Spot[1]: Distance based join indices for spatial data

Tsin Shu YEH
University of Versailles: PRiSM Laboratory
45, av. des Etats-Unis
78035 Versailles
(33) 1 39 25 40 48
Tsin-Shu.Yeh@prism.uvsq.fr

ABSTRACT

Recently, distance based spatial queries have become more and more important for spatial data analysis, data-mining, and geo marketing. Such queries capture metric relationships between spatial entities. They need the computation of spatial join with a metric distance criterion, which is extremely expensive in I/O disk cost. Indeed this is equivalent to make a cartesian product between all entities. Processing these queries is a challenging task due to the huge amount of spatial data and to the conceptual nature of the problems.

Distance based spatial queries is a common problem in Geographical Information Systems (GIS). Several approaches have been presented in recent years, almost all of them based on structures called join indices. Such structures transform the complexity of the spatial join into a traditional problem that can be processed with basic operators of relational data model. However, a join indices file contains the cartesian product of couples of spatial objects with their distances. Consequently, it is properly working only on few amounts of data.

In this paper, we propose a distance based join indices approach for spatial data in order to reduce the cost of storage and processing. The idea is to store only distances of spatial data linked to virtual points called "spots" which avoids storing all combinations of distances two by two. A spot is a roundup of a set of spatial objects. We propose a data structure on disk that uses a clustering method to reduce the disk I/O during the join computation. In theory, and confirmed by empirical studies, this approach outperforms the traditional distance join method by at least one order of magnitude, especially when the data set is large.

Keywords

GIS, join indices, spatial data processing, spatial operators.

1. INTRODUCTION

Many applications require the processing of spatial distance join. Spatial join is the search of couple of data that fulfills a given criterion of distance. Spatial distance queries may play an important role in understanding spatial data and in capturing intrinsic relationships between spatial and/or non-spatial data. For instance, the spatial data mining may use distance based spatial join to discover new knowledge as in our application[1] described in section 2.3. Distance based joins are so critical that commercial softwares as ArcView integrate such operator.

However, as all distances of entities two by two might be interesting, many operators lack efficiency because they calculate the cartesian product of entities. This cartesian product makes it possible to compute distances and to select couples of points whose distance satisfies a criterion. Such cartesian product is very expensive in running time, which is a limit for huge data volumes. That is why algorithms implemented into most commercial software are restricted to compute distances either between entities and a fixed point, or between entities but for very small volumes of data (typically too small to be practical in most cases). Consequently, without an adapted data organization, this involves an important number of I/O.

1.1 Related work

For distance queries, we could outline mainly two categories of works. The first is solving the specific problems of near-neighbor finding. These algorithms work for one fixed-point criterion and they select points which distance to a given point is below some value. This approach is used for seeking objects (images, sounds,...) that are similar to a given pattern. Efficient solutions use hierarchical data organization and typically have an O(log(n)) cost as the M-tree algorithm [2]. However, these approaches are inadequate for selecting all couples of points within a given distance.

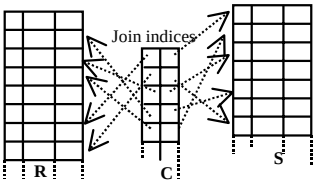

Figure 1: Join-indices principle Figure

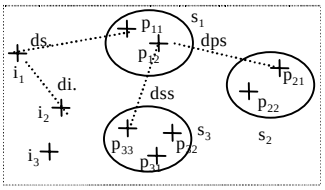

Figure 2: Distances to be considered

Our problem takes place in the second category, which is computing two by two every distance between all the points of two data sets. In order to reduce the prohibitive cost due to cartesian product, various approaches based on the join-indices method have been proposed. The join-indices method has been

ACM GIS '99 11/99 Kansas City, MO USA
© 1999 ACM 1-58113-235-2/99/0011 … $5.00

[1] This work is partly supported by the PSIG'98 "Extraction de connaissances en accidentologie routière".

proposed by P.Valduriez [13] as a valuable technique for increasing speed of the query evaluation in Data Base Management System (DBMS) [12,9]. The efficiency of this method has been demonstrated on the distance based join for a given selectivity [9]. It reduces the cartesian product cost by transforming it into a linear scan of disk blocks. This approach materializes (i.e. precomputes) the functional joins in the form of generalized join indices. This principle consists in storing in a collection of tuples C (cf. figure 1) information concerning spatial relationship `sr` between two entities. Each tuple of C contains an identifier to a tuple of the set R and an other identifier to a tuple of the S set. Both R and S sets contain respectively the spatial description of entities. Thus, this method is interesting when spatial relationship `sr` is very expensive to calculate as a geometrical computation [6]. However, if the selections of materialized join indices exhibit a low selectivity value (i.e. many output tuples are produced), the performance gain of these algorithms is limited [5,11].

