Optimization of Spatial Joins on Mobile Devices

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

[1] Department of Computer Science and Information Systems,
University of Hong Kong,
Pokfulam Road, Hong Kong
nikos@csis.hku.hk

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

[3] Department of Electrical and Electronic Engineering
University of Hong Kong
Pokfulam Road, Hong Kong
sbakiras@eee.hku.hk

Abstract. Mobile devices like PDAs are capable of retrieving information from various types of services. In many cases, the user requests cannot be directly processed by the service providers, if their hosts have limited query capabilities or the query combines data from various sources, which do not collaborate with each other. In this paper, we present a framework for optimizing spatial join queries that belong to this class. We presume that the connection and queries are ad-hoc, there is no mediator available and the services are non-collaborative. We also assume that the services are not willing to share their statistics or indexes with the client. We retrieve statistics dynamically in order to generate a low-cost execution plan, while considering the storage and computational power limitations of the PDA. Since acquiring the statistics causes overhead, we describe an adaptive algorithm that optimizes the overall process of statistics retrieval and query execution. We demonstrate the applicability of our methods with a prototype implementation on a PDA with wireless network access.

1 Introduction

The rapid development of mobile gadgets with computational, storage, and networking capabilities, has made possible for the user to connect to various distributed services and process information from them in an ad-hoc manner. In the common distributed data/service model there exist several services and global information is distributed to them based on theme and/or location. Spatial information is not an exception; spatial data are distributed and managed by various services depending on the location and the service type. For example, there could be a specialized service for querying census data from Long Beach county and a separate service for querying tourist data from the same area. This decentralized model is cheap, efficient, easily maintainable, and avoids integration constraints (e.g., legacy constraints).

In many practical cases, complex queries need to combine information from multiple sources. As an example, consider the spatial join between two datasets which are hosted on two different servers. Server A hosts a database with information about hotels and other tourist services. Server B is a GIS server, providing information about the physical layers in a region (e.g., rivers, urban areas, forests, etc.). The user is a visitor and a nature lover who wants to find hotels which are close to forests. The query can be formed as a *distance join*: "find all hotels which are at most 2 km from a forest".

Since information about hotels and forests are provided by different services, 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. Mediators can use statistics from the sources to optimize the queries. However, there are several reasons why this architecture may not be appropriate or 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 requests may be ad-hoc and not supported by existing mediators. Third, the user may not be interested in using the service by the mediator, if she has to pay for this; retrieving the information directly from the sources may be less expensive.

Thus, we assume that the query should be evaluated at the client's side, on the mobile device. In this communication model, the user is typically charged by the bulk of transferred data (e.g., transferred bytes/packets), rather than by the time she stays connected to the service. We are therefore interested in minimizing the downloaded information from the services, instead of the processing cost at the servers. Another (realistic) assumption is that the services are not willing to share their statistics or indexes with the user. Therefore, information can only be downloaded by means of queries (which are supported by the service), like window queries, for instance.

In this paper, we describe MobiHook[1], a framework for optimizing complex distributed spatial operations on mobile devices such as wireless PDAs. As a special case, we consider the evaluation of spatial joins [3, 7], where the joined information is retrieved by two different services. We provide an evaluation algorithm that aims to minimize the transferred information instead of the processing time of the queries. Indeed, the user is typically willing to sacrifice a few seconds in order to minimize the query cost in dollars.

For efficient processing, we propose a query evaluation paradigm, which *adapts* to the data characteristics. First, the device downloads some statistical information, which is dynamically computed at the servers by submitting aggregate queries to them. With the help of these summaries, the mobile client is then able to minimize the data that have to be downloaded in order to process the query. For the distance join example we first retrieve some statistics which describe the distribution of data in each dataset. Based on these, we can avoid downloading information which cannot possibly participate in the result. For

[1] MobiHook is an acronym for MOBIle, ad-HOk hoOKing on distributed services.

instance, by applying a cheap (in terms of transferred data) query to the forests server, we can conclude that in some (e.g., urban or desert) areas there are no forests. This information will later help us avoid downloading any hotels in these areas.

Every partition of the data space is examined independently and the query optimizer decides the physical operator that will be applied. Therefore, depending on the retrieved statistics, different fragments can be processed by different physical operators (*adaptivity*). The optimizer may also choose to *recursively* obtain more statistics for some partitions, if the overhead is justified, based on a detailed cost model. Retrieving and processing summaries prior to actual data has the additional advantage of facilitating interactive query processing. By processing summary information we are able to provide estimates about the query result and the cost of transferring the actual data in order to process it. The user may then choose to restrict the query to specific regions, or tighten the constraints in order to retrieve more useful results.

The rest of the paper is organized as follows. Section 2 defines the problem formally, describes types of spatial queries that fit in the framework, and discusses related work. Section 3 presents the spatial join algorithm and the cost model which is used by the query optimizer. The proposed techniques are experimentally evaluated in Section 4. Finally, Section 5 concludes the paper with a discussion about future work.

2 Problem definition

Let q be a spatial query issued at a mobile device (e.g., PDA), which combines information from two spatial relations R and S, located at different servers. Let b_R and b_S be the cost per transferred unit (e.g., byte, packet) from the server of R and S, respectively. We want to minimize the cost of the query with respect to b_R and b_S. Here, we will focus on queries which involve two spatial datasets, although in a more general version the number of relations could be larger.

