LG		GQ		GG	
dist	time	speedup	time	speedup	time	speedup	time	speedup
10	176	**1.5**×	180	**1.5**×	30	**14.5**×	28	**16.6**×
5	172	**1.6**×	181	**1.5**×	28	**15.3**×	26	**16.6**×
2.5	171	**1.6**×	193	**1.3**×	27	**15.7**×	28	**15.7**×
1	180	**1.5**×	187	**1.4**×	26	**16.4**×	27	**15.6**×
0.5	191	**1.3**×	177	**1.5**×	27	**15.6**×	25	**17**×

(a) NewsStand

	LQ		LG		GQ		GG	
dist	time	speedup	time	speedup	time	speedup	time	speedup
10	2855	**2.5**×	2690	**2.5**×	987	**7.7**×	966	**8.1**×
5	2269	**2.8**×	1841	**3.2**×	861	**7.8**×	841	**7.9**×
2.5	1431	**3.3**×	1031	**4.4**×	578	**9.2**×	587	**9.1**×
1	464	**4.3**×	411	**5**×	260	**8**×	277	**8.2**×
0.5	258	**4.1**×	223	**4.8**×	144	**8.8**×	147	**8.4**×

(b) World Cities

Table 2: Multi-threaded Experiments: fix s=0.8, vary t

our machine has 16 physical cores (each core supports two virtual threads), and thus it is difficult to achieve linear speedup to 32 if the processors are fully occupied.

As we can see from Table 1 and 2, for all threshold settings, the multi-threaded global partitions achieve far better speedup than local partitions. This is because coarse-grained partitions (in local quadtree and grid partitions) lead to larger task sizes and unbalanced computation distribution. For the experiment on the NewsStand dataset, we discovered that a node contains 10k documents and generates 582k similar pairs, which consumed around 70% of the entire processing time. This intrinsically skewed geolocation distribution increases scheduling difficulties and results in idle cycles when waiting for a large task to finish. Therefore, in the context of skewed geo distributions, a fine-grained multi-threaded approach, enabled by global partitions, appears to be better for STJoin.

To further examine this skewed distribution, we visualized the distribution of task running times of the multi-threaded experiments in Figure 5. This visualization was on the NewsStand dataset with similarity threshold 0.8 and distance threshold 2.5km. Each task is viewed as a point in the graph. The x axis is the running time of a task, the y axis is the percentage of the number of tasks with corresponding running times.

These distributions nicely illustrate what makes multi-threaded speedup of global partitions and local partitions different. First, the distribution shows that a small number of tasks occupy a large portion of total running time, while most tasks consume little processing time. These small number of tasks determine the speedup of multi-threaded experiments. As we can see from Figure 5, for local partitions, the largest task consumes more than 100 seconds, while it only takes around 10 seconds for global partitions. Second, compared to global partitions, tasks for the local partitions have a much wider distribution over running times (from 10^{-3} to 10^{2}), which means local partitions have a more unbalanced computation distribution than global partitions. This unbalanced computation distribution is the reason for low speedups. We also notice that global grid has a larger portion of tasks (around 30%) with small running times than global quadtree. This is due to the smaller overhead of the global grid when consulting the auxiliary data structures to look up the z-order ranges. For the global grid, we can access a grid node by a two-dimensional array, while for the global quadtree we need to visit a map to access a quadtree node. Of course, the former is better for random access.