Unfortunately, in the case of spatial relationships based on a distance criterion, it requires to store all combinations of pairs of points issued of the cartesian product with their distances. So, due to the cartesian product, a huge data volume has to be stored. Scanning such a collection containing join-indices requires important disk I/O and therefore limits performances of this approach. An improvement of this method consists in using a B-Tree structure to store these distances as suggested in [5,9]. But, due to the extremely high data volume, the performance problem remains.

In [9], besides the traditional approach that consists in storing all the distances between points, two other approaches have been studied. The first approach distributes couples of points in several files according to their distance. Each file includes pairs of points with a distance included in an interval. These intervals are defining collections that are two by two disjointed. For a query, only relevant files are scanned. The second approach defines several classes of distances. Each class defines a scale in which a distance has a meaning. For example if one is interested by a distance at continent scale, distances of entities in a village are no longer significant. For a query at a scale, solutions are the union of all significant classes of distances. However, the data volume remains a problem.

In [7] an approach is proposed to calculate the spatial distance join and distance semi-join. Incremental algorithms are presented for computing these operations, which can be used in a pipelined fashion, thereby obviating the need of waiting for their completion even when only a few tuples are needed. The algorithm is based on a queue and using R-trees. Computing the entity pairs situated to a certain distance requires recursive scanning of the R-Tree. The queue guides the course of R-tree and is also used as a cache memory. The incremental algorithms outperform non-incremental approaches by an order of magnitude if only a small part of the result is needed, while the penalty for the incremental processing is modest if the entire join result is required. But the number of I/O remains a problem when computing all the distances fulfilling a criterion.

1.2 Our Contribution

In this paper, we propose a new approach for distance based join-indices. Storing an intermediate data named "spot" is the key idea for reducing the data volume. Conceptually, a spot is a virtual circle on a 2D map with the purpose to delimit a subset of points. Each spot clusters some points and is stored in an individual disk block. We store only the relationship from these points to the spot with a linear cost. Thus, this approach avoids storing all combinations two by two of distances between points.

In such a method, speed increasing is due to complementary information in our proposed data structure. This information describes the distance between these two spots. With this information, a distance query loads in main memory only spot disk blocks that fulfill the distance criterion. For example, if a query searches points which distance is less than d_{max}, then if two spots are separate with a $d-(r_1-r_2) \geq d_{max}$ they is no chance that spots contain points respecting the distance criteria. Here, r_1 and r_2 are spot radius, and d is the distance that separates these two spots. With this key idea, we save a number of disk I/O compared to a traditional approach during the join processing.

1.3 Outline of Paper

This paper is organized as follows. We first introduce in section 2 an algorithm for seeking spots on a map. Then, this result is used in section 3 to be stored in an adapted storage structure for efficiently computing a join indices collection. This join indices computation is presented in section 4. Sections 3 and 4 contain our main contribution. Section 5 presents performance study of our approach. Finally our conclusion is presented in section 6.

2. SPOTS DETECTION

This section describes the algorithm for determining spots from a spatial data set. It's not our main contribution, but it is necessary for the next step. Although more general clustering algorithms (e.g. CLARANS) are existing, our spot search has specific constraints. This algorithm could work with O(n log n) I/O based cost to find spots in a map independently of data volume. The result of this step is a set of spots and a set of isolated points.

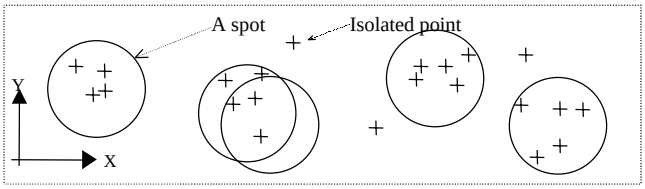

Figure 3: A map with spots

The problem is to group a set of points that could be contained in a spot. The algorithm input is a map which contains a number of points too high for been loaded in main memory as a supposition. Moreover, a map might contain a non-uniform distribution. For example, cities bring together generally more geographical entities than countryside. This increase in density involves an increase in points quantity to be loaded into memory. Consequently it results in a non-uniform distribution of spots in space. Thus, for a given location, the number of spots could be more than one (spots are overlapped) as illustrated in figure 3.