The most general query type that conforms to these specifications is the *spatial join*, which combines information from two datasets according to a spatial predicate. Formally, given two spatial datasets R and S and a spatial predicate θ, the spatial join $R \bowtie_\theta S$ retrieves the pairs of objects $\langle o_R, o_S \rangle$, $o_R \in R$, and $o_S \in S$, such that $o_R \; \theta \; o_S$. The most common join predicate for objects with spatial extent is *intersects* [3].

Another popular spatial join operator is the *distance* join [9, 7, 16]. In this case the object pairs $\langle o_R, o_S \rangle$ that qualify the query should be within distance ε. The Euclidean distance is typically used as a metric. Variations of this query are the *closest pairs query* [4], which retrieves the k object pairs with the minimum distance, and the *all nearest neighbor* query [19], which retrieves for each object in R its nearest neighbor in S.

In this paper we deal with the efficient processing of intersection and distance joins under the transfer cost model described above. Previous work (e.g., [3, 7]) has mainly focused on processing the join using hierarchical indexes (e.g., R–

trees [6]). Since access methods cannot be used to accelerate processing in our setting, we consider hash-based techniques [14].

Although the distance join is intuitively a useful operator for a mobile user, its result could potentially be too large. Large results are usually less interpretable/useful to the user and they are potentially more expensive to derive. She might therefore be interested in processing queries of high selectivity with potentially more useful results. A spatial join query of this type is the *iceberg distance semi-join* operator. 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". In pseudo-SQL the query could be expressed as follows:

```
SELECT H.id
FROM Hotels H, Restaurants R
WHERE dist(H.location,R.location)≤ ε
GROUP BY H.id
HAVING COUNT(*)≥ m ;
```

2.1 Related Work

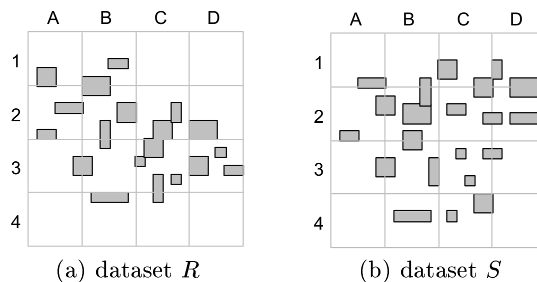
There are several spatial join algorithms that apply to centralized spatial databases. Most of them focus on the *filter* step of the spatial *intersection* join. Their aim is to find all pairs of object MBRs (i.e., *minimum bounding rectangles*) that intersect. The qualifying candidate object pairs are then tested on their exact geometry at the final *refinement* step. Although these methods were originally proposed for intersection joins, they can be easily adapted to process distance joins, by extending the objects from both relations by $\varepsilon/2$ on each axis [9, 12].

The most influential spatial join algorithm [3] presumes that the datasets are indexed by hierarchical access methods (i.e., R–trees). Starting from the roots, the trees are synchronously traversed, following entry pairs that intersect. When the leaves are reached, intersecting object MBRs are output. This algorithm is not directly related to our problem, since server indexes cannot be utilized, or built on the remote client. Another class of spatial join algorithms applies on cases where only one dataset is indexed [13]. The existing index is used to guide hashing of the non-indexed dataset. Again, such methods cannot be used for our settings.

On the other hand, spatial join algorithms that apply on non-indexed data could be utilized by the mobile client to join information from the servers. The Partition Based Spatial Merge (PBSM) join [14] uses a regular grid to hash both datasets R and S into a number of 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 to 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. To avoid ending up with partitions with significant differences in size, in

case the datasets are skewed, a tiling scheme paired with a hash function is used to assign multiple cells to the same hash bucket.

Figure 1 illustrates an example of two hashed datasets. Notice that MBRs that span grid lines are hashed to multiple cells (i.e., the cells that they intersect). The side-effect is that the size of hashed information is larger than the original datasets. Moreover, a duplicate removal technique is required in order to avoid reporting the same pair of objects twice, if they happen to be hashed to more than one common buckets. For instance, the objects than span the border of cells B2 and B3 could be reported twice, if no duplicate removal is applied. Techniques that avoid redundancy in spatial joins are discussed in [5, 12]. Finally, PBSM is easily parallelizable; a non-blocking, parallel version of this algorithm is presented in [12]. The data declustering nature of PBSM makes it attractive for use for the problem studied in this paper. Details are discussed in Section 3. Alternative methods for joining non-indexed datasets were proposed in [11, 2].



(a) dataset R (b) dataset S

Fig. 1. Two datasets to be joined

The problem of evaluating nearest neighbor queries on remote spatial databases is studied in [10]. The server is assumed to evaluate only window queries, thus the client has to estimate the minimum window that contains the query result. The authors propose a methodology that estimates this window progressively or approximates it using statistics from the data. However, they assume that the statistics are available at the client's side. In our work, we deal with the more complex problem of spatial joins from different sources, and we do not presume any statistical information at the mobile client. Instead, we generate statistics by sending aggregate queries, as explained in Section 3.

