

Data Declustering and Cluster-Ordering Technique for Spatial Join Scheduling

Jitian Xiao, Yanchun Zhang, Xiaohua Jia[†], Xiaofang Zhou[‡]

Dept. of Mathematics & Computing, University of Southern Queensland,
Toowoomba Qld 4350, Australia, *Email: {jitian, yan}@usq.edu.au*

[†]Dept. of Computer Science, City University of Hong Kong, Kowloon, Hong Kong

[‡]CSIRO Mathematical and Information Sciences, GPO Box 664, Canberra 2601, Australia

Abstract

The spatial join operations combine two sets of spatial data by their spatial relationships. They are the most expensive operations, yet among the most common operations in spatial databases. In this work we will investigate the optimization issue through data declustering. Firstly, a graph model is developed to formalise the problem, and then a matrix-based data partitioning method is proposed for declustering the non-uniform spatial data. The clusters produced are also maximum-overlapping ordered. When inputting the clusters in this order for spatial joins, the I/O cost can be reduced significantly. The experimental work has shown that 15-35% saving can be achieved when comparing with some other existing methods.

Keywords: spatial joins, data partitioning, declustering, graph and hypergraph.

1 Introduction

Spatial joins are among the most important operations in spatial databases, yet the cost of spatial join can be very high because of the large sizes of spatial objects and the computation-intensive spatial operations. Spatial join operation, such as `crosses`, `intersects`, *etc.*, may take a long time when there is a large number of spatial objects [1]. The cost of spatial join can be reduced by the *filter-and-refine* approach [15, 6, 2] which employs a filtering step followed by a refinement step. In the filtering step, a *weaker* predicate for the spatial predicate is applied on the *approximation* of spatial objects to produce a list of *candidates*; then a refinement step follows to drop the “false hits” in the candidate list by applying a full test of the spatial operation on full geometry of spatial objects. A commonly used filtering predicate for many spatial operations is intersection of *Minimum Bounding Rectangles (MBRs)* of spatial objects. A necessary condition for two polygons to intersect (or to meet many other spatial relationships) is that their MBRs must intersect. This filter-and-refine approach can reduce spatial join cost because the filtering operation can reduce the join search space, and the filter cost is usually much lower than the refinement cost by two facts: the filtering predicate is less expensive to evaluate and object approximations are smaller in size than full geometry (thus incurs less disk access cost).

Lots of efforts have been made to reduce spatial join cost in the last decade [2, 6, 19]. To reduce CPU cost and I/O cost of spatial join processing, data partitioning methods are usually employed to decompose a bigger set of spatial data into smaller data sets so that each set can be processed more efficiently. A typical spatial data partitioning method for spatial data

is based on *space decomposition*, in which the whole space is decomposed into regions such that spatial objects can be partitioned according to their space coordinates or their spatial relationships, such as *enclosure*, *overlapping* and *adjacent* etc., with these regions. Most typical space decomposition methods use one or more spatial indexing mechanisms, like *R-tree*, *R⁺-tree*, *k-d tree* and the *Z-value* based approaches [18, 10]. While these data partition methods take advantage of an indexing mechanism to guide the partitioning, they do not specifically address the data partitioning and clustering from two or more different classes for set join operations. Furthermore, there may be no proper index for the given spatial join to decluster the spatial data from different data classes. To address this, we developed a matrix-based approach to do the spatial data partitioning[24]. This approach is able to reduce a significant amount of I/O cost for the environment where the sizes of the spatial objects concerned are the same or similar. We call this method a *uniform* one. However, spatial data are usually non-uniform in size. The spatial data partition strategy based on the uniform assumption has problems in realistic applications whenever the data size changes significantly. As an extension to this work, we will propose a new model in this paper for partitioning or declustering non-uniform spatial data. The new approach works well for both cases of uniform and non-uniform sized spatial objects. The rest of this paper is organised as follows: in Section 2, we ly introduce the previous work [24], the spatial data partitioning method on uniform objects. In section 3, a new model for the non-uniform spatial data partitioning method is proposed. Some problems and motivations on non-uniform data partition are discussed in Section 4. Section 5 gives the matrix-based partitioning algorithm as well as the complexity analysis. In Section 6 we further consider the maximum overlapping order among the resultant clusters produced by the partitioning algorithm. When inputting the clusters in this order, a more decline in the fetching time from disk may be achieved. Section 7 is the experimental and simulation result. And, the conclusion will be presented in Section 8.

2 Summary on Data Partition in Uniform Spatial Data Model

Assume that the spatial database system is based on the following extensible-relational model: only one spatial column of polygon type is allowed for each table. Other columns are omitted except the so-called *key-pointer* data which consists of a unique objects identifier and the MBR of the object. In other words, the schema of the tables is: (*id*:number; *mbr*:rectangle; *obj*:polygon) where a MBR is represented by the coordinates of its lower left and top right corners. We choose one of the most important spatial join operation – *binary polygon intersection join* – as a representative of spatial join operations. It is defined on table *S* and *T* using the standard relational algebra notations as $S \bowtie T = \sigma_{intersect(S.obj, T.obj)}(S \times T)$ where \bowtie , π , σ and \times denote the join, project, select and Cartesian product operations, respectively. The MBR and the object with identifier *id* are denoted as *mbr(id)* and *obj(id)*, respectively.

The *filter* operation using MBR intersection for $S \bowtie T$ produces a set of *candidates* defined as: $F = \{(sid, tid) \subseteq (\pi_{id}S) \times (\pi_{id}T) \mid intersect(mbr(sid), mbr(tid))\}$. The *refinement* step to perform $S \bowtie T$ using *F* produces the final results $S \bowtie T = \{(sid, tid) \subseteq F \mid intersect(obj(sid), obj(tid))\}$. For simplicity, we still use *F* for this set, and call it a *filter*. Regard *F* as a *sequence* of candidates, the *refinement algorithm* is simply to fetch the objects

referenced by the next candidate in F if these objects are not already in memory, and to check polygon intersection by using the full description of the spatial objects.

For the refinement cost description, we defined a *Spatial Join (SJ)* graph G_F for a candidate set F . $G_F = \{V, E\}$ where the vertex set $V = V^S \cup V^T$, $V^S = \{v \mid v \in \pi_1 F\}$, $V^T = \{v \mid v \in \pi_2 F\}$; and the edge set $E = \{(v_1, v_2) \mid v_1 \in V^S, v_2 \in V^T, (v_1, v_2) \in F\}$. We further define its *SJ* matrix $M_F = [m_{ij}]_{n \times n}$ as follows:

$$m_{ij} = \begin{cases} 1 & \text{if } i = j \vee (v_i, v_j) \in E \vee (v_j, v_i) \in E \\ 0 & \text{otherwise} \end{cases}$$

where $n = |V| = |V^S| + |V^T|$ is the size of M_F . If $(v_i, v_j) \in E$, we say that object v_i and v_j are *joinable*. In the uniformed spatial data model, we assume all of the objects in a spatial join operation are the same or similar in size (for example, c blocks of disk storage), the fetching cost for any joinable object pair (v_i, v_j) is a constant c . Therefore, instead of being the exact cost for fetching the object pair (v_i, v_j) , we let m_{ij} be 1 if v_i and v_j are joinable and 0 otherwise.