This clustering algorithm is characterized by three parameters. The first one is the maximum number of points in a spot. This parameter is chosen with the consideration of storage structures described in section 3. The set of spot points has to be placed in only one disk block in order to optimize the disk I/O during the query execution. The second parameter is the minimum number of points that could be contained in a spot. To the extreme, a spot

that contains one or two points is not worthy to be considered. Indeed, a spot involves an over cost of several I/O during the data access. When the radius increases, the number of points in a spot increases. However the imprecision increases during the data access and involves to load useless disk blocks as shown in the section 4.

2.1 Algorithm principle

The algorithm loads as and when required points in main memory. For this part, it is processing in two steps. First, the algorithm sorts the whole of the points in the data set following the X and Y coordinates.

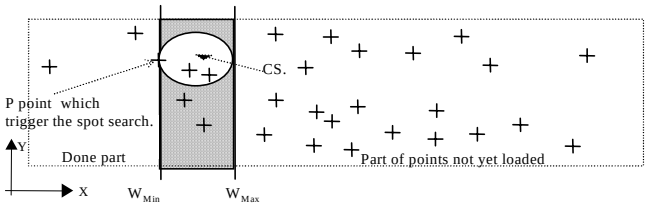

Figure 4: algorithm principle

```
1. Input: data set s, threshold_min, threshold_max
2. Output: a collection r of points or spots
3. mem ← {} ; r ← {} ; W_Min ← 0; W_Max ← 0;
4. While ! eos(s) do
5.  While !eos(s) and card(mem)>sizeof(mem) and
    (W_Max-W_Min < 2*SR) do
6.    Read s, pt
7.    mem ← mem + {pt}
8.    W_Max ← pt.x
9.  Wend
10. m <- Min(mem); spot ← Calc_Spot(m) ; s ← {}
11.  foreach p in mem do
12.   if p∈spot and card(s) < threshold_max then s
    ←s+{p}
13.  done
14.       if Card(s) > threshold_min then
15.           mem ← mem - s
16.           r ← r U { ("spot", s)}
17.       else
18.           mem <- mem - m
19.           r ← r U { ("isolated point", m)}
20.       end if
21.       W_Min ← min(mem).x
22. wend
23. return r
```

Figure 5: Spot search algorithm

For the second step, the algorithm works on a virtual window constituted by two vertical lines thereafter named W_{Min} and W_{Max}. These two lines are sliding down the X-axis in proportion of the algorithm progression. All points contained in the window are in the main memory. The window is slightly wider than twice the Spot Radius (SR). This allows to have in memory all needed points for constructing spots.

For the loading of points in memory, according to figures 4 and 5, when a point in the data set reaches the W_{Max} line, it is loaded in main memory (lines 6,7). In proportion of the loading of points, the W_{Max} line goes to the right. W_{Max} has the same coordinate as the X coordinate of the loaded point (line 8). Since points are sorted following the X-axis this is equivalent to make a sequential read of disk blocks. Thus, the number of I/O is equal to the number of blocks of the data set.

For the W_{Max} boundary, it is possible to load points until the main memory is saturated. This view is not useful in our case for two reasons. The first one is that data read from disk are buffered by the operating system and therefore need a minimum number of disk I/O. The second reason is that the search of spot points in memory is linearly proportional to the number of loaded points.

However, concurrently and during the former processing, the W_{Min} line slides to the right. When a point P from the windows reaches this line (line 10), the algorithm calculates the point CS which is placed at a distance SR of the point P slid in the right direction of the X axis, as illustrated in figure 4. The set of points included in this spot is then analyzed (lines 11-13). There are two cases. If the spot contains a number of points higher than a fixed number, all points contained in this spot constitute a new spot (lines 14-16). This new spot is generated (line 16) and the points included in this spot are taken from the window and therefore from the memory (line 15). In the other case (lines 17-19), the point P is an isolated point. It reaches then the left of the W_{Min} line (line 21). Hence a result is produced with this point (line 19) and it is then taken from the virtual window (line 18).

2.2 Dynamic adjustment of the window size

As the main memory size is limited to the strict necessary for the spot calculation, it might involve a saturation problem. This size is fixed using two criteria: the density contained in the windows at a given time, and the spot radius. When the radius increases, the number of points that a spot will be able to contain will increase, and the main memory size will be bigger.

So, to solve this problem of memory saturation, our strategy consists in reducing automatically the window width when the density of point increases. In order to do that, when the main memory allocated to the window saturates (second test, line 5), the W_{Max} line stops progressing. As concurrently the W_{Min} line continues to progress (line 21), the window width decreases. The search of spots is made of a spot radius equal to half of the window width. As soon as the W_{Min} line takes points from the window following the above described mechanism (line 21), the window size grows larger again to reach a width equal to twice initial SR (line 5).

2.3 An example of car crash survey

The figure 6 illustrates an example of spots seeking applied to a car crash survey in the district of Lille in France. Each point is an accident location. The data set contains more than 30000 points.

