Ad-hoc Distributed Spatial Joins on Mobile Devices

Panos Kalnis[1], Nikos Mamoulis[2], Spiridon Bakiras[3] and Xiaochen Li[1]

[1]Department of Computer Science
National University of Singapore
117543 Singapore
{kalnis, g0202290}@comp.nus.edu.sg

[2]Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
nikos@cs.hku.hk

[3]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
sbakiras@cs.ust.hk

Abstract

PDAs, cellular phones and other mobile devices are now capable of supporting complex data manipulation operations. Here, we focus on ad-hoc spatial joins of datasets residing in multiple non-cooperative servers. Assuming that there is no mediator available, the spatial joins must be evaluated on the mobile device. Contrary to common applications that consider the cost at the server side, our main issue is the minimization of the transferred data, while meeting the resource constraints of the device. We show that existing methods, based on partitioning and pruning, are inadequate in many realistic situations. Then, we present novel algorithms that estimate the data distribution before deciding the physical operator independently for each partition. Our experiments with a prototype implementation on a WiFi-enabled PDA, suggest that the proposed methods outperform the competitors in terms of efficiency and applicability.

1. Introduction

Modern mobile devices, such as mobile phones and Personal Digital Assistants (PDAs), provide many connectivity options together with substantial memory and CPU power. Novel applications that take advantage of these features are emerging. For example, users can download digital maps in their devices and navigate in unknown territories with the aid of add-on GPS receivers. General database queries are also possible. Nevertheless, in most cases requests are simply transmitted to the database server (or middleware) for evaluation, while the mobile device serves only as a dump client for presenting the results.

In many practical situations, complex queries need to combine information from multiple sources. Consider, for instance, the Michelin guide which contains classifications and reviews of top European restaurants. Although it provides the address of each restaurant, the accuracy of the accompanying maps varies considerably among cities. In Paris, for example, the maps go down to the street level (200 feet), while for Athens only a regional map (5 miles) is available. A traveller visiting Athens must combine the information from the Michelin site with accurate data from a local server (i.e., a map of the area together with hotels and tourist attractions), in order to answer the query "find the hotels in the historical center which are within 500 meters from a one-star restaurant".

Since the two data sources in this scenario are unlikely to cooperate, the query cannot be processed by either of them. Typically, queries to multiple heterogeneous sources are handled by mediators, which communicate with the sources and integrate information from them via wrappers. However, there are several reasons why this architecture may not be appropriate or even feasible. First, the services may not be collaborative; they may not be willing to share their data with other services or mediators, allowing only simple users to connect to them. Second, the user may not be interested in using the mediator, since she will have to pay for the service; retrieving the information directly

1-4244-0054-6/06/$20.00

from the sources may be less expensive. Finally, the user requests may be ad-hoc and not supported by existing mediators, as in our example. Consequently, the query must be evaluated on the mobile device.

Telecommunication companies typically charge the wireless connections by the bulk of transferred data (bytes or packets), rather than by the connection time. We are, therefore, interested in minimizing the amount of exchanged information, instead of the processing cost at the servers. Indeed, the user is typically willing to sacrifice a few seconds in response time, in order to minimize the query cost in dollars. We also assume that services allow only a limited set of queries through a standard interface (e.g., window queries). Therefore, the user does not have access to the internal statistics or index structures of the servers.

Formally, the problem is defined as follows. Let R and S be two spatial relations located at different servers, and b_R, b_S be the cost per transferred unit (i.e., byte) from the server of R and S, respectively. We want to evaluate the spatial join $R \bowtie_\theta S$ in a mobile device, while minimizing the cost with respect to b_R and b_S. We deal with *intersection* [2] and *distance* joins [6, 4, 15]; in the latter case, the qualifying object pairs should be within distance ε. We also consider the *iceberg distance semi-join*. This query differs from the distance join in that it asks only for objects from R (i.e., semi-join), with an additional constraint: the qualifying objects should 'join' with at least m objects from S. As a representative example, consider the query "find the hotels which are close to *at least* 10 restaurants".

Previous work proposed the *MobiJoin* algorithm [9] for evaluating spatial joins on mobile devices. MobiJoin partitions recursively the datasets and retrieves statistics in order to prune the search space. While MobiJoin exhibits substantial savings compared to naïve methods, it does not consider the data distributions inside the partitions. In many practical situations this results to inefficient processing, especially when the cardinalities of the joined datasets differ significantly, or when there is more memory available on the PDA.

In this paper, we first analyze the behavior of MobiJoin, concluding that the source of inefficiency is the implicit assumption of uniformity inside the data partitions which results to inaccurate estimation of the repartitioning cost. Based on our findings, we present two novel algorithms, namely Uniform Partition Join (*UpJoin*) and Similarity Related Join (*SrJoin*), that avoid the above pitfalls. Our algorithms examine every partition of the data space and make a decision on the physical operator that will be applied based on (i) the cost of applying the physical join operator, and (ii) the relative uniformity of each space. The aim is to identify and prune areas which cannot possibly participate in the result (e.g., do not download any hotels if there is no one-star restaurant in the area), while keeping the number of aggregate queries at acceptable levels. Depending on the retrieved statistics, different fragments can be processed by different physical operators (*adaptivity*). Our experiments with a prototype implementation on a PDA equipped with WiFi, verify that our methods avoid the drawbacks of the previous approach and can be efficiently applied in practice.

