The Journal of GIS Association of Korea, Vol.7, No.2, pp.271-282, October 1999

다중-속성 색인 기법을 이용한 공간 조인 연산의 성능[1)]

황 병 연[2)]

Performance of Spatial Join Operations using Multi-Attribute Access Methods

Byungyeon Hwang

요 약

본 논문에서는 다중-속성 데이터와 공간 조인 연산을 효율적으로 수행하는 색인 기법인 SJ(Spatial Join) 트리를 제안한다. 또한, 다중-속성 데이터를 다루기 위한 기존의 다양한 알고리즘들을 계산 복잡도와 I/O 연산의 복잡도와 함께 설명한다. 우리는 이 논문을 통해서 제안된 SJ 트리가 기존의 데이터베이스 시스템에서 색인 기법으로 많이 사용되는 B-트리를 일반화한 것이라는 것을 보여준다. 이것은 SJ 트리가 기존의 대부분의 B-트리를 이용하는 저장구조에 쉽게 구현될 수 있다는 것을 의미한다. 공간 출력을 갖는 공간 조인 연산은 R-트리, B-트리, K-D-B 트리, SJ 트리에 대해서 성능평가를 수행한다. 성능평가 결과 제안된 SJ 트리가 점 데이터를 갖는 공간 조인 연산에 대해서 다른 색인 기법들보다 상대적으로 우수한 결과를 보여준다.

ABSTRACT : In this paper, we derive an efficient indexing scheme, SJ tree, which handles multi-attribute data and spatial join operations efficiently. In addition, a number of algorithms for manipulating multi-attribute data are given, together with their computational and I/O complexity. Moreover, we show that SJ tree is a kind of generalized B-tree. This means that SJ tree can be easily implemented on existing built-in B-tree in most storage managers in the sense that the structure of SJ tree is like that of B-tree. The spatial join operation with spatial output is benchmarked using R-tree, B-tree, K-D-B tree, and SJ tree. Results from the benchmark test indicate that SJ tree outperforms other indexing schemes on spatial join with point data.

1. Introduction

A spatial join is one of the most common operations in spatial databases. The term "join" is usually used in conjunction with a relational database management system [9, 16]. In that context, a join is said to combine entities from two data sets into a single set for every pair of

---

1) 이 논문은 한국과학재단 97년도 후반기 해외 post-doc 연수에 의해 수행되었음.
2) 가톨릭대학교 컴퓨터공학과(Dept. of Computer Engineering, The Catholic University of Korea, 43-1, Yokkok 2-Dong, Wonmi-Gu, Puchon, Kyong Ki-Do 420-743, Korea)

elements in the two sets that satisfy a particular condition. These conditions usually involve specified attributes that are common to the two sets. In the spatial variant of the join, the condition is interpreted as being satisfied when the elements of the pair cover some part of the space that is identical.

The spatial join problem has been studied both algorithmically and empirically for a variety of spatial data structures. Spatial join algorithms for regular grid files [10] were first investigated in [1]. The grid file was also used as the spatial data structure when the spatial join was examined from the perspective of creating a spatial join index [15]. In this case, the spatial join indexing simulations were on grid files using differing node-splitting rules. These simulations showed that grid files with a regular decomposition result in considerably fewer leaf node intersections between two joining structures. Spatial joins were also examined using the generalized tree [8], an abstracted hierarchical data structure similar to an R-tree [9]. Using cost models on artificial data, the generalization trees were shown to outperform join indices if there was either high data structure update rates, or high levels of join selectivity. Other studies examined the R*-tree [3] in the context of spatially joining maps composed of large polygons [3, 4]. In this case, various acceleration techniques were compared for improving CPU speed as well as I/O performance. Once a candidate set of polygons was obtained, geometric filtering were used prior to the final exact geometry testing.

