Accelerating Spatial Join Operations using Bit-Indices

Elizabeth Antoine, Kotagiri Ramamohanarao, Jie Shao and Rui Zhang

Department of Computer Science and Software Engineering
The University of Melbourne, Victoria
Australia
Email: {eantoine, rao, jsh, rui}@csse.unimelb.edu.au

Abstract

Spatial join is a very expensive operation in spatial databases. In this paper, we propose an innovative method for accelerating spatial join operations using Spatial Join Bitmap (SJB) indices. The SJB indices are used to keep track of intersecting entities in the joining data sets. We provide algorithms for constructing SJB indices and for maintaining the SJB indices when the data sets are updated. We have performed an extensive study using both real and synthetic data sets of various data distributions. The results show that the use of SJB indices produces substantial speedup ranging from 25% to 150% when compared to Filter trees.

Keywords: spatial join bitmap indices, hierarchical representation, size separation, spatial locality

1 Introduction

Spatial database systems have gained a lot of interest in the research community due to the increasing popularity of automated processes in fields such as remote sensing, cartography and earth sciences. Spatial objects in CAD applications are mostly points, lines and polygons. However, more complex objects appear in applications that include robotics, virtual reality and cartography. Spatial objects take various shapes and manipulating such objects becomes computationally expensive. Approximations such as the Minimum Bounding Rectangle (MBR) are used to ease the processing. An object is represented by the smallest axis-parallel rectangle that completely contains it. In spatial database systems, the type of queries can be broadly categorized into single-scan and multiple-scan queries. Processing of single-scan queries requires at most one access to an object and therefore, the execution time is linear to the number of objects stored in the corresponding relation [Brinkhoff et al., 1993]. Examples of single-scan queries are window and point queries. However, multiple-scan queries access an object several times and therefore, execution time is superlinear to the number of objects. The most important multiple-scan query in a spatial database system is spatial join. The join operation (spatial join) finds pairs of objects (each from a joining data set) that have a specific spatial relationship.

The spatial join, just like the join operation in the relational databases, is a computationally expensive operation. The reason is that for large databases, pages may need to be fetched from disk more than once in order to compute the join. Many indexing techniques have been proposed for the storage and manipulation of spatial data that are able to execute spatial join operations with great efficiency. A method to organize rectangles based on their sizes was proposed by [Abel and Smith, 1983], and a similar approach was proposed by [Six and Widmayer, 1988] to extend grid files to represent rectangles rather than points. The method proposed by [Orenstein and Manola, 1988] uses the hierarchical representation and Z-ordering in evaluating range queries. The Quad-CIF tree proposed by [Kedem, 1982] uses the size separation method to organize rectangles. A similar method was proposed by [Guenther, 1991]. An efficient processing of window query is important for the efficiency of spatial joins. The methods listed above have been proved experimentally to be efficient for window queries but the experimental results for spatial join performance are not available.

The R-tree join algorithm [Brinkhoff et al., 1996] uses a buffer pinning technique that tries to keep the relevant blocks of entities in the buffer in order to minimize block re-reads and is efficient in the processing of spatial joins. Filter trees proposed by [Sevcik and Koudas, 1996] perform the spatial join by combining the hierarchical representation, size separation and spatial locality methods, and the number of block reads to perform the spatial join does not exceed the required minimum. Filter tree join algorithm outperforms the R-tree join algorithm by reading each block of entities at most once. However, for data sets of low join selectivity, the number of blocks processed for Filter trees is excessive compared to the number of blocks that have intersecting entities.

The goal of this work is to provide a method for accelerating spatial join operations by using Spatial Join Bitmap (SJB) indices. The file organization is based on the concepts introduced in Filter trees. The SJB indices keep track of blocks that have intersecting entities and make the algorithm to process only those blocks. We provide algorithms for generating SJB indices dynamically and for maintaining SJB indices when the data sets are updated. Although maintaining SJB indices for updates increases the cost in terms of response time, our method is beneficial when the data set is randomly distributed. For a given set of relations, SJB indices are generated during the first run of the spatial join and are maintained to perform the subsequent join operations much faster.

The sequel of the paper is organized as follows. Section 2 explains how the file organization is done based on the concepts introduced in Filter trees, algorithms for insertion and deletion of entities and algorithms for query processing. Section 3 describes our proposed method, the SJB indices. Section 4 presents an analysis of the SJB indices. Section 5 reports the experimental results, Section 6 presents the related

work and Section 7 gives the conclusion.

2 File Organization for SJB Indices

The file organization is based on the hierarchical representation, size separation and spatial locality concepts introduced in Filter trees [Sevcik and Koudas, 1996]. In contrast to Filter trees, our algorithm is implemented with the assumptions that the spatial entities are not only 2-dimensional rectangles, but also points and lines; and moreover, there are updates on the data entities. The file organization is done in two phases: generate level files, and ordering of entities in level files. To be self-contained, each phase is explained below.

Generate level files:

The concept of hierarchical representation is used here. Each entity is associated with a level that corresponds to a particular granularity of space partitioning [Sevcik and Koudas, 1996]. The algorithm hierarchically partitions the data space and groups the data entities into level files. The number of levels is denoted by L. At every level $l(l = 0, \ldots, L-1)$, the number of partitions is 4^l and each level is composed of $2^l - 1$ equally spaced lines in each dimension. An entity intersected by any line of level l belongs to level $l-1$. This method of partitioning places the large entities at the higher levels and the small entities at the lower levels with high probability. In some cases, entities which are smaller in size may be placed in the higher levels when they overlap with the grid lines of a higher level. The partitioning method reflects the concept of size separation, i.e., entities of different sizes tend to be associated with different levels.

Figure 1 illustrates the process of recursive partitioning and shows how the entities with different sizes are associated with their corresponding levels. Entity i that intersects the grid lines of level l is fully contained in level $l-1$ and hence belongs to level $l-1$. In Figure 1(a), entity a is contained in the $\frac{1}{2}$ by $\frac{1}{2}$ partition and hence associated with level 1. However, entity c is located on the grid line of level 1 and hence associated with level 0. Entity d is much smaller and is fully contained in the $\frac{1}{8}$ by $\frac{1}{8}$ partition so it is associated with level 3 and entity b is in level 2. Figure 1(b) illustrates the partitioning of entities into level files.