Under the assumption of uniformed spatial data model, the declustering of the objects into groups can be converted to find a partition of G_F so that both the number of subgraphs and the number of overlapped nodes among different subgraphs are minimised. This can be done by using a variation of BEA (Bond Energy Algorithm [13, 24]) on matrix M_F . For a given symmetric square matrix M , BEA permute rows and columns to maximise the so called *energy* which measures the sum of *bonds* between any two objects v_i and v_j . After the energy is maximized, M appears in a form of two or more data-clustered submatrices, and is then divided into two submatrices which indicates the graph G_F is decomposed into two subgraphs. Afterward, the recursive permutation takes place for each submatrix until a desired partition is found. In practice, a pre-determined buffer size p , indicating the maximum number of objects in resultant subgraphs, is used to determine when the recursive procedure should stop. The use of p guarantees that the output clusters can all be fitted into the buffer of size p .

Once a desired partition is found, the spatial join operation can be processed in a way that all joinable candidates are processed cluster by cluster. For each cluster, as their buffer cost is less than or equal to the given buffer size p , all of the objects in the cluster can be fetched into the buffer for processing. After a cluster is processed, the data of the next cluster is fetched into the buffer and the same processing applies. Using this strategy, the disk access cost can be decreased since the objects in a cluster can all be held in a buffer of size p and the object duplication among different clusters is minimised. This strategy is especially suitable for a distributed database system as the processing can be done one cluster at one site.

3 A Graph Model For Non-Uniform Spatial Data Partition

As spatial objects are often very large in size, and the data size changes significantly, the spatial data partition strategy based on the uniform assumption applies only for a few cases or for the processing of spatial data indices. It has problems in realistic spatial applications whenever the data size is non-uniform or the proper index is not available for the given join operations. In this section, we develop a new model for the non-uniform spatial data so that the spatial data partitioning approach works for both uniform and non-uniform cases.

For a given candidate set F , define its *Node-Weighted Spatial Join (NWSJ)* graph as an undirected graph $G_F = \{V, E, w\}$ where the vertex set $V = V^S \cup V^T$, $V^S = \{v \mid v \in \pi_1 F\}$, $V^T = \{v \mid v \in \pi_2 F\}$; and the edge set $E = \{(v_1, v_2) \mid v_1 \in V^S, v_2 \in V^T, (v_1, v_2) \in F\}$. For each node $v \in V$, there is a *weight* $w(v) \geq 0$ with it. For convenience, we define an *edge weight* as $w_e(v', v'') = w(v') + w(v'')$ for each edge $(v', v'') \in E$.

In a NWSJ graph, $w(v)$ is normally used to present the size of the object v in respect of some storage unit. Denote n as $|V|$ and $d(v)$ the degree of node v . Figure 1(a) shows a filter, with its object sizes in (b), and (c) its NWSJ graph with weights shown inside cycles of the nodes.

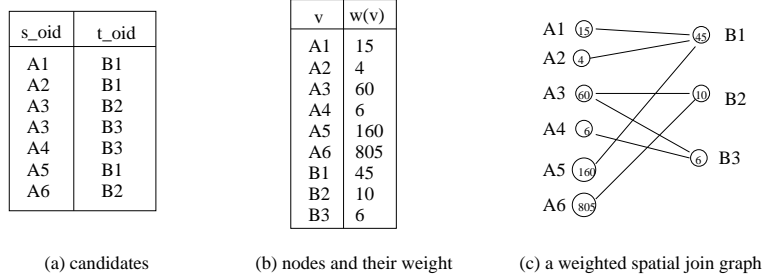


Figure 1: An example of a filter and its NWSJ graph.

In this paper, we focus on how to minimise $C_{I/O}$ for the spatial operations. Obviously, the I/O cost of fetching a spatial object v from databases (or disks) depends on $w(v)$, i.e., the size of v . The bigger the size of the object, the higher the cost of fetching it will be. The total cost $C_{I/O}$ varies in the following range

$$\alpha \cdot \sum_{i=1}^n w(v_i) \leq C_{I/O} \leq \beta \cdot \sum_{i=1}^n d(v_i) \cdot w(v_i)$$

where α and β are constants, the lower bound is achieved when each object is only fetched once in the refinement step, and the worst case is where two objects need to be fetched for every candidate. The two ends meet when $\max_{v \in V} d(v) = 1$.

To reduce the total cost of fetching objects from disk, properly re-arrange the fetching sequence of the input candidates might be a solution. By using a buffer, the objects in a filter can be filled in once a time. After all objects in the buffer are processed, other objects can be fetched into the buffer again and processed sequentially. For practice usage, we assume the size of each joinable object pair is less than or equal to the given buffer size, say p . In this case, the improvement of $C_{I/O}$ depends on the node degrees of G as well as its edge weights. For example, if the sum of weights of all nodes (i.e., objects) in a connected component $g = \{V_g, E_g\}$ of G is less than or equal to p , the I/O cost to process g is $\beta \cdot \sum_{v \in g} w(v)$ no matter how many edges are in g if the candidates in E_g are processed consecutively (comparing to the worst case where the I/O cost to process g can be as high as $\beta \cdot \sum_{v \in V_g} d(v) \cdot w(v)$).

Let F be a candidate set, $G = \{V, E, w\}$ the NWSJ graph of F . Let $G_i = \{(V_i, E_i, w|_{V_i}) \mid V_i = V_i^S \cup V_i^T, V_i^S \subseteq V^S, V_i^T \subseteq V^T, E_i \subseteq E\}$ be a subgraph of G ($1 \leq i \leq m$), where $w|_{V_i}$ is the constraint of w on V_i . Subgraph G_1, G_2, \dots, G_m form a *partition* of G with respect to candidates of F if $(\cup_{i=1}^m E_i \equiv E) \wedge (i \neq j \rightarrow (E_i \cap E_j \equiv \Phi))$. For a given buffer size p , if subgraph G_1, G_2, \dots, G_m form a partition of G and $\sum_{v \in V_i} w(v) \leq p$ for each i ($1 \leq i \leq m$),

we call the partition G_1, G_2, \dots, G_m as an *ideal partition*. As our intention is to process all candidates in a subgraph together, we also call a subgraph *subfilter*. Obviously, we have $(\cup_{i=1}^m V_i^S \equiv V^S) \wedge (\cup_{i=1}^m V_i^T \equiv V^T)$. Note that $V_i \cap V_j \equiv \Phi$ might be not true.

For a given NWSJ graph $G = (V, E, w)$ and a buffer size p , if $w_e(v', v'') \leq p$ for each edge $(v', v'') \in E$, there must exist an ideal partition for G . Once an ideal partition is found, the candidates in a subfilter can be processed consecutively. The objects from both data sets in the subfilter are fetched into buffer for processing. Then all the data in the buffer is thrown away before a new subfilter is processed. Using this strategy, the disk access cost can be minimised if the objects in a subfilter can all be held in a buffer of size p (i.e., $\sum_{v \in V_i} w(v) \leq p, 1 \leq i \leq m$). Therefore, the optimization objective to minimise I/O costs can be formulated as to find an ideal partition of G such that the following two goal functions are minimised

- 1). m (i.e., the number of subgraphs) is minimised; and
- 2). under the condition 1), $z = \sum_{1 \leq i < j \leq m \wedge v \in (V_i \cap V_j)} w(v)$ is minimised.

An ideal partition is called a *best* partition if it satisfies the above two items. Figure 2 shows two different partitions of the NWSJ graph in Figure 1 for $p=1040$. While both of them satisfy condition 1), Figure 2(b) represents a better partition than (a) since its z value is 0 (optimal) and the z value for the partition in (a) is 61 (B_1, B_2, B_3 are overlapped between two subfilters).