Distributed processing of spatial joins has been studied in [17]. Datasets are indexed by R–trees, and the intermediate levels of the indices are transferred from the 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 here 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] system tracks statistics from previous calls to the sources and uses them to optimize the execution of a new query. This method is not applicable in our case, since we assume that the connections are ad-hoc and the queries are unlikely to share modules with previous retrievals from the services. DISCO [18], on the other hand, retrieves cost information from wrappers during the initialization process. This information is in the form of logical rules which encode classical cost model equations. Garlic [15] also obtains cost information from the wrappers during the registration phase. In contrast to DISCO, Garlic poses simple aggregate queries to the sources in order to retrieve the statistics. Our statistics retrieval method is closer to Garlic. Nevertheless, both DISCO and Garlic acquire cost information during initialization and use it to optimize all subsequent queries, while we optimize the entire process of statistics retrieval and query execution for a single query. The Tuckila [8] system also combines optimization with query execution. It first creates a temporary execution plan and executes only parts of it. Then, it uses the statistics of the intermediate results to compute better cost estimations, and refines the rest of the plan. 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 a Mobile Device

As discussed already, we cannot use potential indexes on the servers to evaluate spatial join queries. On the other hand, it is a realistic assumption that the hosts can evaluate simple queries, like spatial selections. In addition, we assume that they can provide results to simple *aggregate* queries, like for example "find the number of hotels that are included in a spatial window". Notice that this is not a strong assumption, since the results of a window query may be too many to be accommodated in the limited resources of the PDA. Therefore, it is typical for the mobile client to first send an acknowledgment for the *size* of the query result, before retrieving it.

Since the price to pay here is the communication cost, it is crucial to minimize the information transferred between the PDA and the servers during the join; the time length of connections between the PDA and the servers is free in typical services (e.g., mobile phones), which charge users based on the traffic. There are two types of information interchanged between the client and the server application: (i) the queries sent to the server and (ii) the results sent back by the server. The main issue is to minimize this information for a given problem.

The simplest way to perform the spatial join is to download both datasets to the client and perform the join there. We consider this as an infeasible solution in general, since mobile devices are usually lightweight, with limited memory

and processing capabilities. First, the relations may not fit in the device which makes join processing infeasible. Second, the processing cost and the energy consumption on the device could be high. Therefore we have to consider alternative techniques.

3.1 A divisive approach

A divide-and-conquer solution is to perform the join in one spatial region at a time. Thus, the dataspace is divided into rectangular areas (using, e.g. 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 (e.g., plane sweep [2]). Like PBSM [14], a hash-function can be used to bring multiple tiles at a time and break the result size more evenly. However, this would require multiple queries to the servers for each partition. The duplicate avoidance techniques of [5, 12] can also be employed here to avoid reporting a pair more than once.

As an example of an intersection join, consider the datasets R and S of Figure 1 and the imaginary grid superimposed over them. The join algorithm applies a window query for each cell to the two servers and joins the results. For example, the hotels that intersect A1 are downloaded from R, the forests that intersect A1 are downloaded from S, and these two window query results are joined on the PDA. In the case of a distance join, the cells are extended by $\varepsilon/2$ at each side before they are sent as window queries. A problem with this method is that the retrieved data from each window query may not fit in memory. In order to tackle this, we can send a memory limit constraint to the server together with the window query and receive either the data, or a message alarming the potential memory overflow. In the second case, the cell can be recursively partitioned to a set of smaller window queries, similar to the recursion performed by PBSM [14].

3.2 Using summaries to reduce the transfer cost

The partition-based technique is sufficiently good for joins in centralized systems, however, it requires that all data from both relations are read. When the distributions in the joined datasets vary significantly, there may be large empty regions in one which are densely populated in the other. In such cases, the simple partitioning technique potentially downloads data that do not participate in the join result. We would like to achieve a sublinear transfer cost for our method, by avoiding downloading such information. For example, if some hotels are located in urban or coastal regions, we may avoid downloading them from the server, if we know that there are no forests close to this region with which the hotels could join. Thus, it would be wise to retrieve a distribution of the objects in both relations before we perform the join. In the example of Figure 1, if we know that cells C1 and D1 are empty in R, we can avoid downloading their contents from S.

The intuition behind our join algorithm is to apply some cheap queries first, which will provide information about the distribution of objects in both datasets. For this, we pose aggregate queries on the regions before retrieving the results

from them. Since the cost on the server side is not a concern[2], we first apply a COUNT query for the current cell on each server, before we download the information from it. The code in pseudo-SQL for a specific window w (e.g., a cell) is as follows (assuming an *intersection* join):

```
Send to server H:
SELECT COUNT(*) as c1
FROM Hotels H
WHERE H.area INTERSECTS w
If (c1>0) then
Send to server F:
SELECT COUNT(*) as c2
FROM Forests F
WHERE F.area INTERSECTS w
If (c2>0) then
SELECT * FROM
(SELECT * FROM Hotels H AS HW WHERE H.area INTERSECTS w),
(SELECT * FROM Forests F AS FW WHERE F.area INTERSECTS w)
WHERE HW.area INTERSECTS FW.area
```

Naturally, this implementation avoids loading data in areas where some of the relations are empty. For example, if there is a window w where the number of forests is 0, we need not download hotels that fall inside this window. The problem that remains now is to set the grid granularity so that (i) the downloaded data from both relations fit into the PDA, so that the join can be processed efficiently, (ii) the empty area detected is maximized, (iii) the number of queries (messages) sent to the servers is small, and (iv) data replication is avoided as much as possible.