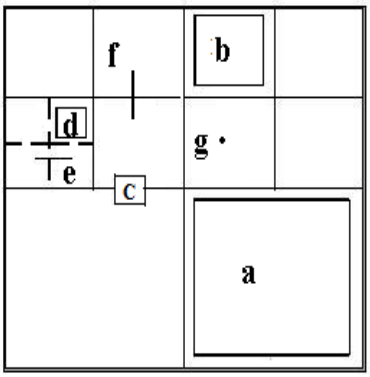
The lines and points fall in the level whose grid lines intersect with the location of the entities or the level of the entity happens to be the lowest level to which the remaining entities fall. Figure 1 illustrates this process of finding the level of entities such as points and lines. In this example, entity e strikes the grid line of level 3 and hence associated with level 2. Similarly, entity f strikes the grid lines of level 2 and belongs to level 1. Entity g is neither contained in any partition nor does it intersect any grid line and hence belongs to the lowest level which is level 3.

Ordering of entities in level files:

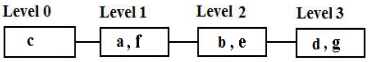
The ordering of the entities in the level files can be done by any space-filling curve that recursively partitions the data space. Our implementation is based on the Z-order curve. For each entity, an entry (entity descriptor) is composed and stored in their corresponding level files. An entity descriptor contains:

- The specifications of the MBR of the entity, $< x_l, y_l, x_h, y_h >$.
- The Z-order value of the entity.

The level files are then sorted based on the Z-order value. Here the concept of spatial locality, i.e., the entities are ordered by their positions along a space-filling curve in order to cause entities in a portion of the multidimensional space to map to contiguous portions of data storage space as much as possible, is used [Sevcik and Koudas, 1996].



(a) Partitioning



(b) Level Files and Entities

Figure 1: Recursive Partitioning Example

2.1 Insert and Delete Algorithms

In this section, we focus on the insertion and deletion of entities in the file structure described above. Insertion of a new entity is done by finding the level of the entity, calculating the Z-order value and adding the entity descriptor of the entity at the appropriate location in the corresponding level file.

Algorithm 1: *InsertEntity*

input: entity IE, level files of data set DS.
begin
  Find the level of IE, l
  Compute Z-order value, Z
  Store the specifications of IE and its Z-order value in the corresponding level file l of data set DS in the order of Z
end

Algorithm 2: *DeleteEntity*

input: entity IE, level files of data set DS.
begin
  Find the level of IE, l
  Locate the entity descriptor based on Z-order value Z in the level file l of data set DS
  Delete the entity descriptor
end

Deletion of an entity is done by finding the level of the entity and its Z-order value, searching the entity in the corresponding level file based on the Z-order value and deleting the respective entity descriptor. Algorithms 1 and 2 illustrate the insertion and deletion of entities.

2.2 Algorithms for Query Processing

In this section, we briefly describe how spatial join and window queries are executed using the file organization structure explained above.

Spatial Join:

The level files are merged to perform the spatial

join which typically resembles an L-way merge sort. The spatial join is performed as follows. Let A and B denote the two joining relations. Let $A^l(Z_m, Z_n)$ denote a page of the l-th level file of A containing entities with Z-order values in the range (Z_m, Z_n), and $B^l(Z_m, Z_n)$ denote a page of the l-th level file of B containing entities with Z-order values in the range (Z_m, Z_n) respectively. For level files $l = 0, \ldots, L-1$:

- Process entries in $A^l(Z_m, Z_n)$ with those contained in $B^{l-i}(Z_m, Z_n)$ for $i = 0, \ldots, l$.
- Process entries in $B^l(Z_m, Z_n)$ with those in $A^{l-i}(Z_m, Z_n)$ for $i = 1, \ldots, l$.

The ranges of i differ in the two steps in order to avoid matching $A^l(Z_m, Z_n)$ and $B^l(Z_m, Z_n)$ twice. As an example of how the spatial join is performed, consider relations A and B in Figure 2. The relations A and B have entities that extend up to level 3. Data entries in partition 6 of level 3 in relation A are compared with the data entries in partition 6 of level 3 in relation B and with the data entries in the enclosing partitions at the higher levels of relation B. The data entries in other partitions are not considered as the partitions are disjoint. In a similar fashion, the data entries in partition 6 of relation B are compared with the data entries in the enclosing partitions at the higher levels of relation A.

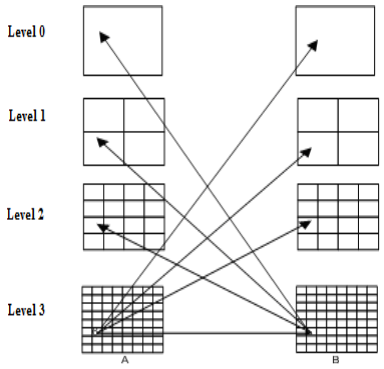


Figure 2: Spatial Join Example

To perform the join, each partition of every level is read only once. In case of spatial join with low selectivity, the number of partitions that have entities is relatively higher than the number of partitions that have the intersecting entities. By keeping track of the partitions that have the intersecting entities, the join phase could be performed in considerably less time. Algorithm 3 illustrates the join phase.

Window Queries:

In order to answer a window query, each level file need to be searched. However, searching within each level is made efficient by identifying the relevant partitions. If k is the lowest level at which the query window is fully enclosed in a partition, then only entities that belong to a single interval need to be processed at levels 0 through k. For the levels below k, there will be generally two or more intervals involved. The detailed explanation and experimental results of window query processing is given in [Sevcik and Koudas, 1996]. Window query processing for SJB indices is similar to the Filter trees as SJB indices are not generated.