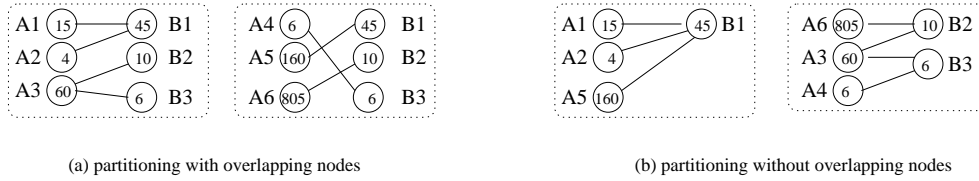


Figure 2: Alternative examples of NWSJ graph partitioning

Of course, the optimal partition depends on the buffer size p . An optimal partition for a special p might not be an optimal one for another p value. For example, when $p = 900$, Figure 2(b) is an ideal partition. But for the case of $p = 820$, both Figure 2(a) and (b) are not ideal. Instead, Figure 3 will become an ideal partition in this case although its z is 10 (not 0).

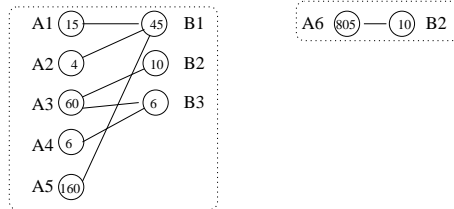


Figure 3: An ideal partitioning for $p=820$

For a given filter F and its NWSJ graph $G_F = \{V, E, w\}$, we define a *Node-Weighted Spatial*

Join (NWSJ) matrix $M_F = [m_{ij}]_{n \times n}$ as follows:

$$m_{ij} = \begin{cases} w(v_i) & \text{if } i = j \\ w(v_i) + w(v_j) & \text{if } i \neq j \wedge (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

where $n = |V| = |V^S| + |V^T|$ is the size of M_F . Intuitively, for the join operation on objects v_i and v_j , the cost of fetching these two objects relates to their sizes, i.e. $w(v_i) + w(v_j)$, while $m_{ii} = w(v_i)$ means that it just needs to be fetched once for each object v_i (here, for simplicity, we conceptually accept the assumption that an object v_i may join with itself). Figure 4(a) shows the NWSJ matrix of the filter in Figure 1. For any two objects v_i and v_j , we define the joinable

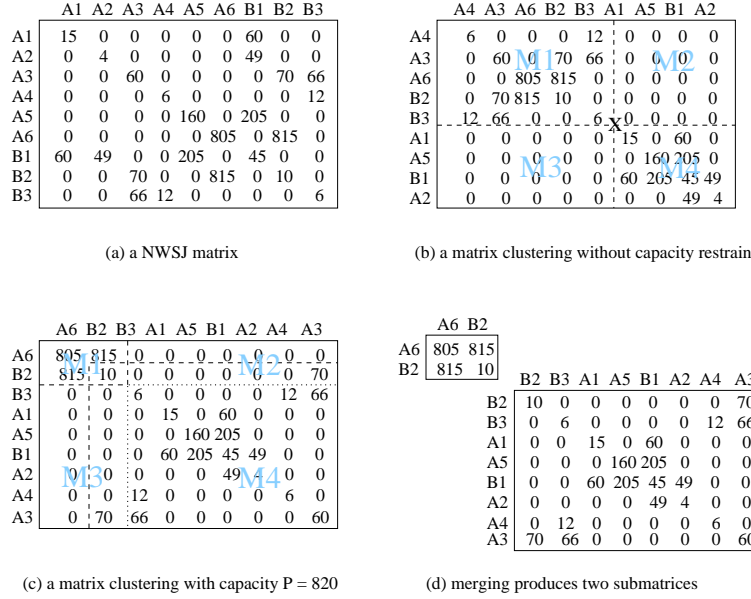


Figure 4: Examples of NWSJ matrix and its clustering

bond between them as

$$B(i, j) = \sum_{k=1}^n m_{ik} \cdot m_{kj}$$

$B(i, j)$ reflects the connection relationship (or *affinity*) between objects v_i and v_j by going through at most one else node. If an object v_k is joinable with v_i and v_j , it contributes to the bond between v_i and v_j . If v_i and v_j are joinable, they themselves will also contribute to their bond respectively. Now define the *energy* of M_F as

$$E(M_F) = \sum_{i=1}^n (B(i, i-1) + B(i, i+1))$$

where $B(1, 0) = B(n, n+1) = 0$. The energy of M is the measurement of how tightly those objects with higher joinable bonds are located closely from each other in the matrix. Intuitively, the higher the energy is, the more closely located those objects with high bonds are in the matrix. The energy of a matrix M can be changed by row and column permutation. After the energy of the NWSJ matrix is maximised, those with higher affinities will appear group by group in the main diagonal of the matrix [13]. By choosing a point, called the *dividing point*

X (as shown in Figure 4(b)), along the main diagonal of M, M can be divided into four submatrices M_1 (the upper-left corner submatrix of M from X), M_2 (the upper-right submatrix), M_3 (the left-bottom submatrix) and M_4 (the right-bottom submatrix). Note that $M_2 = M_3^T$ because of the symmetry. Define the *fetching cost* of a submatrix $M_{k \times l}$ as $C_M = \sum_{i=1}^k \sum_{j=1}^l m_{ij}$. Then C_{M_1} and C_{M_4} represent the maximum buffer size needed to process the objects in clusters represented by M_1 and M_4 respectively, and $C_{M_2} (= C_{M_3})$ represents the buffer size needed to process the spatial join operations for objects crossing clusters. For a symmetric square matrix M , define the *buffer cost* as the minimum buffer size requirement for processing objects in a cluster represented by M , i.e., $C'_M = \sum_i m_{ii}$ which is the sum of the weights of nodes in the subgraph. Therefore, we can summarise our objective to find ideal partitions as to find matrices $M_i (i = 1, 2, \dots)$ such that their C'_{M_i} are less than or equal to but as close as possible to the given buffer size p . Or in graphics terms, for a given NWSJ graph and a given buffer size p , our objective is to find an ideal partition G_1, G_2, \dots, G_m such that m is minimised. One way to minimise m is to increase $\sum_{v \in V_i} w(v)$ for each V_i so that it is as close to p as possible. Thus, the dividing point can be so chosen such that the following goal function is maximised:

$$z = C_{M_1} \cdot C_{M_4} - C_{M_2}^2 \quad (1)$$

By using an algorithm similar to the BEA (we still call it BEA), the objects in a given filter can be grouped based on their joinable bonds. For example, by applying BEA to the matrix in Figure 4(a), we can get a matrix shown in Figure 4(b). We see the creation of two clusters: one is in the upper-left corner (5×5 submatrix) and the other is in the lower-right corner (4×4 submatrix). This indicates how the objects of M should be split. In general the border for this split is not as clear-cut as that in this example. When the matrix is big, usually more than two clusters are formed and more than one candidate partition exists ([16]).

In the case of uniform model, as we do not consider the object sizes, the partitioning of that in Figure 4(b) should be a satisfactory one (by taking each non-zero element as 1). However, in the case of non-uniform, especially when we take buffer size into account, we need to find partitions that can be put into a buffer with given size p . For instance, for a buffer size $p = 820$, due to the buffer cost for M_1 in Figure 4(b) is 887 which is greater than p , this partition cannot be fitted into the given buffer, while the buffer cost for M_4 in Figure 4(b) is 224 which is much less than p . Obviously, as the buffer cost for M is 1101, all objects can not be fitted into one buffer. Is it possible to divide these objects into two groups, where both can be fitted into a buffer of $p = 820$? The answer is yes and the partition shown in Figure 4(c) can be considered as one of these solutions (note that two clusters have an *overlapping* object B2).

