Graph Partition Based Multi-Way Spatial Joins

Xuemin Lin, Hai-Xin Lu, Qing Zhang
School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
lxue, cshlu, qzhang@cse.unsw.edu.au

Abstract

In this paper, we investigate the problem of efficiently computing a multi-way spatial join without spatial indexes. We propose a novel and effective filtering algorithm based on a two phase partitioning technique. To avoid missing hits due to an inherent difficulty in multi-way spatial joins, we propose to firstly partition a join graph into sub-graphs whenever necessary. In the second phase, we partition the spatial data sets; and then the sub-joins will be executed simultaneously in each partition to minimise the I/O costs. Finally, a multi-way relational join will be applied to merge together the sub-join results. Our experiment results demonstrated the effectiveness and efficiency of the proposed algorithm.

Keywords*: Spatial Databases, Spatial Joins, Query Processing.*

1. Introduction

Multi-way spatial join selects, from n sets of spatial objects in the 2-dimensional space, the tuples each of which consists of n objects respectively from the n data sets and satisfies some given spatial predicates. An example of a multi-way spatial join is "find all residential areas that intersect a lake that passes some golf courses". Spatial join is an important but expensive query form to process spatial information in GIS, satellite images, digital video, multimedia documents, etc. A multi-way spatial join can be modelled by a *join graph* whose vertices represent respectively different data sets and edges represent respectively spatial predicates between pairs of vertices. We will formally define a spatial multi-way join in the next section.

Due to high processing complexities and costs, a spatial join is usually executed in two steps [2, 14], *filtering* and *refinement*. In the filtering step, spatial objects are approximated by the isothetic *Minimal Bounding Rectangles* (MBR); and then the MBRs are processed to produce a candidate result set. In the refinement step, each tuple from the candidate set is examined further, for the given spatial predicates, against the real objects instead of their MBRs. In this paper, we will concentrate on the filtering step. Therefore, the multi-way spatial joins discussed in this paper will be restricted to data sets where spatial objects are isothetic rectangles. A special predicate may be in various different forms; for instance, *direction* joins [22], *distance* joins [6], etc. In this paper, we will discuss only one of the most popular predicates - *overlapping*.

Spatial joins with the predicate of overlapping has received a great attention in the last twenty years. A number of different computation methods have been proposed for efficiently processing 2-way spatial joins, such as Z-order elements technique [13, 14], R-tree join [2, 5] and its variations, filter tree join [20], seeded tree join [9], partition based spatial merge join [15], spatial hash join [10], size separation spatial join [7], the scalable sweeping-based spatial join [1], and slot index spatial join [12]. However, there has been little research on the general multi-way (with more than 2 data sets) join until very recently. The papers [16, 17, 18] provided spatial join algorithms based on a synchronous traversal technique on R-trees, while the paper [12] proposed to apply a slot index spatial join method to handle a join of an intermediate join result and an indexed data set.

In this paper, we investigate the multi-way spatial join processing algorithms when no spatial data indexes exist. One way to process a multi-way join is by a sequence of 2-way joins; and then apply some non-index based spatial join algorithms [1, 10, 15, 7] to each two-way join. However, such an approach does not only have to involve extra I/O overheads to repartition every intermediate result set when it does not fit in the buffer but also has to involve some extra computation costs (e.g. if swapping-line algorithm is applied, data sets may have to be scanned more than once).

The experiment results in [1] suggest that a data partitioning based join approach [10, 15, 7] tends to have small

Proceedings of the International Database Engineering and Applications Symposium (IDEAS'02)
1098-8068/02 $17.00 © 2002 **IEEE**

I/O overhead. Consequently, an alternative way to process a multi-way spatial join is firstly to partition the data space into a number of *buckets* [15]; and secondly to process data only in each bucket to produce the local multi-way join results; and thirdly merge the local sub-join results together by a union, like what was suggested in [10, 15] for 2-way joins. Unfortunately, such a data partitioning based method may miss some join result tuples; and we will illustrate them in the next section.

Motivated by the above facts, in this paper we will propose a novel multi-way spatial join algorithm based on a two-phase partitioning technique in the filtering step (that is, join sets of rectangles):

- Firstly, we partition (if necessary) a join graph into a set of subgraphs.
- Secondly, we partition the involved data sets into a number of buckets.

Once the two-phase partitioning is done, we simultaneously execute the joins, which correspond respectively to a subgraph, in each bucket by reading in spatial data only once. After the local joins have done for each data bucket, a relational join will be applied on the sub-join results to "merge" them together to produce the join results for the original query graph. This is the first and the principal contribution of the paper. The second contribution of the paper are the results we obtained for the optimal graph partitioning problem. The third contribution of the paper is that we completely characterise a class of join graphs where a graph partitioning is not necessarily required while applying our join algorithm. Finally, our experiment results confirm the efficiency of our algorithm.

The rest of the paper is organised as follows. In section 2, we present a precise definition of multi-way join, an overview of related work, and the difficulties to process a spatial multi-way join without indexes. Section 3 presents the framework of our two-phase partitioning technique, the fundamental, and the join algorithm. Section 4 reports our experiment results. This is followed by conclusions and remarks.