Algorithm 3: *Join*

input: level files of the joining relations; $DS1, DS2$.
Let $DS1^l(Z_m, Z_n)$ denote a page of the l-th level file of $DS1$ containing entities with Z-order values in the range (Z_m, Z_n) and $DS2^l(Z_m, Z_n)$ denote a page of the l-th level file of $DS2$ containing entities with Z-order values in the range (Z_m, Z_n).
begin
  for $l \leftarrow 0$ **to** $L-1$ **do**
    for $i \leftarrow 0$ **to** l **do**
      process entries in $DS1^l(Z_m, Z_n)$ with entries in $DS2^{l-i}(Z_m, Z_n)$
    for $i \leftarrow 1$ **to** l **do**
      process entries in $DS2^l(Z_m, Z_n)$ with entries in $DS1^{l-i}(Z_m, Z_n)$
  end
  (Note: The ranges of i differ in the two 'for' loops in order to avoid the processing of matching $DS1^l(Z_m, Z_n)$ and $DS2^l(Z_m, Z_n)$ twice.)
end

3 Spatial Join Bitmap (SJB) Indices

SJB indices are generated dynamically as the entities are inserted in the relation. Consider the relations A and B, if an entity is inserted in relation A, then the SJB indices are generated for the partition the inserted entity belongs to. The SJB indices are generated or maintained by processing the inserted entity with the entities in the corresponding partitions of the joining relation B. A more detailed explanation is given in Section 3.3. The SJB index shows the existence of intersecting entities in the corresponding partition: '1' represents the existence of intersecting entities and '0' otherwise.

3.1 Algorithms for generating SJB indices

The SJB indices are generated as follows: Let $S^l(Z)$ denote a partition at level l containing entities of relation S with Z-order value given by Z. Consider relations A and B, for levels $l = 1, \ldots, L-1$:

- For every partition $A^l(Z)$, mark '1' if there are intersecting entities in $B^{l-i}(Z)$ for $i = 0, \ldots, l$; mark '0' otherwise.
- For every partition $B^l(Z)$, mark '1' if there are intersecting entities in $A^{l-i}(Z)$ for $i = 1, \ldots, l$; mark '0' otherwise.

The ranges of i differ in the two steps in order to avoid matching $A^l(Z)$ and $B^l(Z)$ twice. Consider relations $DS1$ and $DS2$ that have entities extending up to level l. Let $pDS1$ be the partition of level l in relation $DS1$ whose data entries are to be compared with the data entries in partition $pDS2$ of level l in relation $DS2$ and with the data entries in the enclosing partitions $(pDS2-1, pDS2-2, \ldots, 0)$ at the higher levels $(l-1, l-2, \ldots, 0)$ of relation $DS2$. For the partition $pDS1$, the SJB index would be generated for $pDS2, pDS2-1, pDS2-2, \ldots, 0$. For relation $DS2$, let $pDS2$ be the partition of level l in relation $DS2$ with the data entries in the enclosing partitions $(pDS1-1, pDS1-2, \ldots, 0)$ at the higher levels $(l-1, l-2, \ldots, 0)$ of relation $DS1$. $pDS2$ is

not compared with $pDS1$ as the processing is already done. For the partition $pDS1$, the SJB index would be generated for $pDS2, pDS2-1, pDS2-2, \ldots, 0$ and for partition $pDS2$, the SJB index would be generated for $pDS1-1, pDS1-2, \ldots, 0$.

As an example of how the SJB indices are generated, consider relations A and B in Figure 2. The relations A and B have entities that extend up to level 3. Data entries in partition 6 of level 3 in relation A are compared with the data entries in partition 6 of level 3 in relation B and with the data entries in the enclosing partitions at the higher levels of relation B. Hence, there will be four entries of either '0' or '1 for the partition 6 of level 3 in relation A. For relation B, there will be three entries of either '0' or '1' for the partition 6 of level 3. The difference in the number of entries is to avoid matching entries twice. To identify the entries of a particular level file, level file name could be stored. Along with entries of '0' and '1', the z-order value that represents the entities in the partition is also stored, which is value 5 for the entities in partition 6.

Algorithm 4: *SJBInsert*

input: level files of joining relation $DS2$;
Z-order value of the inserted entity IE, Z; level of the inserted entity IE, l.
Let l_{upd} denote the file in which the entity descriptor of the inserted entity IE is stored.
Let $DS2^l(Z)$ denote the entities at level l of the joining relation $DS2$, having Z-order value Z that could possibly intersect with entities in l_{upd}.
begin
  Locate the SJB indices for the partition in which the entity is inserted using the level l and the Z-order value Z
  for $i \leftarrow 0$ **to** l **do**
    if $SJB[i]$ *is '0'* **then**
      process entries in l_{upd} with entries in $DS2^{l-i}(Z)$
      if l_{upd} *has intersecting entities* **then**
        $SJB[i] \longleftarrow$ '1'
  end
  Write SJB in SJB indices file of $DS1$ at the appropriate location
  Let $DS2^l(Z_x, Z_y)$ denote the entities at level l of the joining relation $DS2$, having Z-order values in the range (Z_x, Z_y) that could possibly intersect with l_{upd}
  for $i \leftarrow l+1$ **to** $L-1$ **do**
    for $j \leftarrow Z_x$ **to** Z_y **do**
      $SJB \longleftarrow$ ***SearchSJB(SJB indices file of*** $DS2$***,*** j***,*** i***)***
      if $SJB[correspondingbit]$ *is '0'* **then**
        process entries in $DS2^i(j)$ with entries in l_{upd}
        if l_{upd} *has intersecting entities* **then**
          $SJB[correspondingbit]$ is '1'
  end
end

Algorithm 4 is given to understand the process of generating SJB indices when an entity is inserted. Algorithm 5 describes the process of updating the SJB indices when an entity is deleted. Algorithm 6 illustrates the process of searching the SJB indices file of respective relation.