The rest of the paper is organized as follows. Section 2 overviews some previous work on spatial joins. Section 3 introduces the problem formulation and discusses the existing approaches. In Section 4 we present our improved algorithms, while Section 5 contains a detailed experimental evaluation. Finally, Section 6 concludes our work.

2. Related Work

There are several spatial join algorithms that apply on centralized spatial databases. Most of them focus on the *filter* step of the spatial *intersection* join. Their aim is to find all pairs of object MBRs (minimum bounding rectangles) that intersect. The qualifying candidate object pairs are then tested on their exact geometry at the final *refinement* step. Typically, spatial join algorithms presume that one [10] or both datasets [2] are indexed by hierarchical access methods (i.e., R-trees). This is not directly related to our problem, since the remote client cannot access the internal server indexes.

On the other hand, spatial join algorithms for non-indexed data are more relevant. The Partition Based Spatial Merge (PBSM) join [13] uses a regular grid to hash both datasets R and S into P partitions $R_1, R_2, \ldots, R_P$ and $S_1, S_2, \ldots, S_P$, respectively. Objects that fall into more than one cells are replicated into multiple buckets. The second phase of the algorithm loads pairs of buckets R_x with S_x that correspond to the same cell(s) and joins them in memory. The data declustering nature of PBSM makes it attractive for our problem. The Spatial Hash Join algorithm proposed in Ref. [7] is similar to PBSM, in that it uses hashing to reduce the size of the problem to smaller ones that fit in memory. This algorithm, however, uses irregular space partitioning to define the buckets. The construction of the hash bucket extents is computationally expensive; in addition, the whole R has to be read before finalizing the bucket extents, thus this method is not suitable for our settings.

Distributed processing of spatial joins has been stud-

We will use the term 'PDA' and 'Mobile Device' interchangeably in the rest of the paper.

ied in [16]. Datasets are indexed by R-Trees, and the intermediate levels of the indexes are transferred from one site to the other, prior to transferring the actual data. Thus, the join is processed by applying semi-join operations on the intermediate tree level MBRs in order to prune objects, minimizing the total cost. Our work is different, since we assume that the sites do not collaborate with each other, and they do not publish their index structures.

Many of the issues we are dealing with, also exist in distributed data management with mediators. Mediators provide an integrated schema for multiple heterogeneous data sources. Queries are posed to the mediator, which constructs the execution plan and communicates with the sources via custom-made wrappers. The HERMES [1], DISCO [17], and Garlic [14] mediator systems maintain statistics in order to optimize the execution of queries. Our statistics retrieval method is closer to Garlic. Nevertheless, Garlic acquires cost information during initialization and uses it to optimize all subsequent queries, while we optimize the entire process of statistics retrieval and query execution for a single query. The Tuckila [5] system also combines optimization with query execution. Our approach is different, since we optimize the execution of the current (and only) operator, while Tuckila uses statistics from the current results to optimize the subsequent operators.

3. Spatial Joins on Mobile Devices

As discussed previously, the join evaluation cannot be assisted by internal index structures on the servers, since they are unlikely to be published. Therefore, communication with the servers is limited to a small set of operations. We assume that the following queries are available:

- WINDOW query: return all the objects intersecting a window w.
- COUNT query: return the number of objects intersecting a window w.
- ε-RANGE query: return all objects within distance ε from a point p.

It is safe to assume the existence of the COUNT query, since it is typical for the servers to send an acknowledgment containing the size of the query result, prior to sending the actual data. Also, if the ε-RANGE query is not available, we can simulate it by a WINDOW query with the window's sides equal to 2ε.

There are two types of information interchanged between the client and the servers: (i) the queries sent to the servers, and (ii) the results sent back by the servers. The main issue is to minimize the total amount of transferred data for a given operation. The simplest way to execute the spatial join is to download both datasets to the PDA and perform the join there. In general, this is an infeasible solution, since mobile devices have limited storage capability. A divide-and-conquer alternative is to perform the join in one spatial region at a time. Thus, the space is divided into rectangular areas (e.g., using a regular grid), a window query is sent for each cell to both sites, and the results are joined on the device using a main memory join algorithm. In the example of Figure 1, the hotels that intersect A1 are downloaded from R, the restaurants that intersect A1 are downloaded from S, and the results are joined on the PDA. Duplicate avoidance techniques [3, 8] can be employed to avoid reporting a pair more than once. In the case of a distance join, the cells are extended by $\varepsilon/2$ at each side before they are sent as window queries. If the data do not fit in memory, the cell can be recursively partitioned (e.g., PBSM [13]).

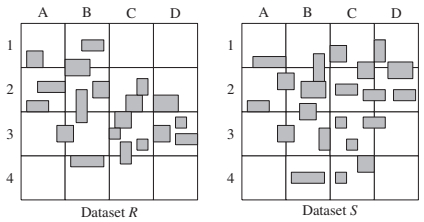


Figure 1. Two datasets to be joined

A drawback of the partition-based technique is that it downloads all objects from both datasets. However, we can achieve sublinear transfer cost by pruning areas that do not contain any results. For example (see Figure 1), if we know that cells C1 and D1 are empty in R, we can avoid downloading their contents from S. The intuition is to apply some cheap queries first, which will provide information about the distribution of objects in both datasets. Since the cost at the server side is not a concern, we first apply a COUNT query for the current partition on each server. Therefore, we retrieve the number of objects in each cell and avoid downloading data in areas where at least one of the relations is empty.