2 Background

A multi-way spatial join can be defined as follows. Assume that $\{R_i : 1 \leq i \leq n\}$ is a set of n spatial relations, and $\{C_{i,j} : (i,j) \in I \ \& \ I \subseteq [1,n] \times [1,n]\}$ is a set of binary spatial predicates. The multi-way join of $\{R_1, ..., R_n\}$ with respect to the given spatial predicates is to compute all n-tuples:

$$\{(r_{1,x_1}, ..., r_{i,x_i}, ...r_{n,x_n}) : \forall i, r_{i,x_i} \in R_i; \forall (i,j) \in I, C(r_{i,x_i}, r_{j,x_j})\}$$

As mentioned in the last section, in this paper we discuss only one predicate - overlapping. We also assume that each spatial data set consists of only isothetic rectangles.

Note that a multi-way spatial join of n spatial tables $\{R_i : 1 \leq i \leq n\}$ is represented by a join graph $G = \{V, E\}$ where $V = \{R_i, 1 \leq i \leq n\}$ and corresponding to each $C_{i,j}$, (R_i, R_j) is an edge. In a graph, a vertex is *adjacent* to another vertex if there is an edge connecting them. The number of the edges *incident* to a vertex is called *degree* of the vertex. Suppose that V' is a subset of V. The *induced* graph by V' from G is the subgraph of G whose vertex set is V' and whose edge set consists of the edges in G connecting only the vertices in V'.

2.1 Related Work

The study of 2-way join has gained much attention in the last two decades. Many techniques have been published for the case when spatial data sets are fully indexed; for instance, those in [2, 5, 14, 20]. Due to an equal importance of the applications where no spatial index exists, the papers [10, 15, 1, 7] studied spatial two-way joins without spatial indexes.

Consider that the number of data objects to be joined may be too large to fit in the main memory simultaneously. The Partition Based Spatial Merge Join (PBSM) [15] divides the data set into a number of different regions; and each region is called a bucket. To avoid missing hits, PBSM suggests to duplicate each object to the buckets which intersect the object. Then, the join algorithm is locally executed in each bucket to produce the local join results. Adding up the local join results from different buckets gives the join result.

Independently, the authors in [10] proposed another data partition based join algorithm - Spatial Hash Join (SHJ). In SHJ, the author proposed a data partition such that each object from one data set is assigned to only one bucket which contains it; this will avoid a post-process to eliminates duplicates among the local join results. However, unlike [15] buckets in SHJ may overlap with each other. Note in SHJ, the data objects from another data set still need to duplicate to the intersected bucket.

The Sized Separation Spatial Join (S^3J) [7] is a non-trivial variation of the data partition based join methods [10, 15]; and it has gone one step further to completely remove data replication in each bucket by applying a multi-resolution idea. It involves a two-step data partitioning. The object space is firstly modelled into a multi-resolution space; and secondly the space for each resolution is partitioned by the Hilbert curve technique. Then the join is executed cross different levels of the multi-resolution space, such that 1) in the same level the join process needs to be carried out only within the same bucket, and 2) in the different levels the join process needs to be carried out only between a bucket from higher resolution level and the bucket in a lower level which contains the bucket in the higher

level.

The Scalable Sweeping-based Spatial Join (SSSJ) [1] is mainly focused on improving the computation in the main memory by a novel application of *interval* tree technique from computational geometry [19].

In this paper, to efficiently do I/O we will develop a data partition based algorithm for processing a multi-way spatial join. Below we first show the difficulties/problems in such a paradigm.

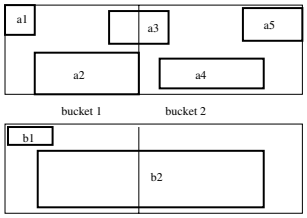

Figure 1. a3 and b2 are respectively replicated into two buckets

2.2 Problems

A *partition based join paradigm* can be generally described below. Firstly, we partition the whole data space into a number of buckets (usually isothetic rectangles). Secondly we assign each object to some (at least one) buckets which intersect the object. Thirdly, bucket by bucket we process the data in each bucket to produce *local* join results. Finally, all the local join results are *merged* together to produce the results of the join. Note that in two-way spatial join, such a merge process is trivial [10, 15] - a union of all the local join results.

The following problems need to be resolved in a partition based join paradigm while processing a multi-way spatial join. As pointed out in [10, 15], an object may have to be duplicated into all the buckets which intersect the object, in order to avoid missing hits. For example in Figure 1, $b2$ has to be duplicated into the two buckets to get join tuples $(a2, b2)$ and $(a4, b2)$. Data replication will cause an extra computation in two ways. Firstly, some pairs may be repeatedly examined in different buckets. For instance in Figure 1, if the objects $a3$ and $b2$ are respectively duplicated into the two buckets then the pair $(a3, b2)$ will be examined twice respectively in each bucket to see if they intersect with each other. Secondly, some joint result pairs may be repeatedly obtained in different buckets; for instance, the pair $(a3, b2)$ is obtained in both buckets. Consequently, it requires an extra sorting process to remove all duplicates [15]. To resolve this, the authors in [10] proposed a non-uniform space partition such that each object from one data set chooses only one bucket, which contains the object, to assign; while the objects from another data set are replicated cross the buckets which intersect the objects respectively. We will apply this idea in our partition based join algorithm for multi-way joins. It will be shown in the next section that to ensure the algorithm correctness, we will have to apply this idea in multi-way spatial joins.