Algorithm 5: *SJBDelete*

input: level files of joining relation $DS2$;
Z-order value of the deleted entity Z;
level of the deleted entity l.
Let l_{upd} denote the file in which the entity descriptor of the entities which belongs to the partition of the deleted entity is stored. Let $DS2^l(Z)$ denote the entities at level l of the joining relation $DS2$, having Z-order value Z that could possibly intersect with entities in l_{upd}.
begin
  Locate theSJB indices for the partition from which the entity is deleted using the level l and the Z-order value Z
  for $i \leftarrow 0$ **to** l **do**
    if $SJB[i]$ *is '1'* **then**
      process entries in l_{upd} with entries in $DS2^{l-i}(Z)$
      if l_{upd} *has no intersecting entities* **then**
        $SJB[i] \longleftarrow$ '0'
  end
  Write SJB in SJB indices file of $DS1$ at the appropriate location
  Let $DS2^l(Z_x, Z_y)$ denote the entities at level l of the joining relation $DS2$, having Z-order values in the range (Z_x, Z_y) that could possibly intersect with entities in l_{upd}
  for $i \leftarrow l+1$ **to** $L-1$ **do**
    for $j \leftarrow Z_x$ **to** Z_y **do**
      $SJB \longleftarrow$ ***SearchSJB(SJB indices file of*** $DS2$***,*** j***,*** i***)***
      if $SJB[correspondingbit]$ *is '1'* **then**
        process entries in $DS2^i(j)$ with entries in l_{upd}
        if l_{upd} *has no intersecting entities* **then**
          $SJB[correspondingbit] \longleftarrow$ '0'
  end
end

Algorithm 6: *SearchSJB*

input: SJB indices file of the relation, z-order value Z, level file l.
begin
  With the level file identifier l and Z-order value Z, locate the corresponding entries in the SJB indices file
  Retrieve the entries of '0' and '1' stored for Z
end

The join after generating the SJB indices is done by looking up the corresponding entries in SJB indices file for every partition. If the value is set to '1', then the processing is done. The processing (join) of the partitions at every level is done in a sorted sequence given by the Z-order value. While the SJB indices are generated, the entries in the SJB indices file are maintained in the same sorted sequence. The level identifier and the Z-order value are used to perform the search operation. The search operation in the SJB indices file could be easily implemented with the efficient searching techniques commonly available in database systems.

3.2 Processing Join Queries with SJB Indices

Algorithm 7: *JoinwithSJB*

input: level files of relations $DS1$ and $DS2$. Let SJB_{DS1} denote SJB indices file of DS1 and SJB_{DS2} denote SJB indices file of DS2. Let $DS1^l(Z_m, Z_n)$ denote a page of the l-th level file of $DS1$ containing entities with Z-order values in the range (Z_m, Z_n) and $DS2^l(Z_m, Z_n)$ denote a page of the l-th level file of $DS2$ containing entities with Z-order values in the range (Z_m, Z_n).

```
begin
    for level files l ← 0 to L − 1 do
        for i ← 0 to l do
            process entries in DS1^l(Z_m, Z_n) with
            entries in DS2^{l−i}(Z_m, Z_n) if their
            corresponding entries in SJB_DS1 is
            set to '1'
        for i ← 1 to l do
            process entries in DS2^l(Z_m, Z_n) with
            those in DS1^{l−i}(Z_m, Z_n) if their
            corresponding entries in SJB_DS2 is
            set to '1'
        (Note: The ranges of i differ in the two
        'for' loops in order to avoid matching
        DS1^l(Z_m, Z_n) and DS2^l(Z_m, Z_n) twice.)
    end
end
```

Algorithm 7 illustrates the spatial join with SJB indices. When join is performed based on the SJB indices, data entries in partition i of level l in relation $DS1$ will be processed with the corresponding partitions in the relation $DS2$ only if the index entries are set to '1'. Consider Figure 2, data entries in partition 6 of level 3 in relation A will be compared with the respective partitions in relation B only when their corresponding index is set to '1'. The SJB indices are stored in an ordered sequence and it resembles the order in which the join is performed.

3.3 Update of SJB Indices

In the real world scenario, although the extent of updates is limited, efficiently handling updates becomes a prime concern. Updates require a simple insertion/ deletion operation, however the SJB indices need to be maintained. The number of pages for storing the SJB indices is highly dependent on the number of level files and it is possible for the SJB indices to be kept in memory for efficient processing of updates.

Consider e as the inserted entity of relation A. The spatial join $(A \bowtie B)$ is performed considering entity e as the only entity of relation A. The insertion of entities falls in the following cases:

- Case 1: The inserted entity is the only entity in the partition that intersects with the corresponding level of the other relation.
- Case 2: The inserted entity falls in the partition which does not have any entity.
- Case 3: The inserted entity falls in the partition which has intersecting entities; hence the bit is already set.

Case 1 leads to the change in the SJB index of the corresponding partition to '1'. Case 2 leads to the insertion of the new set of bits which represents the partition. The SJB index needs to be updated for Cases 1 and 2. Consider the insertion of an entity in relation $DS1$, call *InsertEntity* to insert the entity and call *SJBInsert* to generate/modify SJB indices for the inserted entity. If new set of SJB indices is generated as per Case 2, then insertion of the generated indices at the appropriate location in SJB indices file of $DS1$ is done. Deletion of entities is handled with respect to the partition that it belongs to. Consider f as the deleted entity of relation A. The partition p to which the entity f belongs to is calculated and all the entities of p except the deleted entity is considered as relation A. The spatial join $(A \bowtie B)$ is performed and the SJB index of the respective partition is either changed to '0' or deleted based on the result of processing. Deletion of entities in the relation would fall into the following cases:

- Case 1: The deleted entity is the only entity that intersects with the corresponding levels of the other relation.
- Case 2: The deleted entity is the only entity that occupies the partition.
- Case 3: The deleted entity is not the only entity in the partition that intersects; no action required.

If the deleted entity is the only entity that intersects, then the bit index is changed to '0' for the corresponding partition. If the deleted entity is the only entity that occupies the partition, the bit entries that represent the partition are deleted. Consider the deletion of an entity in relation $DS1$, call *DeleteEntity* to delete the entity and call *SJBDelete* to modify SJB indices for the deleted entity. If Case 2 is valid, then delete the respective SJB indices from SJB indices file of $DS1$.