A different approach makes use of the seeded tree [13]. This structure was designed to speed the more complex spatial join process when one of the two maps being joined is the result of an intermediate operation such as a selection. The seeded tree is constructed by copying the internal node structure of one map into the second map, and then the features are inserted into the second map. This replication of the internal node structure greatly accelerates the join process as there is a one-to-one mapping between internal nodes in the two maps. The application of global clustering to a modified R*-tree was studied [2]. Modified R*-tree with global clustering was found to be more expensive in terms of CPU construction costs and data storage requirements than the standard unclustered R*-tree. Experimentation with the clustered R*-tree showed, however, that spatial joins were significantly improved, primarily because of greatly decreased I/O costs. Finally, in the data-parallel domain, the spatial join has been studied in the algorithmic and empirical context [11, 12]. Experimentation indicated that the data-parallel PMR quadtree [14] significantly outperformed data-parallel R-trees and R+-trees [6] primarily because the PMR quadtree's regular decomposition is well-suited to the data-parallel domain. In the data-parallel domain, communication bottlenecks during a spatial join are greatly reduced by the regular decomposition of the PMR quadtree and the ability to quickly correlate a region in one map with a corresponding region in a second map. This ability to correlate regions will be seen to have a similar effect on the performance in the sequential domain as the size of the output of the spatial join increases with respect to the large of the two inputs.

Recently, a generalized n-dimensional B-tree indexing scheme named BV-tree[7] has been proposed. However, this scheme is not feasible because its structure and algorithms are not specified. Our indexing scheme proposed in this paper is a rather feasible indexing scheme of BV-tree in the sense that its detail structure and algorithms are properly defined.

The rest of this paper is organized as follows. In Section 2, the data manipulation algorithms for our index structure are presented; the computational complexity and the I/O complexity are also discussed. Furthermore, we generalize B-tree index structure to get our index structure. This generalization means that our index structure easily extend existing B-tree index structure for advanced database applications using multi-attribute records. In Section 3, we also measure the performance of spatial join operations with multi-attribute access methods by benchmark testing developed storage manager. Finally, in Section 4, we conclude this paper with comments on limitations of our indexing scheme and future researches.

2. A Multi-attribute Indexing Scheme

The hyper-region set will be used as multi-attribute index over multi-dimensional space. The hyper-region set will be used as multi-attribute index over multi-dimensional space. The hyper-regions in hyper-region set can be sorted by increasingly or decreasingly. Then this sorted set is just an index for multi-attribute. Even though the set is sorted, the size of the set might be too large to fit in primary memory. Thus the sorted set should be stored in a secondary storage device like a disk. One way of storing the set to a disk device is to store sequentially along to the order of the hyper-region. However, this is not a good approach to index the points in multi-dimensional space because sequential read and write operations are required for searching and updating index entries.

Another approach is to use a hashing mechanism to index the points in multi-dimensional space. This might be a good approach if hash addresses would be evenly distributed over the disk address space. The most of index scanning requires one disk access but rarely two or more disk accesses are required in case the hash addresses are duplicated severely. Besides that, the range search is not feasible because the index itself cannot be ordered.

The best approach to index points on multi-dimensional space certainly might be to use a tree structure because of its fast and efficient database operations. As an evidence, the B-tree and its variations have been used as indexing methods through most database management systems. B-tree is a balanced multi-way search tree whose structure provides several different access paths according to a given key attribute. B-tree also provides the same length for all search paths. SJ(Spatial Join) tree has hierarchical structure like B-tree. Each entry of a node except for leaf node has a hyper-region. SJ tree is a new multi-attribute indexing scheme having a B-tree-like structure.

2.1. Basic algorithms and computational complexity

The basic algorithms of SJ tree, include binary search, region split, and a region merger algorithm. They are essential algorithms for maintaining each index node of SJ tree efficiently. The binary search algorithm makes a role to search a specific point within a node. In the region split algorithm, the policy to split leaf and internal node is applied. This policy is very important for node balancing because the node occupation rate is determined by this policy. The region merge algorithm is opposed to region split algorithm. This algorithm determines how to merge less occupied nodes under the given value.