Task (i) is hard, if we have no idea about the distribution of the data. Luckily, the first (aggregate) queries can help us refine the grid. For instance, if the sites report that the number of hotels and forests in a cell are so many that they will not fit in memory when downloaded, the cell is recursively partitioned. Task (ii) is in conflict with (iii) and (iv). The more the grid is refined, the more dead space is detected. On the other hand, if the grid becomes too fine, many queries will have to be transmitted (one for each cell) and the number of replicated objects will be large. Therefore, tuning the grid without apriori knowledge about the data distribution is a hard problem.

To avoid this problem, we refine the grid recursively, as follows. The granularity of the first grid is set to 2×2. If a quadrant is very sparse, we may choose not to refine it, but download the data from both servers and join them on the PDA. If it is dense, we choose to refine it because (a) the data there may not fit in our memory, and (b) even when they fit, the join would be expensive. In the example of Figure 1, we may choose to refine quadrant AB12, since the

[2] In fact, this query may not be expensive, if the server maintains precomputed aggregates or employs aggregate spatial indexes.

aggregate query indicates that this region is dense (for both R and S in this case), and avoid refining quadrant AB34, since this is sparse in both relations.

3.3 Handling bucket skew

In some cells, the density of the two datasets may be very different. In this case, there is a high chance of finding dead space in one of the quadrants in the sparse relation, where the other relation is dense. Thus, if we recursively divide the space there, we may avoid loading unnecessary information from the dense dataset. In the example of Figure 1, quadrant CD12 is sparse for R and dense for S; if we refined it, we would be able to prune cells C1 and D1.

On the other hand, observe that refining such partitions may have a counter-effect in the overall cost. By applying additional queries to very sparse regions we increase the traffic cost by sending extra window queries with only a few results. For example, if we find some cells where there is a large number of hotels but only a few forests, it might be expensive to draw further statistics from the hotels database, and at the same time we might not want to download all hotels. For this case, it might be more beneficial to stop drawing statistics for this area and *perform the join as a series of selection queries*, one for each forest. Recall that a potential (nested-loops) technique for $R \bowtie S$ is to apply a selection to S for each $r \in R$. This method can be fast if $|R| << |S|$. Thus, the join processing for quadrant CD12 proceeds as follows (a) download all forests intersecting CD12, (b) for each forest apply a window query on the hotels. This method will yield a lot of savings if the hotels from that cell that participate in the join are only a few.

The point to switch from summary retrieval to window queries depends on the cost parameters and the size of the smallest partition (e.g., forests). In the next section, we provide a methodology that recursively partitions the dataspace using statistical information terminating at regions where it is more beneficial to download the data from both sites and perform the join on the PDA or download the objects from one server only and process the join by sending them as queries to the other.

3.4 A recursive, adaptive spatial join algorithm

By putting everything together, we can now define the proposed algorithm for spatial joins on a mobile device. We assume that the servers support the following queries:

- 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.

The *MobiJoin* algorithm is based on the divisive approach and it is *recursive*; given a rectangular area w of the data space (which is initially the MBR of the joined datasets) and the cardinalities of R and S in this area, it may choose

to perform the join for this area, or recursively partition the data space to smaller windows, collect finer statistics for them, and postpone join processing. Therefore, the algorithm is *adaptive* to data skew, since it may follow a different policy depending on the density of the data in the area which is currently joined.

Initially, the algorithm is called for datasets R and S, considering as w the intersection of their MBRs. For this, we have to apply two queries to each server. The first is an aggregate query (easily expressed in SQL) asking for the maximum and minimum coordinates of the objects, which define the MBR of the dataset. The intersection w of the MBRs of R and S defines the space the joined results should intersect. If the distribution of the datasets is very different, w can be smaller than the map window that encloses them and many objects can be immediately pruned. The second query retrieves the number of objects from each dataset intersecting w (i.e., a COUNT query).

Let $R_w.count$ and $S_w.count$ be the number of objects from R and S, respectively, intersected by w. The recursive MobiJoin algorithm is shown in Figure 2. If one of the $R_w.count$ and $S_w.count$ is 0, the algorithm returns without elaborating further on the data. Else, the algorithm employs a cost model to estimate the cost for each of the potential actions in the current region w: (1) download the objects that intersect w from both datasets and perform the join on the PDA, (2) download the objects from R that intersect w and *send them* as *selection queries* to S, (3) download the objects from S that intersect w and *send them* as *selection queries* to R, and (4) divide w into smaller regions $w' \in w$, retrieve refined statistics for them, and apply the algorithm recursively there.

Action (1) may be constrained by the resource constraints on the PDA. Actions (2) and (3) may have different costs depending on which of the $R_w.count$ and $S_w.count$ is the smallest and the communication costs with each of the sites. Finally, the cost of action (4) is the hardest to estimate; for this we use probabilistic assumptions for the data distribution in the refined partitions, as explained in the next section.

3.5 The cost model

In this section, we describe a cost model that can be used in combination with MobiJoin to facilitate the adaptivity of the algorithm. We provide formulae, which estimate the cost of each of the four potential actions that the algorithm may choose. Our formulae are parametric to the characteristics of the network connection to the mobile client. For simplicity, we consider distance joins between point sets instead of intersection joins. However, the formulae can be easily adapted for intersection joins.