4 Analysis of SJB Indices

The number of pages read for generating the level files and for the join process is illustrated in [Sevcik and Koudas, 1996]. [Sevcik and Koudas, 1996] explains in detail the distribution of entities across level files for data set of uniform distribution, the distribution of level occupancy, the average level occupied by squares of certain size and how the size separation is achieved by Filter tree structure. As discussed, for the join phase every page of the joining relations is accessed only once. However, for data sets of low join selectivity, the number of pages processed is significantly higher than the number of pages that have intersecting entities. This is observed in the data sets of uniform distribution where the join selectivity is low which leads to the decrease in the time required for processing the pages.

Let P_X represent the number of pages processed for method X, C represent the cost (time) for a page to be processed and T_{perf} represent the difference in performance with respect to time. C_{MI} gives the cost (time) of computing the bit-mapping indices. The difference in performance with respect to response time is given by the following equation:

$$(P_{SpatialJoin} - P_{SpatialJoinwithMI}) * C - C_{MI} = T_{perf}$$

4.1 Space Requirement for SJB Indices

The number of pages required for storing SJB indices is calculated for relations A and B separately. For every level file, the number of partitions is given by 4^l and the page size 2^P for the purpose of implementation is considered to have 2^{12} bits, which is 4KB. For a given relation, the value of l_{mx} is set to the value $L-1$ of that relation. For relation A the number of pages is given by:

$$\frac{\sum_{l=1}^{l_{mA}} 4^l(l+1)}{2^P}$$

For relation B the number of pages is given by:

$$\frac{\sum_{l=1}^{l_{mB}} 4^l(l)}{2^P}$$

4.2 Analysis of Updates

The update maintenance cost for SJB indices is higher than the update cost (direct insertion and deletion) when the SJB indices are not maintained. The reason is that for spatial join with SJB indices, the updated entries are processed individually and their corresponding SJB indices are updated. Let the response time for updating SJB indices be R_{UM_MI} and the response time for the processing of spatial join queries with SJB indices be R_{SJ_MI}. The response time of spatial join operation for the relations with updated entities be R_{SJ} and the response time for updating entities in the relation be R_{UM}. The decrease in the response time when the spatial join is performed with SJB indices balances the increase in the update cost for maintaining the SJB indices. Hence, if the spatial join is performed α times after an update, this method is beneficial if the following equation holds true for $\alpha \geq 1$

$$R_{UM\_MI} + \alpha * R_{SJ\_MI} < R_{UM} + \alpha * R_{SJ}$$

5 Experimental Results

In order to assess the performance benefits of SJB indices, we implemented the Filter tree join algorithm. Filter tree join algorithm performs better than the best R-tree join algorithm proposed by [Brinkhoff et al., 1993], as described in [Sevcik and Koudas, 1996]. Filter trees can perform the join with almost 32% savings in response time with 2.1% buffer space, relative to the R-tree with 5% buffering [Sevcik and Koudas, 1996]. As per the experimental results shown in [Sevcik and Koudas, 1996], Filter trees outperform R-trees and hence we chose to compare SJB indices with Filter trees. In the following, we present the experimental results for the join performance of Filter trees and SJB indices.

5.1 Description of Data Sets

The experiments were conducted on an Intel®Core™2 Duo Processor at 2.00GHz which has the main memory capacity of 2.0GB. The windows experience rating index for the processor is 4.7 out of 5. We experimented with both real and synthetic data sets. For the purpose of implementation, we consider the z-order value is generated at the time of generating level files and will be stored as part of the entity descriptor. This would save the processor time when computing the join results. The real data sets consist of the road segments extracted from the TIGER/Line relation [Tig, 2000]. The relations consist of long beach county roads containing 53,145 MBRs, railways containing 128,971 MBRs, tiger streams containing 194,971 MBRs. The synthetic data sets which are of uniform and zipfian distributions have 250,000, 100,000 and 50,000 entities, respectively. Table 1 presents the data sets used for the experiments.

5.2 Spatial Join using SJB Indices

Figures 3, 4, 5 and 6 show the result of the experiments conducted on data sets of various sizes and data distributions. The benefits of using SJB indices are clearly seen in real data sets as the join selectivity is low compared to the relations of uniform and zipfian distributions.

The data sets of real, uniform and zipfian are sampled to have 25,000, 50,000, 75,000, 100,000, 150,000 and 250,000 entities and the join is performed with data set having 50,000 entities of respective distribution. The use of SJB indices decreases the response time. The experiments conducted on the data sets of uniform distribution show that the percentage improvement in the response time is found to be approximately 95%. The improvement in the response time is due to the decrease in the number of pages processed. The intersecting entities are distributed randomly and not every partition has an intersecting entity, which leads to the decrease in the pages processed. Figure 3 presents the response time for the join of two uniformly distributed data sets, U1 and U2 containing 250,000 and 50,000 entities respectively. Data set U1 is sampled as described above and the join is performed with data set U2 for every sample.

The experiments conducted on the data set of zipfian distribution show that the percentage improvement in the response time is found to be approximately 25%. The decrease in the response time in the zipfian distribution is found to be less when compared to the uniform distribution as the relations of zipfian distribution show high join selectivity in our experiments. The performance of SJB indices is high when the join selectivity is low. The join on data sets of zipfian distribution shows a low percentage improvement in performance due to the high join selectivity. The percentage difference in the number of page processed is 15%.

Figure 4 presents the response time for the join of two data sets of zipfian distribution, Z1 and Z2 containing 250,000 and 50,000 entities respectively. Data set Z1 is sampled as described above and the join is performed with data set Z2 for every sample. Although the number of partitions which have the intersecting entities is less for data sets of zipfian distribution, the number of pages processed is high. The reason is that, the partitions that have the intersecting entities are the only partitions that have the highest percentage of entities. The cost (response time) is reduced by reducing the number of pages processed to compute the join. The difference in the number of pages processed for the Filter trees and SJB indices in case of data sets of zipfian distribution is low, which results in the low performance.