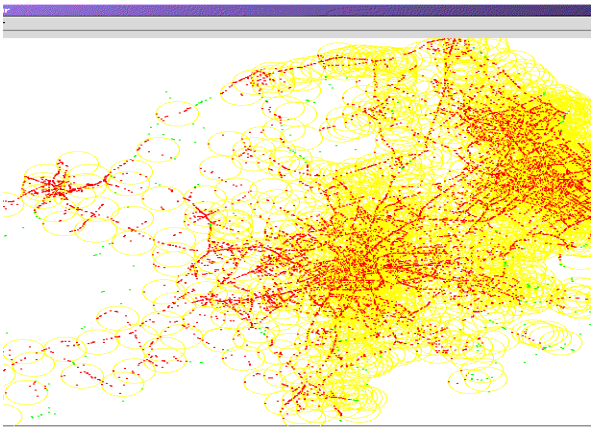

Figure 6: An example of car crash survey

3. THE STORAGE DATA MODEL

This section describes a storage data structure for spots. This Spot Data Structure (**SDS**) is used to build join indices as described in the next section. This physical representation has to store spots and isolated points computed during the previous step. The problem is to find an optimum and adapted representation that minimizes the disk I/O and increases the processing speed while computing join indices.

3.1 Computing distances

The most general case is to consider the seeking distances of entities issuing from two data sets. Seeking distances of entities in a same data set is a particular case. However, to facilitate the illustration, we have represented on the figure 2 the data belonging to the two data sets. Each point belongs to one of the two data sets (in the same way for spot represented by circles).

As our data structure includes two levels, we need to consider several categories of distances. By taking combination of points and spots, we have four categories that are necessary to be stored. Each category can contain distances of type:

- **IxI** for the couples of isolated points. These distances are the cartesian product between all isolated points. The `di` distance in figure 2 illustrated an example.

  Computed distances are:
  $d_{IxI}(I_1,I_2)=\{dist(i_1,i_2)|i_1\in I_1 \wedge i_2 \in I_2\}$ where I_1 (resp.I_2) are isolated points from the first (resp. second) set.

- **IxS** for the couples from the cartesian product between every isolated point and points from each spot. This kind of distance is illustrated by the line `ds` in figure 5.

  All calculated distances are: $d_{IxS}= D(I_1,S_2) \cup D(I_2,S_1)$ with: $D(I,S)=\{dist(i,p)|\forall s \in S,\ i \in I \wedge p \in s\}$

- **SxP** for the couples of points belonging to different spots. This is illustrated by the `dps` line in the figure. Computed distances are : $d_{SxP}(S_1,S_2)=\{ dist(p_1, p_2) \mid \forall\ s_1 \in S_1, \forall\ s_2 \in S_2, p_1 \in s_1 \wedge p_2 \in s_2\}$ where S_1 and S_2 are two spots sets.

- **SxS** for the couples between spots (`dss` in the figure). During the join indices construction, these distances are used by algorithm to eliminate SxP distances computation between points of the two spots (section 4.1).

  All calculated distances are: $d_{SxS}(S_1, S_2)=\{dist(s_1, s_2) \mid s_1 \in S_1 \wedge s_2 \in S_2\}$. S_1 and S_2 are two sets of spots.

3.2 Physical representation

We do not store directly these data according to these 4 categories. Indeed, due to the cartesian product between all these data, the produced volume will be too huge.

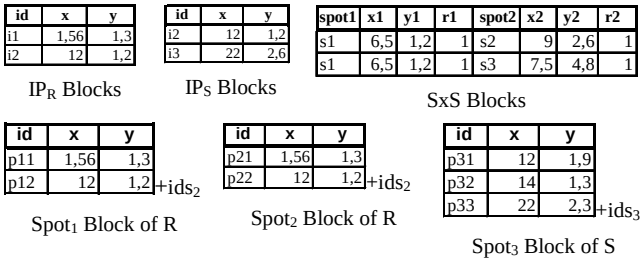

Figure 7: Physical representation for one SDS

For the SDS, only identifiers [8] and information about spots locations, points contained in spots, and isolated points are necessary. To reduce the storage cost, three categories of storage structure are taking place. Thus, according to figure 7, the storage structure consists of:

- **IP_R and IP_S blocks** that store isolated points of each collection. These data are used to calculate the IxI distances. For each point we store the identifier and the coordinates according to the structure of type `{(id,x,y)}`. `id` is an identifier that refers a tuple of R or S collection [8].