The largest amount of data that can be transferred in one physical frame on the network is referred to as MTU (Maximum Transmission Unit). The size of the MTU depends on the specific protocol; Ethernet, for instance, has $MTU = 1500$ bytes, while dial-up connections usually support $MTU = 576$ bytes. Each transmission unit consists of a header and the actual data. The largest segment of TCP data that can be transmitted is called MSS (Maximum

```
// R and S are spatial relations located at different servers
// w is a window region
// R_w.count (resp. S_w.count) is the number
// of objects from R (resp. S), which intersect w
MobiJoin(R,S,w,R_w.count,S_w.count)
1.   if R_w.count = 0 or S_w.count = 0 then terminate;
2.   c_1(w) = cost of downloading R_w.count objects from R
            and S_w.count objects from S and joining them on the PDA;
3.   c_2(w) = cost of downloading R_w.count objects from R, send them
            as window queries to server that hosts S and receive the results;
4.   c_3(w) = cost of downloading S_w.count objects from S, send them
            as window queries to server that hosts R and receive the results;
5.   c_4(w) = cost of applying recursive counting in R_w.count and S_w.count,
            retrieve more detailed statistics, and apply MobiJoin recursively;
6.   c_min = min{c_1(w), c_2(w), c_3(w), c_4(w)};
7.   if c_min = c_4 then
8.       impose a regular grid over w;
9.       for each cell w' ∈ w
10.          retrieve R_w'.count and S_w'.count;
11.          MobiJoin(R,S,w',R_w'.count,S_w'.count);
12.  else follow action specified by c_min;
```

Fig. 2. The recursive MobiJoin algorithm

Segment Size). Essentially, $MTU = MSS + B_H$, where B_H is the size of the TCP/IP headers (typically, $B_H = 40$ bytes).

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. For point objects $B_{obj} = 4 + 2 \cdot 4 = 12$ bytes (i.e., point ID plus its coordinates). Thus, when the whole D is transmitted through the network, the number of transferred bytes is:

$$T_B(B_D) = B_D + B_H \cdot \left\lceil \frac{B_D}{MSS} \right\rceil , \tag{1}$$

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

The cost of sending a window query q_w to a server is $B_H + B_{qtype} + B_w$, i.e., transferring the query type B_{qtype} and the coordinates of the window B_w. Let $R_w.count$ and $S_w.count$ be the number of objects intersecting window w at site R and S respectively. 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:

$$c_1(w) = (b_R + b_S)(B_H + B_{qtype} + B_w) + b_R T_B(R_w.count \cdot B_{obj}) + b_S T_B(S_w.count \cdot B_{obj}) \tag{2}$$

Now let us consider the cost c_2 of downloading all $R_w.count$ objects from R and sending them as *distance selection* queries to S. Each of the $R_w.count$

objects is transformed to a *selection region* and sent to S. For each query point p, the expected number of point from S in w within distance ε from p is $\frac{\pi \cdot \varepsilon^2}{w_x \cdot w_y} \cdot S_w.count$, assuming uniform distribution in w, where w_x and w_y are the lengths of the window's sides.[3] In other words, the *selectivity* of a *selection* query q over a region w with uniformly distributed points is probabilistically defined by $area(q)/area(w)$. The query message consists of the query type, p and ε (i.e., $S_q = S_{qtype} + S_p + S_\varepsilon$).Therefore, the cost of sending the query is $B_H + S_q$. The total number of transferred bytes for transmitting the distance query and receiving the results is:

$$T_B(w, \varepsilon) = (B_H + S_q) + T_B\left(\frac{\pi \cdot \varepsilon^2}{w_x \cdot w_y} \cdot S_w.count \cdot B_{obj}\right) \quad (3)$$

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

$$c_2(w) = b_R(B_H + B_{qtype} + B_w) + b_R T_B(R_w.count \cdot B_{obj}) + b_S R_w.count \cdot T_B(w, \varepsilon) \quad (4)$$

The cost c_3 of downloading the objects from S and sending them as queries to R is also given by Equation 4 by exchanging the roles of R and S. Finally, in case of an intersection join, we can use the same derivation, but we need to know 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 $R_w.count$ and $S_w.count$ (i.e., we can post an additional aggregate query together with the COUNT query).

The final step is to estimate the cost c_4 of repartitioning w and applying MobiJoin recursively for each new partition w'. In order to retrieve the statistics for a new partition w', we have to send an aggregate COUNT query to each site ($B_H + B_{qtype} + B_w$ bytes) and retrieve ($B_H + 4$ bytes) from there. Thus, the total cost of repartitioning and retrieving the refined counters is:

$$c_{CQ(w)} = N_p(b_R + b_S)(B_H + B_{qtype} + B_w + B_H + 4), \quad (5)$$

where N_p is the number of new partitions over w. Now we can define:

$$c_4(w) = c_{CQ(w)} + \sum_{\forall w'} \min\{c_1(w'), c_2(w'), c_3(w'), c_4(w')\} \quad (6)$$

Without prior knowledge about how the objects are distributed in the new partitions, it is impossible to predict the actions of the algorithm when run at the next level for each w'. The minimum value for $c_4(w)$ is just $c_{CQ(w)}$, i.e., the cost of refining the statistics, assuming that the condition of line 1 in Figure 2 will hold (i.e., one of the $R_{w'}.count$, $S_{w'}.count$ will be 0 for all w'). This will happen if the data distribution in the two datasets is very different. On the other hand, $c_4(w)$ is maximized if the distribution is uniform in w for both R and S. In this case, $c_4(w) = c_{CQ(w)} + \sum_{\forall w'} \min\{c_1(w'), c_4(w')\}$, i.e., case 1 will apply

[3] For the sake of readability, we ignore boundary effects here.

for all w', unless the data for each w' does not fit in the PDA, thus the algorithm will have to be recursively employed.