The improvement in performance for the real data sets is shown in Figures 5 and 6. The percentage improvement in response time is found to be 150%, which is due to the low join selectivity and the high

Name	Description	Data Set Size
U1	Uniformly distributed entities	250,000
U2	Uniformly distributed entities	50,000
U3	Uniformly distributed entities	100,000
Z1	Entities of zipfian distribution	250,000
Z2	Entities of zipfian distribution	50,000
Z3	Entities of zipfian distribution	100,000
SD	Tiger streams of Iowa, Kansas, Missouri and Nebraska	194,971
LB	Long beach county roads	53,145
RA	Tiger/Line LA railways	128,971

Table 1: Relations Used

decrease in the number of pages processed while using SJB indices. The percentage decrease in the number of pages processed is approximately 110%. The increase in the performance is due to the size of the entities, which is invariably small in case of the real data sets and the low join selectivity. The entities in the real data sets (SD) are found to be scattered all over the space and the size of the entities is small, which results in the low join selectivity. The improvement in performance for the real data sets (RA and LB) is found to be similar to the results found in real data sets (SD and LB).

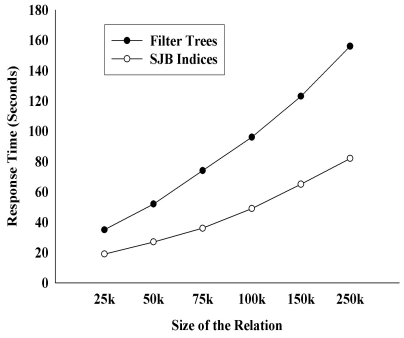


Figure 3: Response Time ($U1 \bowtie U2$)

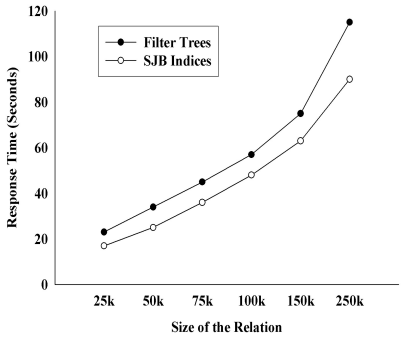


Figure 4: Response Time ($Z1 \bowtie Z2$)

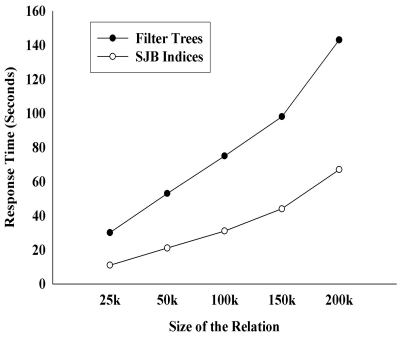


Figure 5: Response Time ($SD \bowtie LB$)

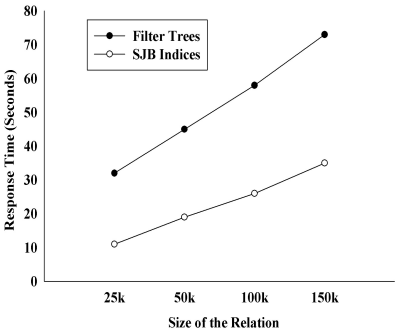


Figure 6: Response Time ($RA \bowtie LB$)

Table 2 summarizes all the experimental results in this subsection and presents the percentage of improvement in response time and pages processed for SJB indices over Filter trees. The experiments show that the improvement in performance is almost 150% for real data sets (TIGER/Line). The percentage of improvement while using SJB indices is computed with the following equation:

$$\frac{V_{FILTER\_TREES} - V_{SJB\_INDICES}}{V_{SJB\_INDICES}} * 100$$

where V_x means response time for method x.

5.3 Updates of SJB Indices

The cost for maintenance of the updates involves the cost of updating the SJB indices. The update maintenance cost for SJB indices is expensive when compared to Filter trees as the SJB indices need to be updated. The cost of processing a spatial join operation after updates is found to be lesser when the SJB indices are used and this could be used to compensate the extra cost involved in maintaining SJB indices for updates. The experiments are conducted on data sets of uniform distribution and zipfian distribution containing 100,000 and 50,000 entities each. The data sets are sampled to have 200, 400, 600, 800 and 1000 entries each and Figure 7 shows the number of times the spatial join has to be performed to compensate the extra cost incurred during updates. The difference in response time while using SJB indices could be used effectively to maintain updates. We found that the following equation holds true and α is < 1 when the updates extend up to 1000 entries for data sets of uniform distribution.

$$R_{UM\_MI} + \alpha * R_{SJ\_MI} < R_{UM} + \alpha * R_{SJ}$$

Improvement	Real (TIGER/Line)		Uniform	Zipfian
	SD join LB	RA join LB	U1 join U2	Z1 join Z2
Time	150%	148%	95%	25%
Pages Processed	110%	105%	60%	15%

Table 2: Percentage of Improvement in Real, Uniform and Zipfian Data Sets

For data sets of uniform distribution, the α value is less than 0.4 for updates of about 1000 entries and the method saves cost even when the spatial join is not performed involving updates up to a certain limit. However, in the real world scenario, the number of updates is quite less than the number of times the spatial join is performed on the data sets. The suggested method can be used for updates up to a certain limit and is highly based on the type of data distribution. For data sets of zipfian distribution, the limit to which the updates are handled efficiently by this method is quite limited as the difference in the response time of spatial join with Filter trees and the spatial join with SJB indices is considerably low.