  It is more economical to store all isolated points in two collections of blocks instead of storing directly all IxI distances for two reasons:

  1. The quantity of isolated points is small. Parameters for the spot seeking algorithm described above trie to minimize the number of isolated points. At the end of this process, all of isolated points are taken in a few disk blocks. In the majority of cases, this allows to load all these blocks in main memory in one scan. The join is therefore executed in main memory.
  2. Computing `IxS` distances is necessary. As all the isolated points are loaded in the main memory, this accelerates the processing.

- **Spot block** that stores all points belonging to spots. This is used to compute `IxS` and `SxP` distances. Each spot block has a structure of type `({(id, x, y)}, ids)`. To each spot block we associate one spot identifier `ids`. For each point in the spot, `id` is the identifier of a tuple in R or S data set to which belongs the point. `x` and `y` are the absolute coordinates of the point on the map. The storage cost is proportional to the number of points in the map.

- **SxS blocks that** are used as join indices at the internal level to find couples of points, which are candidate to the evaluation of distances. It is stored in a structure of type `{(ids1, x1, y1, ids2, x2, y2)}` which contains the location of the two spots. `ids1` and `ids2` are spot identifiers. In our case, `ids` is a reference to a block that contains all points of the spot. The distance is calculated using the two coordinates. This structure stores the cartesian product of all possible associations between spots. It is a kind of join-indices used by the algorithm described in section 4.

3.3 Internal data constraints

The disks I/O are bottlenecks for algorithms implemented in DBMS. So, to decrease these I/O, we adopt two rules which optimize our algorithm execution.

For the first rule, all points belonging to one spot are stored in one block. The maximum number of points in a spot is chosen such as it can contain all of them in one block. This parameter is fixed in the spot calculation algorithm described in the section 2. The aim of this clustering rule is to enable to process in main memory the distance cartesian product of points contained in two spots with a minimum of blocks to load.

The second rule imposes for the SxS blocks to store all the couples of spots sorted by the first spot identifier (column spot$_1$ on figure 7). Each spot s_1 and s_2 in the couple (s_1,s_2)

necessitates to load the corresponding spot block for making the cartesian product. With this rule, we need to load only one time the spot block corresponding to the spot s_1 as it is explained in the next section.

4. COMPUTATION OF JOIN INDICES

This section describes the construction of join indices from SDS. In our spatial query processing, computation of distances based join is achieved in two steps. The first step builds the join indices matching a given distance criterion. The representation of the join indices is traditional [12, 13] as illustrated by C on figure 1. Each join index contains a couple of tuple identifier [8] allowing to seek the tuples of sets R and S.

Then, for the second step, these join indices are used to join the R and S collections that contain spatial values. The result is a collection of tuples produced from the concatenation of R and S collections. These two joins are made by traditional join operators [10]. That is why, we describe only the first step processing.

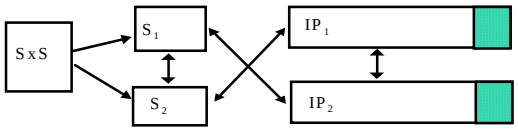

Figure 8: Buffers used for constructing join indices

All spatial queries based on distance are equivalent to distance computations that are included in range value [9]. Considering the low and high boundaries, we mainly discern three categories of query based distances:

1. The first is seeking couples of points which distance is included in an interval. For example, "which inns are located at a distance included between 100 and 350km from NY ?".
2. The second category is computing couples of points separated with a distance at most equal to a given value. For example, "Find all houses at most distant of 10km from the commercial centers". For this case, the inferior boundary of the interval is equal to the particular value 0.
3. The third category is computing couples of points that are distant of at least a certain distance. For example, "Find all houses that are distant of at least 10km from schools". For this case, the superior boundary is set to the $+\infty$ value.

4.1 Reducing I/O based cost

According to SDS, the speed increasing is partly due to data in SxS blocks, which contain the distance information between spots. These data indicate that some "spot blocks" can not contain points verifying the criterion of distance.

There exist two cases where it is not useful to load the spot blocks contents spots. Thus,:

1. for the superior boundary d_{max}, if the minimum distance $d-(r_1+r_2)$ between centers of the two spots is superior to d_{max}. r_1 (respectively r_2) is the radius size of the first spot (respectively the second spot) and d_{max} is the superior boundary given by the query. This means that all couples of points from the two spots are too distant to verify the maximum boundary. Then the spot blocks are not loaded.
2. for the inferior boundary d_{min} of a query, it imposes the constraint that the maximum distance $d+(r_1+r_2)$ has to be inferior to d_{min} value.

If one of these two conditions is verified, the spot blocks are not loaded in main memory. This allows to reduce appreciably the number of disk I/O.