In multi-way spatial join, the "sparsity" of a join graph may cause missing hits in a data partition join paradigm even if **each object is duplicated to all the buckets that intersect with the object**. For example, regarding the join graph in Figure 2(b) we assume that data are partitioned into two buckets (as depicted in Figure 2(a)) where $a1$ is in bucket 1, $c1$ is in bucket 2, and $b1$ and $d1$ are in both buckets. In this example, the joining tuple $(a1, b1, c1, d1)$ will be missed by applying the partition based join paradigm. This is because in both buckets we can only obtain a part of the tuple. In bucket 1, we obtain only $(a1, b1, d1)$, while in bucket 2 we obtain only $(b1, c1, d1)$. Therefore, we cannot obtain the joining tuple in either bucket.

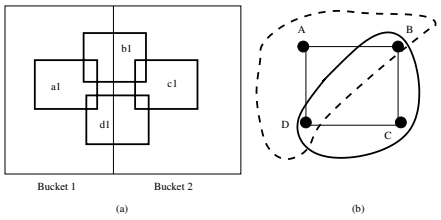

Figure 2. Sparsity missing hits problem

To resolve this *sparsity missing hits* problem, we may decompose the join graph in Figure 2(b) into two subgraphs: the one (depicted by a dotted curve) induced by vertices A, B, and D, and the one (depicted by a solid curve) induced by B, C, and D. Regarding the example in Figure 2(a), in each bucket we then execute simultaneously the local joins, which are respectively represented by the two subgraphs, by reading in the objects only once. Finally, we perform a relational join between two tables (A, B, D) and (B, D, C) with B and D being the join columns. In this example, the join tuple $(a1, b1, c1, d1)$ is obtained from the join of $(a1, b1, d1)$ and $(b1, c1, d1)$. This is the basic idea of our approach.

In the next section, we will present our join algorithm, its I/O management, and the correctness.

Note that that by similar examples, it can be shown that a trivial application of S^3J to multi-way spatial joins does not work either; that is, it is no longer correct in a multi-way spatial join that we only need to handle the join within one bucket in the same level. It is worth to point out that our graph partition based technique presented in this paper is also applicable to an application of S^3J to multi-way spatial

joins, though it will not be presented in this paper.

3 Graph Partition Based Join Algorithm

In this section, we propose a new multi-way spatial join algorithm - Two-Phase Partitioning Join (TPPJ). TPPJ contains two partitioning phases, a graph partitioning and a data partitioning phase. To present our algorithm, the following notation is needed.

A graph $G = (V, E)$ is *star-like* if G has at least three vertices, and in G there is a *peak* vertex v_0 such that every other vertex is adjacent to v_0. For example, the three graphs in Figure 3 are respectively star-like. A graph consists of only two vertices is *2-graph*; that is, a graph has two vertices and one edge connecting the two vertices. A set of subgraphs $\{G_i = (V_i, E_i) : 1 \leq i \leq k\}$ of $G = (V, E)$ *covers* G if $E = \cup_{i=1}^{k} E_i$. Suppose that G is a star-like join graph and v_0 is a peak vertex of G, and P is a data partition for every vertex (data set) in G. We call P *compatible* with G and v_0 if:

- each object o from the data set v_0 chooses only one bucket to be assigned; and this bucket also contains o.
- each object o from other data sets (vertices) is duplicated to the buckets which intersect o.

A data partition P is *compatible* with a 2-graph G if each object from the data sets (vertices) in G is duplicated to the buckets which intersect the object.

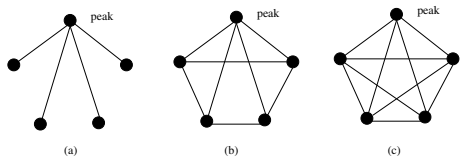

Figure 3. star-like graphs

Below is a framework of our spatial multi-way join algorithm TPPJ. It consists of four steps.

Algorithm TPPJ

Step 1: Decompose the join graph $G = (V, E)$ into a set $S = \{G_i = (V_i, E_i) : 1 \leq i \leq k\}$ of subgraphs of G such that:

- each G_i is either a star-like graph or a 2-graph, and
- S covers G, and
- for each star-like graph G_i, there is a *denoted* peak vertex $v_{i,0}$ with the property that for each pair of star-like graphs G_i and G_j, $v_{i,0}$ and $v_{j,0}$ are not adjacent, and
- any vertex in a 2-graph is not from those denoted peak vertices $\{v_{i,0} : G_i$ is star-like$\}$.

Step 2: Obtain a data partition P for all data sets (vertices) in G such that P is compatible with every star-like G_i and its denoted peak vertex $v_{i,0}$. P is also compatible with every 2-graph.

Step 3: For each bucket, do joins given by the subgraphs.

Step 4: Suppose that for each subgraph G_i, the join results obtained from step 3 over all buckets are maintained in a table denoted by T_i where each column of T_i corresponds to a vertex (data set) in G_i, and every tuple of T_i corresponds to a join result tuple for G_i but consists of only IDs for the corresponding objects. Do the multi-way equi-join on all T_i such that for each pair of T_i and T_j with some join columns, they have to perform an equi-join as part of the multi-way join. The relational multi-way join results will be the results of the spatial multi-way join.

For example, suppose that we have 4 data sets A, B , C, and D where each data set has only one object respectively $a1$, $b1$, $c1$, and $d1$. Assume that the join graph is the one as depicted in Figure 2(b). Below is an illustration of our algorithm.