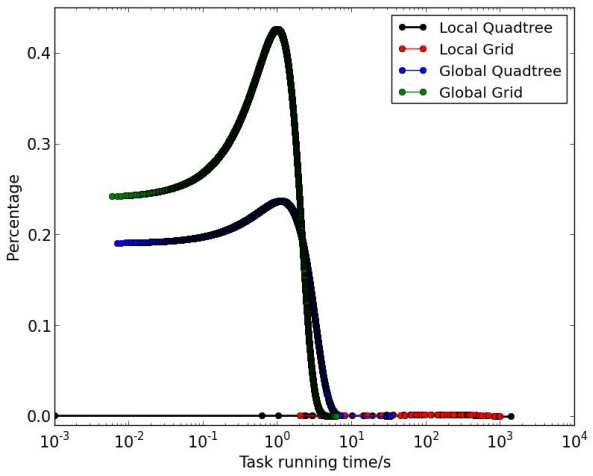
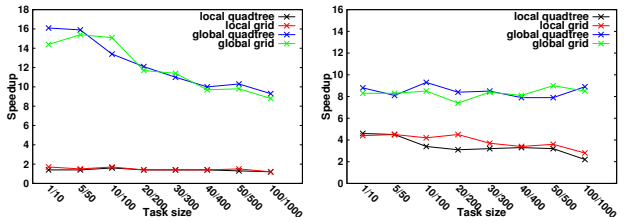


Figure 5: Distribution of tasks' running time

In Table 1 and 2, we also noticed different speedups for the NewsStand and the World Cities dataset. In the NewsStand dataset, the multi-threaded speedup of local partitions is around 1.4-1.7 times, while in the World Cities dataset, the speedup is 4-5 times. We believe this difference is caused by different spatial distributions of these two datasets. In the NewsStand data, there can be many duplicate documents located in a same place, but in the World Cities data, we crawled images and texts distributed in 40 cities in worldwide, and therefore this dataset is more uniformly distributed than the NewsStand and thus achieve a better speedup for local partitions.

4.4 The Finer-Grained the Better?

As we claimed that fine-grained partitioning is good for multi-threaded speedup, can we conclude that the finer the better? To explore this, we performed another set of multi-threaded experiments varying the task size. For this, we fixed the similarity threshold to 0.8 and distance threshold to 2.5km. For local partitions, we vary the task size from



(a) NewsStand (b) World Cities

Figure 6: Multi-threaded speedup varying task size

the granularity of 1 nodes to 100 nodes; for global partitions, the task size ranges from 10 objects to 1000 objects. Results are shown in Figure 6. Labels in x axes represent the task sizes (i.e., the first label '1/10' denotes the local partition task contains 1 node and global partition task contains 10 objects); the y axes show multi-threaded speedup.

In the NewsStand dataset, varying task size doesn't make much difference to local partitions, but the global partitions seem to be more sensitive to the task size, and achieve better speedups for smaller task sizes. However, this finding doesn't hold for the World Cities dataset. In Figure 6b, local partitions obtain the best speedup at the finest task size, and gradually suffer poorer speedups as the task size increases. But for global partitions, they achieve the best speedup at the task size of 100 objects. These inconsistent findings originate from the different sizes and distributions of the datasets. The World Cities data contains a total of 2.23m objects and each quadtree/grid leaf node contains around 3000 objects. Here, a finer-grained setting would be better for local partitions. However, for global partitions, a very fine-grained setting results in a large number of tasks (there are total of 223k tasks in the task size setting of 10 objects), thereby increasing context switch costs. Therefore, we conclude that for global partitions, a suitable task size setting should balance the benefits of a uniform computation distribution and context switch costs; while for local partitions, a finer-grained task setting can be better.

4.5 Set Join Approaches Comparisons

In previous experiments, all partitioning strategies adopt All-Pairs [3] using cosine distance in the inverted indexing and joining phase. To generalize our findings, we also completed the experiments for other set join approaches and other distance measures: PPJ [34] using Jaccard distance measure, All-Pairs [3] using Jaccard distance measure. All the optimizations and experimental settings in previous experiments are used here to ensure a fair comparison.