Multi-attribute binary searching

Binary search is a primary algorithm in computer software area. Traditionally, binary search algorithm is based on a single-dimensional data set. However, multi-dimensional binary search is needed to support multi-attribute data sets. In this section, we describe our binary search algorithm for multi-attribute data sets and we analyze its computation complexity. Our binary search algorithm is like that of single-dimensional binary search. The algorithm is described in Pascal-like pseudo code for simple description. We assume that node is a set of hyper-regions and the number of node entries is m.

```
Algorithm MABS(Hr : set of hyper-regions,
               x : a hyper-region)
Begin
  {Let m be the number of element of given
  hyper-region set.}
  Lb = 1, Ub = m, Mid = (Lb+Ub)/2;
    while not equal to Hr[Mid] and x and
      Lb<Ub do
      if Hr[Mid] <_n x
      then Lb = Mid+1;
      else Ub = Mid-1;
      end if
  end do
  if Lb < Ub return(failure) end if
  return(Mid)
End
```

Binary search based on a single-dimensional data set has $O(\log_2 k)$ computational complexity where k is the number of elements in the data set. In case of multi-attribute data sets like this algorithm, a comparison for equality of two hyper-regions requires m loops for an n-dimensional data set. Therefore, the total computational complexity of this search algorithm is $O(\log_2 mn)$. Almost of n is much smaller than m.

Region splitting

The split algorithm is like that of B-tree except that the split policy is based on multi-dimensional attributes.

```
Algorithm RESP (Hr: set of hyper-region)
Begin
  Node Left, Right;
  {Let Left and Right be nodes to have set of
  hyper-regions.}
  select a hyper-region R, which divides Hr half
  While remaining hyper-regions in Hr do
      take a hyper-region r, from Hr;
      if r <_n R
      then insert r into Left
      else insert r into Right;
```

```
        end if
    end
    return(Left, Right);
End
```

In the above splitting algorithm, **Node** is a data structure for storing a set of hyper-regions composed of an index node. the **while** loop ends at no more hyper-region remaining in *Hr*. The number of repeating of the while loop represents the **computational** complexity of this algorithm. Therefore, the computational complexity of the split algorithm is O(m) where m is the number of hyper-regions in *Hr*.

Region merging

The split algorithm is alike that of B-tree except that the node entry is hyper-region. Two halfway empty nodes are merged into one full node.

```
Algorithm REMG(Left, Right : set of hyper-
                 regions)
Begin
    Node Hr;
    {Let Hr be a node to have result set of
    hyper-region.}

    while remaining hyper-regions in Left and
Right do
        take a hyper-region r, from Left of Right;
        insert r into Hr;
        end
        return(Hr);
    End
```

In Algorithm Split, **Node** is a data structure for storing a set of hyper-regions composed of an index node. The while loop ends at no more hyper-region remaining in the *Left* and *Right* nodes. The number of repeats of the while loops represents the computational complexity of this algorithm. Therefore, the computational complexity of the merge algorithm is O(m) where m is the number of hyper-regions in *Left* and *Right*.

2.2. Data manipulation algorithms and I/O complexity

The data manipulation operations for index management contain retrieval, insert, and delete algorithms. For these algorithms of *SJ tree*, every variable is italicized. Especially, the input variable of each algorithm is boldfaced. For instances, in retrieval algorithm, *root* and **K** in an input variable appear as italicized and boldfaced symbol, respectively. The algorithm is composed of the following four components: heading of algorithm, input specification, output specification, and detail method. The heading part describes the function of algorithm and its parameters. In the input specification part, given input parameter is described in detail. In the output specification part, the result from algorithm is described. Finally, the method part describes the logical flow of algorithm in step by step.

Retrieval

The retrieval algorithm of *SJ tree* is like that of B-tree except for the way of searching in leaf node. The following retrieval algorithm internally uses the binary search algorithm shown in the previous section. In step MS1, the binary search is performed. This algorithm is recursive form (refer to step MS3).

Algorithm Multi-Attribute Search
(Input : *root*, **K** ; Output : *tuples*)

Input : A root of multi-attribute index tree, *root*
Given multi-attribute list (query)
K = (K, K, ..., K_n).
Let (K, K, ..., K_n) be $R_n(K)$

Output : tuples that satisfy input query, *tuples*

Method :

MS1. [Search root node]
Read the *root* node from disk. If root is leaf then perform binary search with $R_n(K)$ within this node. Call the Binary Search algorithm with argument *root* and $R_n(K)$. If the binary search is successful then return results.