In some cases we must repartition the space recursively, in order to identify the empty areas. For example, if we partition the space of Figure 1 in quadrants, CD12 cannot be pruned since it is not empty. However, if we recursively divide CD12, we can identify the empty cells C1 and D1. On the other hand, observe

COUNT queries can be answered fast by data structures such as the aR-tree [11] or the aHRB-tree [12].

that refining such partitions may have a counter-effect in the overall cost due to the overhead of the additional aggregate queries. In this case, it might be more beneficial to stop drawing statistics for this area and perform the join as a series of selection queries. Thus, the join processing for quadrant CD12 proceeds as follows: (i) download all hotels from R intersecting CD12, and (ii) for each hotel apply a window query on S to find the matching restaurants. This method resembles the Nested Loop join algorithm and can be efficient if $|R| \ll |S|$.

3.1. The Cost Model

Let $|R_w|$ and $|S_w|$ be the number of objects from R and S, respectively, intersected by a window w. Below we define four execution strategies and their coresponding cost functions $c_{1...4}(w)$:

- $c_1(w)$ is the cost of performing a *Hash-Based Spatial Join* (HBSJ), by downloading $|R_w|$ objects from R and $|S_w|$ objects from S and joining them on the PDA. If the PDA's buffer cannot accommodate $|R_w| + |S_w|$ objects, $c_1(w) = \infty$.
- $c_2(w)$ is the cost of performing a *Nested Loop Spatial Join* (NLSJ) by downloading $|R_w|$ objects from R, sending them as window queries to server that hosts S and receiving the results.
- $c_3(w)$ is the cost of performing NLSJ by downloading $|S_w|$ objects from S, sending them as window queries to server that hosts R and receiving the results. Notice that $c_2(w)$ and $c_3(w)$ may differ depending on (i) which of the $|R_w|$ and $|S_w|$ is the smallest, and (ii) the communication costs with each of the sites.
- $c_4(w)$ is the cost of applying recursive counting in $|R_w|$ and $|S_w|$, retrieve more detailed statistics, and apply the join algorithms recursively.

The largest amount of data that can be transferred in one packet on the network is referred to as *MTU* (Maximum Transmission Unit). Each network packet consists of the TCP/IP headers (with a typical size of $B_H = 40$ bytes) and the actual data. Let D be a dataset. The size of D in bytes is $B_D = |D| \cdot B_{obj}$, where B_{obj} is the size of each object in bytes. Thus, when the whole D is transmitted through the network, the number of transferred bytes is:

$$T_B(B_D) = B_D + B_H \cdot \left\lceil \frac{B_D}{MTU - B_H} \right\rceil \quad (1)$$

The MTU depends on the physical network layer; Ethernet, for instance, has $MTU = 1500$ bytes, while dial-up connections usually support $MTU = 576$ bytes.

where the second component of the equation is the overhead of the TCP/IP headers.

The cost of sending a query to a server is $B_H + B_Q$, where B_Q is the size of the query string in bytes. Let b_R and b_S be the per byte transfer cost (e.g., in dollars) for sites R and S, respectively. The total cost of downloading the objects from R and S and joining them on the PDA is:

$$\begin{aligned} c_1(w) =& (b_R + b_S)(B_H + B_Q) + b_R T_B(|R_w| \cdot B_{obj}) \\ &+ b_S T_B(|S_w| \cdot B_{obj}) \end{aligned} \quad (2)$$

Now let us consider the cost c_2 of downloading all $|R_w|$ objects from R and sending them as ε-RANGE queries to S. The expected number of points in S_w within distance ε from a point p is $\frac{\pi \cdot \varepsilon^2}{w_x \cdot w_y} \cdot |S_w|$, assuming uniform distribution in w, where w_x and w_y are the lengths of the window's sides. The total number of bytes for transmitting the ε-RANGE query and receiving the results is:

$$T_{dq}(w, \varepsilon) = (B_H + B_Q) + T_B\left(\frac{\pi \cdot \varepsilon^2}{w_x \cdot w_y} |S_w| \cdot B_{obj}\right) \quad (3)$$

Therefore, the total cost of downloading the objects from R intersecting w, sending them one by one as distance queries to S and receiving the results is:

$$\begin{aligned} c_2(w) =& b_R(B_H + B_Q) + b_R T_B(|R_w| \cdot B_{obj}) \\ &+ b_S |R_w| \cdot T_{dq}(w, \varepsilon) \end{aligned} \quad (4)$$

We assumed that the ε-RANGE query processes one point at a time. However, if the database server supports *bucket* queries we can send many ε-RANGE queries simultaneusly, thus redusing the overhead of the TCP/IP headers. In this case, the cost of downloading all $|R_w|$ objects from R and sending them as *bucket* ε-RANGE queries to S is:

$$T_{br}(w, \varepsilon) = (b_R + b_S) T_B(|R_w| \cdot B_{obj})$$

The cost of retrieving all the results from S is:

$$T'_{dq}(w, \varepsilon) = T_B\left(\left(\frac{\pi \cdot \varepsilon^2}{w_x \cdot w_y} |S_w| B_{obj} + B_{obj}\right) |R_w|\right) \quad (5)$$

Therefore, the total cost of downloading the objects from R intersecting w, sending them as bucket queries to S, and receiving the results is:

$$\begin{aligned} c'_2(w) =& (b_R + b_S)(B_H + B_Q) \\ &+ (b_R + b_S) T_B(|R_w| \cdot B_{obj}) + b_S \cdot T'_{dq}(w, \varepsilon) \end{aligned} \quad (6)$$

The cost c_3 is also given by Equation (4) (respectively, (6)), by exchanging the roles of R and S. If

the objects are polygons instead of points, we can use the same derivation, but we need statistics about the average area of the object MBRs intersecting w for R and S. These can be obtained from the server when we retrieve $|R_w|$ and $|S_w|$ (i.e., we can post an additional aggregate query together with the COUNT query).