The single-threaded performance of PPJ Jaccard and All-Pairs Jaccard are shown in Figure 7 and 8. Consistent with previous findings, local partitioning strategies outperform using a single-thread but all partitioning strategies are still comparable. Comparing these three set join approaches for the same parameter settings, we found that PPJ Jaccard is the fastest, All-Pairs Jaccard second, and All-Pairs cosine the slowest. We believe that the inefficiency of All-Pairs cosine stems from the fact that cosine distance requires floating point operations while Jaccard distance only needs integer operations. The PPJ Jaccard approach is more efficient than All-Pairs Jaccard as PPJ employs positional information to increase filtering power. Detailed comparisons between PPJ and All-Pairs Jaccard can be found in [34].

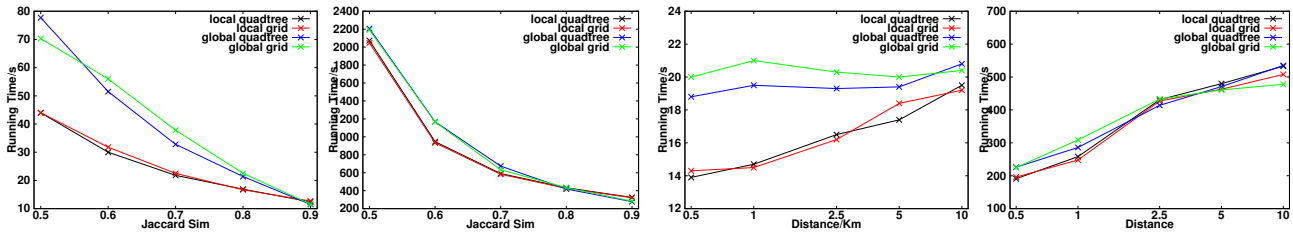
The multi-threaded speedups of PPJ Jaccard and All-Pairs Jaccard are shown in Figure 9 and 10. For the PPJ Jaccard approach, we see that global partitions achieve 8-16 times speedup and local partitions obtain a speedup of 2-5 times. Since we use the default task size for all threshold settings, we expect to observe some fluctuations in multi-threaded speedup. However, for the All-Pairs Jaccard approach, we noticed that the global partitions don't achieve the expected speedup. This is because the All-Pairs Jaccard approach requires objects to be sorted in ascending order of size, thereby conflicting with the global partitions requirement that objects need to be sorted by z-order. As discussed in Section 3.2, for such approaches, we need to visit all neighbor nodes instead of only the four neighbors in the $\{W, NW, NE, N\}$ directions. In this case, the filtering overhead of global partitions increases, and thus the average running time of global partition tasks increases. Nevertheless, global partitions achieve better multi-threaded speedup than local partitions.

5. FUTURE WORK AND CONCLUSIONS

An obvious question in response to our work is: in addition to grids and quadtrees, why not also consider R-trees and variants [15, 4, 21]? We believe that R-trees are not appropriate for several reasons. First, similar objects of a target object x can only be located in the same or neighbor nodes of x (otherwise it will not satisfy the distance threshold). Quadtrees and grids allow us to easily access neighborhoods in a efficient manner. On the other hand, R-trees are primary designed to support fast access in terms of containments relationships. To access neighbors, we would need to visit all nearby MBRs (minimum bounding rectangles) and check the distances to identify the neighbor nodes. This is less efficient than accessing neighbor nodes via z-order in quadtrees or grids. Furthermore, there is no upper bound on the size of a minimum bounding rectangle, which means that we may need to consider a large number of irrelevant objects (i.e., far outside the distance threshold). Second, our global index approach requires a consistent ordering of nodes, which is a intrinsic property of quadtrees and grids but is difficult for R-trees. Indeed, Bouros et al. [5], show that R-trees and their variants are slower than grid partition in terms of single-threaded performance.