MS2. [Search internal node]
If *root* is internal node then search last location in the node where $R_n(K)$ is *strictly-less-than* hyper-region in the location. Let the location be i.

MS3. [Search child]
Search the ith child node with the same search algorithm. Call Multi- Attribute Search recursively with chid node.

End of Algorithm

The I/O complexity of this algorithm is O($\log_k N$) where N is the total number of index nodes and average k-way path exists. The step MS1 is input routine in which one node structure is read from disk. This routine is called $\log_k N$ times recursively from step MS3.

Insertion

The insert algorithm of *SJ tree* is like that of B-tree except that the node entry is not a single value but a hyper-region. The insert algorithm internally uses the region split algorithm shown in the previous section. In step IN3, the region split is performed. This algorithm is in recursive form(refer to step IN3).

Algorithm Multi-Attribute Insert(Input : *root*, *K*)

Input : A root of *SJ tree*, *root*
Given multi-attribute list K = (K, K, ..., K_n)
Let (K, K, ..., K_n) be $R_n(K)$

Output : Node, the index structure is modified.

Method :

IN1. [Search insert location in leaf node]
Search insertion location of leaf node using the Multi-Attribute Search algorithm. Let the location be R. R is a set of regions in node structure.

IN2. [Insert into leaf node]
Insert $R_n(K)$ into R. As a result from insertion, if the node is overflown then go to step IN3. Otherwise, overwrite the node R to the disk. Terminate insert algorithm.

IN3. [Split node]
Split leaf node R into *R1* and *R2* using Split algorithm. Now, *R1* and *R2* are newly created and the R's entries are inserted. Write the node *R1* and *R2*. Insert the disk location of *R1* and *R2* into the parent node of R by calling recursively the Multi-Attribute Insert algorithm.

End of Algorithm

Like the retrieval algorithm, the I/O complexity of this algorithm is O($\log_k N$) where N is the total number of index nodes and average *k-way* path exists. The step IN1 is the search routine in which the leaf node location is searched from the index structure. This routine

calls the Multi-Attribute Search algorithm shown in the previous section. The searching is performed in O($\log_k N$) and the splitting is also done in O($\log_k N$) because the maximum search path is $\log_k N$.

Deletion

The delete algorithm of *SJ tree* is like that of B-tree except that the node entry is not a single value but a hyper-region. The delete algorithm internally uses the region merge algorithm shown in the previous section. In step DL3, the region merge is performed. This algorithm is in recursive form(refer to step DL3).

Algorithm Multi-Attribute Delete(Input: *root, K*)

Input : A root of *SJ tree, root*

Given multi-attribute list $K = (K, K, \ldots, K_n)$.

Let $(K, K, \ldots, K_n)$ be $R_n(K)$

Output : None, the index structure is modified.

Method :

DL1. [Search delete location in leaf node]
Search deletion location of leaf node using the Multi-Attribute Search algorithm. Let the location be R. R is a set of regions in node structure.

DL2. [Delete from leaf node]
Delete $R_n(K)$ form R. As a result from deletion, if the node is underflown then go to step DL3. Otherwise, overwrite the node R to the disk. Terminate delete algorithm.

DL3. [Merge node]
Merge node R and adjacent node R' into the node R using the Merge algorithm shown in previous section if merge is possible. Overwrite the node R to the disk. Delete from the location R' in the parent node of R by calling recursively the Multi-Attribute Delete algorithm.

End of Algorithm

Like the retrieval algorithm, the I/O complexity of this algorithm is O($\log_k N$) where N is the total number of index nodes and average k-way path exists. The step DL1 is the search routine in which the leaf node location is searched from the index structure. This routine calls the Multi-Attribute Search algorithm shown in the previous section. The searching is performed in O($\log_k$ N) and the merging is also done in O($\log_k$ N) because the maximum search path is $\log_k$ N.