4 Problems and Motivations in Non-Uniformed Spatial Data Partition

Normally, after applying BEA to a NWSJ matrix M , there will be two or more candidate clusters in the resultant matrix. At this stage, it is the buffer cost of the clusters that decide whether the created candidate clusters can be fitted into the buffer or not. Assume we use a two-way splitting method to implement matrix splitting, and four submatrices M_1, M_2, M_3 , and M_4 have

been formed. There are several cases to be considered: (a) If both C'_{M_1} and C'_{M_4} are greater than the given buffer size p , then split M into two submatrices so that each one can be declustered recursively; (b) If either $C'_{M_1} > p$ or $C'_{M_4} > p$, but $C'_{M_1} + C'_{M_4} \leq 2p$, then we should consider adjusting or balancing the sizes between two clusters to see if both can be fitted into the given buffer. (c) In both of above cases, re-distributing elements of M_2 (or M_3) should be considered.

4.1 Size-balancing between clusters

Now we consider the above case (b), that is: either $C'_{M_1} > p$ or $C'_{M_4} > p$, but $C'_{M_1} + C'_{M_4} \leq 2p$. Without losing generality, we suppose that $C'_{M_1} > p$ and $C'_{M_4} < p$. From [13], we know that BEA changes row and column orders of the matrix so that it makes the greatest contribution to the *global affinity measure*. In the case of NWSJ graph, after applying BEA to M , those joinable objects with greater affinities will be put closely in the final matrix. That is to say, objects which need greater buffer sizes to do join operations will be put more tightly than others in the group. Therefore, by using SHIFT operation (see Section 4.2) for a number of times ([24]), those objects with greater affinities will be moved to the upper-left corner or lower-right corner of the matrix. During this, check the objective function (1) so that a best dividing point can be found. If this point creates new clusters M'_1 and M'_4 such that $C'_{M'_1} \leq p$ and $C'_{M'_4} \leq p$, we have found two clusters for an ideal partition and made a size-balancing between M_1 and M_4 . If it does not work, continue to decluster M_1 .

4.2 Shift operation

Dividing a submatrix by using z value in objective 1 has a disadvantage of not being able to partition a filter by selecting out an embedded “inner” block ([24]). By using the procedure SHIFT, this disadvantage can be avoided. SHIFT works in a way that moves the leftmost column of M to the right end, and the topmost row of the matrix to the bottom. By calling the SHIFT for n times, every diagonal block gets the opportunity of being brought up to the upper left corner in the matrix. After each application of SHIFT, the z value in formula 1 is re-calculated so that a best dividing point can be re-selected. Of course, when the SHIFT procedure is used, the complexity of the algorithm is increased by factor n . Experiments have shown that the use of the SHIFT improves the quality of spatial join partitioning in most cases.

4.3 Merging objects crossing clusters

In the process of two-way recursive declustering, if M_2 (as well as M_3) is not an all-zero matrix, we should merge it into M_1 or M_4 so that the two-way recursive declustering can be continued without losing joinable information. Because of the symmetry of M , we just consider M_2 . The criteria for merging M_2 into M_1 or into M_4 is to achieve less growth of the total sizes of M_1 and/or M_4 , thus to make less growth to the complexity of further declustering. For instance, suppose M_1 and M_4 are $n_1 \times n_1$ and $n_4 \times n_4$ submatrices respectively, and assume that **merger1**: merge M_2 into M_1 and make a new submatrix M'_1 of size $n'_1 \times n'_1$; **merger2**: merge M_2 into M_4 and make a new submatrix M'_4 of size $n'_4 \times n'_4$. Then we should do **merger1** when $n'_1 - n_1 \leq n'_4 - n_4$ and do **merger2** otherwise. In these cases, size-balancing may be needed if $C'_{M'_1} + C'_{M'_4} \leq 2p$. The

only exception exists when either M_1 or M_4 (we express this submatrix as M_*) gets a buffer cost which is less than p , and after merging M_2 to it, the buffer cost of M_* is still less than or equal to p . In this case, merging M_2 to M_* is better. No matter merging M_2 into either M_1 or M_4 , we guarantee to keep the same number of joinable object pairs as in the original filter, and minimise the size increase of the whole matrix. This holds the key to reduce the algorithm complexity. The work of merging objects and size-balancing can be done in $O(n^2)$ time.

Example 1 Consider the NWSJ graph shown in Figure 1. Let $p = 820$. The NWSJ matrix M for this graph is shown in Figure 4(a). By applying BEA to M , we get a matrix shown in Figure 4(b). As $C'_{M_1} = 887$ and $C'_{M_4} = 224$, size-balancing operation is needed. By applying SHIFT 2 times to the matrix, we get a matrix shown in Figure 4(c), in which the new M_1 is composed of the upper-left 2×2 submatrix with $C'_{M_1} = 815 < p$, and M_4 is composed of the lower-right 7×7 submatrix with $C'_{M_4} = 296$. Because M_2 is not an all-zero matrix, a merging operation is needed. Though merging M_2 to M_1 or to M_4 will cause the total size growth of 1 in both cases, the element 70 in M_2 has to be merged into M_4 in this case. The result is shown in Figure 4(d), the same as the graph partitioning shown in Figure 3. \square

5 Partitioning Algorithm and Analysis

Similar to the algorithm for the uniform model ([24]), the partitioning algorithm for the non-uniform model is a two-way recursive one. However, there are some essential differences between them. First of all, the buffer size p gets a completely different meaning. Secondly, the partitioning objective combines object sizes and joinable affinities between objects so that the resultant clusters can be fitted into the pre-sized buffer and the buffer cost of the resultant clusters can be as close to the buffer size as possible. And thirdly, the dividing point is determined not only by the maximum affinity of joinable objects, but also by the buffer size as well.

For clarity, we assume a procedure *Output-subfilter* will output the corresponding object pairs (or edges) according to the input matrix. Besides, we assume that, for a $n \times n$ matrix M , function *sizeofmtx*(M) returns the dimensional size of M , i.e. n . The following is the Partitioning Algorithm to be used to produce desired clusters for the non-uniform spatial join data model.

Algorithm 1. *Partition*(n, M, p)

Input: n : size of matrix M ; M : $n \times n$ symmetric matrix; p : buffer size;

Output: F_1, F_2, \dots : subfilters;

Begin

- [1] **if** ($\text{bcost}(M) \leq p$) {*Output-subfilter*(n, M); **return** ;};
- [2] permute rows and columns of M to maximise affinity aff and determine the dividing point x ($1 \leq x \leq n - 1$);
- [3] $best = aff$;
- [4] **for** $i = 1$ **to** n **do** {
- [5] SHIFT(n, M);
- [6] compute affinity aff ;

```

[7]         if (aff > best) {best=aff; k = i; };
[8]         }; /*to get a best partition position */;
[9] decompose M into M1, M2, M3 and M4 according to k;
[10] MERGE(M1, M2, M4, p);
[11] n1 = sizeofmtx(M'1); n4 = sizeofmtx(M'4);
[12] Partition(n1, M'1, p);
[13] Partition(n4, M'4, p);
[14] return ;

```

End

Note that in line 10, the procedure *MERGE*() not only merges *M*₂ to one of *M*₁ and *M*₄, but also make balancing between them if necessary.