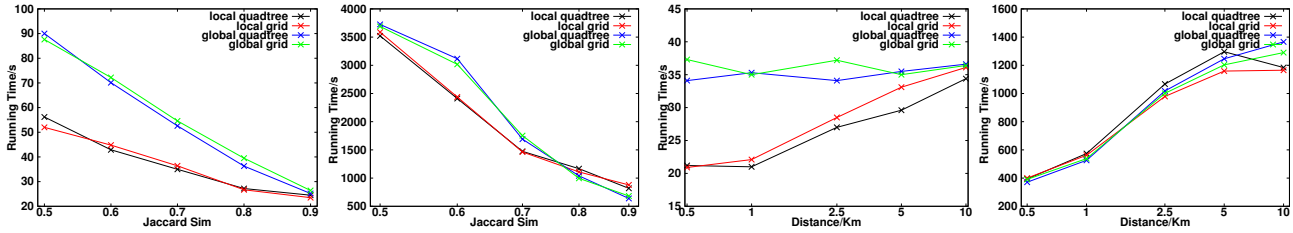
We see two future extensions of our work. First, as join performance is super-linear with the size of collections, we can replace the underlying set join approaches with probabilistic techniques, e.g., based on locality sensitive hashing [10] or some other signature-based approaches [24], to accelerate the join performance. These probabilistic techniques are able to identify most similar object pairs in an approximate but efficient manner. Another possible extension is to scale up our algorithms in a distributed framework [13, 12]. MapReduce [12] is one obvious choice as it offers partitioning and aggregation as basic primitives, which share similarities to our multi-threaded implementations.

To conclude, this work presents a methodical analysis of the STJoin problem in multiple dimensions: local vs. global partitioning, grid vs. quadtree, All-Pairs vs. PPJ, and single- vs. multi-threaded. Although some of these combinations have been previously explored, our experiments thoroughly evaluate performance of STJoin under different variants and parameter settings. From experiments on two real-world datasets, we find that single-threaded implementations of these approaches are roughly comparable in performance.



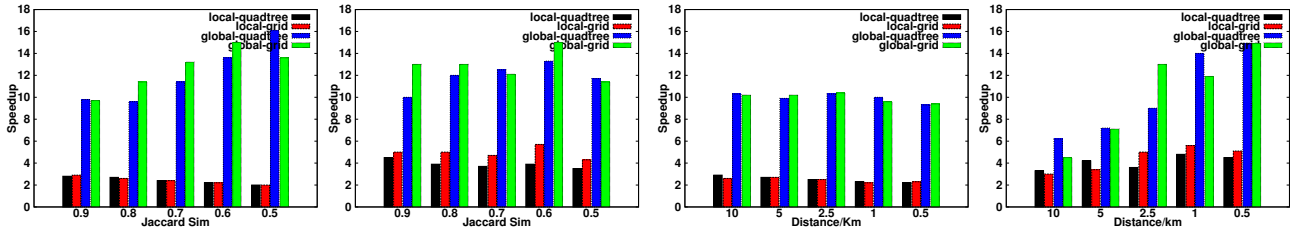
(a) NewsStand, fix t=2.5km (b) World Cities, fix t=2.5km (c) NewsStand, fix s=0.8 (d) World Cities, fix s=0.8

Figure 7: PPJ single-threaded performance.



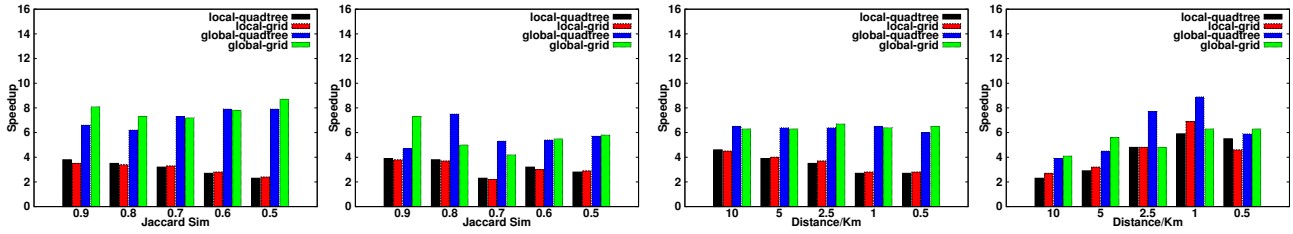
(a) NewsStand, fix t=2.5km (b) World Cities, fix t=2.5km (c) NewsStand, fix s=0.8 (d) World Cities, fix s=0.8

Figure 8: All-Pairs Jaccard single-threaded performance.