Spatial join

The spatial join algorithm of *SJ tree* is like equi-join of B-tree except that the number of key fields engaged in join is two or more. Spatial join between two sets of spatial points is defined as intersection points in commonly overlapped regions. Spatial join needs the most time-consuming I/O operations among database operations. In case of R-tree, most of the node entries are traversed for checking if overlapping between two sets of spatial points exists.

Algorithm Spatial Join(Input : *root1, root2*)

Input : A root of *SJ tree, root1*

Another root of SJ tree, root2

Output : A set of spatial points overlapped between two indexed regions.

Method :

JN1. [Search overlapped entry between *root1, root2*]

For each entry in *root1*, check if the entry lies in the range of *root2*. Let the result entries be an ordered set R. The ordering of entries in R is already determined in the node.

JN2. [Search path down to leaf]
With frist entry in R, search path down to leaf. Let the leaf node be L_{lb}. Then, with last entry in R, search path down to leaf. Let the leaf node be L_{ub}.

JN3. [Traverse from L_{lb} to L_{ub}]
Traverse sequentially leaf node using linked list from L_{lb} to L_{ub}. While traverse each node, check if the entry is found in another *SJ tree*. Return the result set.

End of Algorithm

Spatial join using the *SJ tree* indexing scheme can be performed as speedily as shown in the above algorithm. In the algorithm, the entries of two root nodes are checked if they are overlapped. Because the order of entries in a node was already determined at the starting point of insertion, searching is done only for the first and the last entry of overlapping set *R*.

The complexity of this spatial join algorithm is influenced by the size of the overlapping area between two SJ trees. From JN1 to JN2 $O(\log_k N)$ I/O cost is needed where *k*-way search is performed for N nodes. In the remaining routine JN3, the I/O cost varies with the size of overlapping area.

3. Benchmarking Spatial Join Operations

The performance of a system can be measured by the simulation and the benchmark tests. The simulation is not a concrete implementation but a realistic system that is nearly the same as practical system is constructed. To evaluate the system performance accurately, simulation testing should be performed with a wide range of much realistic test data. After the performance measurement is done by a simulation, the actual system needs to be implemented and to be run under the practical system environment. In spite of the simulation results show a good performance, it might not be always guaranteed to run with best performance under the practical circumstances since the simulation has assumed virtual ones.

The benchmark test is performed with a concrete implementation under the practical system environment. A proposed system is first implemented entirely, and then benchmark data will be applied to the system. However, the benchmark test might be slightly dangerous because the performance of a new system never has been measured. Besides that, the benchmark data are required to reflect practical database circumstances. The best choice for performance measurement is to simulate with realistic data set and then to do the benchmark test under the practical database circumstances.

To accomplish two-step performance measurements such as simulation and benchmark, it takes a lot of time till the evaluation is entirely done. Thus one of two methods needs to be selected to take less of time. If the good performance of the system could be expected somewhat, the benchmark test will be desirable. Because we could expect reasonable performance

for our system as shown in the previous sections we will select the benchmark test as a method of performance measurement. However, concrete implementation require much of time. To take less of time, we implement only a storage manager using our index structure without implementing overall database management system. Besides that, on the benchmark test sets for measuring the performance of our indexing scheme, we use simple data and query file set.

3.1. Design of benchmark tests

While application-specific benchmarks measure which database system is best for a particular application, it could be very difficult to understand them. General benchmarks should be able to measure the performance of almost general-purpose database systems. Moreover, it is necessary to understand them easily.

The benchmark database is designed so that one can quickly understand the structure of the relations and the distribution of each attribute value. Consequently, the results of the benchmark queries are easy to understand, and additional queries are simple to design. The attributes of each relation are designed to simplify the task of controlling selectivity factors in selections and joins. It is also straightforward to build to an index (primary or secondary) on some of the attributes and to reorganize a relation so that it is clustered with respect to an index.