- In step 1, we partition the join graph into two star-like graphs (A, B, D), and (B, C, D).
- In step two, we partition each data set into two buckets as depicted in Figure 2(a) such that in bucket 1 has $a1$, $d1$, and $b1$, and bucket 2 has $b1$, $c1$, and $d1$.
- In Step 3, we do the join for each bucket. In bucket one, we obtain a tuple $(a1, b1, c1)$ for the star-like graph induced by (A, B, C) and null for another star-like graph induced by (B, C, D). In bucket 2, we obtain a tuple $(b1, c1, d1)$ for (B, D, C) but null for (A, B, C).
- In step 4, we implement the relational equi-join between (A, B, C) and (B, C, D) where the join columns are B and C. The tuple $(a1, b1, c1, d1)$ is obtained; and it is also the result of spatial join as depicted in Figure 2(b).

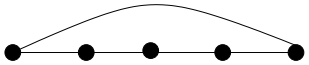

Figure 4. an example

Note that in an ideal situation, we would prefer to decompose the graph into star-like graphs only with the properties required in Step 1; however, it is not always possible

to do this. For example, the graph G as depicted in Figure 4 cannot be decomposed into a set of star-like subgraphs covering G such that the peak vertices are not *adjacent* to each other. This is the reason why we allow 2-graphs in Step 1.

Step 4 is about processing multi-way relational equi-joins; and there are many existing techniques in the literature [11]. In our implementation, we apply a *left-deep* tree based hash join algorithm. Below we will detail every step except step 4. We start with the correctness of TPPJ and the theoretical reasons about why we put those constraints in TPPJ.

3.1 Fundamentals of TPPJ

Consider that our TPPJ is a special case of the partition based join paradigm. In this subsection, we will show that those constraints in TPPJ are sufficient and necessary. That is, we are going to show that the necessity of having the constraints, as well as the correctness of the algorithm. First, we show that if a join graph has at least 3 vertices then only a star-like join graph does not need to be partitioned into subgraphs.

Theorem 1 *Suppose that a spatial join of R_1, R_2, ..., R_n forms a star-like join graph G where R_1 is a peak vertex of the graph; and a data partition P is compatible with G and R_1. Then, the union of all local join results from all the buckets is equal to the join result for G.*

Proof: Suppose that there are m buckets, T_j denotes the local join result set for G in bucket j ($1 \leq j \leq m$), and T denotes the complete join result set for G. Clearly, $\cup_{j=1}^{m} T_j \subseteq T$. Below we prove $T \subseteq \cup_{j=1}^{n} T_j$.

Suppose that $\forall(r_{1,x_1}, r_{2,x_2}, ..., r_{n,x_n}) \in T$, B_l is the bucket containing the rectangle r_{1,x_1} from R_1. Then for $2 \leq i \leq n$, r_{i,x_i} must be in B_l as well; this is because that each r_{i,x_i} has to intersect r_{1,x_1} to qualify for a rectangle in the join result tuple. Consequently, this n-tuple $(r_{1,x_1}, r_{2,x_2}, ..., r_{n,x_n})$ is in T_l. □

Theorem 1 means that if the algorithm TPPJ is applied to a star-like join graph then we do not need to implement the step 1 (graph decomposition) and the step 4 (relational multi-way join). On the other hand, we can show that only a star-like graph does not need a graph partition when implementing the partition based join paradigm.

Theorem 2 *Suppose that the join graph G of a spatial join is not star-like. Then, there is data partition P of the data sets (vertices) in G such that the union of all local join results from all buckets does not cover the join result for G.*

Proof: Since G is not star-like but connected, there must be 4 vertices, say, A, B, C, and D, such that A is adjacent to B and D. However, C is not adjacent to A and B is not adjacent to D. Note that we are not interested in the relationship between C and B, nor that between C and D.

Suppose that a partition P restricted to A, B, C, and D is the one as illustrated in Figure 2(a). Further, suppose that $a1$, $b1$, $c1$, and $d1$ are respectively from A, B, C, and D, and their locations are illustrated as those in Figure 2(a). We assume that $a1$, $b1$, and $d1$ are in bucket 1, while $b1$, $c1$, and $d1$ are in bucket 2. We also assume that there is one object from each data set other than A, B, C, or D (if there are more than 4 data sets) in the join graph intersecting respectively $a1$, $b1$, $c1$, and $d1$; and also intersects each other. Consequently, $(a1, b1, c1, d1)$ must be one part of a tuple in the join result. However, according to a partition-based join algorithm any tuple containing $(a1, b1, c1, d1)$ cannot be in the result; this is because $(a1, b1, d1)$ has to be in bucket 1 while $(b1, c1, d1)$ has to be bucket 2. □

In PBSM [15] and Theorem 1, it is already shown that that if a data partition is compatible with a join graph which is either a 2-graph or a star-like graph, then we can guarantee the correctness of an application of the partition based join paradigm. The examples below show that it is necessary to have a compatible data partition.