Now we analyse the complexity of *Partition*. Let *g*(*n*) be the complexity function of the algorithm for an input *n* × *n* matrix *M*. For simplicity, we first omit procedure *SHIFT* (i.e., not considering lines 4-8 for the moment). Both the best case and the worst case for the algorithm should be considered. First, we consider the best case of the execution, i.e., non-overlapping partitioning. Denote *f*(*n*) as the complexity for this case. For a given matrix *M* of size *n* × *n*, the decomposition of *M* produces two sub-matrices *M*'₁ and *M*'₄. For an ideal decomposition, these two sub-matrices may have the same size of *n*/2 × *n*/2. In this case, we can get a recurrence formula for the complexity function as

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \alpha n^2 & \text{if } n > 0 \text{ and } \sum_{i=1}^n m_{ii} \leq p \\ 2f(n/2) + \alpha n^2 & \text{if } n > 0 \text{ and } \sum_{i=1}^n m_{ii} > p \end{cases}$$

where α is a constant and p the given buffer size. It is easy to prove that $f(n)$ is a monotone increasing function for $n \geq 0$. As the weight of each joinable object pair is less than or equal to p , either $n = 0$ or $\sum_{i=1}^n m_{ii} \leq p$ will become true after running the algorithm recursively for some rounds. Therefore, for a large n , according to the recurrent property of $f(n)$, we have

$$f(n) = 2f(n/2) + \alpha n^2$$

This equation is valid for any n that is a power of 2, say $n = 2^k$. Recall that $f(1) = \alpha$, we get

$$f(n) = \alpha n^2 \cdot \left(1 + \frac{1}{2} + \frac{1}{2^2} + \dots + \frac{1}{2^{k-1}} + \frac{1}{2^k}\right) = \alpha n^2 \cdot (2n - 1)/n$$

or

$$f(n) = O(n^2) \tag{2}$$

If n is not a power of 2, there must exist a k such that $2^k < n \leq 2^{k+1}$, thus

$$\alpha n \cdot (2n - 1)/n \leq f(n) \leq \alpha n \cdot (4n - 1)/2n$$

we still have

$$f(n) = O(n^2) \tag{3}$$

Secondly, consider the case of overlapping. Denote $F(n)$ as the complexity of the algorithm in this case. As discussed before, the procedure *MERGE* produces M'_1 and M'_4 with at least one of its size bigger than $n/2$. Without losing generality, we assume that M_2 is merged into M'_4 , and the average size of M'_4 will be $3n/4$. Then, we can get a recurrent function as

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ \alpha n^2 & \text{if } n > 0 \text{ and } \sum_{i=1}^n m_{ii} \leq p \\ F(n/2) + F(3n/4) + \alpha n^2 & \text{if } n > 0 \text{ and } \sum_{i=1}^n m_{ii} > p \end{cases}$$

$F(n/2)$ is the decomposing complexity of M'_1 while $F(3n/4)$ is the decomposing complexity of M'_4 . Obviously, the function $F(n)$ is monotone increasing. In the worst case, every recursive decomposition would produce a sub-matrix of size $3n/4$. After k times of recursive execution, either the remaining size of the sub-matrix is 0, or the buffer cost, $\sum_{i=1}^n m_{ii}$, of the remaining matrix will be reduced to be less than p . So, for simplicity, we can suppose that $n \cdot (\frac{3}{4})^k = 1$, or say $n = \lfloor (\frac{4}{3})^k \rfloor$ for some k . According to the recurrent relation of $F(n)$, we derive

$$\begin{aligned} F(n) &= F\left(\frac{1}{2}n\right) + F\left(\frac{3}{4}n\right) + \alpha n^2 \\ &= F\left(\frac{1}{2^2}n\right) + 2 \cdot F\left(\frac{1}{2} \cdot \frac{3}{4}n\right) + F\left(\left(\frac{3}{4}\right)^2 n\right) + \alpha n^2 \cdot \left(1 + \frac{1}{2^2} + \left(\frac{3}{4}\right)^2\right) \\ &= \dots \\ &= \sum_{i=0}^m C_m^i \cdot F\left(\frac{1}{2^i} \cdot \left(\frac{3}{4}\right)^{m-i} n\right) + \alpha n^2 \cdot \sum_{i=0}^{m-1} \left(\frac{1}{2^2} + \left(\frac{3}{4}\right)^2\right)^i \end{aligned}$$

where C_m^i is the coefficient of binomial, and the last equation hold for $m = 1, 2, 3, \dots$. For given n , when decomposition continues to some extent, term $\frac{1}{2^i} \cdot (\frac{3}{4})^{m-i}$ will become 0 when m is large enough for $i \leq m$, thus $f(\frac{1}{2^i} \cdot (\frac{3}{4})^{m-i})$ will become 0 according to definition of $F(n)$. Especially, when m increasing to k , we have

$$\begin{aligned} F(n) &\leq \alpha n^2 \cdot \sum_{i=0}^k \left(\frac{13}{16}\right)^i \\ &= pn^2 \cdot \left(1 - \left(\frac{13}{16}\right)^{k+1}\right) / \left(1 - \frac{13}{16}\right) \end{aligned}$$

Using $n = (\frac{4}{3})^k$, $k = \log(n) \cdot (\log 4 - \log 3)$,

$$F(n) \leq \alpha n^2 \cdot \frac{16}{3} \cdot \left(n - \frac{13}{16} \cdot n^{\log 13 - \log 12}\right) / n$$

we obtain

$$F(n) \leq O(n^2) \tag{4}$$

For the case of $n \neq \lfloor (\frac{4}{3})^k \rfloor$, a similar discussion as for equations (2) to (3) can be done and the same result as (4) can also be obtained. As the complexity of our algorithm is always greater than or equal to $f(n)$, and less than or equal to $F(n)$, i.e.,

$$O(n^2) = f(n) \leq g(n) \leq F(n) \leq O(n^2)$$

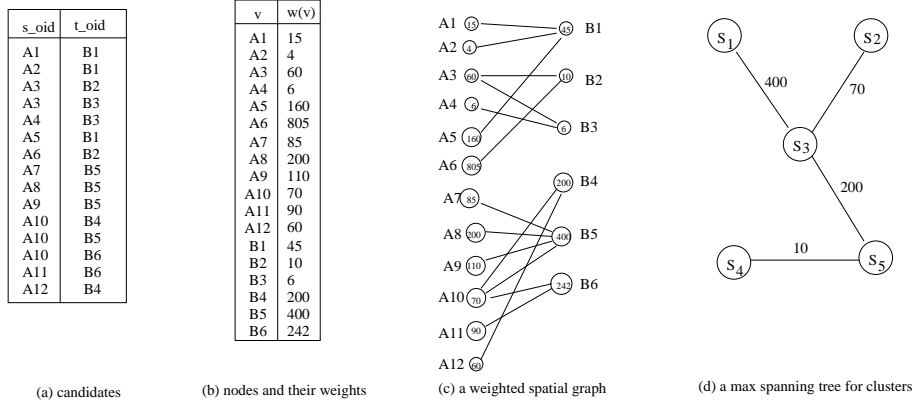


Figure 5: An extended example of a filter and its WSJ graph.

using *MERGE*, the non-zero element 270 (the joinable object acrossing two clusters) is merged into the upper-left submatrix. The resultant submatrices become the following (iii) and (iv):

objl	17	10	9	8	16	7	6	15
17l	242	332	312	0	0	0	0	0
10l	332	90	0	0	0	0	0	0
9l	312	0	70	0	470	0	0	270
8l	0	0	0	110	510	0	0	0
16l	0	0	470	510	400	600	485	0
7l	0	0	0	0	600	200	0	0
6l	0	0	0	0	485	0	85	0
15l	0	0	270	0	0	0	0	200