Benchmark data set should be designed to reflect realistic application specific properties. Especially, spatial database applications mainly use multi-attribute data. This multi-attribute data may be point data or range data. Since our view point in this paper is point data, we assume that multi-attribute point data to be used in benchmark test have the following properties:

(1) A file has limited number of records having two key-fields.
(2) Data type of each field of a record is either integer or character string.
(3) Key-fields are arithmetically comparable with each other.
(4) For spatial distribution, some database may be clustered at a specific region.
(5) Data do not need to be ordered by specific key fields.
(6) Multi-attribute indexing scheme is used to enhance performance.

With these assumptions, we design a set of the following data file having two-key field records. Each key field has the ranged integer type. Both first key and second key field are put on the range [0,10000). This means that maximum 100 million(10^8) points or records could exists in the data space. We limit the number of other fields as just one due to simplicity. The total length of a record is 255 bytes. The data type of the remaining field is a simple character string for simplifying the structure of a record. The spatial distribution of data is defined by clustering factor.

The clustering factor is defined as the area of a two-dimensional subspace. For clustering factor *n*, 70% of the total spatial point data are distributed in the range of $10^{-n} \times S_t$ where S_t is the size of the total data space. The remaining 30% are evenly distributed in the range of the total data space. The ratio 70% is a simple experimental result value.

A 100,000-record file is a commonly-used size

in the practical application area. Occasionally, one million(1,000,000) records file is used, but this size makes it hard to benchmark test due to time consuming. Besides, Wisconsin benchmark uses the database with only 10,000 records.

The *join_1.dat* file has the *tiny* number of records overlapping with *bench_1.dat* file. The "tiny" means that 1% of total number of records. The *join_2.dat* file has the *small* number of records overlapping with *bench_1.dat* file. The "small" means that 5% of total number of records. The *join_3.dat* file has the *medium* number of records overlapping with *bench_1.dat* file. The "large" means that 20% of total number of records. The *join_4.dat* file has the *huge* number of records means that 40% of total number of records. Each join file also has 100,000 records in uniform distribution data

3.2. Performance of spatial join operations

The performance of *SJ tree* is measured for spatial join operations. The spatial join operation is performed on five join files (*join_1.dat, join_2.dat, join_3.dat, join_4.dat* and *join_5.dat*). We performed the benchmark test in SunOS 5.5.1 running on a Sun sparc Ultra-1 computer. Spatial join is an extension of equi-join in relational database system. We measure the performance of

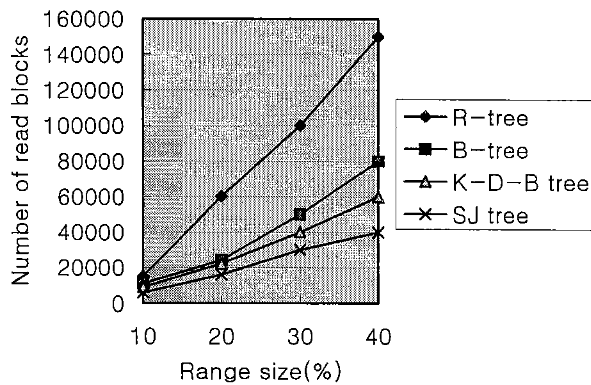

Fig. 1. Number of read blocks on range size

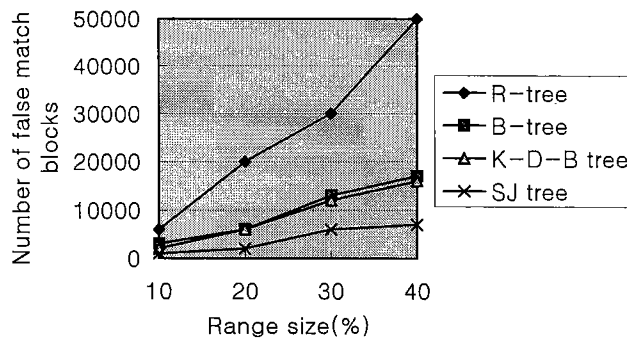

Fig. 2. Number of false match blocks on range size

spatial join by I/O cost that is the number of read blocks(see Fig.1) and false match blocks(see Fig.2) on range size.