Example 1. A star-like join graph has three vertices, A, B and C. Suppose that a, b, and c are the objects respectively from A, B, and C. The whole space is divided into two buckets; the data partition and the locations of a, b, and c are shown in Figure 5(a), where a and b are assigned to bucket 1, and a and c are assigned to bucket 2. □

Example 2. Suppose that A and B are two adjacent vertices in a join graph G. A and B respectively have only one object a and b. Assume that the whole space is partitioned into two buckets. The buckets and the locations of a and b are shown in Figure 5(b), where a is assigned to bucket 1 and b is assigned to bucket 2. We also assume that every vertex in G other than A or B has only one object which intersects a and b; and those objects from other vertices also intersect each other. □

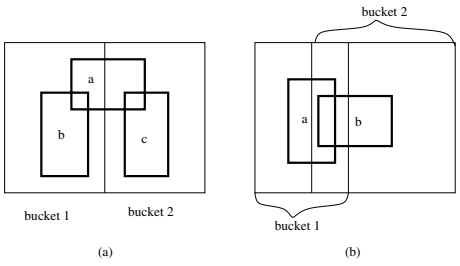

Figure 5. an illustration

Example 1 shows that even for a star-like graph G, if in a data partition P an object from the peak vertex is not contained by any bucket (therefore, P is not compatible with

G and the peak vertex) then the union of local join results may lead to some missing hits. In the example, the join result (a, b, c) cannot be obtained from a union of local join results in the two buckets even a has been duplicated into the two buckets.

Example 2 shows that even for a data partition which is compatible with the peak vertex may still lead to missing hits if an object from a non-peak vertex is not duplicated to all the buckets which intersect the object. In the example, there is actually only one tuple in the join result; however it will be missed in local joins in both buckets.

Now we show the correctness of the algorithm.

Theorem 3 *The algorithm TPPJ is correct; that is, TPPJ can produce the exact solution for any multi-way spatial join (G) where every data set is a set of isothetic rectangles.*

Proof: Suppose that G is a join graph, and G is decomposed into a set of subgraphs G_i $(1 \leq i \leq k)$ by the step 1 in TPPJ

From theorem 1, it follows that TPPJ can produce correct join results for every star-like G_i. The results in paper [15] also implies that our TPPJ can produce the correct join results for every 2-graph G_i.

Clearly, every tuple t in the join result for G can be decomposed into k sub-tuples which are respectively a tuple in a sub-join result (for G_i); and the tuple t can be recovered from those sub-tuples by the relational equi-join. On the other hand, it can be immediately verified that in step 4, any tuple in the result set of the multi-way relational join among all sub-join results also a tuple in the result set of spatial multi-way join (for G). □

Note that it can be immediately verified that the intersection of any pair of local join result sets from two different buckets is empty due to the fact that each object from a peak vertex is assigned to only one bucket.

In the next several subsections, we will show the details for steps 1, 2, and 3 in TPPJ.

3.2 Graph Partition

In this subsection, we investigate the step 1 in TPPJ - the graph partitioning problem. We will first formalise one optimisation problem in the graph partitioning. Then we will show the complexity of the problem, together with a heuristic.

Clearly, Step 1 in TPPJ is feasible. A naive way to do Step 1 is to decompose a join graph into a number of 2-graphs for each edge. However, such a decomposition may have many subgraphs. This not only means that we have to process too many intermediate results but also potentially increases the computational complexity in Step 4 (relational multi-way join) of TPPJ. Therefore, we propose to have a graph decomposition in Step 1 with the minimum number of subgraphs. The optimisation problem is thus described below.

Optimal Graph Decomposition Problem (OGDP)
INSTANCE: A connected graph $G = (V, E)$.
QUESTION: Decompose it into a set of subgraphs with the requirements specified in Step 1 of TPPJ such that the number of subgraphs is minimised.

Theorem 4 *OGDP is NP-hard.*

Proof: In this paper, we show only the basic idea of the proof. The interested readers may refer our full paper [8] for the detail.

We will transform the *vertex cover* [4] problem to a special case of OGDP. For each graph G in the vertex cover problem, we attach $2n^2$ adjacent vertices to each vertex of G to make an instance G' of OGDP; for instance, Figure 6 illustrates such a transformation (from (a) to (b)). Then, we can show that the optimal solution for G' has to consist of 1) the star-like graphs with the peak vertices which form a minimum vertex cover for G, and 2) the remaining uncovered 2-graphs in G'. □

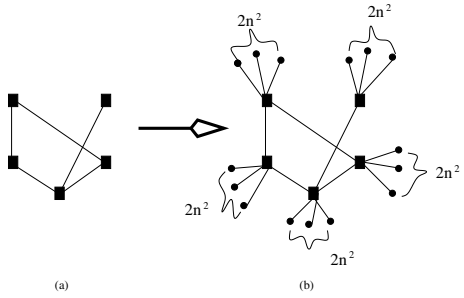

Figure 6. an transformation from vertex cover to OGDP

Now, we present a heuristic for solving OGDP. It applies a greedy heuristic to iteratively choose an available vertex with the maximum degree. Below is the algorithm.

Step 1 - Algorithm OGDP

Step 1.1: Iteratively do the following (till no such vertex to choose):

- choose a vertex v_0 from V with the maximum degree in G, which is at least 2, and then
- delete from V all vertices incident to v_0, as well as remove v_0, and then
- enter into the next iteration.

Step 1.2: For each chosen v_i, form the induced subgraph G_i of G with the vertex set which consists of v_i and all the adjacent vertices of v_i.

Step 1.3: For each edge left, form a 2-graph.

Step 1.4 Return the star-like graphs chosen in Step 1.2 and the 2-graphs in Step 1.3 (if any) to form a graph partition of G.

Note that each vertex v_i chosen in Step 1.1 is the denoted vertex which is not adjacent each other, and needs to be treated specially in the data partitioning phase.