In practice, we expect some skew in the data, thus some partitions will be pruned or converted to one of the cases 2 and 3. For the application of our model, we consider $c_4(w) = c_{CQ(w)}$, i.e., an *optimistic* approach that postpones join processing, as long as refining the statistics is cheaper. While this estimation was effective for most of the tested cases, we are currently working on a more accurate model.

3.6 Iceberg spatial distance semi-joins

Our framework is especially useful for iceberg join queries. As an example, consider the query "find all hotels which are close to at least 20 restaurants", where closeness is defined by a distance threshold ε. Such queries usually have few results, however, when processed in a straightforward way they could be as expensive as a simple spatial join. The grid refinement method can be directly used to prune areas where the restaurant relation is sparse. In this case, the condition $R_w.count = 0$ in line 1 of Figure 2 will become $R_w.count < k_{min}$, where k_{min} is the minimum number of objects from R that must join with an object in S. Therefore, large parts of the search space could potentially be pruned by the cardinality constraint early. In the next section, we show that this method can boost performance by more than one order of magnitude, for moderate values of k_{min}.

4 Experimental evaluation

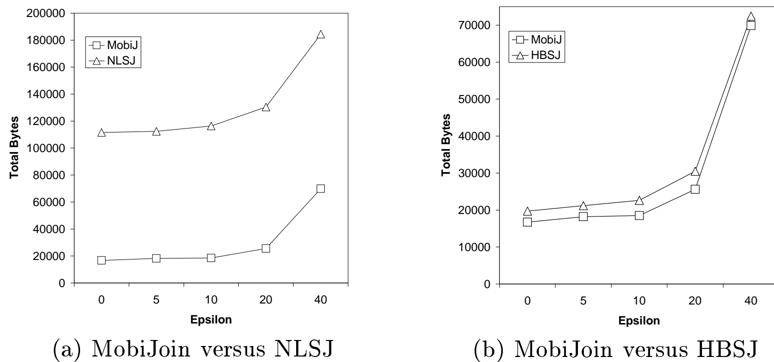
In this section, we study the performance of the MobiJoin algorithm described in Figure 2 and compare it against two simpler techniques. The fist one, called *Nested Loop Spatial Join* (NLSJ), is a naïve algorithm which resembles cases 2 and 3 of MobiJoin. The PDA receives in a stream all the objects from server R and sends them one by one as distance queries to server S, which returns the results. For fairness, we first check the number of objects in each server, and set R to be the one with the smallest dataset. The second algorithm is *Hash Based Spatial Join* (HBSJ), and consists of cases 1 and 4 of MobiJoin. The PDA retrieves statistics from the servers and recursively decomposes the data space, until the partitions of the two datasets fit in memory. The corresponding fragments are downloaded and the join is performed in the PDA.

We implemented our algorithms 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 a wireless interface. The servers for the spatial datasets resided on a 2-CPU Ultra-SPARC III machine with 4GB RAM. In all experiments, we set $b_R = b_S$, i.e., the transfer cost is the same for both servers. We used synthetic datasets consisting of 1000 to 10000 points, in order to simulate typical windows of users' requests. The points were clustered around n randomly selected centers, and each cluster was following a

Gaussian distribution. To achieve different levels of skew, we varied n from 1 to 128. We also employed two real-life datasets, namely the road and railway segments of Germany. Each of these sets had around 35K points.

In the first set of experiments, we compare the three algorithms in terms of the total number of bytes sent and received, including the overhead due to the TCP/IP headers, as a function of the joining distance ε. We used two datasets if 1000 points and set the PDA's memory size to 100 points. In Figure 3 we present the average values of the results over 10 runs with different datasets. Figure 3a compares MobiJoin with NLSJ. It is obvious that MobiJoin easily outperforms the naïve NLSJ by almost an order of magnitude. This is due to two reasons. First, NLSJ transmits a huge number of queries to the largest dataset, which have a high cost (including the cost of the packet headers for each query). Second, MobiJoin avoids downloading any data from regions where at least one dataset is empty. Clearly, performing the join on the mobile device comes with significant cost savings. The results also demonstrate that the overhead of identifying such regions (i.e., the additional aggregate queries) is justified.

Figure 3b compares MobiJoin with HBSJ. MobiJoin is better than HBSJ, but the difference is small (i.e., 22% in the best case). The performance gain is due to cases 2 and 3 of the algorithm, which apply Nested Loop Join for some partitions. If a space partition w contains many points in (say) S but only a few points in R, MobiJoin executes Nested Loop Join in w, which is cheaper than hashing (i.e., downloading both fragments). Note that as ε increases, the cost of each algorithm increases, but for different reasons. In the case of NLSJ it is due to the larger number of solutions that must be transferred. For HBSJ, on the other hand, it is due to the enlargement of the query window by $2 \cdot \varepsilon$ at each side, while for MobiJoin it is a combination of the two reasons.