The results of benchmark show that the performance of R-tree is worst in any overlapping size. Because the ordering of node in the index dose not exist, R-tree should compare all of node entries in two indexes. In addition to that, overlapped regions in the nodes lead to false match path. Other indexing schemes do not allow overlapping regions in the node. Therefore, they show relatively good performance. Among these indexing schemes, SJ tree shows the best performance because the ordering is fit to join tables. B-tree and KDB-tree show average performance because of the false match blocks.

4. Conclusions

The join operation is the principal method of combining two relations. Conceptually a join is defined as a cross-product followed by a selection condition. In practice, this viewpoint can be very expensive, because it involves materializing the cross-product before applying the selection criterion. This is especially true for spatial databases.

In this paper, we proposed a new indexing scheme for efficient spatial join operations, SJ tree. Also, we described basic algorithms for manipulating multi-attribute data. For each algorithm, we analyzed computational complexity and I/O complexity. Moreover, we showed that SJ tree is a kind of generalized B-tree. This fact means that SJ tree can be easily implemented on existing built-in B-tree in almost storage managers.

As an experimental performance measurement, we performed benchmark testing under various realistic data and query sets. From the results of benchmark test we concluded that SJ tree mostly outperforms other indexing schemes *on spatial join*. Since spatial join is a large time-consuming operation, the performance on this operation influences to handle very large database in wide range. SJ tree presented relatively good performance compared to other indexing schemes regardless of the size of overlapping area.

We conclude that SJ tree is a good indexing scheme for efficient spatial join operations. If SJ tree is fully implemented as an index method of storage manager, the performance of many applications using multi-attribute will be remarkably enhanced. In the further work, we will extend the proposed method to handle spatial range data as well as point data.

References

N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, In Proc. ACM *SIGMOD Conf., (1990) 322-331.*

T. Brinkhoff and H. P. Kriegel, The impact of global clustering on spatial database systems, In *Proc. Very Large Data Bases Conf., (1994) 168-179.*

T. Brinkhoff, H. P. Kriegel, R. Schneider, and B. Seeger, Multi-step processing of spatial joins, In. *Proc. ACM SIGMOD Conf., (1994) 197-208.*

T. Brinkhoff, H. P. Kriegel, and B. Seeger, Efficient processing of spatial joins using R-trees, In. *Proc. ACM SIGMOD Conf., (1993) 237-246.*

R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, *Benjamin/Cummings, Red wood City, CA, 2nd edition, (1994).*

C. Faloutsos, T. Sellis, and N. Roussopoulos, Analysis of object oriented spatial access methods, In *Proc. ACM SIGMOD Conf., (1987) 426-439.*

M. Freeston, A general solution of the n-dimensional B-tree problem, In *Proc. ACM SIGMOD Conf., (1995) 80-91.*

O. Gunther, Efficient computation of spatial joins, In *Proc. IEEE Data Engineering Conf., (1993) 50-59.*

A. Guttman R-trees: A dynamic index structure for spatial searching, In *Proc. ACM SIGMOD Conf., (1984) 47-57.*

K. Hinrichs and J. Nievergelt, The Grid file: a data structure designed to support proximity queries on spatial objects, In *Proc. WG'83(Intl. Workshop On Graphtheoretic Concepts in Computer Science), (1983) 100-113.*

E. G. Hoel and H. Samet, Data-parallel spatial join algorithms, In *Proc. Parallel Processing Conf., (3) (1994) 227-234.*

E. G. Hoel and H. Samet, Performance of data parallel spatial operations, In *Proc. Very Large Data Bases Conf., (1994) 156-167.*

M. L. Lo and C. V. Ravishankar, Spatial joins using seeded trees, In *Proc. ACM SIGMOD Conf., (1994) 209-220.*

R. C. Nelson and H. Samet, A consistent hierarchical representation for vector data, *Computer Graphics, 20 (4) (1986) 197-206.*

D. Rotem, Spatial join indices, In *Proc. IEEE Data Engineering, (1991) 500-509.*

S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, and C. T. Lu, Spatial Databases-Accomplishments and Research Needs, *IEEE Trans. on Knowledge Engineering, 11 (1) (1999) 45-53.*