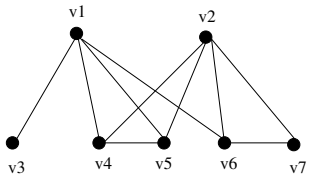

Figure 7. gdp algorithm illustration

For example, regarding the graph in Figure 7 the algorithm will output two star-like graphs respectively induced by ($v1$, $v3$, $v4$, $v5$, $v6$) and ($v2$, $v4$, $v5$, $v6$, $v7$), and one 2-graph ($v6$, $v7$). Note that both star-like graphs include edge $v4$ and $v5$. In fact, we have the option of removing the edge (v_4, v_5) from one of these two star-like graphs. However, adding one more edge to a join graph will make the size of join result smaller. This was why we duplicate (v_4, v_5) in both subgraphs.

Note that the algorithm can run in $O(n \log n)$ time (n is the number of vertices) if we sort vertices in G first according to their degree values. Moreover, it may be immediately shown that the algorithm can guarantee a detection of whether or not a graph is star-like. If a graph is star-like, then the algorithm outputs only one subgraph - the G itself. However, due to NP-hardness of the problem OGDP, the algorithm OGDP cannot guarantee the minimality of the decomposition.

3.3 Data Partition

Suppose that a graph partition has been done in Step 1 of TPPJ. Assume that $S = \{G_i : 1 \leq i \leq k\} \cup \{G_j : k+1 \leq j \leq q\}$ is a set of subgraphs obtained in Step 1 where for $1 \leq i \leq k$, G_i is star-like, and for $k+1 \leq j \leq q$, G_j is a 2-graph. Moreover, assume that for $1 \leq i \leq k$, $v_{i,0}$ is the denoted peak vertex in G_i; that is, every pair of $v_{i,0}$ and $v_{j,0}$ are not adjacent. In step 2 we need to partition the data sets. As mentioned before, a data partition P should be compatible with every G_i and $v_{i,0}$ for $1 \leq i \leq k$, and with every G_j for $k+1 \leq j \leq q$. That is, find a data partition P such that:

- for $1 \leq i \leq k$, every object from $v_{i,0}$ should be contained by one bucket in P and is allocated to that bucket; and
- every object from other data sets (vertices) will be allocated to the buckets which intersect the object (possibly with some duplications).

We use a similar idea as that in [10] to partition the data sets. Our algorithm is described below.

Step 2 - Data Partitioning

Step 2.1: Obtain an initial partition with m rectangle buckets.

Step 2.2: Allocate each object r from every $v_{i,0}$ for $1 \leq i \leq n$ in the following way:

- In case if there is a bucket containing r, allocate r to this bucket.
- If there is no bucket containing r, then get all buckets which intersects r. Then choose the one from those buckets to expand to contain r such that the resultant area of the new bucket after an expansion is minimum. Assign r to this new bucket; and use the new bucket to replace the old bucket and remove any other buckets which are contained by the new bucket.

Step 2.3: Assign each object from the other vertices to the buckets intersecting the object.

In data partition, we manage I/O as follows. We allocate one page for each bucket. Once a page is full, we write it back to the hard disk. To distinguish objects from different data sets, we partition data sets one by one; and the denoted peak vertices $v_{i,0}$ start before the other vertices.

3.4 Join and Partitioning Skew

In Step 3, we execute local joins for the sub-join graphs bucket by bucket. To save I/O costs, in step 3 the data in each bucket will be read in once only into the main memory to simultaneously execute the sub-joins. In our algorithm TPPJ, we implemented a variation of dynamic interval tree based plane-sweeping line technique [3, 19] to do local joins in each bucket.

While partition data, we can roughly estimate the smallest number of buckets we should have in order to make the data in each bucket fit in memory. Suppose the data size of each data set R_i is denoted by $||R_i||$ where R_i is a set of rectangles and their IDs, k is the number of subgraphs, β is a page size, and M is the available memory size. Then, we

compute the minimum number m of buckets by the following formula:

$$m = \lceil \frac{\sum_{i=1}^{n} ||R_i||}{M - k\beta} \rceil \quad (1)$$

As data objects may be duplicated into several buckets, a conservative way is to double the figure calculated from (1). A *partition skew* may still occur; that is, one bucket may have extremely large number of buckets. Though our current implementation of the algorithm does not incorporate any of partition skew resolution techniques, we feel that the technique of "sampling the data sets to get a good initial partition" [10] will be a good choice. It is possible that the partition skew may still exist after using this resolution technique. Consequently, the data in one bucket may not entirely fit in the memory. If this occurs, then the technique of dynamically repartitioning the overflow bucket [15] will be our next choice.

4 Experiment Results

We implemented our algorithm TPPJ on a PENTIUM III/700 running Linux 2.4.7 with 256 main memory and 12GB local disk access. We examined the efficiency and scalability of our algorithm against different data densities, different data sizes, different graph sizes, different graph densities, and different bucket numbers.

In our initial experiment, we down-loaded the two group of data sets from TIGER/LINE file [21], as illustrated in Figures 8 and 9. The 10 data sets in Figure 8 are the road segment and hydrograph data of different counties from Washington state, while the 10 data sets in Figure 9 are the road segment from California State. The data sets from CA have more objects but less density. Note that to implement joins, the data sets from different counties are mapped to the same center by a translation.