The cost of sending an aggregate query and receiving the results is:

$$T_{aq} = (B_H + B_Q) + (B_H + B_A) \quad (7)$$

where B_A is the size of the answer string (usually it consists of one long integer). In order to repartition w in $k \times k$ partitions, we must send k^2 aggregate queries to each site. Next, each partition w' will be processed by the cheapest method $c_{1...4}(w')$. Therefore,

$$c_4(w) = 2k^2 \cdot T_{aq} + \sum_{\forall w'} \min\{c_1(w'), c_2(w'), c_3(w'), c_4(w')\} \quad (8)$$

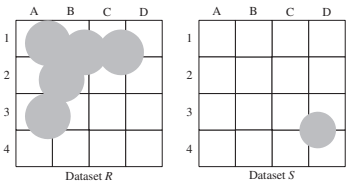
3.2. The MobiJoin Algorithm

The MobiJoin algorithm [9] is the basis of our improved methods. MobiJoin works as follows: First it sends COUNT queries to both R and S to retrieve statistics for a window w. If R_w or S_w is empty, the algorithm returns. Otherwise, the algorithm estimates the cost of $c_{1...4}(w)$ and follows the action with the lowest cost. Each recursive step (action c_4) divides the space into a regular $k \times k$ grid, where k is fixed to 2.

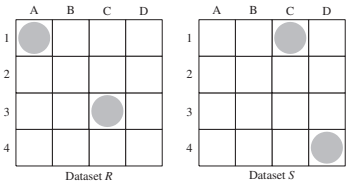
Notice that Equation (8) is recursive and is, therefore, difficult to estimate prior to repartitioning, since we do not know which partitions will be pruned and which will need further refinement. MobiJoin, assumes that w is uniform and small enough so that every subwindow w' will be processed by HBSJ after only one partitioning. Nevertheless, this heuristic may not be appropriate in many practical cases. Figure 2(a) presents such an example: here, $|R| \gg |S|$, therefore, c_3 is the minimum cost and MobiJoin will perform NLSJ by downloading all objects from S and sending them as individual queries to R. However, if one more recursive step is allowed, the entire space can be pruned. Notice that this problem can arise at any level of recursion, so in the general case it will not be solved by simply allowing one additional step.

Figure 2(b) presents a different case: assume that each cluster contains 500 points and the PDA's memory can accommodate 1900 points. c_1 is inapplicable, since HBSJ requires a buffer size of at least $4 \cdot 500 = 2000$ points. Therefore, the space is partitioned in four quadrants and in the next step the empty areas AB12, CD12 and AB34 are pruned. Assume now that we increase the PDA's memory to 2000 points. Since there is enough memory for HBSJ, all points from both datasets are downloaded. Thus by increasing the available resources, the transfer cost is doubled! This problem is amplified by the recursive nature of the algorithm. For instance, if the PDA's buffer is less than 1000 points, quadrant CD34 will be further partitioned and all areas will be pruned.

Pruning all areas after one step is the best scenario for c_4. In this case, $c_4(w) = 2k^2 \cdot T_{aq}$, i.e., only the cost of the aggregate queries. This approximation forces more recursive steps, so it could be a potential solution to the previous problems. Unfortunately, there is the counter-effect of increasing the total cost due to the excessive number of aggregate queries, especially for datasets with relatively uniform areas. Another possible solution is to increase the number k of partitions at each step. However, our experiments revealed two drawbacks: (i) for large buffers the problem persists, and (ii) for larger k the overhead due to aggregate queries increases significantly.



(a) Inefficient Nested Loop join



(b) Inefficient Hash-based join

Figure 2. Drawbacks of MobiJoin

4. Distribution-aware Methods

It is apparent from the above analysis that we need a robust criterion to decide when to stop retrieving more statistics. Next, we present two algorithms, namely the *Uniform Partition Join* (UpJoin) and the *Similarity Related Join* (SrJoin) that solve the previous problems,

MobiJoin would not choose c_2 or c_3, since the cost of downloading 1000 points, sending them one by one as queries and retrieving the results, is larger than c_1.

by considering the data distribution inside w.

4.1. The Uniform Partition Join Algorithm

The motivation behind UpJoin is simple: we attempt to identify regions where the object distribution is relatively uniform. In such regions, the cost estimations of our model are accurate; therefore, we can decide safely which action to perform, without requiring knowledge of the future recursive steps.