(iii)

objl	3	2	5	13	14	0	4	12	1	15	11
3l	6	0	0	0	12	0	0	0	0	0	0
2l	0	60	0	70	66	0	0	0	0	0	0
5l	0	0	805	815	0	0	0	0	0	0	0
13l	0	70	815	10	0	0	0	0	0	0	0
14l	12	66	0	0	6	0	0	0	0	0	0
0l	0	0	0	0	0	15	0	60	0	0	0
4l	0	0	0	0	0	0	160	205	0	0	0
12l	0	0	0	0	0	60	205	45	49	0	0
1l	0	0	0	0	0	0	0	49	4	0	0
15l	0	0	0	0	0	0	0	0	0	200	260
11l	0	0	0	0	0	0	0	0	0	260	60

(iv)

objl	6	8	7	16	15	17	10	9
6l	85	0	0	485	0	0	0	0
8l	0	110	0	510	0	0	0	0
7l	0	0	200	600	0	0	0	0
16l	485	510	600	400	0	0	0	470
15l	0	0	0	0	200	0	0	270
17l	0	0	0	0	0	242	332	312
10l	0	0	0	0	0	332	90	0
9l	0	0	0	470	270	312	0	70

(v)

Continuing to apply BEA with SHIFT on (iii), we get a matrix as in (v), in which the dividing point is 4, the position between 400 and 200 along the main diagonal of the matrix. This time the non-zero element, 470, in the upper-right submatrix is merged into the lower-right submatrix thus the two resultant submatrices are shown as in (vi) and (vii).

objl	6	8	7	16
6l	85	0	0	485
8l	0	110	0	510
7l	0	0	200	600
16l	485	510	600	400

(vi)

objl	15	17	10	9	16
15l	200	0	0	270	0
17l	0	242	332	312	0
10l	0	332	90	0	0
9l	270	312	0	70	470
16l	0	0	0	470	400

(vii)

objl	17	10	9	16	15
17l	242	332	312	0	0
10l	332	90	0	0	0
9l	312	0	70	470	270
16l	0	0	470	400	0
15l	0	0	270	0	200

(viii)

objl	17	10	9
17l	242	332	312
10l	332	90	0
9l	312	0	70

(ix)

objl	9	16	15
9l	70	470	270
16l	470	400	0
15l	270	0	200

(x)

Since the buffer cost of (vi) is 795, less than the given size $p=820$, the recursive execution of the algorithm stops and the first cluster (cluster No.1) is worked out as $\{A7, A9, A8, B5\}$ with candidates $\{(A7, B5), (A9, B5), (A8, B5)\}$. Further decomposition to (vii) produces (viii), in which the dividing point lies between 70 and 400 along the main diagonal. Thus, after merging, two submatrices (ix) and (x) are produced which make cluster No.2: $\{A10, A11, B6\}$ of buffer cost 402, with candidates $\{(A10, B6), (A11, B6)\}$, and cluster No.3: $\{A10, B4, B5\}$ of buffer cost 670, with candidates $\{(A10, B4), (A10, B5)\}$.

Similarly, by applying BEA with SHIFT to (iv), we get matrix (xi) with its dividing point in position 2. The non-zero element 70 is merged to the lower-right submatrix this time, thus we obtain (xii) and (xiii). As both of their buffer costs are less than p , the decomposition stops and cluster No.4: $\{A6, B2\}$ with unique candidate $\{(A6, B2)\}$, of buffer cost 815, and cluster

No.5: {A1, A2, A3, A4, A5, A12, B1, B2, B3, B4} with candidates {(A4, B3), (A3, B3), (A12, B4), (A5, B1), (A1, B1), (A2, B1), (A3, B2)}, of buffer cost 566, are produced.

obj\	5	13	14	11	15	12	4	0	1	3	2	obj\	5	13	obj\	14	11	15	12	4	0	1	3	2	13	
5	805	815	0	0	0	0	0	0	0	0	0	5	805	815	14	6	0	0	0	0	0	0	0	12	66	0
13	815	10	0	0	0	0	0	0	0	0	0	13	815	10	11	0	60	260	0	0	0	0	0	0	0	0
14	0	0	6	0	0	0	0	0	0	0	0	15	0	260	200	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	60	260	0	0	0	0	0	0	12	0	0	0	45	205	60	49	0	0	0	0	0	0	0
15	0	0	0	260	200	0	0	0	0	0	0	4	0	0	0	205	160	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	45	205	60	49	0	0	0	0	0	0	60	0	15	0	0	0	0	0	0	0	0
4	0	0	0	0	0	205	160	0	0	0	0	1	0	0	0	49	0	0	4	0	0	0	0	0	0	0
0	0	0	0	0	0	60	0	15	0	0	0	3	12	0	0	0	0	0	0	6	0	0	0	0	0	0
1	0	0	0	0	0	49	0	0	4	0	0	2	66	0	0	0	0	0	0	0	0	0	60	70	0	0
3	0	0	12	0	0	0	0	0	0	6	0	13	0	0	0	0	0	0	0	0	0	0	0	70	10	0
2	0	70	66	0	0	0	0	0	0	0	60															

(xi)

(xii)

(xiii)

As a comparison, when using the algorithm without SHIFT, the clusters produced by the algorithm are different. Not only more clusters (6 clusters, not 5) are produced which means more recursive executions are needed, but also the sizes of resultant clusters become 815, 802, 795, 306, 300 and 70 which indicate a bad distribution of cluster sizes (thus a bad workload balancing) of the resultant clusters. Also, while the number of duplicated objects among clusters produced by BEA with SHIFT is 4 (A10, B2, B4, B5), it is 6 by BEA without SHIFT. \square

6 Maximum Overlapping Order Among Clusters

As mentioned before, after the desired subfilters (i.e. the clusters of objects) are produced, the objects from both data sets in the subfilters are fetched into the buffer for processing. Then all the data in the buffer are thrown away before a new subfilter is processed. In many cases, however, some objects may cross two or more subfilters in a partition. These objects can be used for processing the following subfilters before they are thrown away from the buffer. Generally, because of large object sizes, utilising the existing objects in the buffer will minimise the overall fetching costs because they need not be fetched into the buffer again. In this section, we discuss this issue by proposing a *maximum overlapping order* method among the resultant subfilters in a partition. We aim to create a maximum overlapping order among the subfilters so that any two adjacent subfilters share as many objects as possible. When the spatial join operations are processed subfilter by subfilter in this order, those overlapping objects shared by the current subfilter and the next need not to be fetched into the buffer again because they are already there. In this way, the overall fetching cost can be decreased.

6.1 Hypergraph modeling

A hypergraph [5] $H = (V, E)$ consists of a set of vertices ($V = \{v_1, v_2, \dots, v_n\}$) and a set of hyperedges ($E = \{e_1, e_2, \dots, e_m\}$) satisfying $\cup_{1 \leq i \leq m} e_i = V$. A hypergraph is an extension of a graph in the sense that each hyperedge can connect more than two vertices. In our model, the set of vertices V corresponds to the spatial object set, and each hyperedge $e \in E$ corresponds to an object cluster resulted from Algorithm *Partition*. By defining $w(v) > 0$ for each vertex $v \in V$, and $w(e) = \sum_{v \in e} w(v)$ for each $e \in E$, H can be extended as a weighted hypergraph, written as $H = (V, E, w)$. A hypergraph $H = (V, E)$ can be considered as a special case of a weighted hypergraph where $w(v) = 1$ for all $v \in X$.