(a) MobiJoin versus NLSJ

(b) MobiJoin versus HBSJ

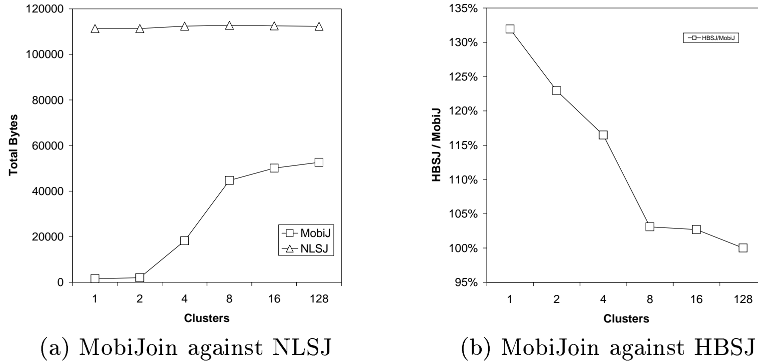
Fig. 3. Transferred bytes as a function of the distance threshold ε

The distribution of the data is crucial for the performance of our algorithms. Intuitively, MobiJoin performs very well on skewed data. If, for example, all data of R are near the lower left corner and all data of S are near the upper right

corner, then the algorithm will terminate after one step, since no combination of quadrants produces any result. On the other hand, if the datasets are uniform there is a lower probability of successful pruning; therefore MobiJoin reduces to HBSJ. In the following experiment, we test the algorithm under varying data skew. We employ a 1000 points dataset and vary the number of clusters from 1 to 128. Fewer clusters result in more skewed data and vice-versa. The comparison between our algorithm and NLSJ is shown in Figure 4a.

For very skewed data, MobiJoin can be as much as two orders of magnitude better than NLSJ. When the data are more uniform, however, the performance of MobiJoin drops, while NLSJ is stable.[4] Still, MobiJoin is two times better in the worst case. This is due to the fact that MobiJoin behaves like hash join when data are uniformly distributed. In such case, it only has to transfer $|R|+|S|$ objects in bulk to the PDA, while NLSJ transfers $|R|$ objects in bulk, then sends $|R|$ objects one by one to server S, and finally receives the results.

The fact that MobiJoin reduces to hash join is further investigated in Figure 4b. There, we present the total number of bytes transferred by HBSJ over the bytes transferred by MobiJoin. When data is skewed, HBSJ is 32% worse, since MobiJoin joins some of the fragments using the nested loops approach. For uniform data, however, the two algorithms are identical.



(a) MobiJoin against NLSJ (b) MobiJoin against HBSJ

Fig. 4. Total number of bytes transferred vs. the number of clusters in the dataset

We also investigated the effect of varying the buffer size which is available in the PDA and present the results in Figure 5. The size of the buffer is presented as a percentage of the total size of the two datasets and varies from 0.3% to 4%. Each dataset contains 1000 points in 4 clusters. In Figure 5a we compare MobiJoin with HBSJ in terms of the total packets transferred. As expected, if more memory is available in the PDA, less packets are transferred. This is especially true for HBSJ; since larger windows can be joined in the PDA, more data are transferred in bulk. MobiJoin, on the other hand, is not affected much, since some fragments are joined by the nested loop approach which transfers

[4] The small fluctuation of NLSJ is due to the different number of solutions

many small packets. In Figure 5b, we present the total number of transferred data for the same settings. Observe that MobiJoin transfers less data in total. Since most services charge the user by the amount of transferred data, the behavior of MobiJoin is preferable.

Notice also the strange trend of the cost in terms of bytes for HBSJ. When the size of the buffer grows from 0.3% until 2%, the cost drops, as expected. However, for larger buffer sizes the cost increases again. This can explained by the stopping condition of the recursive algorithm; data are partitioned recursively until the partitions fit in memory. If the buffer is small, many queries are transmitted and the cost is high. If the buffer is large, on the other hand, the recursion stops early and HBSJ fails to prune sub-partitions that are empty in either of the datasets. There is a range of memory sizes, where a good trade-off between these two cases is achieved.

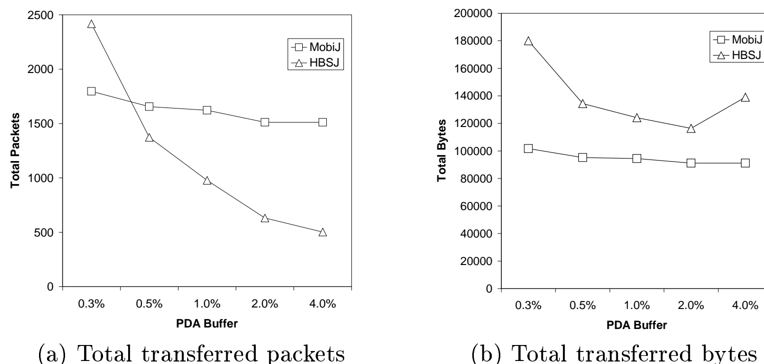
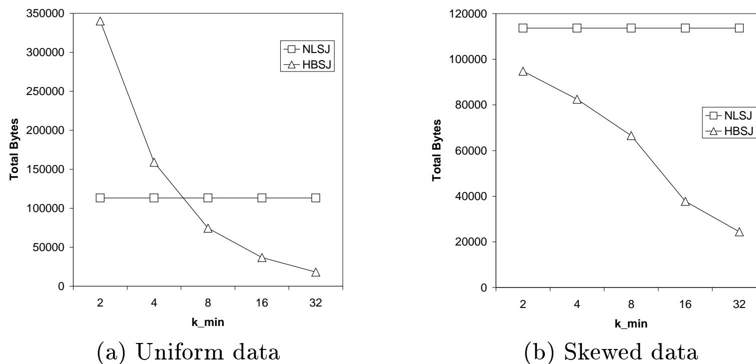