The algorithm (Figure 3) is called with the query window w and the number of objects from datasets R and S intersecting w. Similar to the previous method, UpJoin prunes the areas where at least one of the datasets is empty. However, before deciding which physical operator to apply, UpJoin decomposes w into a regular 2×2 grid and retrieves the number of objects for each cell. Based on this information, it checks whether each dataset $\mathcal{D}, \mathcal{D} \in \{R, S\}$ is uniform, by using the following formula:

$$\left| \frac{|\mathcal{D}_w|}{4} - |\mathcal{D}_{w'i}| \right| < \alpha \cdot |\mathcal{D}_w| \quad (9)$$

where w'_i is a quadrant of w and $\alpha \in (0, 1]$ is a system-wide parameter. If all quadrants satisfy the inequality, $\mathcal{D}_w$ is considered uniform. Notice that Equation (9) implies that all quadrants should have approximately the same number of objects. For some distributions, this requirement creates problems. For instance, a 2D Gaussian distribution whose mean is located at the center of w, would be mistaken as uniform. In practice, this is an extreme case, assuming that $\alpha \simeq 0$. However, such a small value of α tends to over-partition the space, generating significant overhead due to aggregate queries, especially when the entire dataset is uniform. Therefore, we must set α to a larger value, which increases the probability of characterizing $\mathcal{D}_w$ incorrectly. In order to minimize this problem, we submit an additional COUNT query (line 6) if the statistics suggest that $\mathcal{D}_w$ is uniform. The window size of the extra query is equal to a quadrant of $\mathcal{D}_w$, but its location is chosen randomly inside $\mathcal{D}_w$. If the new result satisfies Equation (9), the algorithm decides that the distribution of $\mathcal{D}_w$ is indeed uniform.

In the best case, UpJoin can identify a skewed dataset by issuing only three aggregate queries, since $|\mathcal{D}_{w'4}| = |\mathcal{D}_w| - \sum_{i=1}^{3} |\mathcal{D}_{w'i}|$. However, if the number of objects inside $\mathcal{D}_w$ is small, the cost of the aggregate queries is higher than downloading the objects. Therefore, the algorithm will ask for more statistics only if $\mathcal{D}_w$ is large enough (line 3). Formally, the following inequality must be satisfied:

$$T_B(|\mathcal{D}_w| \cdot B_{obj}) > 3 \cdot T_{aq} \quad (10)$$

```
// |R_w|, |S_w| are the number of objects from R, S
// which intersect window w
UpJoin(w,|R_w|,|S_w|)
1.  if |R_w| = 0 or |S_w| = 0 then return;
2.  for each dataset D, D ∈ {R, S}
3.      if |D_w| is large and D_w is not uniform then
4.          impose a regular 2 × 2 grid over D_w;
5.          for each cell w' ∈ D_w retrieve |D_w'|;
6.          if D_w is uniform then sent a random count query;
7.      else assume that D_w is uniform;
    // Assume c3(w) < c2(w) (the other case is symmetric)
8.  calculate c1(w), c2(w), c3(w);
9.  if c1 < c3 then
10.     if both datasets are uniform
            and there is enough memory then HBSJ(w);
11.     else for each cell w' ∈ w do UpJoin(w',|R_w'|,|S_w'|);
12. else if c3 < c1 then
13.     if the largest dataset is uniform then NLSJ(w);
14.     else for each cell w' ∈ w do UpJoin(w',|R_w'|,|S_w'|);
```

Figure 3. The uniform partition join algorithm

Here, T_{aq} represents the cost of sending a single aggregate query.

Also notice that one of the datasets may have already been characterized as uniform at a previous step. In this case, UpJoin does not request additional statistics (line 3); instead, it estimates the number of objects in the quadrants $\mathcal{D}_{w'i}$, based on $|\mathcal{D}_w|$ and the uniformity assumption. The algorithm will issue additional aggregate queries for $\mathcal{D}$ only when accuracy is crucial, i.e., when applying the physical operators.

In line 8, UpJoin calculates the costs $c_{1...3}$. It is not necessary to compute c_4, since the criterion for repartitioning is the data distribution. In Figure 3 we assume that $c_3 < c_2$ and, therefore, S will be the outer relation if NLSJ is executed; the other case is symmetric.

If $c_1 < c_3$ and there is enough memory on the PDA to accommodate $|R_w| + |S_w|$ objects, the algorithm will join the windows by employing HBSJ. If there is not enough memory on the PDA, the algorithm will decompose the window into several subparts which can be accommodated in the PDA's memory and join them accordingly. However, if at least one dataset is skewed, it is possible that HBSJ will be inefficient (similar to Figure 2(b)). In this case, UpJoin decides to further partition the space.

On the other hand, if $c_3 < c_1$ there is no memory constraint and NLSJ can be applied. Nevertheless, there is also a possibility of inefficient processing, similar to the example of Figure 2(a). To avoid this problem, UpJoin repartitions the window if the larger dataset (i.e., the inner relation R) is skewed. Notice that if the outer relation S is skewed but R is uniform, there is no need to repartition. This is due to the fact that the cost of NLSJ is mainly determined by the number of objects in S. Since R is uniform, it

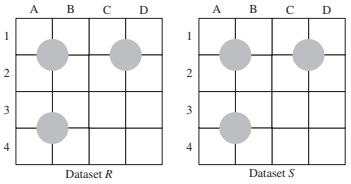


Figure 4. Inefficient UpJoin

is unlikely to contain large empty areas, so it cannot prune any objects from S. Therefore, even if S causes part of R to be pruned, the cost will remain roughly the same. Of course it is possible that in the next step the relationship between c_3 and c_1 changes, in which case repartitioning may be beneficial. However, we found that this rarely happens in practice while our method saves many aggregate queries in most cases.