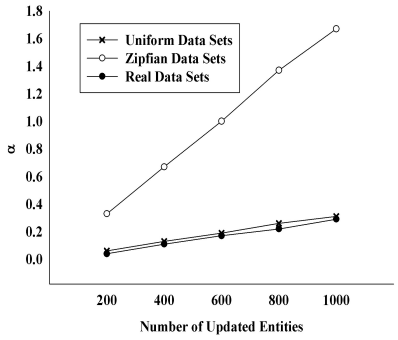


Figure 7: Spatial Joins over Updates

Figure 7 shows that the number of times the spatial join has to be performed (α) for data sets of zipfian distribution becomes greater than 1 when the number of updated entities scales beyond 500. For real data sets, the α is less than 0.4 for updates scaling up to 1000 entries and is found to be quite similar to the results shown for uniform data sets. This method is beneficial for large number of updates when the data sets are randomly distributed and the join selectivity is low.

5.4 Performance of SJB Indices Measured for Different Data Distributions

The percentage of bit set to '1' for uniform, zipfian and real data sets is given in Figure 8. $L1$ represents level 1, $L2$ represents level 2, and so on. It shows that for real and zipfian data sets, the number of bits set is comparatively lesser than the bits set for uniform data sets. For zipfian data sets (Z1 and Z2), large number of entities concentrate on few partitions, while most of the partitions at the lower levels are empty. The percentage of bits set for zipfian data sets is less but this does not decrease the number of pages processed as the intersecting entities are found in those partitions which have large number of entities. The join selectivity is found to be high in zipfian data sets and hence, the number of pages processed does not decrease. For uniform data sets (U1 and U2), the percentage of bits set is comparatively high but reduces the number of pages processed. The entities are equally distributed across the partitions and the number of bits set to '0' relates to the number of pages ignored during processing. Although the bits set to '0' is comparatively less, it affects the number of pages processed considerably as the join selectivity is low. For real data sets (SD and LB), the join selectivity is low and hence the number of bits set is also less.

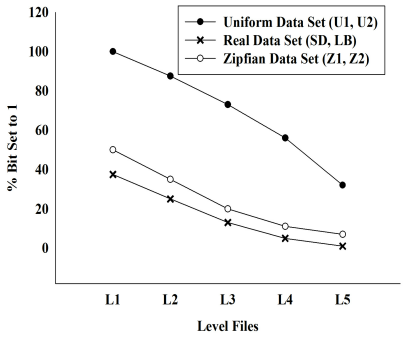


Figure 8: Percentage of Bit Set to '1' for Uniform, Zipfian and Real Data Sets

Figure 9 represents the response time for performing join with the uniform (U2 and U3) and real data sets (RA and LB) in the uniform-real percentage. The real (RA and LB) and uniform data sets (U2 and U3) are sampled to have 100,000 and 50,000 entities respectively. The 100-0 represents the join performance when the data sets are of uniform distribution. The response time considerably decreases as the data sets are mixed with real data sets as can be seen in Figure 9. The decrease in the response time is due to the considerable decrease in the join selectivity and the decrease in the number of pages processed. The 90-10 represents the join performance when the data sets are 90% uniform and 10% real. This shows that there is a decrease in the result set by 25% which results in the decrease of response time. The entities are evenly distributed in case of uniform data distribution and every partition has an entity at all levels. When the data sets are mixed with real data sets in noted percentages, the existence of entity in every partition is not seen and the existence of intersecting entities at almost every partition which is the case in uniform data sets is no longer seen.

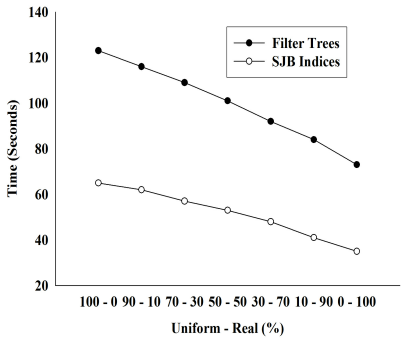


Figure 9: Join Performance for Uniform and Real Data Sets

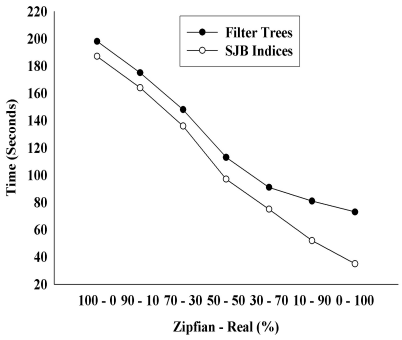


Figure 10: Join Performance for Zipfian and Real Data Sets

The benefit of using SJB indices is clearly seen when the real data sets are used. The benefit of using SJB indices in real data sets is shown in the Figure 9 and Figure 10, which is almost 148% in case of response time and 100% in case of the number of page processed. The 0-100 represents the join performance for the real data sets. The improvement in the response time for the real data sets is due to the decrease in the pages processed and the low join selectivity.

Figure 10 represents the response time for performing join with the zipfian (Z2 and Z3) and real data sets (RA and LB) in the zipfian-real percentage. The response time and the number of pages processed decrease at the scale of 40% as the data sets are mixed with real data sets in noted percentages. The number of intersecting entities decreases substantially when the data sets are mixed with the real data sets and this leads to the decrease in the response time and number of pages processed.

The use of SJB indices does not decrease the response time of zipfian data sets when compared to the uniform and real data sets and it is due to the lack of decrease in the number of pages processed. In case of zipfian data distribution, join selectivity is high and it attributes to the increase in the response time. The high join selectivity is due to the fact that the large number of entities concentrate on a few partitions. The percentage of improvement in response time is shown in Figure 11.

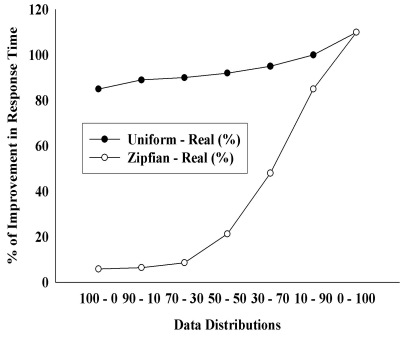


Figure 11: Percentage of Improvement in Response Time while using SJB Indices

The lack of improvement in response time for data sets of uniform distribution is due to lack of difference in the number of pages processed and for the zipfian distribution it is due to the high join selectivity. Figure 11 shows that the SJB indices are beneficial in data sets of uniform distribution when compared to the data sets of zipfian distribution. The high join selectivity results in the decrease of improvement in performance for data sets of zipfian distribution. The SJB indices perform extremely well when the join selectivity is low and the data sets are randomly distributed. However, SJB indices perform better than Filter trees for data sets of various distributions. The SJB indices could be further optimized by choosing the value of L. Beyond a certain level cl, the number of entities found at the levels lower than cl is found to be extremely less (for instance, could be stored in less than one page). In which case, L could be fixed to cl and the entities that fall beyond cl could be restricted to level cl.