In our method, the relationship between a set of spatial objects and their partitioned clusters are mapped into a hypergraph. A hyperedge represents a relationship (affinity) among spatial objects in a cluster, and the weight of a hyperedge reflects the strength of this affinity. The overlapping objects between two different clusters are the intersection of the corresponding hyperedges. And the weight of the intersection reflects the sum of the size of those overlapping objects. So a *maximum overlapping (MO) order* among clusters can be defined as a sequence $(e_{i_1}, e_{i_2}, \dots, e_{i_m})$ which makes $\sum_{k=1}^{m-1} \bar{w}(e_{i_k}, e_{i_{k+1}})$ reach the maximum among all permutations of $E = \{e_1, e_2, \dots, e_m\}$ in a weighted hypergraph H , where $\bar{w}(e', e'') = \sum_{v \in e' \cap e''} w(v)$.

6.2 Finding a MO order in a hypergraph

Obviously, there must exist a MO order for each (weighted) hypergraph. However, the problem of finding a MO order in a (weighted) hypergraph is *NP*-complete [28]. Based on this fact, trying to find an efficient algorithm for producing a *MO* order in a hypergraph should receive a low priority. Instead, we developed an efficient algorithm that gives us an *approximate MO* (or *AMO*) order of relative “high” overlapping weights.

For simplicity, we assume that the weighted hypergraph $H = (V, E, w)$ is a connected one, and the mapping from H to an edge-weighted graph $G = (E, E_H, w_H)$ has been completed, i.e., $(e', e'') \in E_H$ in G if and only if $w_H(e', e'') = \sum_{v \in e' \cap e''} w(v) \neq 0$. In this case, the algorithm to find an AMO order consists of three main steps:

Algorithm 2. *MO-order*(G)

1. Find a maximum cost spanning tree T of G ; /*a maximum cost spanning tree T is a spanning tree in which $\sum_{(u,v) \in E_T} w_H(u, v)$ reaches the maximum among all spanning trees of G , where E_T is the edge set of tree T); */
2. Conduct a *depth-first search* of T ; and
3. Construct a *MO* order according to the output of 2.

The complexity of the algorithm is $O(m^2)$. Let us use an example to illustrate how it works.

Example 3 Continue to consider the Example 2 shown in Figure 5, in which the the buffer size is $p=820$. After data partitioning, five subfilters s_1, s_2, \dots, s_5 are produced as shown in Example 2. Now take $H = (V, E, w)$ with $V = \{A1, A2, \dots, A12, B1, B2, \dots, B6\}$, $E = \{s_1, s_2, \dots, s_5\}$ and for each $v \in V, w(v)$ is shown in Figure 5(b). The weight function $\bar{w}(s_i, s_j)$, indicating the size of overlapping objects between s_i and s_j ($1 \leq i, j \leq 5$), is obtained as in the following table

	s1	s2	s3	s4	s5
s1	795	0	400	0	0
s2	0	402	70	0	0
s3	400	70	670	0	200
s4	0	0	0	815	10
s5	0	0	200	10	566

Now let us look at the execution of *Mo-order*. At the first step, edges (s_1, s_3) , (s_3, s_5) , (s_2, s_3) and (s_4, s_5) are added in sequence to the maximum spanning tree T with their weights 400, 200, 70 and 10, respectively. We assume that a depth-first search of T begins at vertex s_1 in Step 2. The search results in a cluster sequence $(s_1, s_3, s_5, s_4, s_2)$ with total overlapping weight

610 (note that the intersection of s_4 and s_2 is empty thus the overlapping weight between s_4 and s_2 is 0). This order is taken as the final *AMO* order in Step 3. It is not difficult to prove that the order is actually the *MO* order for this example.

If the spatial join operations are processed in this *AMO* order, 400 out of 670 of object volumes in cluster s_3 need not be fetched into the buffer again because they are already there (fetched by cluster s_1 , the preceding cluster in the *AMO* order). Similar situations occurs for s_5 and s_4 . In other words, only those of objects shared by clusters s_4 and s_2 (of volume 70) have to be fetched again. All other objects just need to be fetched once. On the contrary, if we adapt the strategy that all data are thrown away before a new subfilter is processed, the total volume of objects that need to be fetched twice by different subfilters will be 680 instead of 70. \square

In the above example, the *AMO* order generated by *Mo-order* is exactly the same as the *MO* order in H . In general cases, however, by selecting a different starting vertex in *Step 2*, the *AMO* order generated by *Mo-order* may differ from a *MO* order, and the weight sum of the *AMO* order may be less than that of the *MO* order for the given hypergraph. But we can prove that the weight of the *AMO* order produced by the algorithm is always greater than half of the weight of the corresponding *MO* order [28]. In many cases, this bound is quite acceptable, since it was obtained by using an efficient algorithm.

7 Experiments

Previous sections have theoretically illustrated the ability of the proposed declustering method for spatial join processing. Now we focus on detailed experimental evaluations with comparison to some existing fetching methods. To minimise disk access cost of spatial processing, the cost of fetching spatial objects from the database should be reduced. Due to slower disk access, decreasing the number of accesses will contribute to minimise the cost of spatial processing. Therefore, partitioning a filter into subfilters so that each subfilter just needs to be fetched into memory once and only once should be a solution. Generally, a buffering strategy is adapted and a buffer which holds the objects to be joined is used for the join operations. Assume that the objects to be joined in the buffer are fetched into the buffer one by one. The buffer size and the average object size have much to do with the decrease of fetching cost. For a given average object size, if the buffer size is small, the fetching cost might be very high. If the buffer size is big enough to hold all objects, each object needs only to be fetched once thus the lower bound of the total fetching cost could be reached. However, the buffer size is often limited (in particular for large spatial objects). Therefore, it becomes important to utilise the limited buffer effectively. This can be achieved by grouping the candidates referencing common objects together. Our experiment shows that the algorithm *Partition* has displayed its ability for grouping the candidates.

Our simulation shows that candidate sequencing in the filter will affect the overall fetching cost. Some comparison has been done with several different sequencing methods. The first method is the *naive* one, that is, using the original sequences as supplied by the filter. The objects referenced by each candidate is fetched from the disk and stored in the buffer if it is not

already there when the candidate is processed. Once the buffer is full, an object in the buffer will be thrown away using some buffer replacement strategy, such as *LRU (Least-Recently-Used)*, *FIFO (First-in-first-out)* and so on. The second method used here is called a *simple* method. It sorts the filter by one column, for example, *F.s_oid*. As all the candidates referencing to the same *S* objects appear in the filter consecutively, each *S* objects will be fetched only once. Figure 6 shows the results from different groups of tests with buffer size $p=1000$. It gives an example of the comparison of some group fetching by several different fetching methods. In the figure, *P-M* means fetching objects group by group according to the output of algorithm *Partition* followed by *Mo-order*. The unit for *Fetching Time* is in *ms*. From the figure, we can observe that *P-M* method decreases the fetching cost for all group sizes. For any size of group, the *naive* method produces a relative higher cost of fetching, while the *P-M* results in a relative lower cost of fetching. Compared with that of *naive* and *simple* methods, *P-M* reduces about 30~35%, and 15~20% of the fetching costs, respectively.