4.2. The Similarity Related Join Algorithm

The advantage of UpJoin compared to MobiJoin is that it considers the distribution of each dataset before applying the physical operation on each partition. But in some cases, just considering the distribution of separate datasets can not provide adequate information to make the correct choice for the next step. Figure 4 presents such a case: UpJoin will label both datasets as skewed, and then recursively repartition them. However, the distributions of these two datasets are very similar, and we cannot prune any points after repartitioning. Since the objects are clustered in the centers of areas AB12, CD12, and AB34, UpJoin will also label these areas as skewed after repartitioning (line 6 of Figure 3). Therefore, the recursion will continue but the cost of sending the aggregate queries will not be compensated.

Here we present SrJoin, which addresses the above drawbacks. SrJoin attempts to compare the distribution of both datasets and decides the next action based on their relationship. If the distribution of these datasets is similar, applying HBSJ or NLSJ on these subparts (according to the cost model) without requiring knowledge of the future recursive steps is more beneficial. Otherwise, we expect that the data distribution at the next level is also skewed, and pruning can be performed. SrJoin uses the four cells of the current window w to estimate the data distribution for the whole window w. Let $|\mathcal{D}_{wi}|$ be the number of objects inside a cell w_i of each dataset $\mathcal{D}, \mathcal{D} \in \{R, S\}$. Let $|\mathcal{A}_{wi}|$ be the area of w_i. Cell w_i is *dense* if the following equation is satisfied:

$$|\mathcal{D}_{wi}| > \rho \frac{|\mathcal{D}_w|}{|\mathcal{A}_w|} \cdot |\mathcal{A}_{wi}| \quad (11)$$

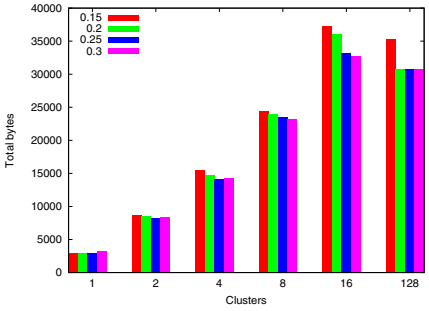
```
// w1,..., w4 are the four quadrants of window w
// |Rwi|, |Swi| are the number of objects from R, S
// which intersect window wi
SrJoin(w,|Rw|,|Sw|)
1.  for each dataset D, D ∈ {R,S}
2.      impose a regular 2 × 2 grid over Dw;
3.  for i=1 to 4
4.      if |Rwi| > ρ(|Rw|/|Aw|)|Awi| then bitR[i]=1 else bitS[i]=0;
5.      if |Swi| > ρ(|Sw|/|Aw|)|Awi| then bitR[i]=1 else bitS[i]=0;
6.  if bitmaps bitR and bitS are equal then
7.      for i=1 to 4
8.          if |Rwi|=0 or |Swi|=0 then continue; // next i
9.          compute cost of c1(wi) and c2(wi);
            // c3(wi) is a symmetric case
10.         if c1(wi) < c2(wi) then apply HBSJ on wi;
11.         else apply NLSJ on wi;
12. else
13.     for i=1 to 4
14.         if |Rwi|=0 or |Swi|=0 then continue; // next i
15.         compute cost of c1(wi) and c2(wi);
            // c3(wi) is a symmetric case
16.         if c1(wi) < 3 · Taq or c2(wi) < 3 · Taq then
                // the dataset must be large
17.             if c1(wi) < c2(wi) then apply HBSJ on wi;
18.             else apply NLSJ on wi;
19.         else apply SrJoin on window wi;
```

Figure 5. The similarity related join algorithm

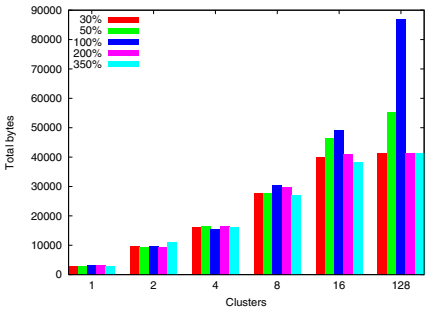
where ρ is a system-wide parameter and $\frac{|\mathcal{D}_w|}{|\mathcal{A}_w|}$ is the average density of window w.

For the current w, two 4-bitmaps are created; one for R and one for S. If a quadrant w_i is *dense*, the corresponding bit is set. Then, we determine the next action for each quadrant w_i. If one of the windows is empty for at least one of R and S, we prune the window as before (no results). If both 4-bitmaps are the same, we assume that the distribution of the two datasets is the same and there is no need for repartitioning. For each quadrant, we choose to apply HBSJ or NLSJ, based on their cost estimation. Notice that if all the points can not fit into the memory, HBSJ is recursively executed and pruning can also be applied at each recursion level.

If the two 4-bitmaps are different, we compute the cost of HBSJ and NLSJ. If repartitioning is more expensive than HBSJ or NLSJ, we choose to apply the cheapest action, as specified by the cost model. Otherwise, we apply repartitioning hoping to prune the search space. Here, we assume that if the data distribution for window w of R and S is different, the distribution of the four quadrants of w of R and S will also be different, and several of the points can be pruned. Therefore, we make an aggressive estimation for the cost of repartitioning, which only includes the cost of the aggregate queries (no points need to be transferred). The complete algorithm is shown in Figure 5.