6 Related Work

[Brinkhoff et al., 1996] is the most widely used algorithm because of its efficiency and the popularity of the R-trees. Several spatial join algorithms have been proposed for the cases that only one of the inputs is indexed by an R-tree [Lo and Ravishankar, 1994, 1995] or when both inputs are not indexed [Koudas and Sevcik, 1997, Lo and Ravishankar, 1996, Patel and DeWitt, 1996]. The join operation is an expensive operation in relational databases and several methods have been proposed for optimizing join operations. A join index facilitates in rapid query processing and for data sets that are updated infrequently, the join index can be very useful [Gunther, 1993, Koudas, 2000, Rotem, 1991, Valduriez, 1987]. [Shekhar et al., 2002] uses the method of clustering to compute the efficient order of page access when the memory buffer size is $< 10\%$ relative to the size of the relations. Given a fixed buffer size and a buffer size which is relatively small than the size of the relations, the join index helps in finding the order of page access which could substantially reduce the number of redundant page re-accesses.

[Chan and Ooi, 1997] examines the issue of scheduling page accesses in join processing, and proposes new heuristics for an optimal page access sequence for a join such that there are no page re-accesses using the minimum number of buffer pages, and an optimal page access sequence for a join such that the number of page re-accesses for a given number of buffer pages is minimum. [Brinkhoff et al., 1993] uses a buffer pinning technique to reduce the number of page re-accesses. Filter trees proposed by [Sevcik and Koudas, 1996] uses the concept of hierarchical representation to perform the join with the minimum number of page reads. Filter trees outperform R-trees [Abel and Smith, 1983] by reading each page only once. [Antoine et al., 2009] illustrates the extension of this work by the authors to employ recursive paritioning for trajectoies; which is the recorded instances of a moving object with respect to time. They have illustrated the methodology for indexing trajectories in the unrestricted space. This recursive paritioning method can be furthur studied on the problem of processing continuous intersection joins over moving objects [Zhang et al., 2008].

7 Conclusion

We have presented the methods for generating Spatial Join Bitmap (SJB) indices and have shown through experimental results that SJB indices improve the performance of spatial join queries substantially. We have presented algorithms to dynamically construct SJB indices and to effectively handle updates and have shown through experimental results that the extra cost incurred while maintaining SJB indices for updates is balanced with the substantial cost saved in processing subsequent joins. This method is highly beneficial in the real world scenario as the number of

times the data set is updated is fairly low when compared to the number of times the join processing is done on the data sets. We have presented the page requirement for storing SJB indices, which is considerably less in every case and can be kept in memory. We have shown that the SJB indices perform better than Filter trees through a series of experiments conducted on data sets of various distributions. We have also shown that SJB indices outperform Filter trees by 150% for real data sets.

Acknowledgements

This work is sponsored in part by National ICT Australia (NICTA).

References

Bureau of the census. *TIGER/Line Census Files*, 2000.

David J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24:1–13, March 1983.

Elizabeth Antoine, Kotagiri Ramamohanarao, Jie Shao, and Rui Zhang. Recursive partitioning method for trajectory indexing. In *Proceedings of the 21st Australasian Database Conference (ADC 2010)*, October 2009.

Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 237–246, May 1993.

Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Parallel processing of spatial joins using r-trees. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 258–265, February 1996.

Chee Yong Chan and Beng Chin Ooi. Efficient scheduling of page access in index-based join processing. *IEEE Transactions on Knowledge and Data Engineering*, 9:1005–1011, November 1997.

Oliver Guenther. Evaluation of spatial access methods with oversize shelves. *Geographic Database Management Systems*, pages 177–193, May 1991.

Oliver Gunther. Efficient computation of spatial joins. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 50–59, April 1993.

Gershon Kedem. The quad-cif tree: A data structure for hierarchical on-line algorithms. In *Proceedings of the 19th International Conference on Design Automation*, pages 352–357, June 1982.

Nick Koudas. Indexing support for spatial joins. *Data and Knowledge Engineering*, 34:99–124, August 2000.

Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 324–335, May 1997.

Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 209–220, May 1994.

Ming-Ling Lo and Chinya V. Ravishankar. Generating seeded trees from spatial data sets. In *Proceedings of the 4th International Symposium on Advances in Spatial Databases*, pages 328–347, July 1995.

Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. *ACM SIGMOD Record*, 25:247–258, June 1996.

Jack A. Orenstein and Frank Manola. Probe spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14:611–629, May 1988.

Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. *ACM SIGMOD Record*, 25:259–270, June 1996.

Doron Rotem. Spatial join indices. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 500–509, April 1991.

Kenneth C. Sevcik and Nick Koudas. Filter trees for managing spatial data over a range of size granularities. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, pages 16–27, September 1996.

Shashi Shekhar, Chang-Tien Lu, Sanjay Chawla, and Sivakumar Ravada. Efficient join-index-based spatial-join processing: A clustering approach. *IEEE Transactions on Knowledge and Data Engineering*, 14:1400–1421, November 2002.

Hans-Werner Six and Peter Widmayer. Spatial searching in geometric databases. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 496–503, February 1988.

Patrick Valduriez. Join indices. *ACM SIGMOD Record*, 12:218–246, June 1987.

Rui Zhang, Dan Lin, Kotagiri Ramamohanarao, and Elisa Bertino. Continuous intersection joins over moving objects. In *International Conference on Data Engineering*, pages 863–872, 2008.