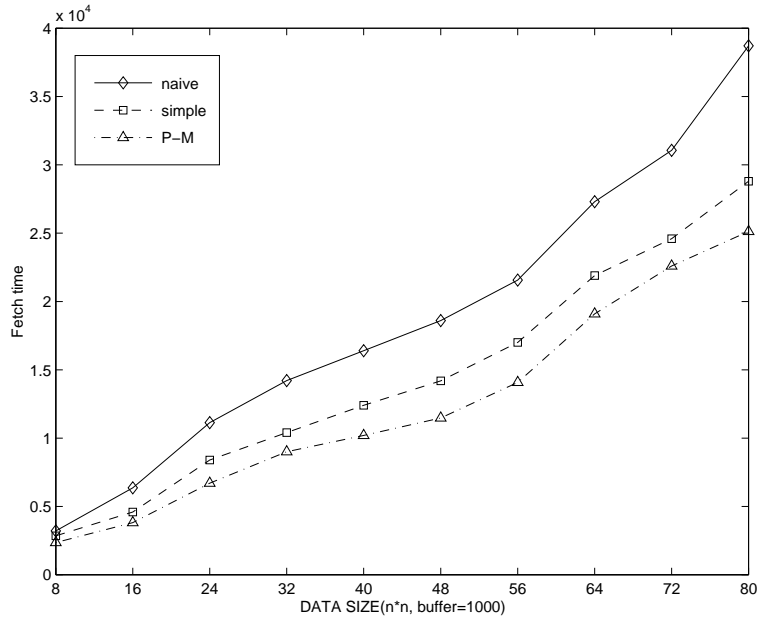


Figure 6: Fetching cost comparison of different strategy (fixed buffer size)

In the application and our experimental work, for each frequent set join operation we use *Partition* Algorithm to establish the corresponding cluster indices and use *Mo-order* Algorithm to generate an AMO order for these clusters before-hand. During the future join process, these clusters are formed according to the pre-calculated indices and fetched into the memory for conducting the join operation in the AMO order.

8 Conclusion

In spatial databases, the I/O cost for fetching objects from disk is very high because spatial data is often large in size. To reduce the cost effectively, properly sequencing the candidates to take maximum advantage of system buffer could be a solution. The uniform spatial data partition strategy fails to work effectively whenever the sizes of objects in realistic applications change

significantly. In this paper, we presented a matrix-based approach for spatial data declustering. Firstly, a graph model was developed to formalise the problem, and then a matrix-based data partitioning method was proposed for declustering the non-uniform spatial data. The clusters produced were also maximum-overlapping ordered. When inputting the clusters in this order for spatial joins, the I/O cost can be reduced significantly. This method works well for non-uniform as well as uniform spatial data. The experimental work has shown that 15-35% saving can be achieved when comparing with some other existing methods.

Acknowledgement: We would like to thank Miss Zhang Weiming for her help on the experiments and simulations.

References

- [1] D. Abel. SIRO-DBMS: A Database Tool Kit for Geographical Information Systems. *International journal of geographical Information Systems*, Vol. 3, No. 2, pp.103-116, 1989.
- [2] D. Abel, etc., Spatial Join Strategies in Distributed Spatial DBMS. *Proc. of fourth international symposium on Large Spatial Databases*, Portland, Maine, August, 1995, pp.348-367.
- [3] W. Aref and H. Samet. Optimisation Strategies for Spatial Query Processing. *Proc. of 17th Int. Conf. on Very Large Databases*. Barcelona, September 1991, pp.81-90.
- [4] S. Berchtold, D. A. Keim, H. P. Kriegel. The X-tree: An index structure for High-Dimensional Date. *Proc. of 22st Int. Conf. on Very Large Data Bases*, pages 28–39, 1996.
- [5] C. Berge. *Graphs and Hypergraphs*, North-Holland Publishing Company 1973.
- [6] Thomas Brinkhoff and Hans-Peter Kriegel and Bernhard Seeger, Efficient Processing of Spatial Joins Using R-trees. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp.237-246, 1993.
- [7] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Parallel Processing of spatial join using R-tree. *Proc. of 12th Int. Conf. on Data Engineering*, 1996.
- [8] G. Chartrand, O. R. Oellermann. International Series in Pure and Applied Mathematics, *Applied and Algorithmic Graph Theory*, McGraw-Hill, Inc., 1993.
- [9] Nicos Cristofides. *Graph Theory, An Algorithmic Approach*, AP Academic Press Inc. (London) Ltd., 1975.
- [10] R. H. Güting, Spatial Spatial Hash-Join. *Introduction to Spatial Database Systems*, The VLDB J. Vol. 3, No. 4, pp.357–399, 1994.
- [11] M. L. Lo and C. V. Ravishankar Spatial Hash-Join, *SIGMOD'96*, pp.247-258.
- [12] H. J. Lu and B. C. Ooi and K. L. Tan. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, 1994.

- [13] S. Navathe and S. Ceri. A vertical partitioning algorithms for database design. *ACM Trans. on Database Systems*, 9, 1984.
- [14] B. C. Ooi. Efficient Query Processing in Geographic Information Systems. *Lecture Notes in Computer Science (471)*, 1990.
- [15] J. Orenstein and F. A. Manola, PROBE spatial data modeling and query processing in an image database application. *IEEE Trans. Software Eng.*, Vol. 14, No. 5, pp.611-629, 1988.
- [16] M. Tamer Ozsü, Patrick Valduriez. *Principle of Distributed Database Systems*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1991.
- [17] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp.259-270, 1996, Montreal, Canada.
- [18] H. Samet, *Applications of Spatial Data Structures*. Addison-Wesley Pub. Co., Inc, 1990.
- [19] H. Samet and Walid Aref, Spatial Data Models and Query Processing. *Modern Database Systems (by W. Kim)*. Addison-Wesley Publishing Company, Inc, 1995.
- [20] S. Sevcik, N. Koudas. Filter trees for Managing Spatial Data over a Range of Size Granularities. *Proc. of 22st Int. Conf. on Very Large Data Bases*, pages 16–27, 1996.
- [21] P. Valduriez. Join indices. *ACM Trans. Database Systems*, Vol. 12, No. 2, pp.219-246, 1987.
- [22] Y. Zhang and M. Orlowska. On fragmentation approach for distributed database design. *Information Sciences: Applications*, Vol. 1, 1994, pp.117-132.
- [23] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH:An Efficient Data Clustering Method for Very large Databases. *SIGMOD Record*, pp103-114, June 1996.
- [24] Y. Zhang, J. Xiao, X. Zhou. A Declustering Algorithm for Minimizing Spatial Join Cost. *LNCS 1276: Proc. of 3th Int. Conf. on Computing and Combinatorics (COCOON'97)*, Shanghai, Springer-Verlag, 1997, pp.363-372.
- [25] Y. Zhang, J. Xiao and A.J. Roberts. Parallel Algorithms for Spatial Data Partition and Join Processing. *Proc. of the third IEEE Int. Conf. on Algorithms and Architecture for Parallel Processing*, Melbourne, World Scientific Publ., Dec. 1997, pp703-716.
- [26] X. Zhou and M. E. Orlowska. Handling Data Skew in Parallel Hash Join Computation Using Two-Phase Scheduling, *Proc. of the IEEE Int. Conf. on Algorithms and Architecture for Parallel Processing*, April 1995, pp.527–536.
- [27] X. Zhou and D. J. Abel and D. Truffet, Data partitioning for parallel spatial join processing, *LNCS 1262: Proc. of the 5th Int. Symp. on Spatial Databases (SSD'97)*, Berlin, Germany, Springer-Verlag, 1997,pp.178-196.
- [28] J. Xiao and Y. Zhang. Maximum Overlapping Order and its Applications in Spatial Join Processing. *Technical Report SC-MC-9820, University of Southern Queensland, 1998*.