4.2 Buffers for processing

To compute distances, the algorithm is based on five buffers (cf. figure 8). The SxS buffer loads SxS blocks which contain join indices between two spots s_1 and s_2. For each spot s_1, the corresponding spot block is loaded in the S_1 buffer. Then the block corresponding to the spot s_2 is loaded in the S_2 buffer.

On S_1 and S_2 buffers, the cartesian product of points is carried out according to the algorithm described below. This processing computes the d_{sxp} distances. On the right of the figure, the IP_1 and IP_2 buffers contain all isolated points. With these buffers, the algorithm computes d_{IxI} and also d_{IxS} distances as described in next section.

The IP buffers have a particular structure. Each buffer is divided in two parts. The first one loads a maximum of isolated points. This buffer has an important size, generally several hundred of disk blocks. The buffer size is big enough to be able to load all IP blocks. This is guaranteed in the majority of cases by the spot calculation algorithm that tries to minimize the number of isolated points. If it is not the case, the second part of the buffer representing one block size (in black on figure 8 is used as overflow buffer. Blocks beyond the n first ones are loaded successively from IP disk blocks of blocks in this part of the buffer. n represents the number of blocks of the first part of the buffer. This organization allows, when it is necessary, to read the overflow blocks and not to remove one of the blocks in the first part buffer. Indeed, the probability to use these first blocks is in our case very high.

4.3 Algorithm principle

```
1. Input : SDS, dmin, dmax
2. Output: join-indices in result collection
3.  d1_sxp,d2_sxp : array [] = {false,...,false}
4.  load buf_IP1 and buf_IP2
5.  r←{Dist_Join(buf_IP1, buf_IP2, dmin, dmax)}
6.  for each p(s1,s2)in SxS buffer do
7.   if spot_dist_is_possible(s1,s2,dmin,dmax) then
8.     if !load buf_S1xP(s1) then load buf_S1xP with s1
9.     if !load buf_S2xP(s2) then load buf_S2xP with s2
10.     r←rU{Dist_Join(buf_S1xP, buf_S2xP, dmin,dmax)}
11.   end if
12.   if  d1_sxp[s1] = false  then
13.     d1_sxp[s1] <- true
14.     if !load buf_S1xI(s1)then load buf_S1xI with s1
15.     r←r U {Dist_Join(buf_S1xI, buf_IP2,dmin,dmax)}
16.   endif
17.   if  d2_sxp[s2] = false  then
18.     d2_sxp[s2] ← true
19.     if !load buf_S2xI(s2)then load buf_S2xI with s2
20.     r←r U {Dist_Join(buf_S2xI, buf_IP1,dmin, dmax)}
21.   endif
22. done
23. return r
```

Figure 9: algorithm to compute indices join

The algorithm receives two boundaries d_{min} and d_{max} to evaluate the predicate. In the case where these boundaries would not receive a particular value described above, the cases to process depend on the category of join indices. Only the three categories above follow a test of predicate. Thus, in the case:

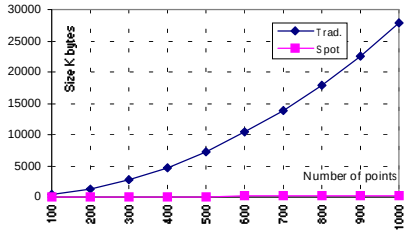

Figure 10: Data volume

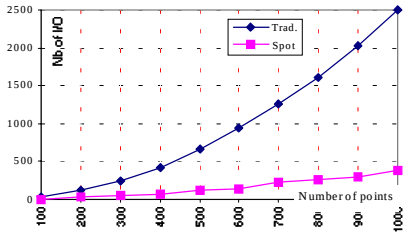

Figure 11: Join computation disk I/O

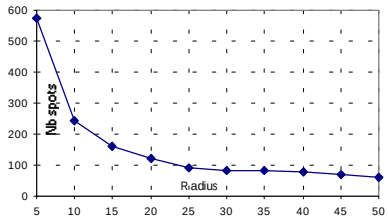

Figure 12: Radius influence on number of spots

- d_{IxI} distances (lines 4-5). For each distance d in d_{IxI} that respects $d \geq d_{min}$ and/or $d \leq d_{max}$ condition, join indices are produced by `Dist_Join` function. To calculate this result, the algorithm loads in the IP_1 and IP_2 buffers data from isolated points blocks (line 4). With these two buffers, the distance join is done in traditional way [10]. Calculations using these buffers are very rapid if the whole blocks of IP blocks may be taken in the main memory. In the opposite case, only the first part is accelerated. For the second part, the cost is equivalent to make a traditional nested-join on secondary memory [10].
- d_{SxP} distances (lines 7-11). The `SxS` buffer loads blocks, which contain the join indices from the two spots (S_1, S_2). For each join index, if the distance between the two spots (line 7) fulfils the conditions provided in section 4.1, points in both spots could verify the distance criterion. If this test is true, for the spot S_1, the corresponding block is loaded in the S_1xI buffer (line 8) when not previously loaded (see the second clustering rule). Then the block corresponding to the spot S_2 is loaded in the S_2xI buffer (line 9) so as to have loaded S_1xS_2. On these two last buffers, the cartesian product of points contained respectively in the two buffers is carried out (line 10).
- d_{IxS} distances (line 12-21). Distance joins are done by `Dist_Join` between S_1 and IP_2 buffers, and symmetrically between S_2 and IP_1 buffers. This distance calculation is carried out only one time for each spot (lines 12-13, 17-18). For the first time in the distance d_{IxS} calculation of spot, the buffer S_1 is loaded when not previously for the former distance calculation.

4.4 Algorithm cost

For the algorithm I/O based cost, we define the function [x] that returns the number of blocks for x and |x| that returns the number of different values for collection x.

The I/O cost of this algorithm is:

$C = C_{IxI} + C_{SxS} + C_{IxS} + C_{SxP} + C_{result}$ with:

- M_1 =max([IP_1]-k,0) (resp. M_2 = max([IP_2]-k,0)). k is the number of blocks for the first part of IP_1 buffer. M_1 determines the overflow blocks to read.
- m_1 = min(k, [IP_1]) (resp. m_2 = min(k, [IP_2])), for the number of blocks read in the first part of IP_1 buffer,
- C_{result}= [result], for writing the result on disk,
- C_{IxI} = $m_1+m_2+ M_1*M_2$ with the cost of the first load of the two IP buffers to compute the d_{IxI} distances,
- C_{SxS}= [S_1xS_2] , for the scan of S_1xS_2 blocks disk,
- C_{SxP}= $|S_1| +\delta|S_1xS_2|$ for the SxP. δ is the selectivity in seeking two spots with a probability to have points for which their distances fulfil criteria (line 7 figure 9). $|S_1|$ is the number of reads in S_1 buffer. Due to the second rule in section 3.3, this read is done at least for the lines 12-16 on figure 9. The minimum reads of $\delta|S_1xS_2|$ is superior at $|S_2|$ for lines 17-20 of the algorithm.
- C_{IxS}=$|S_1|* M_2+\delta|S_1xS_2|* M_1$ for the IxS. This cost is zero if all isolated points could fit in IP buffers.

With the defined buffers and due to the spot seeking algorithm, M_1 and M_2 are often set to 0. We can say that the cost is set to C = $m_1 + m_2+[S_1xS_2]+ |S_1| +|S_1| +\delta|S_1xS_2|$

5. EXPERIMENTAL RESULTS

This section analyzes the behavior of this new method. The comparison measures the relative run time, disk I/O based cost and data volume of the two methods.

The implementation has been carried out in the GeoLis GIS prototype. Code has been partially written in C++ and Tcl/Tk for data visualization interface. The benchmarks were executed on a lightly loaded PC workstation, a Pentium 200MHz with 32 MBytes of main memory running under FreeBSD.

5.1 Data volume measured

For the data volume, the comparison is done between the distance join indices using cartesian product method and our method. The comparison of these two methods is made on identical data sets.

The figure 10 compares the data volume to construct join indices between the traditional and spots methods. For these measures, the size of the spot radius is constant as well as the minimum loading threshold of a spot. For the traditional method, each tuple size is 32 bytes and the quantity of data evolves as the square of point number, which is obvious. For the spot method, each tuple size is more than 32 bytes. The curve shows that the data volume for the spot method is nearly proportional to the number of points. The gain is more profitable as the number of points increases. Contrarily to the traditional method, we notice that the size of join indices does not explode with the quantity as it is shown by the curve. The increase of spot indices size is essentially due to the data in SxS blocks which is due to the cartesian product of spots.

The figure 11 measures the number of I/O-based cost necessary for a spatial query. Each query selects couples of points with a distance included in an interval distance. For the traditional method, the query execution is traditional join algorithm [10] (nested loops). This collection size is equal to the square of the number of points that are stored. In the case of a selection on spot data, the number of I/O is perceptibly reduced due to the effect of

buffers and the organization of data in the spot method. All data are clustered in disk blocks to reduce the number of I/O.

5.2 Spot radius varying

The figure 12 shows the influence of the spot radius on the number of spots. The number of points in the calculation is 5000, the minimum loading threshold is 10. This measure shows that beyond a certain value, the efficiency of the radius decreases. Indeed, the number of spots decreases not appreciably. This is due to the density of points on a map. The number of spots is directly bound to the spot radius.