Fig. 5. Transferred data vs. the PDA's memory size (% of the total size of the datasets)

In the final set of experiments, we tested our algorithms for the case of iceberg queries. We employed the Germany Roads and Railways datasets, and we joined them with synthetic datasets of 1000 points, which represent locations of interest (e.g., hotels). We varied the minimum support threshold k_{min} from 2 to 32. Since we wanted to test the efficiency of pruning due to space partitioning, we compared the HBSJ algorithm against NLSJ. Notice that the settings of the experiment are favorable for nested loop join, since we join a small dataset with a large one. The results are presented in Figures 6a and 6b. In the first figure, the hotels dataset is uniform. This is the worst case for our algorithm, which is 3 times worse than NLSJ, for small values of k_{min}. However, when k_{min} increases, more partitions are pruned, and HBSJ can be as much as one order of magnitude better than NLSJ. Observe that the cost of NLSJ is stable, since it must always transfer the data from the smaller set, regardless of the value of the support threshold k_{min}. Figure 6b presents the results in the case where the hotels dataset is skewed. NLSJ does not change, since its performance depends mostly on the size of the hotels dataset, and not on the distribution. HBSJ,

on the other hand, takes advantage of the skewed distribution and performs better that NLSJ in all cases. In summary, our technique of obtaining statistics information, decreases considerably the cost when k_{min} is sufficiently large and is especially suitable for skewed data.



(a) Uniform data

(b) Skewed data

Fig. 6. Iceberg queries for real datasets. Transferred data vs. the threshold k_{min}

5 Conclusions

In this paper, we dealt with the problem of evaluating spatial joins on a mobile device, when the datasets reside on separate remote servers. We assume that the servers support three simple query types: (i) *window* queries, (ii) *aggregate* queries, and (iii) *distance-range* queries. We also assume that the servers are non-collaborative, they do not wish to share their internal indices, and no mediator can perform the join of these two sites. These assumptions are valid for many practical situations, where users apply ad-hoc joins. For instance, consider two services which provide maps and hotel locations, and a user who requests an unusual combination like "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, instead of the processing cost on the servers.

We developed MobiJoin, an algorithm that partitions recursively the data space and retrieves statistics in the form of simple aggregate queries. Based on the statistics and a detailed cost model, MobiJoin can either (i) prune a partition, (ii) join its contents in hash join or nested loop fashion, or (iii) request further statistics. In contrast to the previous work on mediators, our algorithm dynamically optimizes the entire process of retrieving statistics and executing the join, for a single ad-hoc query.

We developed a prototype on a wireless PDA and tested our method for a variety of synthetic and real datasets. Our experiments reveal that MobiJoin outperforms the naïve approach by an order of magnitude. Our partitioning method is also suitable for iceberg queries, especially for skewed data.

In the future, we plan to support complex spatial queries, which involve more than two datasets. We are also working on refining our cost model, since accurate estimations are crucial for selecting the most beneficial execution plan.

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 Int'l Conference*, 1996.
2. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. of VLDB Conference*, 1998.
3. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *Proc. of ACM SIGMOD Int'l Conference*, 1993.
4. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proc. of ACM SIGMOD Int'l Conference*, 2000.
5. J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 2000.
6. A. Guttman. R-trees: a dynamical index structure for spatial searching. In *Proc. of ACM SIGMOD Int'l Conference*, 1984.
7. G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. of ACM SIGMOD Int'l Conference*, 1998.
8. 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 Int'l Conference*, 1999.
9. N. Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 1998.
10. D. Liu, E.-P. Lim, and W. K. Ng. Efficient k nearest neighbor queries on remote spatial databases using range estimation. In *Proc of Int'l Conference on Scientific and Statistical Database Management (SSDBM)*, 2002.
11. M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. of ACM SIGMOD Int'l Conference*, 1996.
12. G. Luo, J. F. Naughton, and C. Ellmann. A non-blocking parallel spatial join algorithm. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 2002.
13. N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *Proc. of ACM SIGMOD Int'l Conference*, 1999.
14. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. of ACM SIGMOD Int'l Conference*, 1996.
15. 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 Conference*, 1999.
16. H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *Proc. of ACM SIGMOD Int'l Conference*, 2000.
17. K.-L. Tan, B.-C. Ooi, and D. J. Abel. Exploiting spatial indexes for semijoin-based join processing in distributed spatial databases. *IEEE Trans. on Data and Knowledge Engineering*, 12(2):920–937, 2000.
18. A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with disco. *IEEE Trans. on Data and Knowledge Engineering*, 10(5):808–823, 1998.
19. J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. Technical Report CS07-02, HKUST, Hong Kong, 2002.