County (WA)	#obj	density	County (WA)	#obj	density
A: Chelan	14701	0.13	F: Lewis	18440	0.24
B: Clark	11062	0.14	G: Skagit	12264	0.18
C: Cowlitz	16395	0.20	H: Stevens	20193	0.17
D: Grays Harbor	13811	0.20	I: Thurston	11099	0.10
E: Kittitas	15761	0.16	J: Whatcom	11131	0.15

Figure 8. 10 Counties from Washington

County (CA)	#obj	density	County (CA)	#obj	density
A: Alameda	53490	0.10	F: Orange	108919	0.10
B: Contra Costa	44774	0.10	G: Riverside	114186	0.08
C: Fresno	66637	0.04	H: Sacramento	53466	0.07
D: Kern	120878	0.11	I: San Diego	123093	0.10
E: Monterey	39150	0.08	J: Santa Barbara	32493	0.06

Figure 9. 10 Counties from California State

Our experiment was based on 4 groups of query graphs as depicted in Figure 10. The graphs in Group 1 are very sparse. The graphs in Group 2 have a medium density. The graphs in Group 3 are complete bipartite graphs - the graphs with a high density, while the graphs in Group 4 are the complete graphs - graphs with the highest density.

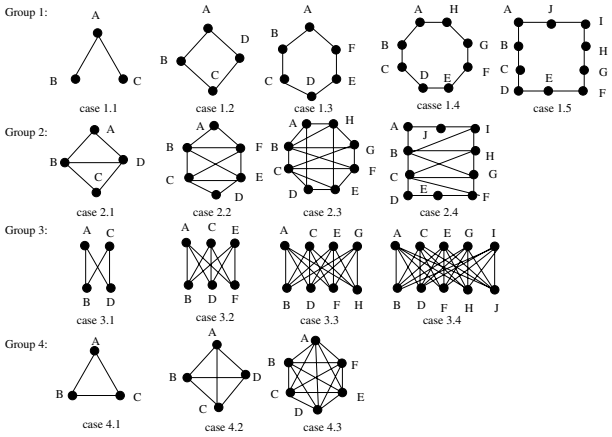

Figure 10. various query graphs

We also experimented the correlation between the number of buckets and the data sizes. As the number of buckets given in formula (1) provides only the basic requirement, there is a trade-off between the data partitioning overhead and local join costs. We implement the algorithm for 5 different bucket sizes, 256, 1024, 2304, 4096, and 5625. For each query graph, the average number of objects takes $15,000$, $30,000$, or more than $60,000$. The average number of objects per set in the complete WA data in Figure 8 is about $15,000$; and thus they are used in our experiment for the case of average $15,000$ objects per data set. Figure 12 shows the experiment results. We then randomly choose $30,000$ objects respectively from each CA data set; the experiment results are depicted in Figure 13 Finally, We use the first six complete CA data sets to implement TPPJ against the query graphs in group 4 where the average number of objects per data set is more than $60,000$. For each query graph, the implementation of TPPJ has therefore done for each combination of an average data size and a bucket number.

From the experiment results, we can see a clear correlation between average data size and the number of buckets; that is, along with an increment of the data size, we should increase the number of buckets to achieve a fast response. The experiment results also suggest that our algorithm is quite scalable regarding the data size. An increment of query graph density does not seem to increase the join costs. Note that we did not provide experiment results against the situations when the number of buckets is 256 but

the average number of data objects in each data set is at least $30,000$; this is because the computation costs of using 256 buckets for large data volume are very high and we would like to concentrate on comparable costs.

5 Conclusion and Remarks

In this paper, we studied the problem of efficiently processing multi-way spatial join without presence of spatial index. We developed a novel graph partition based join algorithm TPPJ. The algorithm TPPJ involves a two-phase partitioning technique. Firstly, it divides the join graph into a set of subgraphs; and secondly it partitions the data space. Then in TPPJ, we run local joins in each bucket by reading data only once. Finally, the sub-join results against the subgraphs are joined together by a relational multi-way join.

In TPPJ, we also investigated a novel optimisation problem of graph partitioning. Our results include the complexity of the problem, as well as an approximate algorithm. Note that our graph partitioning paradigm may also be a necessary pre-process for applying S^3J to multi-way spatial joins. As one of our future work, we would like to implement this idea and compare it with TPPJ. Another future work is to investigate the situation where the data sets are partially indexed; that is, some data sets are indexed while the others are not.

References

[1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. S. Vitter, "Scalable Sweeping-Based Spatial Join", *VLDB'98*, 1998.

[2] T. Brinkhoff, H. Kriegel, B. Seeger, "Efficient Processing of Spatial Joins Using R-trees", *ACM SIGMOD'93*, 1993.

[3] T.H. Cormen, C.E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT press, 1990.

[4] M.R. Garey and D.S. Johnson, *Computers and Intractability: a guide to the theory of the NP-Completeness*, Freeman, New York, 1979.

[5] A, Guttman. "R Trees: A Dynamic Index Structure For Spatial Searching", *ACM SIGMOD'84*, 1984.

[6] G. R. Hjaltason and H. Samet, "Incremental Distance Join Algorithms for Spatial Databases", *SIGMOD'98*, 237-248, 1998.

[7] N. Koudas, K. Sevcik, "Size Separation Spatial Join", *ACM SIGMOD'97*, 1997.

[8] X. Lin, H.X. Lu, and Q. Zhang, "Graph Partition Based Multi-Way Spatial Joins", *full paper*, 2001.

[9] M.L. Lo and C. V. Ravishankar, "Spatial Joins Using Seeded Trees", *ACM SIGMOD'94*, 1994.