(a) NewsStand, fix t=2.5km (b) World Cities, fix t=2.5km (c) NewsStand, fix s=0.8 (d) World Cities, fix s=0.8

Figure 9: PPJ multi-threaded speedup.



(a) NewsStand, fix t=2.5km (b) World Cities, fix t=2.5km (c) NewsStand, fix s=0.8 (d) World Cities, fix s=0.8

Figure 10: All-Pairs Jaccard multi-threaded speedup.

In terms of the underlying similarity join algorithms, PPJ is faster than All-Pairs for Jaccard similarity, but when cosine similarity is desired, All-Pairs remains the preferred choice. However, in a multi-threaded setting, we find that the global index approach yields significantly better speedups than the local approach. Our results suggest that load balancing is a fundamental issue regardless of algorithms, and highlight the importance of evaluating parallel performance in today's multi-core computing environments.

6. ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation under awards IIS-12-18043, IIS-10-18475, IIS-12-19023, and IIS-13-20791. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsor.

7. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[2] J. Ballesteros, A. Cary, and N. Rishe. SpSJoin: parallel spatial similarity joins. In *GIS*, 2011.

[3] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. *WWW*, 2007.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[5] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 2012.

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

[7] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, 2011.

[8] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, 2006.

[9] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2009.

[10] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Annual Symposium on Computational Geometry*, 2004.

[11] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.

[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 2008.

[13] A. Eldawy and M. F. Mokbel. A demonstration of SpatialHadoop: an efficient MapReduce framework for spatial data. *PVLDB*, 2013.

[14] J. Fan, G. Li, L. Zhou, S. Chen, and J. Hu. Seal: Spatio-textual similarity search. *PVLDB*, 2012.

[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[16] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSBDM*, 2007.

[17] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, 1998.

[18] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 1999.

[19] E. H. Jacox and H. Samet. Spatial join techniques. *TODS*, 2007.

[20] E. H. Jacox and H. Samet. Metric space similarity joins. *TODS*, 2008.

[21] Z. Li, K. C. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. IR-Tree: An efficient index for geographic document search. *TKDE*, 2011.

[22] S. Liu, G. Li, and J. Feng. Star-Join: Spatio-textual similarity join. In *CIKM*, 2012.

[23] S. Liu, G. Li, and J. Feng. A prefix-filter based method for spatio-textual similarity join. *TKDE*, 2013.

[24] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

[25] S. Nutanong, E. H. Jacox, and H. Samet. An incremental Hausdorff distance calculation algorithm. *PVLDB*, 2011.

[26] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[27] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *CACM*, 2003.

[28] H. Samet, J. Sankaranarayanan, M. D. Lieberman, M. D. Adelfio, B. C. Fruin, J. M. Lotkowski, D. Panozzo, J. Sperling, and B. E. Teitler. Reading news with maps by exploiting spatial synonyms. *CACM*, 2014.

[29] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

[30] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *VLDB*, 2005.

[31] B. E. Teitler, M. D. Lieberman, D. Panozzo, J. Sankaranarayanan, H. Samet, and J. Sperling. NewsStand: A new view on news. In *GIS*, 2008.

[32] G. Tolias and Y. Avrithis. Speeded-up, relaxed spatial matching. In *ICCV*, 2011.

[33] K.-Y. Whang and R. Krishnamurthy. The multilevel grid file: a dynamic hierarchical multidimensional file structure. In *International Symposium on Database Systems for Advanced Applications*, 1992.

[34] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 2011.