(a) Parameter α for UpJoin



(b) Parameter ρ for SrJoin

Figure 6. Setting the parameters

5. Experimental Evaluation

In this section, we present a detailed experimental study of our methods. All the algorithms were implemented in Visual C++ for Windows Pocket PC. Our prototype run on an HP-IPAQ PDA with a 400MHz RISC processor and 64MB RAM. The PDA was connected to the network through an IEEE 802.11b WiFi interface. The servers for the spatial datasets resided on UNIX machines. In all the experiments, we set $b_R = b_S$, i.e., the transfer cost was the same for both servers. We used synthetic datasets consisting of 1000 points, in order to simulate typical windows of users' requests. The points were clustered around k randomly selected centers, and for each cluster the distribution of objects was Gaussian. In order to achieve different skew levels, we varied k from 1 to 128. We also employed a real dataset (with around 35K objects) representing the railway segments of Germany. Unless specified otherwise, the PDA's buffer size was set to 800 points (i.e., 40% of the total data size for the synthetic datasets).

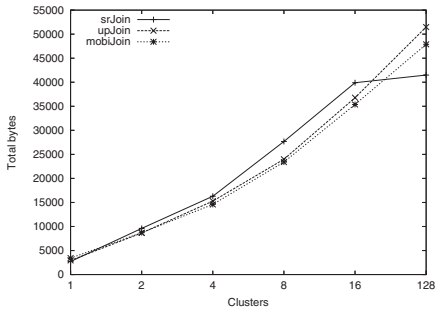
The first experiment examines the effect of parameter α in the UpJoin algorithm. Recall that α is used in Equation (9) in order to identify whether a window is uniform. In Figure 6 we plot the total amount of transferred bytes for different values of α. Each value in the diagrams represents the average of 10 executions with different datasets. Observe that setting $\alpha = 0.15$ tends to over-partition the space, and the overhead of retrieving the statistics increases significantly. On the other, a large value of α is also not desirable, since it can not identify empty areas efficiently. For the rest of the experiments we set $\alpha = 0.25$.

In the next experiment we investigate the effect of parameter ρ in the SrJoin algorithm. We express ρ as a percentage of the average density $\frac{|\mathcal{D}_w|}{|\mathcal{A}_w|}$ of window w. The results for various values of ρ are summarized in Figure 6(b). Setting $\rho = \frac{|\mathcal{D}_w|}{|\mathcal{A}_w|}$ tends to over-partition the datasets when they are uniform (i.e., $k = 128$), resulting in a much higher cost due to unnecessary COUNT queries. The performance of using $\rho = 30\%$ and 200% is quite similar and both of them fit the uniform datasets very well. Considering the overall performance for all cluster settings, we use the value of $\rho = 30\%$ for the rest of the paper.

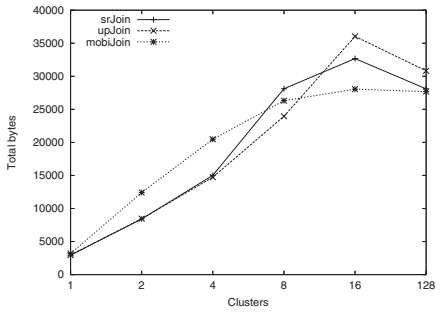
5.1. Comparison against MobiJoin

Here, we compare UpJoin and SrJoin against MobiJoin. In the first experiment we set the PDA's buffer to 100 points. Figure 7(a) shows that all three algorithms have similar performance for skewed datasets (i.e., small number of clusters), with SrJoin being slightly worse for moderate skew levels. However, when $k = 128$ (i.e., uniform dataset) the performance of UpJoin deteriorates, due to the fact that UpJoin tends to create unnecessary partitions for uniform datasets. On the other hand, for uniform datasets, SrJoin performs better that both MobiJoin and UpJoin.

Figure 7(b) shows the results of the same experiment, but for a buffer size of 800 points. The first thing worth noticing is the performance degradation of MobiJoin for skewed datasets. The reason for that is thoroughly explained in the example of Figure 2. Notice, however, that for uniform datasets MobiJoin works well. In this case, many regions are joined with HBSJ, and since the buffer size is large, HBSJ does not need to partition the region and introduce overhead. Observe that the performance of SrJoin exhibits a balanced tradeoff between the good results of UpJoin for skewed dataset and the results of MobiJoin for uniform datasets.



(a) Buffer = 100 points



(b) Buffer = 800 points

Figure 7. Comparing the three algorithms

5.2. Experiments with Real Data

The next experiment models the situation where a large dataset (e.g., the map of a city) is joined with a much smaller dataset (e.g., the hotels of the city). We use a real dataset of the German railway segments which contains around 35K objects, and a 1000-point synthetic dataset. The PDA's buffer is set to 800 points and we vary the skew of the small dataset. In Figure 8(a) we present the results for the *bucket* versions of the algorithms (see Section 3.1). Observe that the heuristic of MobiJoin performs poorly for real-life datasets, since it chooses to execute NLSJ most of the time. Both UpJoin and SrJoin easily outperform MobiJoin, especially for skewed datasets.

5.3. Comparison against Indexed Join

SemiJoin [16] is a distributed spatial join algorithm, which requires that at least one of the datasets is indexed by an R-tree. A revised version of SemiJoin was implemented in our PDA/server prototype. SemiJoin assumes that the servers collaborate with each other. Therefore, the MBRs and the qualifying objects can be directly transferred from one server to the other. In our environment, however, we assume that the servers are non-cooperative. Consequently, the PDA will act as the mediator between the two servers. If both datasets are indexed by R-trees, the algorithm identifies the smaller dataset, based on the information provided by the two R-trees. Without loss of generality, we assume that R is the small dataset. The algorithm chooses one level of MBRs from S and transfers them to server R, using the PDA as a mediator. Then, all the objects of R inside these MBRs will be transferred back to S, through the PDA. The final join step is performed at server S, and the results are returned to the PDA.