[10] M.L. Lo and C.V. Ravishankar, "Spatial Hash Joins", *ACM SIGMOD'96*, 1996.

[11] P. Mishra and M.H. Eich. "Join processing in relational database", *ACM Computing Surveys*, 24(1):64-113, March 1992.

[12] N. Mamoulis, D. Papadias, "Integration of Spatial Join Algorithms for Processing Multiple Inputs", *ACM SIGMOD'99*, 1999.

[13] J. Orenstein, "Spatial Query Processing in an object-Oriented Database System", *ACM SIGMOD'86*, 326-336, 1986.

[14] J. Orenstein, "A comparison of spatial query processing techniques for native and parameter spaces", *ACM SIGMOD'90*, 343-352, 1990.

[15] J.M Patel, D.J. DeWitt, "Partition Based Spatial-Merge Join", *ACM SIGMOD'96*, 1996.

[16] D. Papadias, N. Mamoulis, and B. Delis, "Algorithms for Querying by Spatial Structure", *VLDB'98*, 1998.

[17] D. Papadias, N. Mamoulis, and Y. Theodoridis, "Processing and Optimisation of Multi-way Spatial Joins Using R-trees", *ACM PODS'99*, 1999.

[18] H. Park, G. Cha, and C. Chung, "Multi-way Spatial Joins Using R-Trees: Methodology and Performance Evaluation", *SSD'99*, LNCS 1651, Springer-Verlag, 229-250, 1999.

[19] F. Preparata and M. Shamos, *Computational Geometry*, Springer-Verlag, 1988.

[20] K. C. Sevcik and N. Koudas, "Filter Trees for Managing Spatial Over a Range of Size Granularities", *VLDB'96*, 16-27, 1996.

[21] "Tiger/Line files (Redestricting Census) (tm). 2000", Technical Report, U.S. Bureau of the Census, 2000.

[22] H. Zhu, J. Su, and O. Ibarra, "On Multi-Way Spatial Joins with Direction Predicates, *SSTD'01*, LLNCS 2121, Springer-Verlag, 217-235, 2001.

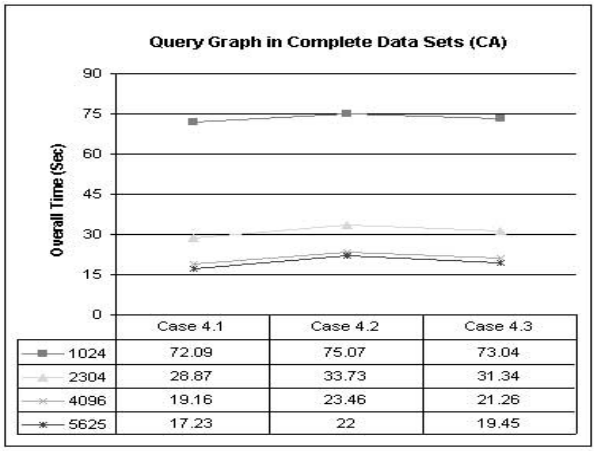

Figure 11. average more than 60,000 objects per data set

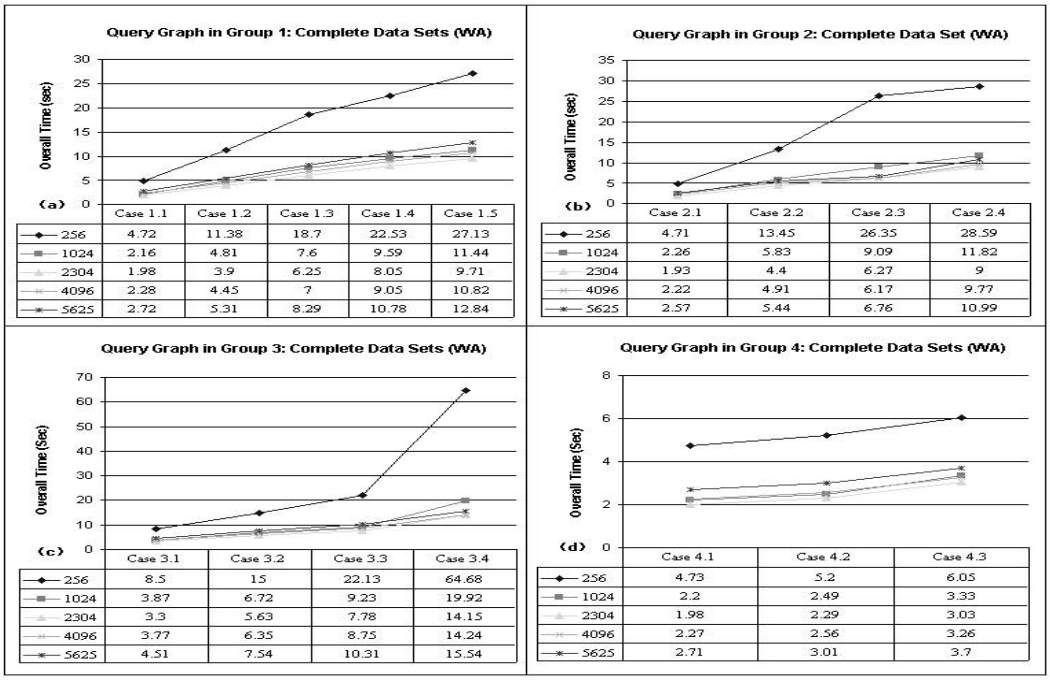

Figure 12. average 15,000 objects per data set