5.3 Processing time

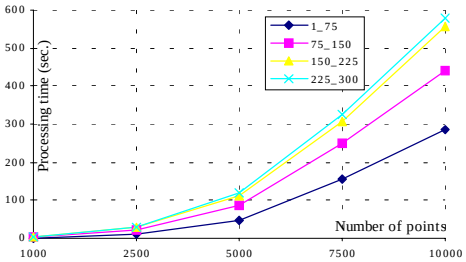

Figure 13: Processing time

The figure 13 shows the run time for a distance based query. These 4 curves correspond to variations of interval of distances of several queries. These intervals increase and show that queries are depending on interval of distance seeking. This result shows that weaker is the interval of distances, stronger is the selectivity of join indices. At the opposite, when the seeking interval increases, a more important number of points is involved by the seeking process. The method is more efficient when we are seeking couples of points distanced by a small value.

6. CONCLUSION

Computing distance based join is a fundamentally problem for spatial query and data analysis. However, as all distances - two by two - between entities can be interesting, it implicates to compute the cartesian product which necessitates an important number of disk I/O and generates a huge data volume.

In this paper we have developed a new data structure based on join indices. The key idea consists in using virtual circles named "spots". A spot is a virtual point that gathers a set of points. Only the relationships of points belonging to their virtual spots are stored in an adapted data structure. This avoids to store the cartesian product in a traditional join indices approach. The data structure clusters data in 3 types of blocks, which optimize the disk I/O-based cost.

Our prototype implementation and the performance study have shown that the algorithm is superior to traditional methods. Experimental results show that it optimizes the storage size while the penalty for the processing is modest but outperforms non-spot approach by an order of magnitude.

Currently, this algorithm is used in a research project for a spatial study about car accidents. More precisely this algorithm is used in the development of databases for data mining study. To compare, the computation time of a contiguity matrix made under ArcView GIS takes more than 1 hour. The same matrix calculated under our prototype takes about less than 8 minutes for 30000 points.

7. ACKNOWLEDGEMENTS

The author thanks K. Zeitouni for her help and discussions having led to this paper, and is grateful to a number of critical reviewers. A special thank to S. Chanchole for his valuable contribution in improving this paper.

8. REFERENCES

[1] T. Brinkhoff, H.P. Kriegel, R. Schneider, B. Seeger, Multi-Step Processing of Spatial Joins, in proceeding of Spatial SIGMOD 94. Intl. Conf. on Management of Data. Minneapolis. Vol. 23, Issue 2, June 1994.

[2] P. Ciaccia and M. Patella and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Space. Proceedings of the 23rd VLDB Conference Athens, Greece, 1997, pp. 426-435

[3] M.J. Egenhofer and J. Sharma, Topological Relations Between Regions in R2 and Z2. Advance in Spatial Databases, 5th International Symposium, SSD'93. pp 316-331. Singapore, June 1993, Springer-Verlag.

[4] M. Ester, H-P. Kriegel, J. Sander and X.Xu, A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In Proceeding of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, 1996, pp. 226-231.

[5] O. Günther, Efficient Computation of Spatial Joins, In Proceeding of Data Engineering, April 19-23, 1993, Vienna, Austria, pp 50-59.

[6] R.H. Güting, M. Schneider.,Realm Based Spatial Data Types : The ROSE Algebra. VLDB Journal, vol. 4, pp. 100-143, 1995

[7] G.R. Hjaltason and H. Samet, Incremental distance Join Algorithms for Spatial DataBases, Sigmod 98, Seatle. USA. Pp. 237-247.

[8] S.N. Khoshafian and G.P. Copeland, Object Identity. In Proc. of the ACM Conf. on Object-Oriented Programing Systems and Languages (OOPSLA), pages 408-416. Nov 86

[9] Wei Lu and Jiawei Han, Distance-Associated Join Indices for Spatial Range Search. Eighth International Conference on Data Engineering, Feb 2-3, 1992 Tempe, Arizona, pp. 284-292.

[10] D. Maier, The Theory of Relational Databases, Computer Science Press, 1983.

[11] P.O' Neil, and G.Graefe, Multi-tables joins through bitmapped join indices. SIGMOD Record, 24(3), 8-11, Sep 95.

[12] Doron Rotem, Spatial join indices, Proc. 7th Conf. Data Engineering, Kobe, Japan, 1991, 500-509

[13] P. Valduriez, Join indices. ACM Trans. on Database Systems, 12(2); 218-246, June 1987.