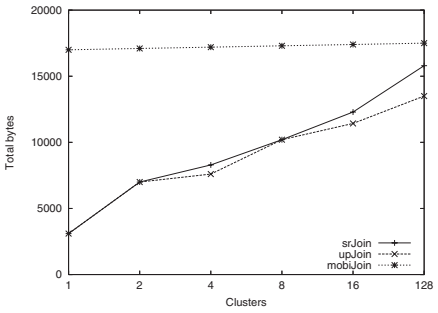
Here, we compare the performance of the *bucket* version of UpJoin and SrJoin against SemiJoin. Again, the use the German railway segments dataset and a 1000-point synthetic dataset. The results are shown in Figure 8(b). Both UpJoin and SrJoin have lower cost for skewed datasets while, for uniform datasets, SemiJoin is better. The cost of SemiJoin comprises of two parts — the cost of transferring the MBRs, and the cost of transferring the objects. For all cluster sizes, the cost of transferring the MBRs is identical, since we use the MBRs of the second to last level of the R-trees of the real dataset. However, the cost of transferring the objects varies, according to the distribution of the synthetic dataset. Therefore, SemiJoin is not ideal for skewed datasets, but for uniform datasets it is more efficient in pruning the empty space.

Notice that in practice, SemiJoin cannot be applied in our problemn, because the servers are unlikely to publish the internal structures of their indexes. The results, however, demonstrate that our algorithms achieve very good performance despite the absence of indexes.

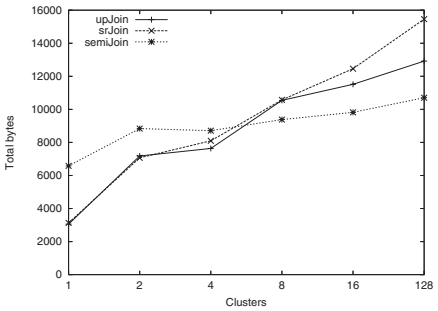
6. Conclusions

In this paper, we deal with the problem of executing spatial joins on mobile devices, where the datasets recite on separate remote servers. We assume that the servers are primitive, thus they support only three simple queries: (i) a *window* query, (ii) an *aggregate* query, and (iii) a *distance-selection* query. We also assume that the servers do not collaborate with each other, do not wish to share their internal indexes, and there is no mediator to perform the join of these two sites. These assumptions are valid for many practical situations. For instance, there are web sites providing maps, others with hotel locations, but a user may request an

Without using the bucket query submission the trend of the results was similar but the absolute values were higher.



(a) SrJoin and UpJoin vs. MobiJoin



(b) UpJoin and SrJoin vs. SemiJoin

Figure 8. Performance for the real datasets

unusual combination, such as "find all hotels which are at most 200km away from a rain forest". Executing this query on a mobile device must address two issues: (i) the limited resources of the device, and (ii) the fact that the user is charged by the amount of transferred information and wants to minimize this metric instead of the processing cost at the servers.

We showed that the existing partitioning and pruning method is inadequate in many practical situations. Motivated by this fact, we developed the UpJoin and SrJoin algorithms; UpJoin and SrJoin retrieve statistics in the form of simple aggregate queries and exam the data distribution before deciding to (i) repartition the space or (ii) join its contents by a nested loop or a hash-based method. While UpJoin evaluates the distribution of each dataset, SrJoin uses the relationship of the distribution of the two datasets to decide the next step action. Our experiments with a prototype implementation on a PDA equipped with a WiFi interface, verify that our methods avoid the drawbacks of the previous approach and can be efficiently applied in practice. In the future we plan to support complex spatial queries, involving more than two datasets.

References

[1] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of ACM SIGMOD*, pages 137–148, 1996.

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

[3] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proc. of ICDE*, pages 535–546, 2000.

[4] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. of ACM SIGMOD*, pages 237–248, 1998.

[5] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proc. of ACM SIGMOD*, pages 299–310, 1999.

[6] N. Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *Proc. of ICDE*, pages 466–475, 1998.

[7] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. of ACM SIGMOD*, pages 247–258, 1996.

[8] G. Luo, J. F. Naughton, and C. Ellmann. A non-blocking parallel spatial join algorithm. In *Proc. of ICDE*, pages 697–705, 2002.

[9] N. Mamoulis, P. Kalnis, S. Bakiras, and X. Li. Optimization of spatial joins on mobile devices. In *Proc. of SSTD*, pages 233–251, 2003.

[10] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *Proc. of ACM SIGMOD*, pages 1–12, 1999.

[11] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *Proc. of SSTD*, pages 443–459, 2001.

[12] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *Proc. of ICDE*, pages 166–175, 2002.

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

[14] M. T. Roth, F. Ozcan, and L. M. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. In *Proc. of VLDB*, pages 599–610, 1999.

[15] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *Proc. of ACM SIGMOD*, pages 343–354, 2000.

[16] K.-L. Tan, B.-C. Ooi, and D. J. Abel. Exploiting spatial indexes for semijoin-based join processing in distributed spatial databases. *IEEE TKDE*, 12(2):920–937, 2000.

[17] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with disco. *IEEE TKDE*, 10(5):808–823, 1998.